TECHNISCHE UNIVERSITÄT DARMSTADT

Enriched Event Streams: A General Platform For Empirical Studies On In-IDE Activities Of Software Developers

Dissertation approved by
Technische Universität Darmstadt
Fachbereich Informatik

for the degree of
Doktor-Ingenieur (Dr.-Ing.)

by
Sebastian Proksch M.Sc.

born in Offenbach am Main, Germany

Examiner:	Prof. Dr. Mira Mezini
Co-Examiner:	Prof. Dr. Walid Maalej
Date of Submission:	31.05.2017
Date of Defense:	25.08.2017

Darmstadt 2017

D17

License

Revisions

November 19, 2017: Polished for Publication

- Added License
- Added Revision Overview
- Added Acknowledgements
- Removed Thesis Statement
- Minor Corrections of Typos, Grammar, and Broken References

May 31, 2017: Submitted Version for Thesis Defense

Abstract

Current studies on software development either focus on the change history of source code from version-control systems or on an analysis of simplistic in-IDE events without context information. Each of these approaches contains valuable information that is unavailable in the other case. This work proposes *enriched event streams*, a solution that combines the best of both worlds and provides a holistic view on the in-IDE software development process. Enriched event streams not only capture developer activities in the IDE, but also specialized context information, such as source-code snapshots for change events. To enable the storage of such code snapshots in an analyzable format, we introduce a new intermediate representation called *Simplified Syntax Trees* (SSTs) and build C$\boxed{\text{A}}$RET, a platform that offers reusable components to conveniently work with enriched event streams. We implement FeedBaG++, an instrumentation for Visual Studio that collects enriched event streams with code snapshots in the form of SSTs and share a dataset of enriched event streams captured in an ongoing field study from 81 users and representing 15K hours of active development. We complement this with a dataset of 69M lines of released source code extracted from 360 GitHub repositories. To demonstrate the usefulness of our platform, we use it to conduct studies on the in-IDE development process that are both concerned with source-code evolution and the analysis of developer interactions. In addition, we build recommendation systems for software engineering and analyze and improve current evaluation techniques.

Zusammenfassung

Aktuelle Studien über Software Engineering konzentrieren sich entweder auf den Änderungsverlauf von Quelltext in Systemen zur Versionskontrolle oder auf eine Analyse von einfachen in-IDE Ereignissen ohne jegliche Kontextinformationen. Beide Ansätze nutzen wertvolle Informationen, die im anderen Fall nicht zu Verfügung stehen. Diese Arbeit stellt *angereicherte Ereignisströme* vor, eine Lösung, die das Beste beider Welten vereint und einen ganzheitlichen Blick auf den Prozess der Softwareentwicklung in der IDE ermöglicht. Angereicherte Ereignisströme erfassen nicht nur die in-IDE Aktivitäten eines Softwareentwicklers, sondern enthalten auch spezialisierte Kontextinformationen, beispielsweise Momentaufnahmen von Quelltext bei Änderungsereignissen. Um die Speicherung solcher Momentaufnahmen in einem analysierbaren Format zu ermöglichen, stellen wir eine neue Zwischendarstellung vor, *Vereinfachte Syntax Bäume* (SSTs), und erzeugen C$\boxed{\text{A}}$RET, eine Plattform, die wiederverwendbare Komponenten bereitstellt, um komfortabel mit angereicherten Ereignisströmen arbeiten zu können. Wir implementieren FEEDBAG++, eine Instrumentierung für VISUAL STUDIO, die angereicherte Ereignisströme mit Momentaufnahmen von Quelltext in Form von SSTs sammelt. Wir teilen einen Datensatz von angereicherte Ereignisströme, den wir in einer andauernden Feldstudie von 81 Teilnehmern gesammelt haben und der 15T Stunden aktiver Entwicklungsarbeit enthält. Wir ergänzen ihn mit einem Datensatz von 69M Zeilen von veröffentlichtem Quelltext, den wir aus 360 GITHUB Repositories extrahiert haben. Um den Nutzen unserer Plattform zu zeigen, führen wir Studien über den in-IDE Entwicklungsprozess sowie über die Entstehung von Quelltext mit ihr durch. Zusätzlichen erzeugen wir Empfehlungssysteme für Software Engineering und analysieren und verbessern aktuelle Techniken für deren Evaluation.

Publications

This thesis is based on the following publications, parts of them are used verbatim.

- *"Enriching In-IDE Process Information with Fine-Grained Source Code History"* [193].
- *"A Study of Visual Studio Usage in Practice"* [4].
- *"A Dataset of Simplified Syntax Trees for C#"* [189].
- *"FeedBaG: An Interaction Tracker for Visual Studio"* [3].
- *"Evaluating the Evaluations of Code Recommender Systems"* [190].
- *"How to Build a Recommendation System for Software Engineering"* [191].
- *"Intelligent Code Completion with Bayesian Networks"* [192].
- *"Towards Standardized Evaluation of Developer-Assistance Tools"* [187].
- *"Enriched Event Streams: A General Dataset For Empirical Studies On In-IDE Activities Of Software Developers"* [188].

I was also involved in the following publications hat are not covered in this thesis. These works are supplemental to this thesis or build upon results.

- *"On the Positive Effect of Reactive Programming on Software Comprehension"* [212].
- *"Addressing Scalability in API Method Call Analytics"* [26].
- *"Method-Call Recommendations from Implicit Developer Feedback"* [2].
- *"Empirical Study on Program Comprehension with Reactive Programming"* [211].

My master thesis [186] laid the ground work for one of the papers that is included in this thesis [192]. Compared to the original thesis, significant extensions were made: the addition of parameter call sites, a significant extension of the evaluation (both programmatically and through additional experiments), a complete revision of the textual description, and a fundamental improvement of the implementation.

Editorial notice This thesis is my own work, but it would not have been possible without the valuable feedback from my co-authors, colleagues, students, and from the anonymous reviewers of my work. Out of respect for all these brilliant minds, I will refer to "we" and "us" throughout this thesis, whenever I refer to its creator.

Preface

When I look back, I never thought of myself as a researcher. I started my own business while being a Bachelor student and I was very involved with getting "real work" done for my customers. Back in the time, I was very close to quitting my studies and I did not really understand the value of science. This changed once I realized that science is not only about theories that are being discussed in an ivory tower, but that it can actually have a very practical impact. I stopped working as a professional software developer and dedicated my time to learn the art and science of software engineering in a more structured way. I focused my Master studies on this field and finished my studies with a thesis that introduced a recommendation system for software engineers. During this time, I discovered my passion for software engineering tools. I happily accepted the offer to join the software technology group at TU Darmstadt as a PhD student, which allowed me to continue working in that area. These last five years have been a journey that I have not undertaken alone. In the following, I would like to express my deep gratitude to several people that I owe a big thank you.

My time as a PhD student was significantly formed through my participation in the *Software Campus*, a German federal project that aims to improve the education of Germany's next generation of managers by funding a project of significant size. In my project, I envisioned the instrumentation of an IDE to better understand developers and to use the collected data as the base for better evaluation of developer tools. Overall, this has been the core idea of all the work that is presented in this thesis. I'm super grateful for this support that allowed me to establish my own research agenda early on and I would especially like to thank my supervisor, Mira Mezini, for giving me the freedom to follow this agenda. The path has not always been straight, but I highly appreciate that you always believed in me and trusted my judgement.

The funding I received in the Software Campus allowed me to hire Sven Amann as a helping hand for implementing the initial FeedBaG data collector. Sven is one of the best programmers I know and also happens to be a great researcher. I was really lucky to find someone with such a rare combination of skills. I will never forget our countless discussions about design and architectural questions, about the planning of the project that was required to coordinate our "army of assistants", and the pair programming sessions that lead to the high-quality standard of our tools. Thanks Sven, the work presented in this thesis would not have been possible without you!

One of the best things that happened to me during my PhD was having the chance to work together with Sarah Nadi who joined our *recommenders subgroup* as a postdoc. You always took your time whenever I needed help and you provided me with very valuable feedback and many suggestions on my work. I am really wondering how many red pens would have been required for all your corrections of my papers, if you would have chosen actual print-outs over your tablet. I'm very grateful for your passion and dedication and I could not have asked for a better advisor. I'm super happy for you that you have found the Professor position that you were looking for and that you deserve. Canada is far away, but one of the best things of pursuing an academic career is that we will regularly meet again at international conferences.

I was lucky to have had an in-official second advisor, Guido Salvaneschi. In the beginning, we just shared an office, but what started with cookies and extended discussions about life, soon became a great example of cultural exchange. Highlights have been the extended hiking trip in Bergamo or the introduction to the Hessian culture of *Apfelweinkeltern*. We are working in different areas of software engineering, but you have an excellent overview over the software engineering community as a whole and I learned a lot from you about the academic world. I admire your calm and thoughtful nature and I highly value your external perspective on my own work that made me re-think my own ways and believes more than once.

My special thanks go to Gudrun Harris, the heart and spine of the software technology group. You were always available for a good chat over a coffee and always had an open door whenever I needed to let off steam. However, your real super power is knowing the secret ways through the workflows of the university administration, it feels like you know *everybody* over there. I'm very grateful that you had my back more than once, when I missed deadlines or messed up some paperwork.

In addition to my colleagues with whom I have directly worked together on research topics, I'd also like to thank all my co-authors Veronika Bauer, Moritz Beller, Georgios Gousios, Gail C. Murphy, Annibale Panichella, and Andy Zaidman, as well as all my excellent students that worked with me on the KaVE project through labs, theses, or as payed assistants: Dennis Albrecht, Andreas Bauer, Marcel Brand, David Dahlen, Uli Fahrer, Roman Fojtik, Gerrit Freise, Waldemar Graf, Florian Jakob, Sven Keidel, Mattis Kämmerer, Ulf Karrok, Can Pekesen, Simon Reuß, Rameez Saleem, Jonas Schlitzer, Ivan Todor, Felix Weirich, Alexander Weitzmann, Markus Zimmermann. It was not only very fun to work with you, your fresh perspectives on my work also constantly challenged my ideas and provided me with new impulses and ideas for improvement. You added countless and invaluable contributions to my work, both in terms of research and implementations, and I appreciate that you stayed with me, even though I had very high expectations on your work.

Getting negative feedback on your work is generally discouraging, but nothing can be as devastating as a paper rejection, when you gave all your blood and sweat. Every student will have to experience this themselves and it took me a while to get used to it. However, after playing the papers game for some time, I'm convinced now that the system works and that reviews, as brutally direct as they can be, help to further improve the quality of one's work. With a bit of distance to the immediate disappointment after rejects, I found value in most reviews and I would like to thank all the anonymous reviewers of my work for their time and effort, even reviewer 2.

It is nice to have close colleagues that make the time in the office so much more enjoyable. I would like to thank Ben Hermann, Johannes Lerch, Michael Reif, and Leonid Glanz for the occasional ice cream or *Mettbrötchen* breaks, all you can eat Sushi lunches, burger dinners, or fun chats over lunch at Mensa.

I'd also like to thank the remaining members of the software technology group with whom I had the pleasure to work over the last years. Andi Bejleri, Oliver Bracevac, Ervina Cergani, Joscha Drechsler, Michael Eichberg, Matthias Eichholz, Sebastian Erdweg, Sylvia Grewe, Sven Keidel, Ralf Mitschke, Christian Klauß, Edlira Kuci, Ragnar Mogk, Andreas Sewe, Jan Sinschek, Jurgen van Ham, Manuel Weiel, and Pascal Weisenburger. A day in the office would not have been the same without you.

I would like to thank Waldi Maalej who provided substantial feedback on the final draft of my thesis. Your suggestions were very helpful and I think that incorporating them has significantly improved the thesis. I would also like to thank Harald Gall who decided to hire me at the University of Zurich, even before I defended my PhD thesis. Thank you for your support and for your faith in me.

Finally, and most importantly, I owe a big thank you to my family and to all my friends. I know that I have not been the easiest companion over the last years and that I was quite stressed-out at times, but I'm eternally grateful that I could always rely on you when it counted or when I needed you.

I have the best family one can wish for. I would like to thank my parents for supporting me throughout school, university, and my academic career so far without ever questioning this path. It started early, when you gave me the chance to attend the Kreuzburg gymnasium, I know that this had put a high financial and organizational burden on you. I'm grateful that you endured it to build the foundation of my education and that you kept supporting me until I managed to stand on my own feet. My grandparents helped me a lot in my times of need, as well. You have shown me how important it is to care for each other as a family and I will always strive to be the good hearted, honest, and generous person that you taught me to be. Unfortunately, not all of you were still around to celebrate my graduation with me, but I will never forget you. Last but not least, I'd like to thank my brother Tobias. Whenever I need a hand or an honest second opinion for a decision, I know that you will always be there for me. I like that we do not always agree on everything. We are complementing each other and I'm very grateful for having you in my life.

Last, but certainly not least, I would like to thank my beloved girlfriend, Catherine. You are the only person that has witnessed first-hand all the highs and lows in these stressful past years and you helped me to survive. You have pushed me through periods of ~~procrastination~~ distraction, you have been the best motivator, whenever I was frustrated, and you have ensured that I take a break from time to time to gather new strength. You are constantly reminding me that living is so much more than just work, but you have always supported me nevertheless, even when this meant joining a phone conference during a city tour or working on a paper during our summer vacation. This thesis is as much a result of your care as it is of my work and I'd like to thank you for your support and understanding. You are bringing balance to my life and I could neither imagine how to be without you, nor would I ever want to.

My time as a PhD student has felt like a five-year roller-coaster ride. There have been so many ups and downs that I lost track. It has been the most exhausting and, at the same time, the most rewarding experience of my life, so far. I would like to thank everybody who was part of this journey, I could not have done it alone.

Contents

Part I:
Prologue

The first part of this thesis will pave the road for the content to come. We will start with a motivation of this work and provide background information to frame the work appropriately. After formulating a concise problem statement, we will list the contributions of this thesis and provide a high level overview over its content and their organization. As a preparation for later discussions, we will then introduce the state of the art in several areas that are related to this work.

1 Introduction

To assist developers in their everyday work, an understanding of developers' activities is necessary, especially how they develop source code. Ideally, this can be done by observing developers in their actual work environment. Several researchers have conducted such observational studies to, for example, assess developer productivity [140], evaluate the user interface of particular tools [157], or study software evolution [44]. Unfortunately, conducting such experiments is very expensive and researchers typically resort to other alternatives.

Two popular alternatives are studying historic development information *after the fact* through the artifacts that are created during the development process (e.g., source code files) and analyzing developer interaction data that was automatically captured while using an Integrated Development Environment (IDE). Examples of the former include using historic data to help developers with change summarization [163], bug localization [38], deriving related files based on their co-change probability [258], and predicting change locations [265]. Examples of the latter include analyzing developer interaction data to recommend artifacts related to the developer's current activities [102], improving developers' productivity [103], and understanding how developers spend their time [148]. We argue that considering historic data and in-IDE interaction data separately misses out on the bigger picture that is necessary to gain a complete understanding of software development.

Analyzing changes committed to public version-control systems (VCS) is the most common way of analyzing software artifacts after the fact (i.e., historic information) and is an important topic in the area of mining software repositories [80]. Some of the approaches in this area work on the level of changed lines [194], others are based on syntactic information [38], and some need fully-resolved types in the sources [265], meaning that the code has to be compilable first. With the rising number of open-source projects hosted on platforms such as GitHub or SourceForge, such data has become easily accessible. However, only looking at the artifacts developers create has the inherent problem that any conclusions drawn are affected by the granularity of the collected artifacts. Many intermediate modifications that are subsequently reverted or again modified will not be observed in such data. Previous work by Negara et al. [166] shows that VCS history is not representative of the actual code evolution.

To overcome some of the disadvantages of using VCS data, some researchers refine the granularity of the snapshots by auto-committing intermediate code versions from the IDE to a VCS whenever the developer makes a change in the editor [166, 217] or saves a file [231]. This approach creates a very fine-grained history of commits, which allows a closer inspection of the steps taken by the developer. Finer-grained information provides more insights about how developers write code, enabling us to produce better tools to support them. However, source code snapshots leave behind other relevant information that is available in the developer's IDE. One example is resolved type information, which is important for any static analysis on the snapshots. While it is usually available in the IDE, it can only be reconstructed from a captured code snapshot by compiling the snapshot. This is typically a hard task, as dependencies might be unavailable or compilation might rely on a project-specific environment. Another example is the edit location of the developer, which is known in the IDE, but can only be approximated from a historic change set. The more files involved and the more coarse-grained the history is, the fuzzier this approximation becomes.

Capturing only source code snapshots also leaves behind any information about the development process. The snapshots themselves do not explain how the current piece of code was developed, which might include usages of refactoring tools or code-completion engines. Previous work recovers this information using heuristics [72, 82], but it cannot generally be reconstructed precisely.

Lack of context may lead to missing or incomplete conclusions when studying developers and the code they produce. We argue that *capturing richer and more detailed context information is needed to facilitate the reasoning about captured code snapshots and to get more insights into the development process.* Previous studies have tracked developers' activities within an IDE to capture information about the development process, usually in the form of an event stream (e.g., [4, 102, 148, 230]). However, most of these studies only record command invocations without specific details about them (e.g., they capture that a specific refactoring was invoked, but not to which part of the code it was applied). This makes it hard to interpret the event stream and to align it with source code changes. To the best of our knowledge, only few studies combine the analysis of fine-grained source code evolution with in-IDE process and tool information (e.g., [13, 44, 231]).

In this thesis, we propose an approach to create development artifacts that combine process information with source changes. We capture an *enriched event stream* of development activities in the IDE that stores not only simple information about executed *commands*, but also *context* information to enrich the usefulness of these events. This provides a holistic picture of the developer's work and makes it possible, even after the fact, to answer questions that touch both source-code changes and information about the development process. We capture a significant dataset in a field study and create an infrastructure around it for experimentation. To demonstrate usefulness of our platform, we will conduct studies on the in-IDE development process that are both concerned with source-code evolution and the analysis of developer interactions. In addition, we will build recommendation systems for software engineering, for which we will analyze and improve the current evaluation techniques.

1.1 Scope of This Work

Studies in development activities and the evolution of source code are two vast fields that cannot be covered exhaustively in this thesis. We are especially interested in the following parts of these fields that will be introduced and discussed in this thesis.

Preservation of In-IDE Activities Create software involves many development activities in the IDE. We are interested in preserving them in a general meta-model that allows reuse of the captured activities in various studies on the development process. Writing code is a major activity in the development process that will be central in our model. Our goal is to capture a holistic picture of the in-IDE process that contains context information, which allows reasoning about the intent of the developer, and we want to design a platform and data schema that is extensible for future research questions.

Studies on the In-IDE Development Process Preserving interactions provides the required means to better understand the development process. Modern development environments support developers with many tools and integrated features and can be overwhelming for newcomers. We are interested in finding ways that allow to study questions like *"How do developers use IDEs?"*. In addition to such general questions, we also want to be able to answer detailed questions about development activities that concern very specific tools like *"Which parts of the code get changed after a failing test?"*. We want to answer these questions in large-scale empirical studies that include a significant number of developers.

Source-Code Evolution The most visible part of the development process is source code. We are interested in understanding how developers write it and how their programs evolve. *Application programming interfaces* (API) are a crucial aspect of modern source code development. A fine-grained source-code history allows use to study their evolution and the evolution of their clients. Moreover, the aligned combination with process information makes it possible to answer more advanced questions that go beyond source code like *"Which files or method implementations did the developer open and read while working with a specific API?"*. So far, the studies have often been restricted by the coarse commit-granularity of repositories. Our preserved activities provide the required details to ask more differentiated questions, e.g., *"What is the effect of using various history granularity levels when analyzing developer behavior, e.g., time-based, commit-based, or activity-based intervals?"*.

Building Recommendation Systems Developers use several tools in their daily work that support their development activities in the IDE, e.g., recommendation systems for software engineering (RSSE) that help them understanding and using APIs. We are interested in building our own recommendation system for developers and in analyzing how such systems are actually used in practice. Creating such tools involves very sophisticated static analyses. We are interested in finding ways that make it easier to statically analyze incomplete and potentially invalid source code.

Evaluation of Development Tools Researchers provide sophisticated evaluations to prove the value of their novel development tools. However, a lack of standardized data schemas makes it hard to replicate or reuse existing works and conducting empirical studies is very expensive. Researchers often fall back to artificial alternatives that beg the question *"How realistic are artificial evaluations?"*. We are interested in improving the standards of such evaluations and want to analyze whether recorded interactions of real developers can be used to establish a more realistic benchmark.

Tools & Datasets A relevant concern across all the previously described areas is the necessity to have analyzable data and shared tools that can be used in recurring tasks. We are interested in creating the required infrastructure to build such tools and in providing shared datasets that can be used by others in various studies.

A major research direction integrates information from additional non-IDE data sources, such as mailing lists, issue trackers, or meta-data of version control systems (e.g., HIPIKAT [36], RELEASE HISTORY DATABASE [57], SOFTCHANGE [69], ALITHEIA [78], or

Software Analysis Service [70]). While clearly related to our work, this line of work is orthogonal. It is concerned with a more holistic picture of the development process, but does not consider many in-IDE activities. Another interesting line of research captures information about in-IDE time of developers that goes beyond development activities and source code. Examples are biometric information (e.g., which code fragments they looked at [207]) or emotions in the development process [108]. Such information could be used to further enrich our stream of events. Integrating these directions into our work opens opportunities, but also certain challenges. Discussing these effects was out of scope for this work, but should be investigated in the future.

1.2 Problem Statement

Within the scope of this work, we have identified several problems that affect the experiments and empirical studies that are typically involved in these lines of research.

Hard to Get Started Researchers usually have a good understanding of a question they would like to answer when they start working on it. However, it is often not clear in advance which kind of data is required for finding the answer. Creating the tooling for the experiment is an incremental process: An initial prototype is developed and used to identify missing features or unhandled corner cases. As a result, the prototype is extended and the cycle is repeated until the required maturity is reached. This approach involves human developers and has two problems. The available resources might not suffice to create a tool stable enough to be used in a production environment right away and there might not always be a test group available that can provide this kind of feedback, which makes it hard to get started.

Expensive Designing an empirical study that involves tools and collecting data from a significant number of participants is an elaborate process and conducting it is very expensive. In addition to building the tool in question, it is necessary to design the study and to create the required infrastructure, i.e., the environment in which the experiment is run as well as the tooling that measures the performance of the subject. Finding participants is often the most challenging part of the process though. Depending on the setup and the goal of the experiment, participating in an experiment takes some time, so it might even be required to pay the subjects.

High Risk Building and running a program is cheap and fast and developers are used to *trial and error*. Unfortunately, this approach does not work for empirical studies. A study can typically not be repeated with the same subjects, because it would introduce learning effects. Consequently, the preparation of the study needs to be very thorough to avoid any complications in the execution. This high up-front effort for the experiment directly increases the risk for the experimenter. An experiment might need to be refined, if an experiment does not provide the expected results or if a methodological mistake was discovered after the fact. The necessity of repeating an experiment can be disastrous for the researcher, because it might require the acquisition of new participants.

No General Data Schemas Researchers that conduct empirical research naturally focus on a specific research question and limit the data collection of the experiment to the required data. Even though two tools produce the same kind of output, they might be hardly comparable, because the dataset of the one approach might miss information that is crucial for the other. We think that a lack of general data schemas that are applicable to more than one research question makes it hard to reuse existing data collections and limits the creation of a common platform with shared tools. No benchmarks exist for a specific kind of development tool that are general enough to allow an evaluation across differing approaches.

Hard to Replicate and Reproduce Recent studies on the state of reproducibility of research on mining software repositories has shown that the published studies are often not reproducible. Robles [206] finds that this is mostly the case, because created tools are often not available. Gonzáles-Barahona et al. [75] have analyzed the reproducibility of mining studies and found that the creation and publication of reusable tools and datasets (including raw data, training and validation data, and results) is important for reproducibility, which is often not achieved. Hemmati et al. [84] confirm this as a crucial, though often neglected, step of mining studies.

Unrepresentative Results Empirical studies are often conducted with students recruited from the close surrounding of the evaluated project or with only a very limited number of participants. Beel et al. [12] criticize these evaluation practices of recommendation systems. In addition, researchers often use synthetic benchmarks to avoid cost and risk of an empirical study. Beel et al. [11] find that synthetic evaluations of recommendation systems may be the cause of incorrect evaluations and that human evaluations can come to different conclusions. While both findings are derived in another area of research, the described factors are also relevant in software engineering. This begs the question, do the reported results of affected experiments in this area suffer from the same problems?

Difficult Project Setup Research on source-code evolution or RSSE typically involves a static analysis of source code that is downloaded from public source-code repositories. The analyses very often rely on resolved typing information, which requires the setup of all referenced project dependencies. Recovering these dependencies and setting up an environment in which a project can be build is hard. Dependencies might no longer be available or a project-specific setup might be required.

Invalid Source Code Sophisticated frameworks exist to facilitate the creation of static analyses on released source-code. However, in-IDE tools often have to analyze source code that is incomplete or even syntactically invalid. They cannot rely on frameworks and are often limited to more basic analyses. As a result, doubled effort is required to maintain two different static analyses. In addition, a discrepancy may exist between both analyses that affects the performance of the system.

The contributions of this thesis are motivated by this list of identified problems. We will revisit them in the conclusion of this work and will discuss there to which extent the problems can be avoided or mitigated using the results of our work.

1.3 Contributions

In this thesis, we introduce KAVE, a platform that enables researchers to work with a fine-grained change history and connect it to the development process. Figure 1.1 shows an overview of our infrastructure, including the tools C$\boxed{A}$RET and FEEDBAG++ and the data structures we built to enable this connection. We now briefly explain the figure and use it as an outline to introduce the contributions of this thesis.

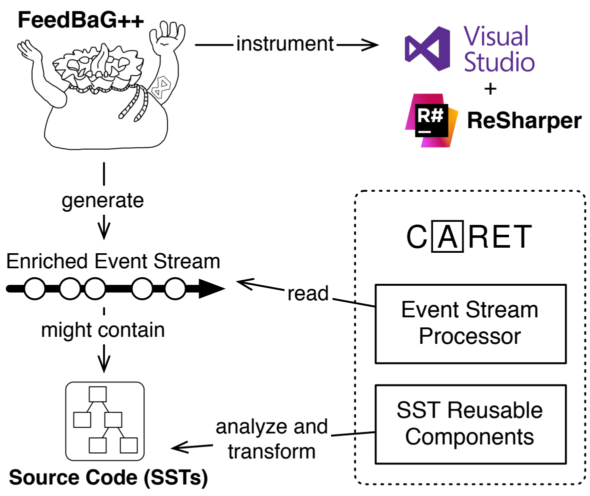


Figure 1.1: Overview over the KAVE Platform

Enriched Event Streams We introduce a meta model for in-IDE development. It captures the interactions of a developer in the IDE, enriched with context information that explains the actions. We especially focus on source-code evolution as the primary artifact of software engineering and capture snapshots of the edited files.

Simplified Syntax Trees We create a novel meta model, *Simplified Syntax Trees* (SST), for source code that features fully-qualified references to types and type elements. The representation is motivated from requirements we extracted from related work on RSSE and it facilitates writing static analyses. SSTs can be used to capture incremental and self-contained in-IDE source-code snapshots that are contained in some events of the enriched event stream (e.g., change events).

Based on both meta models, we have created a standardized infrastructure and tools that facilitate working with the representations.

Infrastructure We have build bindings for the meta models in JAVA and C# and provide tooling that allows to persist them to exchange data or for storing intermediate results. A reusable source-code transformation exists that can transform C# programs to their SST representation. We provide the transformation both in a bulk-transformation for repositories and as an in-IDE component that is part of the infrastructure that we built to facilitate the development of tools and research. We maintain a central service that is able to collect uploaded data from many developers all around the world.

FEEDBAG++ We created an extensible interaction tracker for VISUAL STUDIO, FEEDBAG++, that captures all the actions developers perform in-IDE. It generates an *enriched event stream* that not only captures which events take place, but also stores relevant context information about them. FEEDBAG++ can be used by anybody to generate new datasets. We share insights into its development and present a concept for privacy control that allows a deployment even in restrictive companies.

CⒶRET To facilitate research on developer interactions, we created a platform, CⒶRET, that makes it easy to work with the meta models. The platform provides several

Composable Analyses and REusable Transformations for recurring tasks. Apart from preprocessing components, we put a special focus on components that support working with the captured source-code snapshots and provide basic building blocks like a points-to analysis, or a method-inlining component.

To show the usefulness of the representations and the tooling, we used both in technical research that was enabled by the conceptual contributions of this thesis.

Datasets We have compiled two datasets that are available for research using C$\boxed{\text{A}}$RET. Using FeedBaG++, we have collected a significant dataset of developer interactions in a public field study. The dataset includes data from 81 developers that was captured during 1.527 individual days, totaling 15K hours of active in-IDE development time. We have also compiled a dataset of released source code from GitHub that can be used by source-based approaches to learn models or create statistics. The dataset complements the interaction data and contains released examples for observed API usage in the field study.

Applications To show the overall usefulness of the KaVE platform, we replicated several related works and also answered novel research question with it. The experiments that we have conducted based on C$\boxed{\text{A}}$RET are part of the evaluation, but we see many more cases, to which the enriched event stream data is applicable.

The remainder of this thesis is structured as follows. Part I will be completed by a survey of the state of the art in related research areas in Chapter 2. We will design two meta models in the following Part II to capture both developer interactions (Chapter 3) and source code snapshots (Chapter 4). Part III describes how these meta models are being instantiated. To this end, Chapter 5 contains details about the implementation of our infrastructure and research platform. Chapter 6 introduces the interaction tracker FeedBaG++ that instruments the IDE. Chapter 7 presents the details about C$\boxed{\text{A}}$RET, the platform we have build around our meta models. Part IV contains a detailed presentation of how we have used our tooling in our own research. We will start in Chapter 8 by introducing the datasets that we have compiled. Afterwards, we will show how our platform can be used for studies on the development process (Chapter 9), for building recommendation systems for software engineering (Chapter 10), and for realistic evaluations of such systems (Chapter 11) to show the usefulness and applicability of the various components available in C$\boxed{\text{A}}$RET. We will discuss the threats to validity in Chapter 12. Part V concludes this thesis. We reflect on our experiences and insights from the experiments (Chapter 13), present an outlook on future work (Chapter 14), and summarize this thesis (Chapter 15).

2 Background and Foundations

The work presented in this thesis touches many related research areas. In this chapter, we will survey a broad selection of previous works to establish the necessary background and to build the foundations for the work presented later. Our survey will be split into two separate sections that address different levels. Section 2.1 will provide the required background for this thesis by introducing several applications that motivated our work. Section 2.2 will discuss existing solutions to inherent problems of the applications, like satisfying their information need. We will discuss all related work in this chapter and refer back to them in the remaining parts of this work.

2.1 Application Domain

Three applications motivated the research presented in this thesis. In this section, we will introduce these lines of work as a background for later considerations about requirements or the design of a system. We were interested in empirical studies on the developers that analyze how they work and we will introduce related work in Section 2.1.1. Over the last years, many systems have been proposed that assist developers in their daily work, affecting their decisions, and making them more productive. We will review these *Recommendation Systems in Software Engineering* (RSSE) in Section 2.1.2 to understand how developers can be better supported and also how the creation of such tools can be further improved. Finally, the most important artifact of software engineering is source code. Developers spent much time creating and maintaining it and we were interested in studying these activities. Section 2.1.3 surveys existing works on source-code evolution.

2.1.1 Empirical Studies on Developer Activities

Many works in the software engineering area propose tools that are built for developers, so understanding the real needs of these target users is a crucial task. An example is creating RSSE, which is a complex problem that consists of several subsequent tasks that need to be solved to get from an idea to a working recommendation system [191], e.g., framing the actual problem of the user or identifying a proper visualization of the proposals. This challenge is not specific to RSSE though and examples can be found in many research lines. Especially in the exploratory phases before a tool is being built, often traditional methods are used to examine software developers. Qualitative methods are applied to explore the field or to get a holistic picture of the observed activities, when it is unknown in advance, which properties should be observed.

Exploration A great way to explore a new idea or to check assumptions in the field is having experts answering questionnaires [23, 154, 159, 251]. The results allow to verify or invalidate intuitions or to identify important topics to focus on. Questionnaires are often used as a first step to guide later experiments and to avoid inappropriate expectations. However, if the researcher cannot fall back upon previous results or if the research question touches novel concepts, questionnaires are not an appropriate technique and more observational techniques need to be applied.

In these cases, it might be necessary to built a prototype or a mock-up of the system under analysis. Once this is built, developers can be observed using it or interacting with it. For example, Bragdon et al. [19] propose a new paradigm for code editing, CodeBubbles, in which source code is no longer organized by files, but in visualized *bubbles* that represent source code blocks. These *bubbles* float in a pane and get connected when the developer starts navigating the code base. To evaluate the usefulness of the approach, the authors conduct a qualitative experiment, in which professional developers work with a prototype of the editor on specific tasks. They capture the screen of the participants and their verbal statements during the experiment to enable an analysis of the interactions and a reconstruction of the mental model after the fact. The same methodology was used by Get et al. [68] in a formative study they conducted to understand how developers manually perform refactorings. Another option is to conduct structured interviews with users of the prototype to gain general feedback about the system or the idea [90, 244].

Qualitative techniques are typically combined with other experiments and are often used in subsequent steps. For example, a questionnaire might reveal interesting questions that are then analyzed in an observational study with a group of experts, or insights of an observational study might be discussed with experts in unstructured interviews afterwards. Many more options for qualitative research exist [130, 191], e.g., *AEIOU* [248], *Contextual Inquiry* [17], *Think Aloud Protocol* [62], *Laddering* [197], *Literature Reviews* [18], *Concept Mapping* [173], *Personas* [49], or *Scenarios* [25]. The challenge is to pick the appropriate technique for the research question, an exhaustive discussion of these works is out of scope for this work though.

Having a working system provides a great opportunity to research its usage. However, building such a system usually means spending a huge upfront effort, which dramatically increases the risk. Optimally, researchers should make sure they do the right thing before actually making it to save time and energy. The slogan "fake it until you make it" does not only hold for entrepreneurs, but also for research prototypes. To show the usefulness of a system it is not required to actually build all its parts. In "Wizard of Oz" experiments [133], parts of the system are faked to make the system seem to work, without having to implement it. For example, in the context or RSSE, such techniques can be used to decide whether a recommendation system adds value for the user, before implementing the whole recommender. Creating a fake recommender avoids any negative influence of bad recommendations and allows evaluating UI choices of the presentation independently in a traditional usability study [157].

Development Process Another line of research in software engineering does not focus on a specific tool, but is interested in understanding the general development process. In this case, a holistic picture of the process is required that shows how developers are working in their regular work environment.

An early work in this direction was an experiment of Perry et al. [178] who analyzed the impact of social processes on development. To this end, they used time cards to collect information about the tasks of the observed developers and the amount of time they spent on them. In a first experiment, the let the developer fill out these cards, in a subsequent second experiment, they also filled out the cards themselves, after creating a development diary while watching the developers. Their results revealed several insights regarding social aspects of software engineering. While being able to find examples with significant differences, they concluded that the self-estimation of developers correspond sufficiently with the externally measured numbers.

A similar experiment was conducted several years later by Meyer et al [140]. They analyzed the perceived productiveness by developers and compared it to observed activities that led the developers to this feeling, with a special focus on context switches. They started with a questionnaire to identify important properties of a "productive day", followed by a *structured observational study* [151] (manual observation and creation of log) with eleven participants. After screening the resulting data, they got back to the participants and conducted semi-structured interviews to get a better understanding of the observed data.

While these two examples use manual observation and log creation in their experiments, it is also an option to record the activities and study the recorded stream after the fact (e.g., [19]). If a manual observation is not an option, for example, if the required effort is to high, infeasible (e.g., when the data collection spans several days or weeks) or if only a low level of detail is required, the findings of Perry et al. suggest that the process log can also be kept by the participant himself [178]. An example of this approach is a study by Böhme et al. [23], who analyzed how professional software developers are debugging programs. They sent out an initial questionnaire to learn about typical debugging activities. To complement the answers with practical data, they compiled a set of 27 bugs from open source projects and hired 12 professional developers to debug and fix them. The developers produced bug-fix patches, reported the time they spent on the different tasks, and elaborated the cause and the fix of the bug. Böhme et al. derived insights about the debugging process from the time cards and from the detailed responses about the tasks and also compiled a benchmark from the fixes that serves for an automated evaluation of bug detectors.

We do not claim that this selection of studies on the development process is in any way exhaustive. The presented studies only serve as examples and helped us to illustrate several ideas about empirical research on developers.

Overall, empirical research in software engineering and especially on developers is typically very exploratory. Qualitative approaches are often used as a first step in a series of experiments. Researchers usually apply these techniques, when they either do not know in advance in which information they are interested in, when the interesting information is too complex to capture automatically, or when they want to find out why a specific phenomenon can be observed.

In empirical research, researchers have to plan their studies or experiments, find participants, execute their plan, and analyze the data. This requires a lot of time, which might be the reason, why most of the time techniques with a low entrance barrier are used (e.g., interviews with a small number of participants or questionnaires). Once participants are involved that actually have to develop software in the empirical study, the required effort and the cost significantly increase. In addition, the resulting quality strongly depends on the selection of the participants and the task design.

2.1.2 Recommendation Systems in Software Engineering

Recommendation systems in software engineering are a strong motivation for our work. We are interested in understanding how developers can be assisted in their daily work and also how the creation of such assistance tools can be further improved. In this section, we will survey several RSSE to better understand this use case and we will focus on two areas. First, how is static analysis applied on source code and which properties are required from an intermediate representation. Second, we carefully review the evaluation methodology of RSSE approaches.

While we did no systematic literature review, we still wanted to cover a broad selection of techniques. Therefore, we consulted a recent comprehensive survey [204] and the RSSE book [205] to identify related work and also added several popular RSSE and some recent publications at premier software engineering conferences.

Static Analyses

In modern software development, *Application Programming Interfaces* (API) are a means to encapsulate responsibilities and facilitate code reuse and they are widely used in statically typed programming languages. Typical RSSE approaches apply static analyses to extract information about the usage of these APIs from the source code of many projects. Some RSSE techniques learn general patterns from source-code by treating it as text or just on the syntax level (e.g., [87]). This makes these approaches applicable to dynamically typed languages like JavaScript, but they are either restricted to very general tasks (e.g., they can predict the next syntax token) or have limited precision (e.g., they could propose invalid completions). While we include these work in our survey, we want to leverage the additional information that is provided by APIs and, therefore, mainly focus our survey on source-based RSSE that consider the additional information that is available in the structure and the context of source code. We will group the surveyed recommenders by their recommendation task and present a brief introduction to each paper or technique. We will discuss the static analysis and highlight unusual assumptions or special requirements of an approach. Special focus will be the scope of the analyses (i.e., method or class scope, inter/intra-procedural analysis) and the applied measures for disambiguation of code elements (i.e., whether fully-qualified types and method signatures are used).

Call Recommenders A call recommender suggests most likely subsequent method call(s) or parameter(s) to a developer. While the base line does only consider the type system to propose all possible methods, the intelligent systems presented in this section can reduce this list to a likely subset. This does not only make it easier to find a wanted method, it also helps novices to identify previously unknown methods.

McCarey et al. [137] introduced one of the early call recommendation systems that proposed which methods should be used in the current class under edit. The underlying static analysis is based on a simple traversal of the class syntax tree, in which all invoked methods are captured. Due to the simplified encoding, they ignore statement order in their analysis. Additionally, the analysis does not differentiate between different method declarations of a class. Therefore, it is not necessary to track the control flow or to normalize complex expressions.

Bruch et al. [22] also built a call recommendation system that was later extended by Proksch et al. [192]. The underlying approach in both cases identifies object instances in an intra-class analysis and extracts all method invocations on each instance, as well as a description of the surrounding source code. The original publication considered object types, enclosing method, and method calls. Additionally, they replace all captured method references with the first occurrence of the method signature in the type hierarchy. The extended work adds definition sites, parameter sites, and enclosing class. The analysis does not preserve order information of the captured method invocations. The authors use an advanced intra-class points-to analysis to identify all available instances and they implemented a tracking approach that follows the control-flow into method invocations in the same class.

Zhang et al. [261] propose a recommender system that focuses on parameter call sites. They extract several features from the structural context that describe the method invocation and its parameters: the called method, the enclosing method, methods that are called on the receiver, and methods that are called on the parameter. They do not consider order information and collect all called methods in a set. Method parameters can be nested expressions, however, they skip parameters if the expression is too complex. Thus, there is an opportunity to improve their technique by applying normalization of such expressions. Since they do not mention any extended static analysis in their description, we assume that they heuristically use variable names to distinguish different target objects.

Heinemann et al. [83] mainly use identifiers to predict method calls. They extract all identifiers used in the source code and split names on camel-case humps. Even though control structures are not part of their model, they consider control structure keywords in the tokenization. They apply text-mining techniques such as stemming or removing stop words to unify the collected tokens. Their static analysis collects all method invocations from the source code, together with the n preceding tokens.

Amann et al. [2] implemented a tool that learns correct API usage from interactions of developers in their IDE. When code completion is triggered in the IDE, they extract features from the structural context around the trigger point that include the type, definition, enclosing statement, expression type, enclosing method. They also store the selected proposal from the completion popup. The paper does not perform a points-to analysis, but it implements a heuristic for the identification of definition sites that is based on the tracking of parameter names, variable declarations, and assignments. The enclosing method context is rewritten, therefore, it is necessary to look-up the info in the type system.

Raychev et al. [195] solve the task of call recommendation by mapping it to a text-mining problem. They model sequences of methods calls as sentences. Missing method calls are similar to holes in a sentence that can be filled by calculating likely candidates. They use an intra-procedural static analysis that extracts all sequences of method invocations in a method body. They use an underlying points-to analysis to differentiate between different object instances.

Asaduzzaman et al. [5] present CSCC, a context-sensitive code completion system that is built on a simple and efficient algorithm. The approach tokenizes source-code by traversing an AST. Tokens will be created for Java keywords, types, and, method names. However, types and method are not resolved and only the simple names as written in the source file are used. For each method invocation found in source-code, the approach considers all tokens of the previous four lines as context. They mine public repositories and extract many mappings from context to invocation. The actual proposals are inferred in two steps. First, a hashing technique is applied to eliminate the majority of irrelevant method calls. Second, more expensive text similarity metrics are used to calculate the ranking. The evaluation shows in a comparison to the BMN recommender [22] that simple technique might provide a similar performance or even outperform more advanced algorithms.

Snippet Recommender Similar to call recommenders, snippet recommenders propose likely completions to the developer. However, the results typically involve multiple statements or method calls and involve more than just the direct element on which completion was triggered.

Nguyen et al. [171] developed GrouMiner to find patterns in API usages from source code repositories. Their approach is only based on the syntax tree, they do not consider resolved types. The patterns, called Groums, contain ordered information about method calls, object declarations, and control points. They do not extract information about the structural context, but they do make use of the fields that are declared in the class. The analysis does not use a points-to analysis, but uses a simple intra-procedural heuristic for the data-flow detection that is built on variable names. In their follow-up work, Nguyen et al. [169] propose the GraPacc snippet recommender that uses these patterns. In addition to the extraction of the graph-based features from the Groums of the code under edit, they also tokenize the code to create token-based features to be able to support un-parsable code as well. They only consider keywords and variable names. The underlying models as well as the queries do not contain information about access modifiers. These are only added later during active completion tasks in an IDE (as opposed to offline evaluation).

Prospector (Mandolin et al. [127]) and PARSEWEB (Thummalapenta et al. [240]) are two recommenders that propose ordered sequences that involve different API types. Both recommenders suggest call sequences that show the developer how to get from one API type to another.

Prospector extracts several "elementary" code elements to build its "signature graph", the repository that contains all information that is collected about a framework. They capture field accesses, static and non-static calls, and types and store them as basic "Jungloids", composable elements that describe reachable types. By analyzing source code, they also find examples of meaningful up-casts and valid down-casts, which they add to their repository. Order is not explicitly encoded in the input data; their synthesis approach naturally generates the right order, because their Jungloids always contain a tuple that describes how to get from one type to another, an implicit kind of order. They perform an inter-procedural and inter-class analysis that slices the program to find an elementary Jungloid that creates the target expression. They then track the control flow into the subsequent method calls.

PARSEWEB works a little different internally. They rely on examples returned from external code search engines. Since such examples often do not compile, they simply parse the snippets into syntax trees and transform them into a graph-based IR that supports branches but not concrete control structures. Their approach considers static and non-static calls and casts. The only type information used are the imports in a class. Heuristics are used to infer types in method signatures and method invocations.

The proposed snippets of both approaches contain variable names. However, they are not part of the model; unique identifiers are generated on demand.

Natural Language Processing Language models are usually used in linguistics. Brown et al. [21] were the first to leverage the repetitiveness of natural text to predict tokens in sentences. This technique and others from the field of natural language processing have recently been applied in research on software engineering.

Hindle et al. [87] have shown that the source code of programs is regular and can be captured in models, similar to natural language. They provided a more general recommender that is based on natural language processing (NLP) techniques that treat the source code as plain text. While it does not propose complete snippets of source code, it can always propose a next token to write. This could be, for example, a keyword, which makes it a general recommender that is able to do more than just call completion. The technique includes all code elements of the respective language, excluding comments. Based on our understanding of the available description of the work, we assume that the authors either use a parser for each language and tokenize by traversing the AST, or they tokenize based on the grammar (lexing). In both ways, the text is tokenized according to the language specification, which makes it one of the exceptions in our survey that is solely syntax-based and which does not depend on a resolved type system. It is important to note that this design decision does not generalize to other NLP-based systems (e.g., [195]). The nature of the approach makes a points-to analysis unnecessary. Normalization and a tracking approach that adds tokens from private helpers would be applicable and the effect on the cross-entropy of the language model could be further evaluated.

Franks et al. [64] present CACHECA, a code completion tool that is based on a cached language model. In previous work, the authors have shown that software can be represented in a language model [87] and they have built an n-gram based recommender to predict tokens. A follow-up work has shown that software is also very local [242], a property that is not captured well by n-gram models. The authors overcome this limitation by adding a cache to the n-gram model. While the n-gram model captures global regularities, the cache captures the local regularities. The underlying algorithm works on untyped tokens, it cannot stand alone and has to be combined with a type-aware component to create real fully-qualified proposals, i.e., in an IDE integration and for an evaluation. The authors integrate the approach in the regular ECLIPSE completion engine.

Santos et al. [213] built the tool APISTA that helps developers using APIs. All API related operations (i.e., construction, static calls, instance-based operations) are represented in tokens. The authors mine repositories and build token sentences by passing through all methods of the contained classes. The control structures of a method are traversed in all possible ways to collect the operations. Nested or chained expressions are normalized to a sequence that reflects the order of execution. The approach uses a minimalistic naming scheme for types and references. More specifically, they qualify types and methods, but do not capture the signatures of methods. An inlining strategy that distinguished public and private methods is applicable, but might lead to an explosion in the number of extracted sentences. The extracted token-sentences are used to build a language model based on n-grams. Queries in the IDE are constructed by extracting the tokens that precede the edit location. The unqualified proposals are then matched with completion candidates that can be matched with real entities from the type hierarchy that is only available in-IDE.

Code Search Code search is quite similar to snippet recommendations. The difference is that proposals point to existing examples that were observed in repositories or the local workspace, instead of making probabilistic recommendations. The proposals cannot be directly integrated into the current editor, but will point to source code that can be used by the developer to understand a correct usage of the API in question.

Several approaches in this category use interclass tracking to follow the control flow in project-specific files. To be able to this, these approaches require structural context information to “find” the target in a look up and the type hierarchy to find implementations of interface methods.

Zhong et al. [262] create MAPO, a code search tool that mines method sequences from API usage examples. Methods calls include constructor calls, static and non-static calls, as well as casts. While they respect the order of calls and follow control structures to create all possible sequences, the control structures are not actually included in the model. The analysis is intra-procedural and the enclosing method is not extracted in the model. They analyze client code, but only include methods in their analysis that are declared in a reusable framework, i.e., included as a dependency, and track into all other method calls. They do this inter-class, as long as the files belong to the same project.

Holmes et al. [90] present the code search tool Strathcona for the Eclipse IDE. The tool builds queries by extracting structural facts from a source-code fragment selected by the developer, i.e., declaring type, super type, fields of declaring type, method declarations, as well as all referenced types/methods/fields. All references are stored with fully-qualified names, if resolvable by JDT, and point to the concrete implementation that is specified. This could be improved by abstracting these references and by pointing to more general types, which would also open the possibility to apply an inlining transformation. On the server side, the structural context is used to find related source-code examples in a database that is populated with facts extracted from open-source projects. Four heuristics that quantify the *relatedness* to the query are calculated and are combined for the final ranking that is shown to the developer.

Moreno et al. [154] propose MUSE, a tool that provides developers examples of how to use a particular method. The tool requires the source code of the target API as well as a list of its clients. Whenever a call to one of the target API public methods is observed in the client code, an intra-procedural backward slice is created that shows how to get to this call. The slicing builds a program-dependence-graph that requires keeping track of method calls, order, variable names, assignments, and control structures. Their approach includes all statements and expressions that are allowed in a method body by the Java language; this excludes structural information such as class declarations. When queried, the proposed example contains a ranked list of (rendered) slices that also contains explaining Javadoc, if available. We consider Javadoc as a separate artifact to source code, because while it is maintained in the source code of the framework, it is usually available as a separate download since binary distributions do not contain comments anymore. We believe that their approach can benefit from normalization to get rid of ambiguities. However, points-to analysis will not necessarily be useful here.

Hummel et al. [92] present CodeConjurer that supports developers by searching for working code that follows a specified UML-like syntax. The involved static analysis is based on the structural context only. The internal representation of the search engine only contains fully-qualified names for classes and all methods found in examples, as well as a link to the location where the file was originally found. The selection and ranking is completely based only on these signatures and identifiers. After including a snippet, type resolution of missing types is achieved by guessing the correct location, given the fully-qualified name of a class, and by downloading it to the local workspace. Since the lookup is very simple here, the technique would not require any of the transformations or additional analyses we discuss.

Wightman et al. [251] provide a set of curated code snippets that are searchable by developers in their IDE with the SnipMatch tool. The snippets represent reusable pieces of source code. They define a search pattern that is used to find them, they may define typed parameters, and they are stored in a simple source-code like syntax with placeholders. The IDE integration allows searching for snippets textually, the results are filtered and ranked. While the general algorithm is type unaware, i.e., the server does not know about available types in the context, the type hierarchy is necessary on the client, as the filtering, the pre-filling of matching variables, and the ranking relies on typing information from the current programming context.

While SnipMatch was designed to work with a manually curated list of snippets, it is an obvious next step to mine these snippets automatically. In fact, several years after the publication, Allamanis et al. propose Haggis [1], a tool that mines *code idioms*. They mine released code for very frequent syntactic patterns that are used across several input projects, as these represent idiomatic ways to solve a task. The tool works on ASTs and encodes the types of variables, but uses only simple method names without signature. The authors mine a *probabilistic tree substitution grammar* that explains the sources found in the input projects best. From this grammar, code idioms are inferred through a sampling of the probability distribution. In their evaluation they show that the mined idioms are indeed valid and they successfully integrate several of the mined idioms into the SnipMatch repository.

Documentation Programmers often rely on documentation when learning about an API, but manually created documentation is hard to maintain and is thus often incomplete or outdated. Automated documentation generators try to solve this problem by mining source code repositories. They extract information that describes how to use an API and create documentation that can be consulted by developers to better understand a system. The documentation can have very different forms. We discuss approaches that cover a wide range.

Michail [143] proposed CodeWeb, a tool that can be used to identify "reuse relationships" in source code. The developer can consult these tuples to learn about the correct API usage. An example of these rules is "you override X and you also override Y", which is based on information from the structural context; another example is "you implement I and override M", which uses a type-system look-up to find the required information. The features they consider are class inheritance, member overrides, and invocations (including constructors). However, the captured calls are on a class level though ("existence fact"). The authors later extend their work to include the complete inheritance hierarchy in their collected facts [144].

Zhong et al. [263] present Java Rule Finder, a tool that infers rules about a correct usage of a framework directly from its source code. The rules are prepared in a textual format and serve as a browsable documentation for developers. To learn the rules, the authors consider extracted facts with information about the type hierarchy, invocations, fields, and field accesses. They use these facts to encode information about the source code in a special graph-based notation.They did not use control structures and seem to just collect field read/writes, as well as invocations. Therefore, points-to analysis and normalization are not applicable. Tracking, however, seems applicable and might reduce the number of facts that are extracted in their approach, because less method relations need to be stored.

McBurney et al. [135] propose a system that automatically generates the documentation of an API method. Instead of analyzing the implemented behavior in the method, they look at callers of the method and extract descriptive information from there. Their system builds a call graph and uses the page rank algorithm to find regularly called methods. To do this, they need method calls and the enclosing method declaration. They also identify nouns and verbs in the tokens by splitting the identifiers found in the signatures, as well as those found in the assigned variables. We believe that tracking might potentially improve the quality of the page rank output. Normalization might also increase the preciseness of a statement by separating unique steps. However, points-to analysis is not applicable.

Ponzanelli et al. [181] propose PROMPTER, a tool that proposes relevant postings on STACKOVERFLOW to developers while they are working in the IDE. It extracts the current programming context from the source code under edit. The context contains fully-qualified names for types and methods, the source code of the enclosing element at the edit location, and a list of all types and methods used in the enclosing element that are defined outside of the project. Based on this context, a query component splits, stems, and normalizes the search terms to create a textual query that is used to request relevant posts from the server. The client then ranks the returned threads based on the programming context with more semantic heuristics, like *matching types* or *topic*. While types and methods are fully-qualified in the context, the STACKOVERFLOW threads do not contain such information. The authors resolve referenced types heuristically, but have to resort to unqualified names if further information is not available.

Anomaly detection Anomaly detection approaches learn characteristics of a typical/-correct program. The underlying models are then used to detect deviations from this established norm. A very related area is bug detection, in which extended static analyses are used to prove the definitive existence of a problem in the code. Due to the similarities of both areas, we will discuss them together.

Monperrus et al. [152] propose DMMC to detect missing method calls. They extract object usages that describe how an object instance is used. They encode the type of the object, the enclosing method, and all calls on it. They track the object inter-procedurally, but intra-class. They implement a points-to analysis. Normalization is not necessary, because their traversal of the syntax tree does not need to handle nested expressions.

Li et al. [117] present PR-MINER, another detector for missing method calls. The tool extracts facts from a method such as the type of variable declarations, variable names, assignments, and calls and uses a prefixing strategy to prevent name collisions in different scoping levels. By applying frequent items mining, the tool ends up having a list of programming rules. The available source is then checked for violations of these rules. To remove false positives, they track method calls in children calls and in the call-graph of the parent (in which the current location is a child). They eliminate the violation report if the missing call is found there.

Pradel et al. [184] present an approach to build finite-state-automatons that describe valid protocols for using a specific type. The approach is based on their own framework that can be used to mine method sequences [183]. The concrete mining approach is exchangeable in this framework, but the most precise implementation uses a points-to analysis to identify objects and collects method calls that happen on or with each object. The approach also makes use of prior work by Jaspan et al. [95], which provided an extended points-to analysis to judge whether objects are contained in other objects, e.g., in collections. Both analyses work intra-procedurally, so there is an opportunity to apply tracking to increase the size of the sequences. Normalization is not applicable.

Overall, static analyses of typical RSSE approaches are often *optimistic*, in contrast to pessimistic analysis in security or sound analyses in program optimization. The corresponding analyses are typically very lightweight and often sacrifice soundness for other properties. Most approaches apply statistical means after the static analysis to identify common patterns. It is therefore less critical if the static analysis under-approximates or even ignores cases that would be filtered out afterwards anyway.

The underlying approaches often do not require a compilable project environment to extract further information or to improve the analysis. However, types and type elements should be fully-qualified for disambiguation, which is often not even necessary for the underlying approach, but for its evaluation.

Evaluation

For reasons elaborated in the introduction of this section, we are interested in source-based RSSE. We want to help users learning and using APIs with a recommendation system and we are fascinated by the idea to improve the capabilities of existing code completion systems like IntelliSense. Tool smiths that build such systems always face the challenge to find a proper evaluation strategy. In this subsection, we will survey the evaluation strategies of related RSSE to understand typical approaches. Evaluation approaches are always specific to a given recommendation problem and the most related evaluations to our envisioned system are those of other method-call recommenders. We relax our survey criteria though and also consider other source-based RSSE to get a more general impression about typical evaluations in this area.

Many RSSE are being evaluated through controlled experiments that involve human subjects. To quantify the performance of the system, researchers analyze if participants successfully complete a task (e.g., [88]) or measure how long subjects take for completion (e.g., [127]). In other more qualitative evaluations, experts judge the usefulness of the proposals (e.g., [88, 154]) or the subjects rate their experience with the tool (e.g., [41, 261]). All these are valid evaluations and they often create additional qualitative insights.

However, controlled experiments also have their downsides. Their nature limits their scope and makes it hard to generalize the results. Many tools also work with external services (e.g., Q&A websites [181]), which hinders replication, or require very task-specific data like navigation information (e.g., [41]), which makes it hard to design appropriate tasks. Most importantly, designing a controlled experiment involving humans takes a lot of time. In addition, the risk of a failed experiment is also quite high, because a study cannot be simply replayed, when for example a bug is discovered after the fact. As a result, it is much more common to find artificial evaluations of RSSE in the literature, which motivated the research questions of our work. In the following, we will discuss several representatives of artificial evaluations that we found in the literature. The goal is not to present an exhaustive list of prior evaluations, but to introduce several high-level ideas by example.

Heinemann et al. present a method-call recommender [83]. In addition to considering the structure of a program, the approach considers identifiers, such as variable names. The recommender tokenizes source code into an event stream and learns a model. The evaluation iterates this stream and tries to predict the method, every time an invocation is encountered, and measure the quality of the proposals. They assume linearity and do not include information found after the query point.

Zhang et al. propose a recommender that predicts parameters for method calls [261]. For evaluation, they use published source code and query the recommender at each observed parameter. Queries contain all observed information from the code with the exception of the parameter that is to be predicted. They also conducted a user study that analyzes the perceived usefulness and opinions of the participants to get qualitative feedback for their tool. However, they do not present quantitative data about the performance of the recommender or the correctness of the proposals.

Bruch et al. [22] propose a method-call recommender based on the Best Matching Neighbor (BMN) algorithm. Queries are automatically generated from API usages observed from code repositories. A query consists of a subset of the observed method calls in the API usage and the evaluation measures how well the recommender predicts the removed calls. The authors use two strategies to generate such queries: (1) a *no calls included* strategy that mimics the situation where a developer triggers code completion when she starts to implement a method and (2) a *first half* strategy that keeps only the first half of the method calls to mimic the situation where a developer triggers code completion after she wrote parts of a method.

Follow-up work replaces the BMN algorithm by a Pattern-Based Bayesian Network (PBN) [192] as the recommender engine. The evaluation also uses both the *no calls included* strategy and the *first half* strategy. In addition, queries are generated with a *random half* strategy that randomly selects which half of the method calls is kept to mimic that developers may not write code in a linear fashion. The random selection is repeated and the results are averaged.

GraPacc [169] recommends code snippets that are related to the current context. Patterns are mined from the source code of some Java projects to create the recommender, which is then evaluated on several other projects. For the evaluation, all method calls are extracted from the method bodies in the validation projects. These sets of methods are divided into two parts: the first part is used to query the recommender, the second part as the expectation. The evaluation measures the fraction of method calls that are contained in the second part and the proposal. This evaluation technique is similar to the *first half* evaluation followed by Bruch et al. [22].

MAPO [262] is a miner and recommender for API usage patterns. The miner identifies API patterns in a large pool of released source code. In the IDE, the current context is matched against these patterns to retrieve related code snippets. The authors use code snippets selected from a tutorial book in the evaluation, which are considered correct and complete. In their queries, they use all context information and the first method call. Given such a query, the recommender suggests related code snippets, which are manually matched with the expectation.

Prospector [127] is a recommender that is queried with a tuple of two API types: an input type from the current context and a target type that the developer wants to obtain an instance of. Prospector returns a sequence of method calls that would return an object instance with the respective type. For the evaluation, the authors manually picked 20 examples of programming problems that they deem realistic and to which Prospector is applicable. For the queries, they always assume that the developer knows both types.

Guervo et al. present InSynth [79], a tool to synthesize type-correct expressions. As such expressions can be complex structures that contain nested sub-expressions, the approach effectively recommends code snippets. For the evaluation, they manually create 50 query/expectation pairs as benchmarks, taken from several open-source projects. A query is a program snippet in which a single expression is removed for the benchmark. The evaluation measures whether InSynth can synthesize that expression again. Since the target expression is selected arbitrarily, the evaluation approach resembles the random removal approach used for PBN [192]. Unlike the PBN evaluation, the benchmark set is manually created, rather than automatically.

Raychem et al. [195] develop a recommender that suggests multiple missing method calls in a piece of code. They traverse the syntax tree and reduce it to a sequence of method calls. Missing method calls are *holes* in this sequence. The recommender calculates the most likely sequences of method calls that fill the holes. Three kinds of queries are used for evaluation: (1) a single hole at the end of the program, (2) multiple holes that are manually introduced, and (3) one or more random holes that are automatically introduced. The first query strategy assumes linearity in code development; the other two assume non-linearity. The authors do not describe whether there is a limit on how many holes are introduced; our understanding is that only a small percentage of the method calls are removed for querying.

Note that the three last papers are somewhat different from the others, since they also include queries manually created by the tool smith. This is done to create realistic, meaningful queries that a real developer might trigger. However, since the creation of such queries is very subjective and not based on any input from actual developers, we consider these evaluations as artificial.

Some approaches conduct controlled experiments and capture all interactions of the subjects for later experiments. However, existing approaches either capture data that is not appropriate to build or evaluate source-based RSSE (e.g., [102, 103]) or they do not perform automated evaluations and evaluate results manually (e.g., [165]). To the best of our knowledge, there have been no attempts to create a benchmark of developer interactions for an automated evaluation of RSSE before.

2.1.3 Understanding the Evolution of Source Code

After capturing source code changes, another line of research analyzes these changes to gain knowledge. They learn how source-code evolves and create tools that use the knowledge about past activities to support developers in the future. Some of the cited papers might have already been introduced in the previous section, but we decided to clearly separate the discussion of the data acquisition with the applications.

General Source-Code Evolution Source code is the central artifact of software engineering and developers use it as a means to instruct the computer and to communicate their ideas. Some researchers are interested in answering general questions about the evolution of source code to understand how teams or individuals write source code.

Schneider et al. [217] use their recording of fine-grained changes in the IDE to analyze and improve team-collaboration. They analyze the changes to identify the location and the kind of changes performed by a developer. They visualize the change history and propose a metric to identify other developers that work in close proximity. These information is shown to all members of the team to improve awareness about the *locality* of the changes other developers are performing, with the goal to improve the coordination and communication in the team. While all parts of the approach are implemented, the authors miss to evaluate the proposed tool.

Spacco et al. [231] use Marmoset in an empirical study to capture fine-grained source code history. They were interested in analyzing how novice programmers write code and how this involves unit testing. To this end, they analyzed the evolution of student projects in a software engineering course. From the fine-grained change history, they can recover test results, test coverage, and code style warnings for each committed version. As a result, they find a statistically significant correlation between failing tests and the reported warnings. They also find saves seem to be a good stability indicator, as around 80% of their snapshots compile (all taken after save operations).

In contrast to many other works before, Xing et al. [254] realized that line-based changes are too coarse grained to visualize differences between two versions of an object-oriented program. To be useful for developers, a differencing tool must work on a semantic level, because changes are easier to understand and to follow. To this end, they propose a meta-model to represent structural elements of the JAVA language, e.g., class structure (fields and methods), blocks, and different kinds of types (i.e., class, interface, array, primitive). The meta-model also stores details about the elements, e.g., modifiers for visibility. They propose a new algorithm for matching the AST nodes that makes use of a new name similarity metric and a structural similarity metric. The evaluation shows that their differencing tool can provide good support, but that the matching of nodes is sensitive to infrequent commits. This is an interesting motivation to capture fine-grained history as proposed in this thesis.

Fluri et al. [60] analyze how source code and comments co-evolve. They use their tool CHANGEDISTILLER to extract version pairs from a VCS and analyze how the sources have changed. To this end, they parse the source code to an AST and distinguish changes on the statement level. They distinguish source-code changes and changes in comments and determine the tree edit operations required to get from one version to the other. This combination allows the analysis of the relation of these changes, which reveals, for example, that newly created code rarely gets commented.

Negara et al. use the history captured by their tool CODINGTRACKER to compare fine-grained changes to the granularity captured in VCS [166]. They find in their work that the source-code history in VCS is incomplete, imprecise and that it is not representative for actual source-code evolution. They conclude that more-detailed history information is required to study source-code evolution, a finding that underlines the relevance of the work presented in this thesis. In a follow-up work, Negara et al. use the same data to mine patterns in these change information [165].

API Evolution Application programming interfaces (API) are an important in object-oriented programming. They keep software systems manageable, facilitate maintainability, and are used everywhere. Programs evolve, because either new requirements emerge and the functionality needs to be extended or if breaking-changes in the API of referenced libraries of frameworks impose adaptation in the client. Framework authors rarely provide sufficient information about how to adapt the breaking changes. A whole line of research emerged that analyzes API evolution, learns from change patterns, and supports developers understanding changes and adapting client code.

Xing et al. [255] present DIFF-CATCHUP, a tool that can identify changes in an API. Developers can use the tool for code search when they encounter a broken API after an update to get proposals how to fix it. This knowledge is gained by analyzing working examples of an API. Using their previous work, UMLDIFF [254], they extract the source-code changes that explain how to update a program from the one framework version to the other. This change information can be searched for obsolete API methods and their corresponding fixes. They also provide the visualization tool JDEVAN, which allows exploring the identified recurrent changes.

We et al. [253] propose AURA, the *automatic change rule assistant* that assists developers migrating client code to new framework versions. The approach is implemented as a plugin for ECLIPSE and works on a type-resolved AST to extract rules. The core contribution of the approach is that *one-to-many* or *many-to-one* change rules are supported, specific cases of API evolution that would have lead to misleading or incomplete change rules in existing systems. AURA combines a call dependency analysis and a text similarity analysis, which are executed in multiple iterations. Both analyses have disadvantages on their own, but the novel combination increases the recall of the detected change rules, when compared to other approaches.

Mileva et al. [146] study API evolution from the perspective of an object usage. After upgrading a program to a new framework, the required changes should be consistently applied in the whole code base. This problem is solved with their tool LAMARCK that can be used for mining *evolution patterns* and for detecting missing changes. They extract *temporal properties* for a program that describe how an API is used in a specific method. These properties are extracted in a static analysis and contain both order information (e.g., call of `m1()` happened before call of `m2()`) and data flow information (e.g., return value of `foo()` used as parameter in call of `bar(...)`). The authors compare the *temporal properties* of different version of the same program and extract *change properties*, which can be mined for common *evolution patterns*. Using these patterns, LAMARCK can detect unfixed code issues related to framework evolution in several open-source projects.

Hora et al. [91] present APIEVOLUTIONMINER a tool that analyzes API evolution in code repositories at the revision level. They use changes in method invocations between two versions of a program to identify API changes. While this information is extracted from an AST, it does not rely on resolved typing information. Method names are encoded with a combination of the receiver name (variable name or class name for static calls), *simple* name of the method, and parameters (parameters that are more complex than primitive expressions are removed). As a result, the approach is applicable to both statically and dynamically types languages, because it does not involve any strict typing information. The authors use frequent itemset mining to identify sets of method that change together, in addition, rules like this can also be provided by experts, but a small case study shows that their automatic approach is able to mine meaningful rules.

Merging Conflicts arise when multiple developers work on the same piece of source code at the same time. A whole line of research works on resolving these conflicts and merging simultaneous changes in concurrent versions of a program.

Lippe et al. [118] propose a new paradigm of merging, which models the operation that induces the change instead of the traditional *two-way* or *three-way* merge of line-based changes. In their merge tool CAMERA, irrelevant information is ignored in the visualization and the included differences are much more concise and are based on operational semantics instead of line-changes. For example, part of such a change set is a variable name refactoring; instead of highlighting all affected places, the simple information that a variable was renamed is much easier to grasp and visualize. To enable such a smart visualization, domain knowledge is necessary to define and identify the operations, e.g., which changes can be ignored. When user intervention is required to solve a conflict, a smarter UI might be able to allow a reordering of the transformations (e.g., pretty printing in parallel to a small change in a line). The user then just has to decide whether to reorder or ignore specific operations, instead of working on the plain text.

Refactoring are cross-cutting changes that affect many locations at once, which usually breaks merging approaches of existing VCS. Executions of refactoring tools are logged in modern IDEs, but it is still necessary to align this information with source-code changes. Dig 2008 [48] address this and introduce MolhadoRef, a refactoring aware VCS. In addition to storing the actual change, they also capture a corresponding refactoring log that provides information about the operation that caused the change. The authors present a new merging algorithm that leverages the new information to improve the resolution of conflicts. They show in experiments that more merge conflicts can be solved automatically than in a traditional CVS.

Refactoring Recovery Even if no refactoring log is recorded in the first place, it is often possible to identify the executed refactorings after the fact. Several works are concerned with recovering refactorings from changesets, to make them more concise.

Weissgerber et al. [249] analyze the VCS history of a project, preprocess the data and identify file sets that have been changed by the same commit. By applying techniques like clone detection, they can reduce the changes to sets of likely refactorings and can distinguish several structural refactorings like *move class* or *rename method*. Their approach parses source code and works on the AST. They store fully-qualified information about types and type-elements to uniquely identify them.

Dig et al. [47] propose RefactoringCrawler, which is very similar to the previous tool. They can identify potential refactorings that were applied to one version of a file to get to another one, by providing several detectors for specific changes in the changes. The approach works on AST with fully-qualified code elements. It features a semantic analysis that can resolve cases in which multiple refactorings have been applied to the same structural element with a shared log of executed refactorings.

Prete et al. [185] present the template-based tool Ref-Finder that can identify complex refactorings that consist of several atomic refactorings. The author express properties of refactorings as logic rules and use a logic engine to find refactoring sequences that explain observed changes between two versions of a program. The approach works on AST and extracts facts about a program that require fully-qualified type information, e.g. used interfaces, inheritance relations, or method overriding.

Foster et al. [63] propose WitchDoctor a tool that detects the manual execution of several refactorings by a developer. Once detected, the tool offers to finish the remaining steps of the refactoring automatically. The detection is done by comparing the current AST of a file to a reference version, but seems to be restricted to structural changes. WitchDoctor shows a preview of the potential completions to developers. The previews are created by reverting all edit operations and delegating the execution to the built-in refactorings available in Eclipse. If selected, the current AST is replaced with the one shown in the preview. In their evaluation, the authors mimic manual refactorings and compare the results to the execution of their tool. The authors report several challenges they had to overcome in their IDE integration, e.g., that many ways exist to achieve the same result or that source-code that is to be analyzed often contains unparsable parts. They report that getting access to actual coding sessions to understand how source code is refactored manually is a key challenge for them. This is an encouraging point for this work, as the required data can be provide with the tools presented in this thesis.

A common problem for refactoring detection is the *late awareness* of a developer who only realizes the need of a refactoring, after it was already started manually. As a result, the refactoring might interleave with unrelated changes. Ge et al. [68] address this issue in their ECLIPSE plugin BENEFACTOR, which can identify and auto-complete manual refactorings, even when parts of the current changes might be unrelated. In a formative study, they learn about typical workflows in manual refactorings and model the operations that describe how the AST would change for several refactorings (e.g., clipboard-cut a node, paste the node again, update another node) Typically multiple workflows exist for the same refactoring. They detect similarities of the current changeset to these workflows. If developers decide to auto-complete a refactoring, a "selective undo" is executed that is able to preserve the unrelated changes.

Defect Prediction Software is constantly changed and some of the changes introduce bugs in the program. A whole line of research exists that analyzes software changes with the goal to predict defects early on in the development process. Over the last years, the techniques went from calculating simple metrics (e.g., [162]) to learning complex classificators (e.g., [86]). All approaches that we classified into this category base their prediction on change information. Other techniques exist that infer this kind of prediction by statically analyzing released code. We will discuss these approaches later in Section 2.1.2.

Real reliability information and bug counts are only available late in the development cycle of large software systems, but defects should be predicted early on. Nagappan et al. [162] use code churn to predict defect density. While absolute code churn is a poor predictor for defect density, the authors present novel measures based on change history and process information that capture more local information. They cover high-level properties like lines-of-code, number of files churned, lines added/deleted/updated, or number of times edited. In the experiments of the authors, these metrics turn out to be highly predictive for the number of defects introduced in a binary between two releases.

Existing metrics for defect prediction are often based on static properties (e.g., lines-of-code, code churn) or historic properties (e.g., file age, number of authors). Lee et al. [116] present new *micro interaction metrics* that include the interaction history of the developer, for example, file level properties like effort, or task-level properties like distraction or repetition. The authors extract the interaction history from data captured with MYLYN and use it to predict defects that are marked as bugfixes in the dataset. The dataset is split into two parts and the authors identify bugs that were introduced in the first part and fixed in the second, meaning that interactions are filtered that do not exist in either of the two parts. The authors build linear-regression models to predict the number of defects and show in their results that the system is able to identify bugs with a high F1 value and that the micro interaction metrics outperform traditional metrics used in the literature.

Livshits et al. [120] follow a different route and use a simple static analysis of the evolving program code to extract change information. They propose DYNAMINE, a system that finds common error patterns through the combination of mining a software repository and a dynamic analysis. Their approach parses the source code and extracts added methods from the AST of two revisions of a file. They investigate all files in isolation and do not resolve types, they report that this step would be too expensive. As a result, they do not distinguish types in their static analysis and resort to using unqualified type names. In addition, methods are qualified by the number of arguments. They mine association rules from the added methods to find patterns of methods that were added together and use these patterns in a dynamic analysis approach that counts the observations of the pattern at runtime. This is achieved by instantiating state machines that correspond to the mined patterns for each created object. The transitions in the state machines are counted to identify violations and validations of the mined patterns.

The history in traditional VCS contains tangled commits that mix changes related to more than one task. Many approaches use changes found in repositories to build models and to compare approaches and all of them depend on the accuracy of the mined information. Herzig et al. [86] analyze the effect of mixing these unrelated changes on defect prediction. The authors find through manual classification of 7,000 change sets that up to 20% of the commits in a repository contain tangled code. They propose an untangling algorithm that is able to partition changesets into more cohesive subsets. After introducing artificial tangling, they show results that suggest that existing techniques are significantly influenced in their predictive power when compared to the untangled version of the code. We could interpret these results as a sign that the mix of unrelated changes in repositories is caused by a coarse-grained history. A more fine-grained commit history might avoid some of the tangled cases or might make untangling changes easier, as shown by Dias et al [45].

Code Completion Code constantly changes and writing it is repetitive and follows specific rules. One line of research analyzes changes made by developers to derive change patterns that can be proposed to other developers in the same situation.

Fluri et al. use ChangeDistiller [60, 66] to extract information about program history from VCS. They use this data to analyze bugfixing behavior in large software systems [61] and find that around 15% of bugs are fixed by wrapping a statement in a new context, e.g., an `if` is added to guard the invocation of a method call. This suggests that unguarded method calls pose a stability threat and they propose the tool ChangeCommander that can help developers to identify and fix these cases. To achieve this, the authors extract changes to the context of method calls from repository versions, more specifically, cases in which method calls were being wrapped in a new context (e.g., wrapped inside a *null check*). Their technique parses source-code to an AST that does not contain resolved types and methods, only a heuristic resolution is achieved by applying ZBinder [180]. Expressions are normalized in a preprocessing step to cope with complex expressions that may be contained in `if` conditions (e.g., `x || y`) They identify patterns by aggregating changes that affect the same method and calculate the *support* for each change. ChangeCommande is integrated into the Eclipse IDE and recommends missing calls in the source-code, the proposals are ranked by their support in the mined changes.

Robbes et al. [201, 203] use the semantic program history that is captured by their previous work SpyWare [202] and extract additions of method calls. They use this data to build a novel benchmark for method completion systems. Each recorded method addition is used as a query, in which a prefix of the added method's name is used to request a proposal. The accuracy of the completion system is then measured by the ability to find the added method. In a second step, they propose several novel code completion strategies that leverage information from the change history to improve code completion through filtering and ranking of the proposals (e.g., by *recent-creation* or by *recent-modification*) and implement them in their completion tool OCompletion. Traditional, *pessimistic* completion strategies propose all possible methods. The proposals always contain the missing method, at the cost of a very long list of proposals that needs to be searched. Their results show that *optimistic* completion strategies could reduce the cognitive load of developers. Even though they only propose a subset of the available methods, i.e., three proposals, the proposals contain the missing method in 75% of the cases. The tool was extended to class name completion using similar heuristics. Their re-implementations of the various strategies heavily rely on the project environment (to find all available types and methods) and on the type systems (to resolve types and find methods defined in the type hierarchy).

Nguyen et al. [167] mine change patterns from changes extracted from commits. They authors postulate that it is better to focus the mining on the repetitiveness of changes than on the repetitiveness of complete source-code, because the intent of changes is captured better than in statistical models created on a history of snapshots or approaches that use techniques for natural language processing (NLP). While a typical change contains many operations that are usually related, pure snapshots mix them with existing code and NLP approaches only consider preceding tokens while additional changes could have been made at various locations of the source code. Their approach considers both worlds, they extract a change context (i.e., the changes introduced by a commit) and a code context (i.e., the preceding tokens). Changesets with a low *support* are rejected assuming that these are project specific. They build ApiRec, an association based approach that considers both kinds of contexts to infer method call proposals. Both strategies are combined in a weighting calculation that is optimized with gradient descent. Their results show that the approach outperforms existing method-call completion systems. Considering the finding of Negara et al. [166] that VCS do not contain a representative program history, we assume that this approach would highly benefit from using program history that is more fine-grained, as proposed in this thesis. It might be necessary to aggregate the changes first though, e.g., by partitioning all changes by programming session or by files switches, to create consistent changesets related to the same intent.

Nguyen et al. [170] use topic models to improve the recommendation of change patterns. They develop TasC, a model that can predict changes that are likely for an observed program history in the IDE. To this end, they mine the VCS history of open-source projects and extract all changes between two versions of a program. These changesets consist of pairs of AST subtrees that are reduced to illustrate the changes. A latent dirichlet allocation (LDA) is applied in the approach to identify *tasks* ("topics" in LDA terminology) among all changesets ("documents") and to create a vector that explains how much the various source-code tokens ("words") used in the extracted change pairs ("sentence") hint towards the different tasks. When a completion is triggered in the IDE, all extracted change pairs are ranked by a function that combines their frequency and their similarity to the topic of the current changeset. The approach works on ASTs and does not seem to use typing information, but resort to a tokenization of the source code. To remove noise, the authors abstract away literals and use alpha-conversion to simplify variables names.

Overall, many approaches exist that analyze changes. This is of little surprise, with source code being the central artifact of software development and being subject to constant change. Our survey presented approaches that use changes to analyze general evolution, API evolution, or merging of concurrent changes, to detect usages of refactoring tools from changesets, to infer rules for defect prediction, or to propose recommendation systems for code completion tasks. These approaches did not only differ in their goals, but also in their techniques, algorithms, and data structures.

Most approaches extract the program history from public VCS and analyze changes on the commit level. Some approaches use more fine-grained data captured from IDE interactions or use additional context information tracked from these interactions (e.g., refactorings). The VCS based approaches are often tailored to technical features of VCS, e.g., line-based changes. Some approaches raise this abstraction level and focus on structural changes or even reconstruct change operations.

Very basic or very general approaches work with textual changes. They do not make use of any typing information nor do they leverage any semantic knowledge about the source-code. The more specific and complex the approaches get, the more likely that a semantic representation is used with ASTs being the most common representation. The most advanced techniques use changes on the semantic level, either by using data recorded in-IDE or by extracting semantic operations from changesets. Many approaches focus on the executable parts of the program (e.g., method invocations or nesting), others also consider non-executable parts like comments (e.g., [60]).

While many approaches work on VCS, most of them preprocess the information and annotate or enrich the data to facilitate their analyses. Many report difficulties or limitations when working with the raw data from repositories. The most common problem is the setup of the project environment to resolve type information in the source code. Advanced approaches need to consider the types to disambiguate referenced elements in source-code or to extract information from the type hierarchy, e.g., in code completion tools [203] or when detecting refactorings [249]. Many works just assume complete access to all code and dependencies to solve this issue or use heuristics for the type resolution (e.g., [91]).

2.2 Solution Domain

Research on the different applications that we have surveyed before can often not be conducted directly. Instead, it usually necessary to solve more basic challenges first, most notably, the representation and capturing of any data that is required for the analysis. In this section, we will review such foundational approaches that represent partial solutions, which -in combination- enable the various applications.

To this end, we are going to survey existing trackers that can be used to capture developer activities. We will focus on interaction trackers in Section 2.2.1 and on source-code changes in Section 2.2.2. We learn about important properties of intermediate representations for source code in a survey of existing formats in Section 2.2.3.

Especially RSSE strongly rely on static analysis of source code, a task supported by various platforms. We will review such platforms in Section 2.2.4 to get inspiration for a platform around tracked source code that facilitates the creation of re-usable analyses. Finally, we will introduce existing datasets in Section 2.2.5 that are available to study developer activities or source-code evolution.

2.2.1 Tracking Development Activities

Many researchers are working hard to understand the development process. If research is interested in novel concepts and it is required to involve human subjects (e.g., to get feedback on a concrete idea), researchers have to resort to traditional qualitative studies. However, other information like details about specific interactions could be easily recovered from artifacts of an experiment after the fact - if they were automatically captured in the first place. There is a great demand for software that facilitates qualitative studies (e.g., an automated data collection) and datasets that can be analyzed without conducting new experiments.

In this section, we will present several works that present interaction tracker for different IDEs. Not all of them capture the whole development process, some of them are only interested in a subset of the in-IDE events. In addition, existing tools can be clearly separated into trackers that capture interactions and trackers that capture fine-grained source-code changes. In this section, we will present the former and introduce the latter in the following section.

Holistic Non-IDE Trackers Some works want to capture the development process as a whole and also consider development events from outside the IDE. Humphrey coined the term *personal software process* as a model for software engineering [93]. Developers monitor their own work to help them assess their capabilities and to improve quality of the products. Systems like HACKYSTAT [99, 100] or PROM [224] were created that provide an automated tracking for developers. The systems hook into development environments, to capture specific development activities, but are also extensible to arbitrary information sources (e.g., office tools or build systems) by adding additional plugins to the systems that add sensors. The systems propose to capture such data in centralized databases to calculate metrics for both the developer and the management. This corresponds to the systems we have discussed before that store command ids and timestamps, but generalizes the data representation.

The HACKYSTAT system [99, 100] collects various measures about the development process of a team from different sources. The measures are calculated by instrumentations, so called *sensors*, that capture a specific aspect of the development process (e.g., file churn). The captured metric is forwarded to a centralized server and stored in a specific XML-based database. The interesting idea is that all members of the team participate and that the collected metrics are a mixture of personal metrics that are captured locally, for example in the IDE, and team metrics like build duration that are collected centrally, for example on the build server. The system automatically aggregates the collected data to generates reports that can be used by team members and managers to support the decision-making process in a project.

Maalej et al. [121, 124, 125] propose INTI, a tool that increases productivity of developers. The core idea of this line of research is to identify the *intent* of a developer and to link all related artifacts from different data sources (e.g., source code, mailing lists, bug trackers). To achieve this goal, the authors instrument the work environment of a developer through several *sensors* that monitor interactions of the developer with these artifacts. The combination of interactions and artifacts provides the required *context*, from which the intent can be derived. Given the formalization of the current developer context, it is possible to avoid big context switches in the planning of the next tasks, share the context in communication threads, reproduce a previous context, or to recommend related artifacts that are used in similar contexts.

In this thesis, we limit the collection of interaction data to in-IDE data. While extending this to other information sources is enticing, we do not consider it to minimize privacy issues that could prevent a practical data collection in the field. The design and implementation of our system should allow extensions in this direction though.

General Purpose In-IDE Trackers An early work into this direction was presented by Kersten et al. who built MYLYN [102], formerly known as MYLAR. The tool automatically extracts information about how the developer is using the ECLIPSE IDE. They focus on counting tool usages frequencies and discuss which parts of the IDE developers typically use. It tracks development events in the IDE like the selection of specific elements or the invocation of commands. Usages of a program element are boolean events, MYLYN counts their frequency, but it does not differentiate how long a program element has been selected or been interacted with. The authors analyze how developers work in the ECLIPSE IDE [156].

At some point, ECLIPSE received an integrated *usage data collector* (UDC[13]), that allowed to analyze interactions of ECLIPSE users, independently of MYLYN. The captured data was very shallow though and only contained executed commands and a timestamp, which makes it hard to answer complex questions with it. For the lack of consumers of the produced data, the UDC project was abandoned in the meantime, while MYLYN is still under active development.

ECLIPSE might be a popular choice among researchers, because JAVA is an important language and because the open-source availability of the ECLIPSE IDE makes the implementation of an interaction tracker easier. However, interaction trackers have also been built in several other IDEs.

Developers are overwhelmed with details when navigating, to validate this, Minelli et al. built DFLOW [147, 150]. It captures development events in PHARO, an IDE for Smalltalk, to identify how activities developers spend their time. They differentiate two kinds of events: *meta events* and *low-level events*. *Meta events* contain information about specific development events that are actively triggered by the developers. The authors differentiate the categories navigation, debugging, and editing. *Low-level events*, on the other hand, are triggered as a side effect by interacting with windows (e.g., open or move) or by using input devices like mouse and keyboard. DFLOW also captures events that describe source-code changes, but it is limited to structural changes like adding a method or removing an instance variable and does not contain information about changes within a method body.

Snipes et al. [230] introduce BLAZE, an extension for VISUAL STUDIO that tracks which commands a developer invokes. The original use case was a gamification system that aims to improve the navigation habits of developers. The authors envision an application of the very same technique in several other in-IDE tools [228], e.g., debugging or refactorings, and can successfully apply the captured data to analyze more general questions (e.g., Damevski et al. [39], Singh et al. [227]). Both the tool and a subset of the captured data are available for download.[15]

CODEALIKE[22] captures in-IDE actions and provides developers with a dash-board that shows various statistics about their time they spent coding and a measure of their productivity. The approach is somewhere in between a general in-IDE tracker and a holistic tracker: the focus is clearly on capturing in-IDE information, but the authors also provide a browser integration that monitors developers usage of websites [34]. No details are available about the kind of data that is captured by the plugin, but -based on the statistics we have seen- our best guess is that the captured data represents a simply time-based log of command ids as captured by MYLYN or BLAZE.

Specialized Trackers Some works are not interested in the holistic picture, so they limit the data collection to specific information that is important for their concrete research question that relates to analyzing developers.

Beller et al. [13] analyze the testing behavior of JAVA developers. They build WATCHDOG to capture testing-related development events in ECLIPSE (a later addition adds INTELLIJ support), for example, which tests are being executed, which files are being read or edited, or when the IDE is being shut down. All data processing is done on the client (e.g., hashing of method names) and only aggregates *intervals* are sent to a central server that are tailored to research question.

Another tracker for ECLIPSE was created by Vakilian et al. [245]. They propose two tools that work in collaboration to collect data about the refactoring process of developers: CODINGSPECTATOR and CODINGTRACKER. The first is a tool that records interaction data about invoked commands for automated refactorings, e.g., the time, the kind of refactoring, or how the refactoring command was invoked. CODINGTRACKER on the other hand captures a fine-grained history of source-code changes that allows replaying the textual history of the source code. They could use the captured data to analyze how professional software developers use automated refactoring tools in the IDE [244]. The combination of both tools make the general approach very similar to the interaction tracker we want to build in this work. However, CODINGSPECTATOR is limited to information about refactoring commands and would need to be extended for the general case. A more detailed discussion about the applicability of CODINGTRACKER will be contained in Section 2.2.2.

EPICEA [44, 46] is a tracker for PHARO, which puts its focus on capturing edit related operations in the IDE. Similar to DFLOW, the focus of the tool is on preserving a fine-grained history of source-code changes on a structural level (e.g., method rename), ignoring details of changes in a method body. The captured data of the tool goes beyond source code though and also contains information about executed refactorings, test execution, and usage of version control systems, making it to one of the tools that are most similar to our own interaction tracker presented in this thesis.

Yoon et al. [259] present FLOURITE, an event-logging plugin for the ECLIPSE IDE. The plugin instruments the IDE and captures of code-editing related events from the editor, down to the level of individual keystrokes. The combination of the fine-grained change information with an initial snapshot of the file enables a reconstruction of all intermediate edit states. In addition, the plugin captures several commands that are related to text editing, e.g., line marking, search and replace, or save. They enrich these commands with additional information that describes the action, e.g., the search string used in a search. The authors provide a visualization of the captured data (later extended in [260]) and present first insights in captured data from a lab study.

The developer tracker PROJECTWATCHER [217] has a slightly different goal. While its focus is on capturing a fine-grained source-code history, it also tracks development activities within ECLIPSE. The collected data is restricted to *activities* that relate to source-code locations in which the developer works though, e.g., a method or a class. The collected information is shared within the project team and a recommender points to other developers that work in close proximity within the code base, e.g., in the same package, to facilitate team-collaboration.

Another interesting example of *local* tracking is presented by Holmes and Walker [89] who turn the tracking direction around. Instead of tracking developer interactions and sharing them on a central server, the remote interactions with a central server are distributed to the clients. The system detects external change events that are related to the current work of a developer, like structural changes, changes to JAVADOC, or new commits in the VCS on the same tag or branches. The developer is notified about any detection to create awareness and to allow an early conflict resolution.

A development tracker in the broader sense was proposed by Roehm et al. [208]. While the work is not concerned with developers, they track the activities for users of a diagram tool, which is generally comparable to the work in an IDE. Their light-weight tool tracks a high-level trace of activities together with some context information. They store four kinds of events: general information about the application (e.g., start and stop), the ids of executed commands, low-level activities regarding the user interface like clicks, and fine-grained change information for the diagram. If an exception occurs, these traces can be attached to a bug reports to the reproduction easier. Even though they do not capture very detailed information about the control flow, they show that their visualization of the captured usage history indeed helps developers significantly in reproducing a bug.

Conceptually, all these specialized trackers are similar to the general-purpose variations in the sense that they also have to instrument the IDE and provide infrastructure to make the data collection work. However, the specialized trackers only capture a subset of the available information and ignore information that goes beyond the core research question.

Navigation Trackers A very specific direction of research that has been intensively researched recently is navigation. According to Ko et al. [107], up to 35% of a developers time is spent navigating the code base under edit. Observational studies like Bragdon et al. [19] suggest that this hinders productivity and that navigating sources (and remembering the path) is too hard. While they conclude that novel interaction concepts are required, another line of research follows a different route. Existing techniques are improved by learning from past interactions. A large number of works analyze navigation behavior of users to learn about related items that could be relevant to other users in the same situation. Early work in this area by Wexelblat et al. [250] identified recurring patterns in web navigation paths to improve user experience in subsequent sessions that search for the same content. This metaphor was later adapted to software engineering in two similar works by DeLine [41] (TeamTracks) and Singer et al. [225] (NavTracks). Related files and artifacts are identified by analyzing the in-IDE navigation traces of developers and recommended again.

While Mylyn [102] features a general-purpose interaction tracker, the captured *task context* can be used to characterize the current task in the IDE. Kersten et al. create a degree-of-interest model from this context information to identify relevant resources in the project and in the IDE. They improve programmer productivity and cut navigation overhead by hiding the irrelevant information for the current task [103].

Overall, it becomes clear that existing interaction tracker are all build around a use case. There is little support for capturing a holistic picture of the development process from within the IDE that includes both all invoked commands and source code. Limiting the collected data might help find participants and might also be sufficient for the research question at hand, but it surely limits the usefulness of collected interaction data. We do not find a *silver bullet* among all surveyed interaction tracker that could provide the kind of data in which we are interested in for this thesis.

2.2.2 Tracking Source-Code Changes

This section introduces several approaches that track changes of source code to make it available for research on source-code evolution. The discussion of papers that mix this data acquisition step with an application or with a specific research question is split between this and the next section to clearly separate both topics.

Source-Code Changes in Repositories The easiest way to obtain versioned source code is public repositories of open-source projects. It is of little surprise that many researchers study source-code evolution based on the contained source-code. The granularity of changes that a researcher can extract from a repository differs and strongly depends on the commit behavior of the individual developer. In addition, technical differences exist between the various *version control systems* (VCS). Current state of the art VCS like GIT capture changes in the granularity of changed lines, in contrast to earlier VCS like SVN or CVS that only differentiate file versions.

Many works that study source-code evolution do not use a specific meta model for their studies on the evolving source code and fall back to plain-text techniques. Early works work on basis of added or deleted lines. An example is the famous UNIX command line tool `diff`, which implements Myers algorithm [158] to highlight the changed lines for two different files. Later work exploits the fact that source code follows a syntax. It is possible to parse it into *abstract syntax trees* (AST) and study the evolution of the source-code with *tree differencing* [28] algorithms to achieve a finer granularity than purely line-based approaches.

The two current state of the art approaches for tree differencing are CHANGEDISTILLER [59] and GUMTREE [56]. Both create an *edit script* that describes all changes between two file versions through add, delete, update, and move operations on the tree. The approaches differ in their conciseness, a better algorithm might be able to express a changeset in a shorter edit script.

CHANGEDISTILLER is part of the EVOLIZER platform [66], an extensive infrastructure that integrates evolution information from source-code repositories with information from bug trackers. A preprocessing step extracts the essential information about the VCS history, which is then available for research through a meta-model that models changes. Part of the EVOLIZER infrastructure is an ECLIPSE integration that also captures fine-grained in-IDE change information [61]. This captured data is not persisted though and only used to identify change patterns that are related to the current activities of the developer.

Tracking Fine-grained Changes The great disadvantage of VCS data that is taken from repositories is that the granularity of the captured changes is very coarse-grained [109]. Researchers postulate that more fine-grained change information is needed to improve research on source-code evolution and that change should be modeled as a first class element in the process [199, 217]. Ebraert et al. [55] follow this line of argumentation and propose that a program and its history should be represented as a sequence of changes rather than explicit versions. Several years later, Negara et al. [166] underlined this when they found in a study that the history contained in a repository is not representative for source-code evolution.

In the meantime, several tools are available that can collect a fine-grained source code history directly from within the IDE. Two early works in this direction are PROJECTWATCHER [217] and MARMOSET [231], both tools can be installed as plugins in ECLIPSE. Once installed, they transparently track a fine-grained development history of the JAVA source-code under edit. Every in-IDE edit operation of the developer is automatically committed to a CVS repository. The technology choice to use CVS leads to a line-based change persistence in both cases.

Robbes et al. created SPYWARE [199, 202], a very similar tracker for source-code changes in Smalltalk. The tool automatically tracks any source-code edits down to the statement level, the important difference to previous works is that no textual representation is used, but that information about the change operation are stored on a semantic level. For example, it stores that a method was renamed (incl. the new name) or if a change was introduced by the execution of a refactoring.

Negara et al. introduce the Eclipse plugin CodingTracker [245] to track change operations. They go a different route in their design than previous works and represent the changes as fine-grained incremental updates to the source code as add, delete, or update operations that allow a precise replay of the development history. An important feature of the tool is that it can be combined with CodingSpectator [245] that tracks high-level in-IDE activities from the developer. The combination of both tools captures interaction data that is conceptually very close to the idea presented in this thesis, even though it captures only a subset of the available information.

Another close work to ours is the work by Dias et al. [44, 46]. They introduce Epicea, an extension for the Smalltalk IDE Pharo, which captures code changes on a structural level and preserve information from the edit process about refactorings, test runs, and version control. This makes it quite similar to this thesis, but Epicea only captures actions related to source code whereas we are interested in capturing general interactions. Epicea stores source code changes incrementally in a log, using the Ring format (see Section 2.2.3). Due to the nature of the dynamically typed host language, Smalltalk, the snapshots do not contain resolved type information other than the structural information about classes and methods for the current file under edit. Changes are stored incrementally, for example, in case of a name refactoring, the only stored information is specific to the refactoring: the type of the refactoring as well as the old and the new name of the method. This design makes it necessary to instrument every individual refactoring command to capture the specific information. Method bodies are only stored as text and will be parsed to an AST on demand.

Another work for Smalltalk is presented by Steinert et al. [234]. They capture fine-grained local histories with their tool CoExist to facilitate the exploration of software systems by embracing change and try and error programming. They postulate that this *explore-first programming* encourages developers to test ideas without the fear of breaking the current program state. The tool creates a distinct program version after each saved change. This history is only stored locally as a means to organize changes and is not persisted, nor meant to be shared with others, but developers can use it to easily revert the source code to any prior state by a simple click of a button.

Overall, many approaches exist that can track changes. We have surveyed approaches that are based on analyzing repositories and approaches that track changes directly from within the IDE. By reviewing these tools we have found that these tools can be differentiated in several dimensions.

The first dimension is the nature of the changes that are stored or processed. Some approaches only consider textual changes, while others support structural or semantic changes. Most of the time, the limitation to textual changes is simply caused by the VCS technology that is used to store the change (e.g., [217, 231]) or required if the change process should be reproduced as closely as possible (e.g., [245]). In-IDE tracker, on the other hand, can leverage available process information to identify structural changes (e.g. added AST nodes) or even semantic changes (e.g., rename method refactoring). Tools like Evolizer [66] try to close this gap and recover semantic information from a VCS history after the fact and represent the results in a semantic meta-model. It is an interesting question for future work if these approaches would benefit from more fine-grained change information or to analyze the effect of various change granularities (e.g., by change, save, commit, or fixed time interval).

The second dimension relates to the way the history is represented. Some approaches store complete snapshots for each program version, other only store incremental change information. Modern VCS are change based (e.g., line-based changes in GIT), but it is also very easy to *check out* complete snapshots from the history, so modern VCS usually can provide both and it depends on the approach what is used. This is similar for in-IDE trackers: Some use or store complete snapshots of the code under edit (e.g., [217, 231]) while others store incremental updates (e.g., [46, 199, 202, 245]). Both approaches have their advantages, e.g., consistency in snapshots or smaller change size for incremental storage, and no consensus exists for the best model. As in every incremental system, incremental updates make the history very volatile though. The corruption or loss of a single incremental step corrupts the whole history. So, depending on the use case, storing changes in a purely incremental fashion is a strong threat to consistency.

The third dimension is if or how typing information is considered in the tracker. Textual approaches usually do not consider type resolution by design. They assume that the captured code is complete, i.e., compilable and all dependencies resolvable, and leave the processing of the changes to the researcher. Structural or semantic in-IDE trackers do not have this problem, because the developer already sets up the necessary environment for the respective project. However, only few in-IDE trackers actually preserve the resolved types. A lot of research in this area addresses Smalltalk, in which only structural changes can be meaningfully represented with types, the method bodies are dynamically typed and can be represented in a simplistic AST. Resolved types are a crucial requirement for analyses in strongly-typed languages like JAVA or C# though. However, tracker in these languages seem to follow the same assumption as their VCS based counterparts and expect that the researcher has access to the complete source code to resolve types after the fact.

It is impossible to simply choose the one right thing to do in each dimension, each decision strongly depends on the use case. We will come back to these in later chapters, in which we will discuss the requirements and design decisions of our platform.

2.2.3 Intermediate Representations

While parsing source-code to an AST is the most straightforward way to create static analyses, doing so has several implications and might not be the easiest way. First, it is usually required to compile the source-code, at least if types are required. This is hard because necessary dependencies might not always be available or because projects might rely on a specific build environment. Second, working on an AST also makes it necessary to handle the complete semantic of the analyzed language, which involves considering all peculiarities of the language, like optional references in a program that need to be resolved. Related work has proposed several intermediate representations and exchange formats for source-code to ease these tasks. They facilitate different kinds of analyses and their features differ, depending on the use case.

Gomez et al. [73] introduced RING, a unified meta model for PHARO. The base model resembles a simple AST-like representation of the corresponding source code structure. Extensions exist that model changes and history of source code [243]. The model does not directly support method bodies, it only stores their textual representation that can be parsed into an AST on demand. This is only practical, because the AST of PHARO methods is relatively simplistic, i.e., it can be represented in 12 nodes. It is also not required to store typing information in the model because Pharo is dynamically typed and does not contain explicit references to types or type elements.

FAMIX [241] is a meta model that supports various programming languages such as Java, Smalltalk, or Python. Originally, is has been used to enable language-independent refactorings [51]. It has been extended though to support history, namely in the extensions Hismo [71] and Orion [114], which enables using it for work on source-code evolution. The model does not capture source-code on an AST level, but only on the level of program entities (e.g., variable declarations and method invocations) and does not model the exact control flow in a method. References to types and type elements are fully resolved though and part of the intermediate representation.

Necula et al. [164] introduced the C intermediate language (CIL) as an IR for C code. Their transformation from C to CIL resolves ambiguities in the source language and ensures unique type names by moving all type declarations to the top level and disambiguating their names. They do not deal with types declared external to the program being transformed.

Proteus [247] converts source code into an IR called Literal-Layout AST (LL-AST). The approach focuses on the problem of preserving document layout, i.e., comments and formatting, in automated code transformations. It deals with many C/C++ specific problems, such as preprocessors and macros. As their main concern is preserving document layout, they also include comments and formatting. We argue though that, while both need to be preserved when modifying source code, they are mostly irrelevant to investigations on how developers work.

JavaML [7] is an XML-based IR for source code. The transformation to JavaML inserts tags around source elements to identify the element's nature, e.g., whether it is a keyword, a type name, or an identifier. This allows analysis and transformation of source code using standard XML tools, such as XQuery or LINQ, instead of custom programmatic processing of ASTs. Apart from adding these annotations, the source code is left unchanged. The strength of this approach is that it makes it easy to process parts of the text separately, for example to create *island grammars* [153].

Existing analysis toolkits like Wala↗6, Soot↗5, or Opal↗2 either directly work on byte code or provide their own intermediate representation that is close. For example, Jimple [111] is a typed three-address representation that was developed as part of the Soot framework to simplify control- and data-flow analyses of JVM Bytecode. Generating a Jimple representation for a Java class requires its .class file, i.e., it must be compilable. While this is a common property for all frameworks (incl. Java byte code), it represents a limitation for our use case. Namely, we often encounter incomplete or uncompilable code during edit sessions in the IDE editor. This is also the reason why approaches that work with *unclean* source code are often based on the very robust Eclipse JDT parser instead (e.g., snippets taken from StackOverflow [236]). Another limitation is that this representation is quite far from the original source-code, which makes it both harder to learn plain source-code patterns from it and also to represent them later in a human-readable form. Finally, the biggest advantage of these frameworks opens up, if the class path is complete, because powerful inter-procedural static analyses exist in this case. In our use-case, it is not possible to capture the whole class path, as dependencies may change on every edit, and static analyses are typically restricted to the method scope, as we saw in the previous section.

Gousios et al. [77, 78] postulate that research on software engineering lacks standardized representations, which leads to duplicate development effort. They propose Alitheia, a platform that features standardized representations for several artifacts of the development process to facilitate the creation of reusable components. The great advantage of their data representation is the integration and connection of very different data sources and that the authors provide connectors to acquire data from a large set of projects and systems. Part of this is an intermediate representation for source code, which resembles a language-neutral and XML-based abstract syntax tree that holds the required meta-data to enable calculation of quality metrics. To the best of our knowledge, the representation goes down to the line level and it seems impossible, or at least inconvenient, to write more complex static analyses that consider control-flow or data-flow information. In addition, the representation does not preserves resolved type information. Overall, the goals and the identified problems of Gousios et al. align well with this thesis, however, their solution goes in a different direction. Their focus is on an integration of different data sources at the expense of details. We decided against building on top of ALITHEIA's source-code representation, because our focus was on defining the most-simple representation for the needs of RSSE, but it does not seem possible to easily extend the representation to our needs.

The M3 [10, 94] source-code model is part of the Rascal standard library [106], a meta-programming language for static analyses that runs on the JVM. M3 represents source code as abstract syntax trees that contains fully qualified type information. They include references to types and type elements in a special URI syntax that is used to reference the actual sources of the referred element. The implementations of the model are language specific and exist for RASCAL, PHP, and JAVA. One of the biggest disadvantages in our case is that M3 does not support C#. This project is very related to this thesis, but it evolved in parallel so we did not consider it as an alternative when we were designing our solution. It seems to be in a stable state by now, although it has a strong focus on Rascal as a language and the reusable calculation of metrics and visualization of source code.

An interesting concept that eases static analyses by enforcing well-typed programs by design is structural editing [237]. Instead of editing source-code as plaintext, the idea is to edit the program structure such that changes have to conform to the allowed composition of constructs. Every edit state creates a valid program, the missing parts in the program are represented as *holes* that are to be filled later. As a result, incomplete or broken source-code is avoided by design and programs are always parseable, which significantly increases analyzability. In the context of source-code evolution, however, edits can lead to incorrectly-typed fragments, for example, when a method signatures changes. Cyrus et al. [174] propose the HAZELNUT calculus that uses holes not only for missing parts, but also as wrappers for mistyped parts of a program, a technique that could be applied to capture evolving parts that are temporarily not compilable without sacrificing the benefits of analyzability.

GROUMs [171] encode source code as directed acyclic graphs. Invocations and control structures are encoded as action and control nodes and data-flow and control-flow as edges between those nodes. However, they are restricted to the method level and do not encode structural information such as class declarations. While they have not been used in that way, the GROUM representation could be used in shared datasets. GROUMs could be extended to include the missing structural information, but the main limitation for our use case is that references to types and type elements are only rudimentary qualified. For example, type references do not feature a namespace and method references do not contain the parameter list. The described transformation that creates GROUMs from source code simplifies the representation by truncating nested or chained expressions and only preserves the type name of the expression.

In addition, there have been several proposed standardized data structures and storages for usage in a variety of studies that focus on meta data. Bevan et al. [16] and Draheim et al. [50] each proposed approaches for storing source code and metadata such as evolution traces. OSSmole [32] aims to prepare data from heterogeneous sources for homogenous access by analyses. The authors encourage researchers to share their analysis scripts to enable reproducibility and reuse. Sourcerer [8] works with a database of metadata and source code of projects that can be queried via SQL. The datastore is mainly specialized for code search and does not contain detailed information about the contents of methods, such as control structures or call order. While these works all store source code and are therefore related to this work, we argue that their storage format is unfit for the intended use case of this thesis.

Overall, many intermediate representations exist and all of them are optimized for a specific use cases, e.g., preserving versioning properties, abstraction of source code, representations for meta data, or to represent code under edit. The various representations introduce interesting concepts, like representing versioning, abstracting the source code, preserving types and providing means to follow references, storing typed holes, or -the very pragmatic requirement- being available in C#. A general solution would combine all these concepts, but does not exist. We could not find an existing solution that could be used in this thesis.

2.2.4 Reusable Program Analysis Platforms

Analyzing the software development process and its tasks is an established research area. Due to the many commonalities and chances for reuse among different studies, there is a great chance for reuse. It is therefore of little surprise that various platforms have already been proposed that provide reusable components for data cleaning, preprocessing, and for static analyses of source code. In the following, we will go through these platforms to analyze their applicability to the use case discussed in this thesis.

RASCAL [106] is a DSL for analyses and transformations of source code. The core approach is language agnostic, with language-specific constructs being provided as reusable libraries. The language environment provides resolved type information when working directly on the abstract syntax tree and it needs compilable sources.

The Boa project [52, 53, 54] aims to simplify writing static analyses that can scale to the ultra-large scale of source-code available in public repositories. They approach this endeavor by providing both a domain-specific language, a mining infrastructure, and curated datasets (see also Section 2.2.5) to facilitate the analysis of large-scale repositories. To access the data, Boa users write their own analysis programs in a domain specific language and execute them online, only the results are made available in a text format for further processing. The analyses use a simple visitor-like structure that can be used to traverse the abstract syntax trees. This makes the analyses very scalable and keeps the overhead for testing initial hypotheses very low. However, the capabilities are also very limited and the platform is unsuitable for more advanced static analyses, such as pointer analysis.

Static analysis frameworks such as Wala,[6] Soot,[5] and Opal[2] provide reusable modules for common static analysis tasks. They work on low-level IRs, such as Java Bytecode or Jimple, which facilitate static analysis. We cannot use these frameworks to create our IR and their representation has limitations for our intended use case. For example, they do not directly include control structures and make it necessary to recover them from the intermediate representation, which makes it hard to learn usage patterns in which control flow is important (see Section 2.2.3 for a complete reasoning). However, we can convert our captured information into their formats and make use of the rich set of static analyses provided by these frameworks, should it be required for a more sophisticated static analysis.

srcML [31] is an infrastructure for the exploration, analysis, and manipulation of source code, which was built around JavaML (see discussion in Section 2.2.3). It generalizes the idea of JavaML by abstracting over programming languages. The format and transformation handles C, Cpp, C++, C#, and Java. Similar to JavaML, the transformation annotates source elements by XML tags, but leaves the source code otherwise unchanged. This especially means that both JavaML and srcML do not provide resolved type information to facilitate analysis tasks. The platform provides several reusable analysis on top of srcML. We are aware of the slicing analysis tool srcSlice, the visitor implementation srcSAX, and stereocode which annotates code elements with stereotypes[18] (only supports C++). The limitations of the representation with respect to our use case made us decide against using it as the basis for our research.

The Ontological Adaptive Service-Sharing Integration System (OASIS) [97] is an integration methodology to achieve reusability and interoperability of existing tools. This goal is achieved by defining the shared conceptual space of multiple tools and mapping these concepts to their representations for each individual tool. While the original work implements OASIS for software reengineering tasks, we want to create an OASIS-like platform for recommender systems in software engineering.

Alitheia [77, 78] is a platform that facilitates research on process and quality metrics in a diverse and large database of open source projects. A meta-model of the development process integrates various artifacts in standardized representations, i.e., source code, issue trackers, and mailing lists. This standardization represents the means for providing reusable components on top of the representation that calculate the process and quality metrics. The ultimate goal is a reduction of the fragmentation and duplicated development effort of research tools. The authors focus on the integration of various data sources and abstract the artifacts to the detail level that is required for the metric calculation. However, this comes at the expense of generality. For example, the source code representation is unfit for more advanced static analyses, like data-flow or control-flow analyses (see Section 2.2.3), which prevented us from building the tools of this thesis on top of Alitheia.

The distributed and collaborative software analysis platform SOFAS [70] provides reusable tooling and datasets for research on software evolution. It integrates data from various sources, such as version-control systems, issue trackers, and mailing lists, in their corresponding representations. For example, source code is captured in a FAMIX meta model. To the best of our knowledge, it currently neither provides a representation of source-code with fully-qualified type information nor static analyses that we could have reused.

To summarize, all reviewed platforms always present a tradeoff between the potential sophistication and generality of static analyses and the ease of use. None of the existing platform fits the needs of this thesis. To facilitate the creation and evaluation of RSSE, we need a platform that standardizes the data structures to open up the possibility to build reusable components and to save effort. In addition to the requirements for the intermediate representation that we have discussed before, it is necessary to provide several components for data processing, advanced static analyses, and transformations that relieve toolsmiths from many of the recurring tasks they need to analyze source code. All reviewed platforms are either too specific and prevent the kind of static analyses we need to write in our domain or they are too general, leaving a lot of responsibility at the users side, which requires additional overhead and makes them hard to use. We conclude that it is necessary to build a new platform that is focused on our specific use case.

2.2.5 Datasets

The three pillars of empirical research are replicability, reproducibility, and reusability. *Replicability* can be reached through standardized data schemas and shared tools. *Reproducibility* requires that the body of an experiment contains all information that is required for other researchers to repeat an experiment. *Reusability* facilitates research in a specific context through reusable tools and datasets. In summary, empirical research relies on establishing standardized data schemas and sharing tools and datasets to achieve these three properties.

One of the biggest challenges for researchers is to find significant and reliable datasets. While versioned source code is available in many public repositories of open-source projects, it is much harder to get access to more fine-grained change information or to activities that describe the in-IDE development process. Once the important properties of a target domain have been identified, extensive datasets can be generated through simulation [179, 219]. Unfortunately, the in-IDE development process is very complex and also not yet fully understood to be able to simulate it. As a result, it is required to create appropriate datasets.

In this section, we will introduce and discuss three different kinds of datasets that are related to this thesis: datasets of versioned source-code that facilitate analyses, datasets of meta-data that provide additional details about source code, and datasets that represent in-IDE development activities through recorded developer interactions.

Source-code A great effort in software engineering research goes into analyzing source code. Currently, the most common way seen in literature to make the subject of a source-based experiment reusable by others is to refer to the precise projects that have been used in the experiment. Modern libraries are often released in provisioning systems like MAVEN and their stable versions make it possible to use unique names and versions. Other experiments analyze the source code found in public repositories, for example if a finer change granularity is required or if specific source-code properties should be analyzed that are not preserved in the compiled form (e.g., generic types or control structures). Researchers can refer to commit ids or timestamps to specify the source-code use in the experiment. However, both approaches have significant drawbacks. A drawback that is shared by both approaches is that the setup of the experimental environment is left to future dataset (re-)users. This does not only mean additional effort, but -more importantly- also that the environment is likely to change: dependencies of the contained projects might get updated or are no longer available. As a result, it is preferable that source-code datasets include their dependencies, both for convenience and for consistency. In addition, modern version-control systems support rewriting the history, which is common for several development styles (e.g., when using feature branches, while keeping a linear master branch). This rewriting breaks traditional assumptions like stable commit ids or a strictly ordered timestamps of commits, which makes it harder to reliably identify source code in repositories.

Several approaches make source code available for research through in large and stable datasets. The PROMISE repository [215] was an early advance to provide a platform to share datasets and tools used in mining studies. In the meantime, the repository has been superseded by the TERA-PROMISE repository that is part of the *Software Engineering Open Science Portal.*[38] The repository contains more than just source-code and provides a number of datasets and tools for tasks such as software-defect prediction, software-cost estimation, and requirements tracing.

The QUALITASCORPUS [239] is a curated collection of open-source JAVA source code for empirical research. The corpus contains dependencies for most contained projects and describes how the missing dependencies can be installed. The contained sources of the current 20130901 release are slightly dated though and do not contain examples for the most recent extensions to JAVA, e.g., streams or lambda expressions. The corpus was created with the goal to facilitate the reproducibility of studies. It comes in multiple flavors, one version of the dataset contains the most recent releases of 112 projects. A second version contains 579 releases of 15 different projects. An extension of the corpus has been presented that contains source code and labeled code clones [238].

Sawant et al. [214] extract a dataset for usages of API methods and annotations from a large number of GITHUB repositories. From all libraries that have been referenced in the repositories, the authors select the five most popular open-source projects that are reasonably large and actively developed. The authors analyze how the API usage of these libraries changes across different framework versions. The dataset introduces a data schema that models projects and classes, but no source code. It is only preserved, which method calls and annotations are contained in a class declaration.

The BOA project [52, 53, 54] provides an infrastructure for writing analyses with an ultra-large-scale repository (see also Section 2.2.4). The project provides curated data sets that contain fine-grained source code from many projects hosted on GITHUB (~ 7.8M projects) and SOURCEFORGE (~ 700K projects). The source-code is stored in a custom AST format that supports most JAVA constructs. It stores fully-qualified information for types and type member, but cannot be used to not resolve references. In addition, the dataset contains further meta-data about the projects and the commit history, even though it is not possible to recover branching information. The BOA platform is designed for a online-analyses and also the datasets are not available for download. BOA reduces the setup overhead that is always required to run analyses on a local dataset. However, it also limits the complexity of the analyses and the data exchange for the results has to follow strict rules.

Source-Code Under Development Changes extracted from the commit history of repositories are coarse-grained [109] and not representative for actual source-code evolution [166]. Other means are required to capture intermediate states of source code to be able to find and analyze the problems programmers are facing during development. Several tools exist that can record source-code changes during regular coding activities (e.g., [202, 217, 231, 234]) to enable studies of the fine-grained evolution of source code. While Spacco et al. [231] report that 80% of these snapshots can be compiled, these works usually capture immediate states *on save*, loosing many intermediate *edit* steps in between. Unfortunately, to the best of our knowledge, no actual dataset was ever published that contains a reasonable amount of fine-grained changes to source code. The lack of such a dataset was one of the motivations for our work. We emphasize the value of such datasets and advocate for not only creating them, but also for making them available to other researchers.

A different form of incomplete source-code that is still under development can be found on question and answer sites like REDDIT or STACKOVERFLOW. These sites have become an important data source for empirical research on software engineering. While existing research was mainly focused around the textual parts of the posts, programmers ask many questions about a specific coding problem when they are stuck with their solution. Their questions often contain incomplete code snippets, which are completed or rewritten by the community. The high heterogeneity of the posts makes it challenging to use them though. Posts mix textual parts, source-code (which itself is often not complete, invalid, or simply does not specify the programming language), and data in formats like XML or JSON. Ponzanelli et al. [182] solve this with an *island grammar* [153] that can be used to parse STACKOVERFLOW posts into *heterogeneous AST* (H-AST). The grammar supports JAVA, XML, JSON, stack traces, and text fragments and can be used to transform released STACKOVERFLOW data dumps.[24] Tools like BAKER [236] can then be applied to recover typing information in these code snippets, making it possible to use fully-qualified references for types and type elements in static analyses, without having to compile the respective snippets.

Meta-data Another kind of dataset exists that does not focus on source code, but on the meta data that describes projects. In the context of this thesis, these datasets are interesting as an additional data source that can be integrated into analyses to enrich existing data. For example, to quantifying the representativeness of experimental results and to optimize for it [161, 187]. Several such datasets and platforms exist.

OSSMOLE [32] was one of the first advances to create a high-quality database of FLOSS project information for research. The authors achieve this through publishing standard analyses that enable replication of results and through facilitating reuse of analysis scripts by others. The collected data contains project related information (e.g., name, homepage, primary language) and developer oriented information (e.g., number of developers, developer roles), and issue tracking data (e.g., bug open, update, or close). The tool supports preprocess data from various data sources like SOURCEFORGE and stores them in a unified datamodel that can be used by analyses.

FLOSSMETRICS [85] is a similar platform for open-source data analytics. Several retrieval systems capture various metrics (e.g., lines of code or cost estimation) and meta-data (e.g., commit logs, mailing lists, or bug trackers) for more than 1,000 projects. While the original work has published the dataset, the research projects has evolved into the GRIMOIRE LAB, a merge of several projects that are concerned with analytics of open source software.[39] The new project does not provide datasets, but publishes and maintains all their tools that are required to analyze open-source projects through their public artifacts. The tools support various data sources, for example, repositories, bug trackers, mailing lists, wikis, or instant messengers.

GHTORRENT [76] is an effort to make the vast amount of development activities on GITHUB available for research. The project stores the development events of GITHUB repositories (for example, activities that include push, fork, or branch operations). The captured data is enriched with meta data about the involved repositories (e.g., primary language) and corresponding users (e.g., unique id) and is frequently released. The dataset is more specific than the previously discussed approaches and only contains data for a single platform (GITHUB), but its dataset is huge.

OPENHUB[29] is a website that curates metadata for a large number of open-source projects. The provided data consists of general meta-data (e.g., repository urls, main programming language, license) and of several metrics regarding the source code (e.g., lines of code), the contributors (e.g., activity), and historical data regarding the evolution of the project. The database can be accessed through a REST-based API.

Other approaches do not stop at the project level to capture metric, but go down to the program level. Bevan et al. [16] propose KENYON, a general solution for logistic problems of fact extraction and storage faced by empirical researchers that analyze development artifacts like source code. KENYON has a focus on software evolution and provides an extensible infrastructure that allows extracting facts through custom extractors. The resulting database can be used for feature analyses, analyses of dependence graph history, or as a data collection framework for data-driven tools that are derived from the source code, e.g., recommendation systems.

SOURCERER [8] is an infrastructure that supports indexing and analysis of open source projects on a large scale. The datastore is specialized for code search (e.g., [9]) and captures the source code found in the repository in a reduced meta model. The model is similar to a call-graph, but it also contains structural information like fully-qualified references to types and type elements that link the different entities contained in the dataset. It does not contain detailed information about the contents of method bodies, such as control structures or call order.

Draheim et al. [50] propose BLOOF, which extracts historical information from the version control system of an open-source project to provide the infrastructure for research on software evolution. The change process is transformed into a meta-model that reduces commits to developers, files, and several basic information about the change (e.g., number of lines changed), reflecting a more process-centric viewpoint of the change activity. The meta-model is stored in a database that serves as an additional layer for process information over the actual version control system that can be queried for the source code.

Interactions This thesis studies the in-IDE developer behavior, ephemeral information that has to be tracked while it is happening, because it cannot be reconstructed after the fact from artifacts. It is not only a lot harder to collect, the process also involves many privacy issues. As a direct result, very few datasets exists of such data.

One of the earliest and most extensive datasets of developer interactions is the public dataset of the ECLIPSE USAGE DATA COLLECTOR[14]. The intention of this project was to provide a means for plugin developers to analyze which functionality of ECLIPSE was actually being used by their users. To this end, the dataset contains a log of executed commands and activated windows, grouped by user. The log is very shallow and does not contain further context information or source-code. The dataset was compiled over a period of 20 months and has a significant size of several gigabytes. In the meantime, due to the lack of users, both the development of the collected and the data collection has been suspended though.

The BLAZE [230] tool can be installed in VISUAL STUDIO 2010 to track which commands are invoked by the developer in the IDE. Singh et al. [227] deployed it within ABB INC. and tracked activities of almost 200 developers, covering more than 30K hours of active development time. Both the tool and the collected data are available for download.[15] While the data that is being captured by BLAZE originally contains some context about the activity (e.g., hashes of file names), the published part of the dataset only contains the invoked command ids together with a timestamp, grouped by developer, the additional context information is being removed. In the meantime, the development of the tool and the data collection is being continued by CODEALIKE[22] in a public setting, but, unfortunately, no dataset has been published so far.

Two extensions of the Pharo IDE are concerned with analyzing developer behavior and are closely related to this thesis. Minelli et al. [150] proposed DFlow to capture information about in-IDE activities. To this end, they capture source-code changes on a structural level (e.g., method rename), windows that are being interacted with, and the layout space occupied by open windows. Their dataset covers 750 hours of development work of 17 developers. Dias et al. [46] propose the tool Epicea that tracks edit related operations in the IDE. The tool also preserves source-code changes on a structural level (e.g., method rename), but ignores method bodies. The captured data also contains information about executed refactorings, test execution, and usage of version control systems. Their dataset [45] was collected over 4 months from seven participants. Both tools have a slightly different use case than this thesis, but were highly related. Unfortunately, the authors of both projects decided against publishing their datasets of captured developer interactions.

Overall, no existing work has introduced a dataset or data schema fit to holistically capture in-IDE development activities, in which we have been interested in for the work conducted in this thesis. The existing dataset usually focus on either source-code or on general activities and do not combine both worlds. Although we could not reuse an existing dataset, several of these works have inspired our research. We want to combine existing ideas into a data schema that brings both worlds together, we want to create reusable tools that are based on this data schema, and we want to capture a dataset that can be shared with other researchers.

Chapter Summary

The goal of this thesis is to study the in-IDE time activities of software developers, which touches many related research areas that have been discussed in this chapter. We have split our review into two parts. In the first part, we have discussed three lines of research that motivated our work as possible applications of our results. The discussion has reflected on empirical studies on software developers (Section 2.1.1), recommendation engines in software engineering (Section 2.1.2), and source-code evolution (Section 2.1.3). In the second part, we have discussed approaches that present solutions to inherent problems of these applications. We have covered trackers that capture development activities, like interaction tracker (Section Section 2.2.1) and tracker for source-code changes (Section 2.2.2). Afterwards, we have presented several intermediate representations for source code as candidates for the preservation of source-code changes (Section 2.2.3). Finally, we have introduced existing platforms that support the creation of re-usable static analyses (Section 2.2.4) and existing datasets that can be used for empirical studies on developers or code (Section 2.2.5).

After our literature review, we conclude that no existing solution exists that is fit for our intended use case. No existing tracking system allows the capturing of development activities together with the amount of context information that we have envisioned in the introduction to this thesis. Also a new intermediate representation for source code is required that can be used to capture in-IDE source-code snapshots and on top of which sophisticated static analyses can be written. Finally, reusable components need to be created to make working with the captured data practical. While not being reusable for our use case, the surveyed works have heavily influenced our own solutions to the challenges that are presented in the remainder of this work.

Part II:

Meta Models for In-IDE Development

In this second part of the thesis, we will introduce the meta models that we have created for representing the development process. We have created different models that cover two specific purposes. We will first introduce how we model development activities that can be observed in-IDE. After that, we will propose a meta-model for source code that facilitates static analyses and studies on source-code evolution. Both models build the basis for recording development activities for later replay and analysis.

3 Meta Model for In-IDE Developer Activities

Developers work in their IDE on various tasks and with many tools. In order to analyze their actions after the fact, it is necessary to store them in a form that can be serialized and shared. The typical way to store interactions is to model them as a stream of events (e.g., click-through data in the web, system log in applications), but these representations are usually very *flat*, in the sense that they do not provide any details about the event that occurred. If it is possible at all, researchers have to reconstruct the context of the interaction by analyzing the stream.

In this chapter, we will introduce a new meta model for in-IDE developer activities. We will introduce *enriched event streams*, an event-based representation that captures additional context information to facilitate later analyses of the developers activities. We will present our design process and the conceptual design of our meta model in Section 3.1. We will then motivate and introduce the supported activities in Section 3.2, describe how the meta model can be extended in future work in Section 3.3, and will discuss limitations of the current state in Section 3.4.

3.1 Designing Enriched Event Streams

Before deciding on the content of concrete events, it is required to decide on a conceptual level how developer activities will be preserved and how the meta model is realized on an abstract level. The definition of a data structure that can store this information is required to be able to replay and analyze the in-IDE development process. In contrast to the common *flat* event streams (e.g., traditional system log data) that are typically reduced to a single line in a log, we wanted to preserve detailed information about the development activities. Previous work has shown that such context information can be used to describe the intent of a developer, which enables, for example, the identification of similar tasks [122, 123].

In this section, we will first formulate our requirements for such a representation, then discuss our relevant design decisions that influenced the design, and present an overview of the conceptual design of enriched event streams.

3.1.1 Requirements

A general meta model for in-IDE developer activities has to fulfill several requirements to be useful and applicable to a wide range of research questions. To identify a set of sensible requirements, we have reviewed related work and also foresee challenges ourselves that should be facilitated by an appropriate design of the meta model. A meta model that is used in shared dataset has a long life expectancy and is used by different developers. To make it usable in practice, it is also required to define requirements that ensure it can evolve over time. We have collected requirements and identified the following set that should be fulfilled by the meta model.

Basic Information In our literature review, we saw several approaches that track the activities of developers. Most approaches represent the interactions on the level of executed commands (e.g., [39, 148, 156, 230]). The amount of captured context data differs, but some basic context information is universally required across all approaches, including the timing of the interactions (e.g. [148]) and some notion of the location or the target of an interaction (e.g., [102, 156]). In addition, some approaches track how an activity was triggered (e.g., [266]) and which user caused the interaction (e.g., [13]). A new meta model should cover all these basic information.

Distinguish Automated Activities The developer is the only actor in the IDE, however, the IDE can also be involved with processing tasks that were initiated by tools that are scripted or that react to actions of the developer. For example, the user might initiate a name refactoring and, as a result, several automated changes will be initiated in the affected files that refer to the name. The automated changes have a different flavor and the meta model should allow to differentiate them from the manual ones. Related work on source-code evolution solves this by considering a refactoring log that may be maintained by the IDE (e.g., [48]). We want to be able to recover the same information from our meta model, so it should be possible to represent the invocation of arbitrary IDE tools.

Different Granularities Our review of related work on source-code evolution has shown that source-code changes are being analyzed on different granularities. Typically, either *on-commit* (e.g., [60]) or *on-save* granularity (e.g., [231]) is used. We want to go beyond that though and capture a more fine-granular *on-change* granularity. We want to couple this with the required context information that allows to replicate the more coarse-grained states or to derive custom strategies to merge changes.

In addition, some requirements are only relevant, when the context of this thesis is being considered. We try to unify over existing approaches and want to establish a general meta model for development activities. The model should be applicable in a wide range of existing and future applications and should be the base for reusable datasets. As a result, we identified requirements that have not been formalized in related work so far, because they only emerge in our specific use case.

Arbitrary Context Related work often captures only little information from the environment to represent development activities, which makes it possible to use simple data schemas that can be represented in tables with only a few columns (e.g., [39]). However, our use case differs, because it requires the creation of arbitrary complex context classes. The final design should facilitate working with the meta model in an object-oriented manner, instead of the SQL query level.

Generality Even though we focus on a specific IDE in this thesis, the data structure itself should not be specific. While the concrete interactions inside different IDEs might differ, the high-level actions stay the same so instead of focusing on a specific functionality only available in a specific IDE (e.g., a specific tool for code search), the focus should be the high-level development activity (e.g., navigation).

Extensibility It is very likely that future work requires extensions to the meta model. For example, to represent activities that take place outside of the IDE (e.g., switch to other applications), that happen away from the machine (e.g., meetings), or that happen on an even lower level than actual tool interactions (e.g., biometric information). While such context information is out of scope for this thesis, the design should facilitate such extensions.

While it is the responsibility of the toolsmith to ensure that an extension does not break existing clients, the design of the data structure should define clear ways of extension to help avoiding breaking changes.

Versioning A meta model should not introduce breaking changes during its evolution and stay consistent. In reality, this is hard to avoid. Extensions may be introduced that add or change information. Clients might want to distinguish the original data points from the extended one or they might want to ignore the original data points all together. Even when the meta model itself does not change, the instrumentation could change that tracks the activities from developers. For example, through a bug fix that affects a specific activity. Clients that analyze this activity need to be able to identify affected data to ignore it. It is, therefore, crucial to version captured data to indicate how it was created, which enables selective filters.

Abstract Concepts It seems required to include as many details as possible in the meta model to represent the original activities as precise as possible. However, our specific use case makes it necessary to reflect on this. We want to preserve development activities in a distributed setting, which results in large event streams. It is required that this amount of data is still manageable and that it can be distributed over the internet. In addition, we want to avoid privacy problems that are caused by including too much user information in the meta model. To support these non-functional requirements, we should introduce abstractions for the data that allow us to reduce the whole pool of available context data to the essential information.

This sums up the requirements that we have identified in our review of related work and that we have considered in our design.

Revisiting Existing Solutions In our survey on related work, we found several existing solutions that are close to our work and address similar requirements.

HACKYSTAT [99], PROM [224] or INTI [121] go beyond in-IDE tracking and apply sensors to various applications. Their main goal is to capture a holistic picture of the development process. They collect data about the developer interactions with different artifacts, but stay on a coarse detail level for the captured information.

Several in-IDE trackers exist, e.g., MYLYN [102], or BLAZE [230]. All of them focus on the process level, i.e., the invoked commands in the IDE, and only include basic information about the targets and selections for the commands. They do not allow the preservation of more complex context data and we did not see how their existing meta models could be extended to support our requirements.

Notable exceptions are the two trackers DFLOW [150] and WATCHDOG [13]. DFLOWS meta model can also capture window interactions and structural navigation in the source code and WATCHDOG can preserves complex testing information in its meta model. However, the former puts the focus of the meta model on visibility and navigation, the latter does not capture individual events, but aggregates the information on the client and only stores intervals. Both were not applicable in our case.

The works that are the closest to this thesis are the combination of CodingSpectator and CodingTracker [244], as well as Epicea [45]. Both capture developer activities and align them with source code changes. However, their meta models are both restricted to activities that are directly related to source changes and would need to be extended to the general case. In addition, Epicea captures source code changes only on the structural level and CodingTracker does not preserve resolved typing information. We would have needed to exchange the representation for source code in both cases.

Overall, we conclude that all existing representations are not applicable in our use case. We had to design a new meta model that is tailored to our requirements.

3.1.2 Design Goals and Decisions

After identifying the requirements from related work, it is necessary to decide on some details for the design. These decisions may be necessary, because no requirements exists or to decide between alternative solutions. In the following, we will discuss several design goals and our decisions that lead to the final design.

Event Stream To make the meta model applicable to a variety of research questions, we propose to combine traditional event stream (e.g., subsequent interactions) with related context information in an *enriched event stream* that contains both.

Some of the observed activities have dependency relations (e.g., recall the example of a refactoring that causes subsequent events). While it would be possible to include these dependencies in the meta model, e.g., by representing them as a tree of actions in which parent actions cause children actions, we decided to not represent these dependencies. However, to preserve the cause of an action and to mark the ones that have been automatically triggered, we include the trigger cause, which can represent *by-mouse*, *by-shortcut*, or *automatic* triggers of actions.

User Identification We don't want to enforce the identification of a user and we decided to separate users only through our organization and management of the collected data. However, developers might work in multiple instances of their IDE at once and these activities might be merged in a single log. We would like to be able to distinguish the individual sources in such a log. To solve this, we will include a unique session identifier in the meta model that allows to group activities by individual IDE session. In addition, we want to support an optional user identification and want to ask for additional background information, if users are willing to share this information.

Separate Concerns The applications that motivate the work in this thesis require information on very different levels. Empirical studies on developers require information about the high-level activities, while for example recommendation systems in software engineering are mainly concerned with source code, which is merely a development artifact. Our meta model should support all these different levels though.

We want to clearly separate the different concerns and do not want to mix unrelated concepts. We decided that the high-level concept that is represented by the meta model should be the development *process*, i.e., activities, interactions, and events that happen in the IDE. In addition, we want to be able to attach additional information that provides *context* to understand these activities. The context should be allowed to be arbitrarily complex, for example, we believe that the source code under edit is also just an extended context information. However, we want to establish clear responsibilities. Information related to the process should be stored in the enriched event stream, all the rest should be stored in specialized context.

To illustrate this point, let us review the requirements of works on source-code evolution. Works typically analyze subsequent versions of a program and might use a refactoring log to improve the classification of changes. In this case, some of the required information relate to the process (i.e., the invocation of refactoring tools) and should be represented in the enriched event stream. However, we think that the source-code representation should not be concerned with details about the evolution and should be restricted to the syntactical level. To keep the coupling between both concepts low while keeping the representation useful, we will include anchor points in the source code that allow to link process information with the source-code, e.g., the cursor location in the file while the snapshot was taken.

Stability Indicators Due to the lack of available data, most existing research has analyzed the *on-commit* granularity and only few approaches have analyzed the *on-save* granularity of source-code changes. We think that the difference is crucial for understanding how developers work, because the more fine-grained the change history, the closer it is to the train of thought of the developer. However, incremental compilation and constant syntax checking of a program during edit operations make regular saves unnecessary. Tooling that tracks activities in the IDE has access to the intermediate states though and to get a more precise picture of the development in between save operations, we want to represent source-code changes with an *on-edit* granularity. However, one potential problem with capturing such code snapshots from the IDE is that the change granularity might be too fine to find meaningful patterns, because the changes are no longer grouped. We argue that having some basic versioning indicators such as *on-save*, or *on-commit* correlated with the *on-edit* code snapshot can allow different groupings of snapshots and the analysis of the data at different levels of granularity, depending on the intention and heuristics used.

Implementation We want to put the focus on having a stable implementation of the meta model with a clear object-oriented design that clearly separates concerns and that introduces consistent and coherent classes. The implementation should facilitate the creation of expressive analyses in which the dataset can be filter and reduced to a subset of the information that is relevant for a given research question.

Having an implementation of the meta model is also required to apply established serialization techniques (e.g., OR-Mapper or XML/JSON serialization frameworks) to serialize the data into a common format (e.g., document-oriented No-SQL databases, plain files, or SQL tables) that is most convenient for further processing.

Understandability The terminology should reflect high-level concepts that can be understood by a wide range of researchers (e.g., code completion) instead of using IDE specific terms (e.g., IntelliSense, the code completion of Visual Studio). If it is necessary though to decide for a terminology, because conflicting terms exist (e.g., a *solution* in Visual Studio and a *workspace* in Eclipse), then the decision for the terminology should be consistent throughout the corresponding domain for further collisions (e.g., *namespace*/*package* or *assembly*/*Java archive (JAR)*). To resolve such cases, we decided to stick with the terminology of Visual Studio and C#, whenever in doubt.

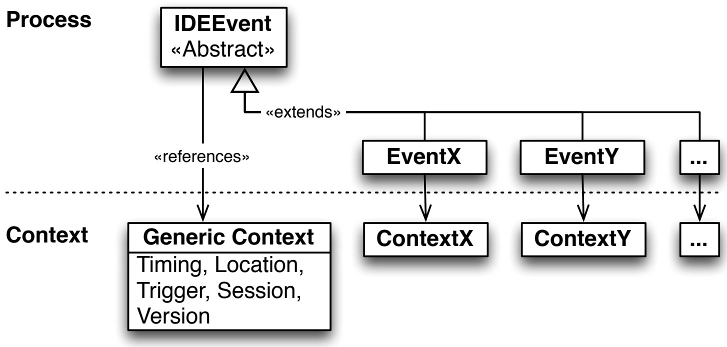


Figure 3.1: Design Concept for Enriched Event Streams

3.1.3 Design Overview

After considering all requirements that we have either identified from related work or because of their relevance for our own use case, we concluded that we cannot reuse an existing meta model and that the creation of a new meta model is required. After discussing the design goals and decisions that have further influenced our design, we have designed the data structures for a new meta model as shown in Figure 3.1.

Our conceptual design can be divided into two layers: the first layer represents the *process* in a hierarchy of `IDEEvents` that denote the different kinds of interactions. Each event has a second layer, in which additional *context* information about a captured interaction is stored. The basic information (e.g., timing) that is universally used is stored as generic context information in the base class. In addition, each derived event class can capture specialized context (e.g., test results or code snapshots). A third layer that is not depicted is a naming scheme for both *IDE components* (e.g., identifiers of windows or file names) and *code elements* (see Section 4.2) that allows to unambiguously refer to locations, targets, or code elements.

Generic Context Every event of the event stream carries some generic context information that is inherited from the abstract base class `IDEEvent`. This generic context information can be categorized into five different areas.

Timing We preserve timing information about the event, more specifically, we capture when the event was triggered. Some events also take some processing time (e.g., a compilation). We also support the storage of the duration in such cases.

Current Location We capture information to preserve locality and to describe "where" the developer did something. We use our naming scheme do denote the window and the document that were active at the time the event was triggered. Having this information allows to denote the location of an interaction, e.g., the event was triggered while working in "window y" or while working on "file x".

Trigger We preserve information about how an event was triggered. For user-triggered events, we differentiate between mouse click or by keyboard shortcut, but we can also mark events that were automatically triggered, e.g., edits in a document that are automatically performed as a result of an executed refactoring.

Session We avoid having a personal identification mechanism for users and rely on our

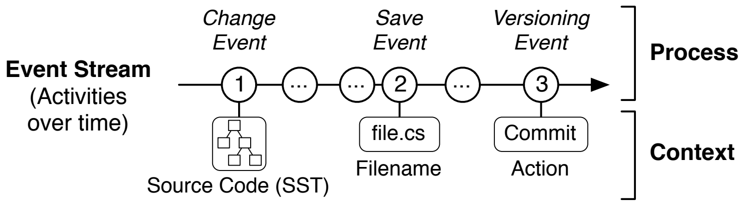


Figure 3.2: Example Enriched Event Stream

management and organization of the captured data to group all tracked interactions of a single user, e.g., by serialization into a single file. However, sessions of the same user can overlap when multiple IDE instances that are running in parallel. These can be disambiguated through attaching unique session identifiers to the events.

Version The version field indicates the version of the tooling that created the event. This can be used to filter older and invalid events or to restrict an analysis to a specific range of versions of said tooling.

The described information is relevant for every event, but some parts can be superfluous. For example, instant events like *File Save* have a duration that is zero, so we allow omitting the latter in such cases to reduce the memory consumption.

Specialized Context In addition to the generic context, concrete event classes that extend `IDEEvent` can store their own specialized context data, which is possible in two ways. Additional data can be directly stored in the corresponding context as a field, which is advised for primitive data types like numbers or strings (e.g., an executed command id in case of command events). However, some context information is more complex and should split into separate classes. For example, while the results of a test run could be stored directly within a `TestContext` class, this would complicate their analysis and abstracting the information into a specialized class (e.g., `TestRunResult`) that stores the name, the duration and the result of a single test, is preferred.

Factoring out isolated concepts ensures coherent classes and prevents a mix of concepts within a single class. However, the more specialized context classes are introduced, the more effort is necessary to create an implementation of the meta model on new platforms. This tradeoff has to be decided per case though. We do not introduce any conceptual limitation for the types that are used in the specialized context, with the exception of technical requirements that will be subject of Chapter 5, in which the infrastructure will be discussed.

Illustrative Example We show a concrete example of an enriched event stream in Figure 3.2 to illustrate the concept of the split into the two levels: *process* and *context*. For the example, assume that a developer is writing source code in a file, saves the file and commits it to the repository.

On the *process level*, we capture single events in a stream that have an *event type*. The ones that are relevant for our example are *source code change*, *file save*, or *versioning action*, but other events might happen in between (e.g., *window close*), which are left our right now for brevity. The events build a stream that allows following the different activities of the developer after the fact. However, it is hard to reason about the course of actions without any further context information.

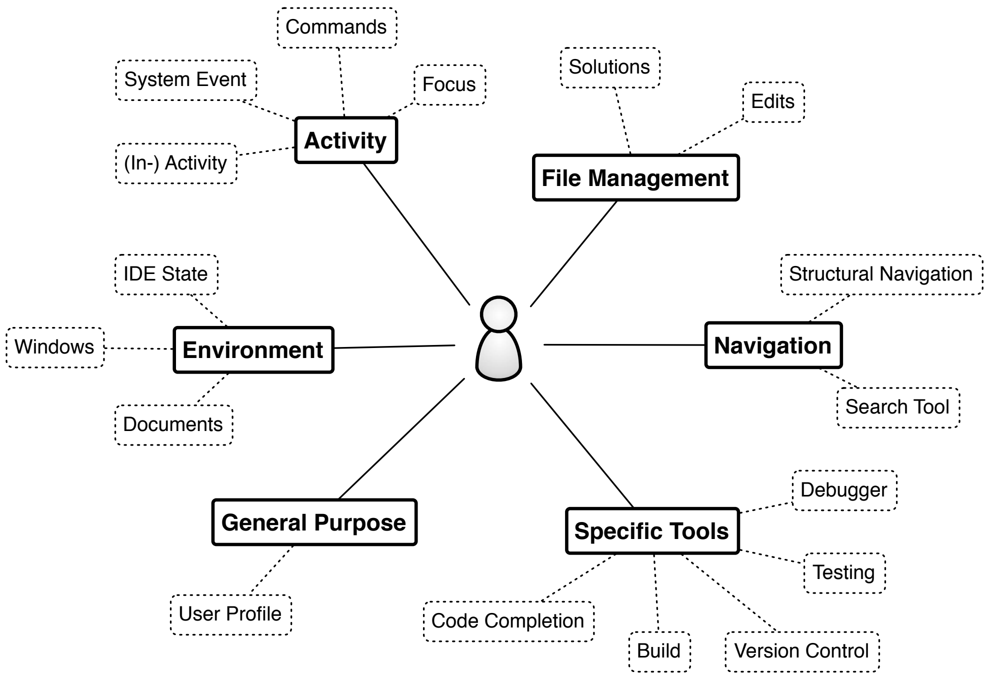


Figure 3.3: In-IDE Development Activities Supported in Enriched Event Streams

To support the investigation of development activities, we allow the storage of additional information on the *context level.* The context information includes generic information (e.g., the time at which the events were triggered), but also additional context that is specific to the individual event types. For example, the *edit event* contains a snapshot of the current file under edit, which allows an analysis of the edited source code after the fact. The *save events* store the name of the file that was saved and the way in which the save action was invoked (e.g., by short cut or by menu selection). The *version control event* indicates the name of the current solution and the version control action, i.e., that the changes have been committed. All these events also contain other information that we left out for brevity in this illustration.

3.2 Supported Development Activities

In the last section, we have designed a conceptual representation of a new meta model for developer interactions. Now that the conceptual representation is decided, it is required to select the development activities that should be supported in the meta model. Figure 3.3 shows an overview of different activity categories that are supported in enriched event streams and breaks them down to concrete development events.

Preliminary versions of our meta model had the goal to replicate previous work, so they were limited to details that have been discussed before. For example, executed commands (e.g., [156]), active periods (e.g., [148]), or navigation behavior (e.g., [34, 230]). Over time, we have gradually extended the meta model though with activities that we commonly saw in our user base or that enable the analysis of interesting research questions. For example, our own research agenda was concerned with RSSE, so we added a specific event that can be used to represent several details about the usage of code completion by a developer. Others have been studying testing behavior of developers [13] or about debugging activities [23]. Finally, to cater for the information-need that has already been identified in related work [148] and to support future work, we added additional events for general activities that do not directly correlate with a development activity, like scrolling, moving the mouse pointer, or locking the screen.

In the following, we will introduce all supported activities. We will leave out some technical event types, e.g., crash events that help us to debug crashes in our instrumentation logic after-the-fact, because they are irrelevant for the conceptual discussion. An exhaustive list is available on the project website.[11]

3.2.1 Activity

The core question that is answered with interaction data preserved in the IDE is "what was the developer doing?". At the highest level, this question can be answered by listing all tools and command that were executed. However, unsupported by the traditional system log of executed commands, some "non-activities" do not leave a trace, because they either do not correlate to a development activity or because they do not involve user interaction at all. We capture the following events that are related to general developer interactions and to the aforementioned "non-activities".

Commands All modern IDEs implement the Command pattern to decouple UI design from the execution of the business logic. We are interested in all Command Ids that are invoked, which are easy to intercept by registering listeners with the IDE. Sometimes, it might be the case that simple buttons that are clicked in side-activities, are not registered as a command though. For these cases, we are also interested in capturing clicks on arbitrary buttons in the UI as a fallback, for which we will capture the caption as the command id. Roehm et al. [208] have implemented a similar handling by tracking UI events in addition to command ids.

(In-)Activity Interactions of a developer do not necessarily correlate with commands. For example, a developer could scroll in a file or read source code. Even though the developer interacts with the mouse or the keyboard, no commands are executed. Previous work pointed out that it is helpful to capture this kind of information to distinguish short breaks from real interruptions [4, 148]. The event does not contain any additional information besides the basic context. The event type serves as a marker for activity of developers that points to periods in which they have been interacting with the IDE, which might not necessarily result in other events.

Focus An interaction tracker that is installed in the IDE is mostly restricted to capturing information that is available in this context. Accessing information outside of the IDE is typically very hard. Unfortunately, a developer spends a significant amount of time working with other development tools outside of the IDE.

To detect context switches to other applications away from the IDE, we capture the information when the IDE gains or looses the window focus. With this information, we can safely distinguish in-activities in the IDE from periods in which the IDE is open, but not used. In combination with the (In-) Activity events, it is reliably possible to distinguish cases, in which the developer was idling in the IDE while thinking about a problem, or switching to another task.

Technically, the focus events are represented as *Window Events* that will be discussed in the later section regarding navigation.

System Events Sometimes, it is not easy to decide if a developer actively uses the computer even though the IDE has the focus. Depending on the operating system of the developer, it might be possible that the IDE has the focus even though the computer is unused. For example, if the screen is locked under Windows, the active application does not loose focus. Other examples are power saving modes, in which the computer is effectively turned off, or remote desktop connections to a network machine that continues running the IDE even after the developer disconnected. The captured activity events alone do not provide any indication that allows identifying these cases.

To improve this situation, we introduce system events that describe changes of the system status. We capture screen saver activations, screen locks, changes to the power mode and remote connections.

The combination of the four different kinds of activity events already provides a very good overview over the actions that happen in the IDE. While these events do not provide any detailed context information that tell about concrete tasks of a developer, they already allow to reason about the used tools and different kinds of activity in the IDE. This information allows to replicate existing research. For example, it is possible to identify the used tools (e.g., [156]) or the time budget of developers (e.g. [148]). In addition, the information builds the foundation for analyzing the development process. For instance, it is possible to identify periods in which a developer changed files, these results could be a first preprocessing step that reduces the data for deeper analyses, which analyze how developers typically edit.

3.2.2 File Management

Working on software naturally includes editing source code files and these sources of a component typically span over multiple files. It also involves managing the file and folder hierarchy, as different components of a system are organized in several projects that depend on each other. These projects are usually bundled together on a higher level to build a product, this concept is called a *solution.* While this terminology is tied to the C# programming language and the file organization in VISUAL STUDIO, the concept applies to other IDEs too. For instance, ECLIPSE defines a workspace as a collection of existing projects.

Solutions On the highest level of file management, we capture actions that relate to solutions. These are infrastructural tasks that are not directly related to programming, i.e., removing a file, which provide additional context about the current task. We capture two kinds of information that describe how the developer modified the infrastructure of the project.

Target For each event, we store the IDE component that was selected for the action. As these kinds of actions are typically invoked in a browser-like window (VISUAL STUDIO: Solution Explorer, ECLIPSE: Package explorer), the typical targets are either the *solution* itself, one of its items (i.e., *projects* or other children), or files that belong to a project.

Type The type describes the concrete action that was invoked. We support changes to the solution (i.e., open, close, rename), changes to items or projects in a solution (i.e., add, delete), as well as changes to items in a project (i.e., add, delete)

We expect that solution information is valuable context information that provides a profile about the developer session. This information can be useful for classifying the task a developer is working on. For example, it is possible to identify the creation of files and -therefore- deduce when new source code is being developed in contrast to maintenance tasks, for which existing files are edited.

Edits Writing software boils down to editing source code files and the enriched event stream preserves information about the different edits performed by the developer. Changes are typically very small, so we designed the event in a way to capture an accumulated view of changes and information about their extent. Events contain the following information.

Number of Changes Counter that shows how many change operations in the same file have been accumulated before creating this event.

Size of Changes Measurement of the extent of the change. We preserve the information how many characters have been changed in total.

Snapshot If technically possible, we take a snapshot of the current file after the change.

A crucial information for edit events is the file name of the file that is being edited. This information is already being captured in the basic information that is stored for every event and does not need to be stored again.

Edit events provide valuable context information that help to identify periods in which the developer was actively working on source code. It is possible to distinguish phases in which the developer was reading and understanding code from periods in which code was being produced. Any intermediate snapshots that are taken as part of these events are valuable for research on source code evolution. These snapshots create a very fine-grained history of source code that is not available in traditional version control systems, in which the change granularity is a lot coarser grained. While Roehm et al. [208] monitor users of a diagram tool, they follow the same approach and track edits and changes to the diagram.

Overall, having all the data about the file management provides a very fine-grained picture of the navigation in the project structure and its maintenance. This data is applicable in many research areas, including finding navigation traces [41, 115], detecting related files [225], predicting co-changes [265], or source code evolution [15].

3.2.3 Navigation

Developers navigate in their IDE all the time to find information. Research distinguishes two kinds of navigation (as defined by Singer et al. [225]): unstructured *browsing* (e.g., when a developer does not know what to look for and tries to find it by navigating to related documents that are semantically linked) or directed *searching* (e.g., when a developer knows what to look for, but not where). We designed enriched event streams to capture enough information to enable reasoning about both kinds.

The captured command events already indicate which tools are used in-IDE, including the ones that are concerned with navigation, and having the trace of opened files in document events allows to reason about the visited documents. In addition to this, we capture more specific kinds of navigation related events. The actual invocations of these tools are described through detailed context information.

Structural Navigation We capture *navigation events* to represent navigation in the structure of the type system. We capture both navigation within a document (e.g., moving the cursor to another method) and *ctrl-click* navigation that is regularly used by the developer to navigate to referenced code elements (e.g., method calls).

The following information is captured to describe concrete usages:

Type We capture the kind of the interaction that took place. We distinguish keyboard navigation (i.e., arrow keys), location indications through mouse interactions (e.g., click on a variable), and structural navigation (i.e., *ctrl-click*).

Location The location at which the navigation event was created, we store the enclosing construct with a fully-qualified identifier (e.g., the name of the surrounding class or the enclosing method). The interpretation depends on the navigation type: Indicative events (e.g., keyboard moves) report the current location, every time a change is observed. Navigating events (i.e., ctrl-click) preserve the location in the code before the requested navigation takes place.

Target Events that involve a transition (i.e., ctrl-click) also capture the navigation target. For example, a control-click on a method call would store the fully-qualified identifier of the target method as the target.

Information about the structural navigation provides important context for research that requires the current location in which the developer is reading or editing. Storing the enclosing construct creates a more fine-grained trace of related locations than just tracking the open documents. Examples of research that need this information is prior work that tracked the eye movement ([96, 105, 207]).

Search Tool All serious IDEs offer an in-IDE search tool that can be used to find information. The options reach from a simple text-based search in the current document to a semantic search in which searches for specific syntax elements can be expressed. We are interested in capturing the invocation of such search tools. While the invocation of the tool itself is already captured as a command event, we wanted to further distinguish actual invocations of a search. We added an additional mark that defines whether the search was executed or whether it was aborted.

Having search events allows analyzing the search behavior of developers. Previous work has already done this on code search engines, enriched event streams allow to extend this line of research to reasoning about in-IDE searching. While the current representation is limited to basic information, extending the support for this kind of instrumentation seems to be an excellent opportunity for future work for further analyses. To enable this, the events should preserve additional information about the content of the search itself. For instance, the term that was searched or at which of the resulting proposals the developer has looked at. Another opportunity is to support additional search tools. Right now, it is not possible to distinguish searches that happen within a single editor window from global searches that find information in the whole solution.

Overall, navigation events provide a deep insight into how developers navigate or browse in their code base. Code bases continuously grow and supporting developers by providing better tools that support finding the information is key. It is necessary that we further analyze how developer search for required information.

Navigation events in the enriched event streams help both in the sense of being a source that can be analyzed to understand the process and as a real representation of actual navigations that can be used to validate new tools in this area. They capture a fine-grained and detailed trace of visited locations and distinguish *manual* navigation and tool-supported navigation (e.g., ctrl+click) and, in combination with document events, a fine-grained and detailed trace of visited locations can be build. Further extended by more detailed information about search tools, this data represents an excellent opportunity for future work.

3.2.4 Environment

To round up the picture of the general usage of the IDE, we were interested in capturing information about IDE sessions and about the environment in which the developer is working. More specifically, we wanted to preserve information about the state of the IDE (e.g., when it is being (re-)started) and the configuration of the local working environment. This includes preserving interactions with different kinds of windows and their configurations, as well as the navigations between documents.

IDE State Events We introduced IDE state events as a means to describe the lifetime of the IDE. While the start and shutdown of the IDE could be approximated from an event stream by the absence of command execution, a restart or a crash of the IDE is not easy to detect algorithmically. We figured that this information could be interesting when reasoning about the event stream, so we decided to capture starts and shutdowns in the event stream.

Phase Indicates when the state was recorded. We distinguish *startup*, *shutdown*, and *during runtime*, even though the latter is not used so far.

Windows We capture a list of identifiers of all open windows. The identifiers include both the window caption and the window type (e.g., editor or tool window).

Documents We capture a list of all documents that were open at the time at which the event was created. The document are identified by their file-system path, relative to the solution file.

Marking (re-)starts and shutdowns of the IDE and the corresponding context information enables to track changes in the environment. Roehm et al. [208] follow a similar approach and track restarts of the application that was instrumented by them.

Knowing about restarts also helps to keep a consistent view on the environment. For example, to correctly keep track of all open windows and documents. Since some windows and documents are not explicitly opened or closed by the developer, but are opened automatically on startup or not closed before shut down, it is helpful if these automated window and document events are also included in the event stream.

Window Events The complexity of modern software systems has supported the rise of integrated development environment. An analysis of the trends in IDE usage reveals that most developers use window-based IDEs.[45] We wanted to track how developers interact with windows, to capture how they navigate between files and which tools they use in their daily work. We were also interested in capturing information about how they interact and place their windows. We store the following information.

Window On every interaction with a window, we store an identifier that includes the window caption and the window type of the window that has been interacted with.

Action We distinguish several kinds of interactions with the window. We support *create*, *activate* (focus gain), *move*, *close*, and *deactivate* (focus loss).

Window events provide many interesting insights into how developers use their IDE. First of all, having the focus gain of editor windows allows to restore a fine-grained trace of the files that have been visited by the developer. In addition to this, the other information allows to analyze the time developers spend with positioning and resizing their windows. With this information, we can replicate prior work that identified the amount of time a developer spends configuring the IDE [148].

Documents We track which documents the developer has been working with and collect information about how files have been navigated. For instance, which files have been opened for reading or which ones were touched. *Document events* provide the runtime information about changes to the list of open documents.

Filename We store the name of the file that has been interacted with, relative to the containing solution.

Action Type Kind of interaction that happened. We distinguish *open*, *close*, and *save*.

Having this information provides additional context about the navigation in the code base. We can deduce, which files were involved in a task, which ones were intensively studied or directly closed. The save events also represent a kind of micro versioning information as we assume that a "save" marks intermediate versions that are more complete and more stable than the versions in between. Spacco et al. [231] report that in their dataset, after a save, source code compiles 80% of the time.

In summary, the information about the IDE and its environmental changes provide a consistent view on the way the IDE is used by the developer, in terms of windows and open files. It also provides information to reason about these, e.g., when windows are relocated or documents are saved.

With these information, it is possible to identify phases in which the developer spent the time with configuration (or "UI fiddling" as coined by [148]) as those environmental events could support these analysis as they can be used as an indicator that developers were not working. In addition, is it possible to get a consistent view on the interactions with files, which could also serve as additional events for tracking the *navigation* in the IDE, complementary to navigation events.

3.2.5 Specific Tool Usage

Software development is a combination of various activities. One of the strengths of enriched event streams is that they do not stop at the general level, but that they are extensible to very low-level details. While the general activities capture this wide view on the process, we identified essential activities that are at the core of software engineering and added support for them to enriched event streams. This allows capturing more than just the generic context information for these activities.

These essential development activities can be captured by tracking tools that are (or should be) used by everybody and capture an extended context that provides concrete information about an execution of the tool. For instance, in addition to the fact in the general stream that a build was triggered, the event stream will capture in a more specialized event which projects have been built, how log it took and the build result. The corresponding events are being introduced in the following.

Build Before software can be run, debugged, or tested, it is necessary to build it. Depending on the programming language and the IDE that is used for coding, this can happen incrementally in the background while working on the source code. In our case, we are targeting C# and VISUAL STUDIO, in which building is an activity that has to be triggered regularly by the developer. We store the following information in *build events* to preserve these actions.

Action The build action that was triggered (e.g., clean or build).

Scope Distinguishes the element that is build (e.g., *on solution* or *on project*).

Targets A list of targets that are build. Depending on the scope, the list of targets includes all nested targets (solution) or all selected targets as well as their dependencies (project).

All general information of the event relates to the accumulated build. So for example, the duration of the whole event refers to the total build time. However, we break this down to each target and also provide the following individual information.

Name Name of the target that was build, usually the name of a contained project.

Configuration Configuration options that affected this build. We support to store options that are specific for the solution, options set for the project, and information about the platform for which the target is being build. The configuration fields are optional and can also be used to store arbitrary information that can be represented in a string.

Timing We capture timing information for each build target and store individual start times and durations.

Result We also store information about the result of the built. We only distinguish successful builds from unsuccessful ones and do not store additional information beyond this.

We designed build events to be independent of the build tool that is used. While the terminology follows the internal build system of VISUAL STUDIO, the captured information is general enough to be applicable to other build systems or programming languages (e.g., MAVEN with JAVA).

Code Completions One of the core features of modern IDEs that is widely used and that goes beyond simple editing helpers like copy and paste is the code completion feature. Prior work has already shown that this is one of the most important tools in modern IDEs, a finding that we could replicate in our own research (see Section 9.2).

The developer can trigger code completion manually, but sometimes it is also automatically triggered by the IDE. Code completion events are the most complex events in our event stream, because we capture a lot of context information. Every time it pops up, we capture the following information.

Proposals We capture the list of the proposals offered by VISUAL STUDIO. As this list can be multiple thousand entries long in some cases, we decided to only capture the first 300 proposals.

Selections We capture the interactions of the developer with the proposed items and store which item have been selected and for how long.

Context The events also contain a snapshot of the surrounding source code that includes a marker for the trigger location at which code completion was invoked.

Termination Multiple ways exist to end the invocation of a code completion. We distinguish *application*, *cancellation*, or *filtering* (refinement of the typed token).

An important information for code completion is the current location in the editor at which it was invoked. We encode the location as part of our intermediate representation of the context snapshot. Following this line, all information about proposals and selections are stored in a fully-qualified form that preserves information from the type system. Details about our representation for code elements and the syntax for our intermediate representation for the source code will be discussed in Chapter 4.

There are several opportunities to use code completion events in research. Examples include using them as ground truth in evaluations of recommender systems [190] or studying source code evolution that is based on the fine-grained snapshots [167].

Debugger Debugging programs is one of the most common activities of a developer. We wanted to preserve information about how developers debug.

Mode Mode of the runtime at the creation time of this event. We distinguish *run* (regular running program), *design* (coding "perspective" of VISUAL STUDIO), *break* (halted debugger), and several modes that relate to exceptions during debugging.

Reason Defines why the event was created. For instance, when the program execution hits a breakpoint, or when the developer pressed the *step next* command.

Action This optional field can be used to store more information about the action that was executed when the event was created, usually the default action that is executed when a breakpoint is hit.

While there is some information that is not covered (e.g., addition of breakpoints), we can already recover many information about how developers debug. For instance, how often they debug, how long debug sessions last, how code is navigated in debugging sessions (e.g., usages of step over or step out). Together with the window events and navigation events, it is also possible to reason about how the call stack window is used to jump within the execution stack. We also can recover when they switched to other debugging related windows like the inspector for local variables or the immediate window, we would miss this information if they only look at it though without activation of the window. These events provide the required information to analyze the debugging behavior of developers (e.g. [14, 23]).

Testing Not all developers test in a structured way, but writing unit tests is nevertheless considered good practice and essential, especially in larger projects. We were interested in capturing information about this activity. Testing is interwoven with other development activities. For instance, snapshots of the testing code are already being captured in code completion events. An important information that is missing, however, is how developers execute their test and react to the results. We wanted to capture information about these test runs.

Whenever a test run is started, we are collecting all test methods that are being run. Parameterized tests create multiple entries, one for each parameter combination passed to the test method. We store whether the test run completed successfully or whether it was aborted and store the following information for all test cases individually in *test run events.*

Method Fully-qualified identifier of the test method.

Parameters This optional field can be used to preserve the concrete parameters of a test run that are being passed to a test method.

Timing We store timing information for each test case individually and capture both the start time of the test and the duration of the execution.

Result We capture the result for each test case. The terminology of the results slightly differs between various testing frameworks, but the semantic boils down to the states of JUnit.[23] We distinguish *Success*, *Error*, *Failure*, and *Ignore.*

With the data about test execution, we can reason about how developers test. Previous work in this area analyzed how often developers test and how they react to failures and we can replicate this line of research with the data. As our data contains more holistic view on the development, it is possible to go one step further and connect the testing events with an additional analysis of follow-up events, e.g., the following edit events, to get a more detailed view.

Version Control Controlling the version of the source code is key for teamwork, so developers commit their changes to repositories hosted on services like GitHub or Bitbucket. As analyzing the commits is the typical granularity at which researcher are analyzing source code evolution, we also wanted to capture this information in the enriched event stream.

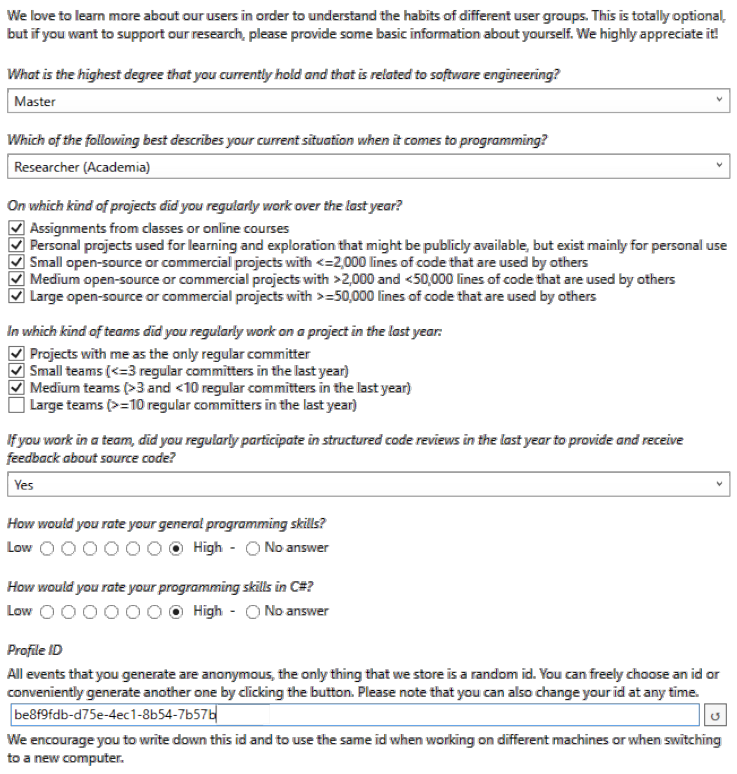
Several VCS exist nowadays that are commonly used among developers. The most popular system is Git, so we picked its terminology for our events. However, given the abstract nature of the data we capture, it is easy to support other features by mapping their commands to these concepts.

We were not interested in capturing every detail in the first step and restricted the design of version control events to preserving commands that are related to a solution. This design is based on the assumption that only one solution is checked into a repository. This is clearly an approximation and changes could be committed that do not relate to the solution. However, we think that this case would be easy to detect as we would see subsequent commits without seeing any changes in the IDE.

Each time, an action is detected, we identify the name of the solution in question and, in addition, we capture all VCS actions that have happened in the corresponding repository since the last event was fired. We capture the *Version Control Action* that has been performed. We use the corresponding Git commands to qualify the action (e.g., add, commit, push). We also store the *timing* information that defines when the action was invoked.

Tools for version control are integrated in the IDE, but nevertheless, they are a prime example for tools that are very often being used outside of the IDE, namely in a terminal or in external tools. Therefore, to capture information about version control, any tracking implementation should also capture information from outside the IDE. This also means that events can happen while the IDE is shut down, so our design allows capturing multiple actions in one event.

Once a dataset of version control events is collected it is very useful for research on code evolution or if subsequent versions of source code are being distinguished.


We love to learn more about our users in order to understand the habits of different user groups. This is totally optional, but if you want to support our research, please provide some basic information about yourself. We highly appreciate it!

What is the highest degree that you currently hold and that is related to software engineering?

Master

Which of the following best describes your current situation when it comes to programming?

Researcher (Academia)

On which kind of projects did you regularly work over the last year?

- ✓ Assignments from classes or online courses
- ✓ Personal projects used for learning and exploration that might be publicly available, but exist mainly for personal use
- ✓ Small open-source or commercial projects with <=2,000 lines of code that are used by others
- ✓ Medium open-source or commercial projects with >2,000 and <50,000 lines of code that are used by others
- ✓ Large open-source or commercial projects with >=50,000 lines of code that are used by others

In which kind of teams did you regularly work on a project in the last year:

- ✓ Projects with me as the only regular committer
- ✓ Small teams (<=3 regular committers in the last year)
- ✓ Medium teams (>3 and <10 regular committers in the last year)
- ☐ Large teams (>=10 regular committers in the last year)

If you work in a team, did you regularly participate in structured code reviews in the last year to provide and receive feedback about source code?

Yes

How would you rate your general programming skills?

Low ○ ○ ○ ○ ○ ○ ● High - ○ No answer

How would you rate your programming skills in C#?

Low ○ ○ ○ ○ ○ ○ ● High - ○ No answer

Profile ID

All events that you generate are anonymous, the only thing that we store is a random id. You can freely choose an id or conveniently generate another one by clicking the button. Please note that you can also change your id at any time.

be8f9fdb-d75e-4ec1-8b54-7b57b

We encourage you to write down this id and to use the same id when working on different machines or when switching to a new computer.

Figure 3.4: Questionnaire: User Profile

Our design captures various change granularities, when combining Version control events with edit events, code completion events, and document events. Edit events and code completion events regularly provide snapshots, leading to a very fine-grained history of a source code file. Document events report save actions from the developer that regularly mark stable intermediate versions. These versions can be seen as the aggregation of many intermediate edits. Version control events mark a much more coarse-grained granularity of source code. They are traditionally used in research as they are easily available in public repositories, so it makes sense to include them in the versioning too.

3.2.6 General Purpose

We created some general purpose events to capture information that is not relevant for the activity stream, but interesting for an analysis later on.

User Profile Events All events are captured anonymously, to preserve some information about the subject that shared the data with us. Primarily, we wanted to capture information about the education and skill level of the participant. However, we wanted to be able to identify experienced programmers, as we expect that the collected events from experienced developers are of a higher quality than events from other groups. Therefore, we also included information about recent projects and team experience.

We designed a questionnaire that contains several questions about the background of the subject, a screenshot of the questionnaire is shown in Figure 3.4. We store the answers in a special user profile event that contains the following information.

Education The education level of the participant. We ask for the highest computer-science degree that has been achieved so far. We distinguish *None*, *Autodidact*, *Training*, *Bachelor*, *Master*, and *PhD.*

Position The current position in which the participant is mainly developing software. We distinguish *Hobby Programmer*, *Student*, *Researcher (Academic)*, *Researcher (Industry)*, and *Professional Software Engineer.*

Skill To get an idea about the programming skills of the participant, we ask for a self-estimation of the programming skills. We ask for programming skills in `C#` and for general programming skills and allow answers in a 7 point Likert scale.

Project Experience We ask about the kinds of projects the developer has been working on. We distinguish several project types that range from *projects as part of a course* to *large projects.* The descriptions distinguish some projects types that are *used by others*, as we expect them to have a higher quality.

Team Experience We also ask about the kinds of teams the developer has been working in. We distinguish different sizes of projects by the number of committers over the last year. We also ask about whether the participant regularly participates in structured code reviews.

Profile While being optional, we asked our users to provide a profile id that allows us to identify all events that have been uploaded by them and merge them together.

Asking the users to answer these questions serves multiple purposes. First and foremost, they provide us with demographic information about our users. We can make sure that the collected data is representative for a wide range of users and backgrounds. Given a dataset that is large enough, it also becomes possible to distinguish different user groups. Second, we tried to capture information that provide insight in team experience Lastly, the user profiles are created when sharing the data so they also serve as a proxy that allows to reconstruct the date (and therefore the frequency) of the upload.

Providing these information is completely optional, even the user id can be chosen freely or can be completely omitted. More details about the related discussion of privacy are provided later in Section 6.2.2.

Overall, the events in the *general-purpose* category are not pointing to actual behavior of the developer, but they are important for the analysis of the event stream. While *User Profiles Events* provide important information for any researcher that analyses a captured event stream, the *Info Events* and *Error Events* provide debugging information about the integration and instrumentation of an interaction tracker.

3.3 Extending the Meta Model

We identified early on that *Extensibility* is one of the requirements for the design of the meta model. We made it to one of the core principles of the design of enriched event streams. We support different extension types and will elaborate them in the following: through *creating* new events, through *adding* new information to existing events, through *changing* existing events, or through *revising* the tooling that captures the event stream.

Creation The easiest way of extending the event stream is creating a new event type to capture new data. The addition should comply with the requirements defined in section 3.1.1, but the only strict requirement to make it compatible to the rest of the platform is to extend the base class `IDEEvent`. Clients are not required to process all events, so simply adding a new type does not break any client.

Addition Another similarly easy way of extension is to add new information to existing event types, e.g., by introduce new fields in the data structure or by adding new cases in an enumeration. An addition will not break the source code of existing clients, but it imposes the responsibility onto clients to distinguish the original events from the extended ones, if the added data changes the semantics of the captured events. For example, assume that an event would store the name of successful refactoring invocations. If an extension enables the distinction of successful invocations from (formerly untracked) aborted invocations by adding a new field, it is required that all existing clients are extended to handle the different cases appropriately. Their code will still compile, but they have to filter for the successful invocations now.

Change A more complex option to extend the data structure and capture additional data is to change an existing event type. For instance, by changing a boolean flag in an event that marks successful execution to an enumeration to distinguish more than two cases. Supporting this is more complex and involves two steps. First, it is advised that the change is provided together with an upgrade functionality that can be used to read data that was captured before -in the old form- and transform it into the new form, to avoid a data loss. Second, when a change breaks existing clients, it should be considered if the old interface could be preserved with a transformation function that provides the old functionality with the new data. This compatibility layer can be marked as deprecated, but it helps to upgrade existing clients.

Revision Extensions of the interaction tracker that change or revise existing instrumentations are an example of extensions to the event stream that do not introduce changes to the meta model. For instance, a researcher could add support for formerly unhandled cases, instrument new tools that are can be represented in the existing events, or simply fix a bug that invalidates previously captured events. These changes are invisible from the perspective of the event stream. However, it might be important to distinguish such cases or, in case of bugs, exclude events generated from the original instrumentation. Therefore, it is of utmost importance that extensions are versioned to make it easy to identify affected events.

To allow this, we introduced a versioning scheme that uses both a version number (e.g., `1.2.3`) to specify the major, minor, and build versions, and a qualifier that can be used to mark extensions that deviate from the default implementation. For instance, our reference implementation creates version numbers in the form `1.2.3-Default`. Should any researcher fork our implementation of the interaction tracker, we would suggest to adapt the qualifier accordingly (e.g., `1.2.3-NewX`) until the extension is integrated back into the main development branch.

Overall, it is hard to design a data structure that is both very extensible and stable at the same time. It is even harder to guarantee that extensions do not break existing clients. Our design presents a middle-ground between these factors. We make the data structures easy to extend in a way that allows a direct integration into the existing pipeline. On the other hand, we place the burden on the researcher to make sure that extensions are backwards compatible, because this is hard to check automatically. As the last resort, we added a versioning scheme to events that make it easier to filter for events that are compatible to a specific client.

3.4 Limitations

We designed enriched event streams with three applications in mind that could benefit from such fine-grained information about development activities. Even though we have reviewed a wide range of related work, it might be that we missed to identify crucial requirements or did not foresee future requirements that would break our design. In addition to these general threats, we are aware of the following concrete imitations of the current state of our meta model.

The current version of the meta model can only be used to represent in-IDE activities, it does not provide the required entities to draw a holistic picture of the development activities. This is the case for other activities on the same machine (e.g., switches to other applications) or activities that are conducted away from the computer (e.g., phone call). While said activities cannot be easily tracked automatically, the data could be manually added to the event stream after the fact, e.g., by entering the results of an observational study to the automatically captured data. Our design is extensible and additional classes could be added to the meta model.

The meta model does not model the developer and the corresponding roles in the current development task that is tracked. We have introduced user profiles as a first step to request at least basic information from our users, like education or work experience. However, it is not possible yet to model the role of the user in the current solution (e.g., developer, tester, integrator) or in the current session (e.g., normally project lead, but every once in a while also developer of a feature). Future work should analyze, how the registration of accounts and solutions that is done in related work [13] could also be integrated into our solution.

Right now, the meta model can used to store observed activities. However, all captured data is “unlabeled” so far, i.e., it is not possible to request user feedback and include this in the event stream to explain the captured activities. Several related works have introduced techniques to regularly interact with their users to request information (e.g., [210, 266]), which can then be used to characterize the recent activities, e.g., perceived productivity, whether they feel stuck, their current goal or role in their current task. Future work should include user feedback in the meta model.

The current representation for version control activities is limited to the most basic information, i.e., *when* has *which command* been invoked. This abstracts from many details and is based on very optimistic assumptions, e.g., that a `git add` always includes all open changes. However, modern version control systems like GIT provide powerful mechanism to control the versioning of changes and they are typically line-based, which allows adding and commiting changes on a very fine level. As a result, the optimistic assumptions do rarely hold and the lost details might be interesting for works on source-code evolution. Future work should provide additional ways to represent version control activities like storing a repository url, capturing affected files or lines, or preserving the if or the message of a given commit.

The meta model already supports debugging activities, but it can only represent high-level events in the work flow (e.g., debugging started or exception occurred) or the invocations of debugging related commands that are issued by the developer. Future work should identify the most important activities of debugging sessions and extend the meta model accordingly, e.g., represent the location of breakpoints, the usage of variable inspections, or the navigator that can be used to directly jump to specific locations in the stack of the current program execution.

Overall, most of the described limitations originate in the nature of the data collection and the data that is to be represented. While it is certainly tempting to preserve as much information as possible in the meta model, it is required to introduce abstractions, because it is not feasible to preserve each and every details. Capturing too many details increases the event stream unnecessarily, which would makes a collection in a distributed setting infeasible. In addition, it might make possible users skeptical about their privacy. The right abstraction level will always be a tradeoff.

Chapter Summary

This chapter has introduced our meta model for in-IDE developer activities, *enriched event streams*. The individual interactions are modeled as events in a stream, which are enriched with additional context information. Capturing such detailed data about the interactions bears great potential and allows applications to more than just observational studies. The captured data is valuable for research on a much broader level and could help analyze the effects of many tools that involve developers.

We have illustrated in detail which requirements and design decision lead to the current design (Section 3.1). After defining the meta model on the conceptual level, we have elaborated on our current selection of activities that can be represented in the meta model and we have described the context information that we preserve for them (Section 3.2). Extensibility of the meta model is one of our core requirements to make the meta model future-proof and we have introduced which extension types we support (Section 3.3). Finally, we have described limitations of the current state and we have sketched how they could be mitigated in future work (Section 3.4).

4 Meta Model for (Evolving) Source Code

Developers spend their time on a wide variety of tasks, but the most important artifact is source code. A study on the development process must also consider the development of source code on a very fine-grained granularity. The literature review in Section 2.1.3 has shown that several works have studied source code evolution. Our requirements are different to the traditional studies on source code evolution though, because we want to cater for several use cases at the same time. While existing work focuses for example on differencing or merging of changes, we are especially interested in *application programming interfaces* (API). A complete research area is dedicated to creating recommendation systems for API usage, however, analyzing the evolution of APIs and the evolution of their usage has rarely been the focus of research so far. Furthermore, we do not only want to study source code evolution, we also want to use the captured data in the area of RSSE both as input for pattern mining and in more realistic evaluations. As a result, we need to capture more than just the changed lines or syntactical changes of fine-grained source-code changes in our enriched event stream, we also need to preserve typing information. Our survey of existing intermediate representations in Section 2.2.3 did not find one that satisfies our requirements so we had to design a more appropriate one ourselves.

In this chapter, we will introduce a meta model for object-oriented code that captures incremental, but self-contained snapshots of in-IDE source-code changes. It can also be used for source code found in a repository. We will start in Section 4.1 by presenting our requirements and by elaborating the design goals and considerations that drove the work. The resulting design consists of three parts. Section 4.2 will introduce a new naming scheme that can be used to address code-elements in. We will discuss our strategy to preserve relevant typing information in Section 4.3. Afterwards, we will present an intermediate representation for source code, simplified syntax trees, in Section 4.4. The limitations of our design will be subject of Section 4.5.

4.1 Designing a Meta Model

We needed a new meta model for object-oriented source code that we could use to capture snapshots in the enriched event stream to preserve a fine-grained development history. At the same time, it should be applicable to represent source code found in repositories. In addition to enabling studies on source-code evolution, the meta-model should allow static analyses and should support common analysis tasks. This section will present the design process for our meta model. We will formulate the requirements and introduce our design goals and considerations that drove our design. At the end, we will present an overview over the resulting high-level design.

4.1.1 Requirements

We want to use our meta model for studies on source-code evolution and to build recommendation systems for software engineering. Our survey of related work has revealed several requirements that have to be addressed by the model to make it applicable for these lines of research and for our specific use case.

Syntax Trees Most advanced approaches work on abstract syntax trees (e.g., [66]). The representation must be parseable to a syntax tree that resembles the structure of the original source code. We can relax the requirements though and don't expect an identical tree, e.g., through dropping rarely used node types like modifiers.

Fully-qualified References For many techniques, fully-qualified type references are crucial (e.g., [2, 22, 87, 92, 127, 135, 143, 144, 152, 154, 192, 262, 263]). While some approaches do not fully-qualify types (e.g., [87, 169, 171, 240]) or only use approximations through minimalistic naming schemes (e.g., [5, 91, 213, 240]), we believe that their underlying techniques could be improved with such information. Other approaches do not require fully-qualified types in their core algorithm, but they need it in their IDE integration, for example to match proposals, and, therefore, also for an automated evaluation. A general meta model must reference types and type elements in a fully-qualified way that allows to unambiguously refer to them.

Type System Using these fully-qualified references, it must be possible to look up details in the type system (e.g., [2, 22, 90, 117, 135, 143, 144, 152, 192, 262, 263]). We are mostly interested in API types, so this requirement can be relaxed for types that are declared in the current project.

Support Static Analysis Several approaches apply advanced static analyses to extract information from the source code. In our survey, we saw approaches using points-to analyses (e.g., [195]), others apply tracking analyses and follow the control flow into invoked methods (e.g., [127])), and a few approaches use *slicing* to extract relevant parts of code snippets (e.g., [154]). The source code representation must contain all information that is required for sophisticated control-flow or data-flow analyses and that its implementation provides appropriate means of traversing the tree (e.g., through a visitor). On a structural level, the aforementioned typing requirements are crucial for this, but it is also required to make sure that all source-code constructs are being captured that are required for these analyses. More specifically, all constructs that relate to control flow (e.g., `if`) or data flow (e.g., *assignment*). In addition, some approaches also require *casts* when analyzing how instances of certain types can be obtained (e.g., [127, 154, 240, 262]).

Granularity Existing approaches typically rely on the commit history and assume full access to the repository to recover missing information. In contrast, enriched event streams will contain fine-grained development snapshots of edited types, which only provide an incremental view on the development process and no access to the edited code base. Moreover, developer might even delete individual events, which makes the history inconsistent. As a result, the meta model must support taking self-contained snapshots that can be analyzed individually.

Scalability Many snapshots have to be taken for enriched event streams. We plan to capture them in field studies from large numbers of developers. This use case prevents us from simply capturing all required information about the development context in code snapshot each time a change occurs, because this would include all sources, dependencies, and configuration files of the project. The captured data has to be uploaded by the participants, which makes it necessary to capture incremental snapshots to keep the amount of data small.

Embrace the IDE Snapshots should not only focus on the source code, but should leverage any available in-IDE information that it expensive to recover later. For example, all types can typically be resolved when source code is edited, because all dependencies are fully set up, but this state is very hard do achieve after the fact. Information like this should be preserved in the snapshot instead of relying on a heuristic resolution later. Other examples of such information are the edit location in the source file or invocations of refactoring tools.

Align Snapshots While *enriched event streams* contain many information about the software development process. To get a holistic view on source-code changes, they have to be aligned with tool interactions in the IDE. For example, consider the case in which subsequent snapshots indicate a variable renaming. To study the evolution, it is interesting to distinguish cases in which the developer used a refactoring tool from cases in which the renaming was done manually. *Enriched event streams* contain this information, but it would be very useful to know the location in the source code in which the tool was invoked, a piece of information that is readily available in-IDE. Our meta model should go beyond the syntactical content of a source file and also include information that makes it easy to align source code and the process information, for example, by adding anchor elements to the source code representation (e.g., edit location) that allow to align the process information.

Some of the surveyed approaches have requirements that go beyond the meta model and that concern the development process. An example of this is partitioning changes into coherent sets of related source-code changes. In our design, such requirements are solved in the *enriched event stream* and do not impose further requirements on the source-code model. The source-code model is supposed to focus on the AST level of a program and present a consistent unit for analysis.

Analysis Challenges and Future Requirements We identified several recurring analysis tasks and challenges that could be simplified by an appropriate design of the meta model. In addition, we also anticipate requirements of future work. On top of the basic requirements for the representation, we identified the following extended requirements for which IR support would be nice to have.

Entrypoints The approaches work on varying scopes to perform their analyses. While some approaches analyze complete classes (e.g., [87, 137, 143, 144]) and others even perform inter-class analyses (e.g., [127, 154]), most approaches work on the method level (e.g., [90, 169]). The structural context is also used by some approaches to identify non project-specific parts of an API that can be reused (e.g., [262]). The meta model should differentiate method declarations that are reusable by others, e.g., public or protected methods, from project-specific helper methods.

Complex expressions Static analyses that consider control or data flow need to resolve the evaluation order of sub-expressions, such as nested calls `m1(m2())` or chained calls `o.m1().m2()`. Untangling these complex expressions is often intertwined with a specific analysis task (e.g., [213, 261]), which increases complexity and decreases maintainability of the implementation. A meta model that only stores normalized source code would avoid this problem by design.

Frameworks A widely unexplored dimension in studies on both source code evolution and RSSE is the API version of the considered types. APIs change, often in a non-compatible way, or they get exchanged with an alternative better suited for the task. A version information is required to distinguish these cases. Modern provisioning frameworks such as NuGet, Maven, or P2 make it easy to extract the respective name and version of dependencies. However, simply storing the project dependencies (e.g., in a VCS) does not help in understanding the impact of different versions for recommending the right code changes to the developer. While the change of project dependencies suggests a library migration, it does not indicate anything about the required code changes. We argue that library references should become a part of the typing information to make their relationship explicit.

Generics Generic type parameters are supported by Java and C#, yet their evolution in source code has not been studied so far. Generic type information should be part of the meta model to pave the way for this kind of research.

This wraps up the requirements that we have identified in our literature review. Please note that no single client required all of them, they represent the union over all clients.

Revisiting Existing Solutions Two existing intermediate representations fulfill several requirements. The FAMIX meta model was designed for language independent refactorings. M3 is used as the intermediate representation by the meta-programming language Rascal. Both approaches support fully-qualified type names and provide means to lookup references to types and type names. Both provide a lightweight approach to store single files, so they satisfy the scalability requirement. Another strong advantage of both intermediate representations is that they come with a complete platform that provides reusable static analyses and refactorings.

Unfortunately, both representations also miss out on some requirements. Both intermediate representations store the information in a tree-based structure. However, only the M3 approach stays close to source code, the FAMIX model stores an incomplete tree that only contains program entities like variable declarations and invocations. None of them encodes information that goes beyond source structure, e.g., code anchors. In addition, neither approach fulfills any of the extended requirements.

4.1.2 Design Goals and Decisions

Our survey of related work has revealed several requirements for the meta model, but many details have not been decided yet. In addition, some requirements conflict and require a tradeoff in the design. We will go through our general design goals now and discuss the design decisions that have influenced the final design.

Optimism One of the core principles of our design is optimism. The *traditional* use cases of static analyses are security and program optimization. Static analyses in these fields are always *sound* and *pessimistic* in the sense that every corner case has to be considered. The approaches assume that the project environment is complete and that information from the type system can be looked up on demand. As a result, static analyses in these areas are very precise and they have to, because they are searching the needle in the haystack. The use cases considered in this thesis are different in the sense that all considered approaches and static analyses are *optimistic* in their nature. They involve statistics, try to find common cases instead of corner cases, and usually accept unsoundness, heuristics, and approximations. Most works use statistical means or apply machine learning techniques to generalize from examples, which is unsound by design. While we try to create a representation that is as precise as possible, we always keep the use case in mind and allow the design to be *optimistic.*

Generality The intermediate representation should be able to represent source-code that is being extracted from repositories as well as being captured in development sessions in-IDE. While some information is not available in repositories (e.g., cursor location), analyses in-IDE are confronted with other challenges (e.g., invalid code). A representation must be flexible enough to unify both cases, which makes the creation of tools and analyses easier for developers.

Changes as Snapshots A fundamental question of our design was how to model the change process. The survey of related work has revealed that both incremental updates and snapshot-based approaches are used in practice. While Robbes et al. [199] postulate that changes have to be modeled as first class elements to be of use, Evolizer [66] can successfully reconstruct semantic changes from a VCS history. In addition to the very fine-grained source-code history that is stored in the enriched event stream, we also capture information concerning executed refactoring or indicators for different change granularities and we expect that a fine-grained snapshot history might be as valuable for us as a change-based approach.

The decision influences the analyses of our two main use cases though. An incremental history is more convenient for studies on source-code evolution, because semantic changes can directly be interpreted. A preprocessing is required to identify the same information from snapshots. On the other hand, typical static analyses require complete source code and these snapshots need to be reconstructed from incremental changes. Independently, enriched event streams will be collected in a distributed setting and, for privacy reasons, users will have the option to remove parts of the event stream at their discretion. Loosing parts of an incremental change history is disastrous and leads to the corruption of the whole history of the affected file. This scenario is very likely, so storing the change history in a purely incremental fashion makes it fragile and is a strong threat to consistency. As a result, we decided not to use incremental information and to store consistent snapshots of edited files. While the amount of data that is stored for a single file increases, in some cases significantly, we gain consistency guarantees that are invaluable in such a fragile environment.

Type System It is infeasible to store a snapshot of the whole project environment on each change. Yet, some analyses require information about the source code that goes beyond the current file, e.g., complex slicing approaches that follow the control flow into other classes (e.g., [154, 262]). To solve this dilemma, we will combine the idea of storing consistent snapshots of singles files, with an incremental approach to store information about the project. In addition to the syntax tree that we capture for directly edited local types, the in-IDE tracking also captures a *type shape*, minimal information about their type hierarchy and structure that is required to look up details later. It contains the fully-qualified name of the type and the full transitive type hierarchy, i.e., the base class and all implemented interfaces. On the structural level, the *type shape* also captures all declared elements, i.e., events, delegates, fields, methods, properties, and nested types.

We accept that this strategy prevents us from recovering information about local types that have not been edited, but we follow the *optimistic* assumption that related files are edited together. The recovery is easier for types defined in publicly released assemblies like the `mscorlib`. Based on their stable version numbers, structural information can be extracted in an offline preprocessing step for many public libraries and made available such that types can be looked up. Our strategy of reducing the information captured from the type system presents a reasonable tradeoff between captured data size and data consistency. In addition, most approaches that we have surveyed only consider single files anyway, so we don't expect this to be an issue.

Source Code Our survey revealed that no consensus exists about the preferred form in which source code is analyzed, textual or tree based. We took the pragmatic decision to capture an abstract syntax tree (AST), because it is easier to pretty-print a tree structure into plain text then to built a parser for a textual representation. Our representation captures whole types and all included declarations to cater for the various scopes required by the surveyed works. We do not dive into nested types though and only capture them in a separate syntax tree whenever they are edited.

We decided to model the AST structure with semantic nodes and have a specific node type for each statement and expression type. While this is similar to the M3 [10] representation, it is in contrast to Boa's [52] simplistic model that only consists of seven node types. We think that the semantic representation makes it easier for researchers to reason about the AST and we accept the additional implementation effort that is required to move the reference implementation of the model to a new programming language. An advantage of this approach is that language keywords (e.g., `new`, `if`, or `while`) don't need to be explicitly stored, because they can be reconstructed.

We decided to include all instructions in our representation that are required to represent source code written in Java and C#. The outmost node is a type declaration for which we include, in addition to the type name, all type members (i.e., delegates, events, fields, methods, and properties). We also include the bodies of the method and property declarations and capture the containing statements and expressions.

The core element of analyses that are concerned with APIs is method invocations. In addition, we also want to support more sophisticated static analyses that need to follow the control flow (data flow or control flow analyses). We capture all blocks (e.g., `while`), statements (e.g., *assignment*), and expressions (e.g., *invocation*). Some of our surveyed approaches ignore variable names or use *alpha conversion* to normalize them. However, we argue that they should be preserved in a meta model, because previous work has shown that variable names transport context information [83]. Our representation should also support casting expressions as several approaches rely on them when learning valid *down-casts* [127]. In addition, we want to have the option to represent literals like numbers, booleans, or null values. Of course, all referenced types and type elements are stored in a fully-qualified form with our naming scheme.

Reference Generalization Most approaches that consider types only store the statically referenced type. In source code, these types are very often project-specific and do not bear any reusable information (e.g., `myproject.MyArrayList.add()` will not occur in any other repository, but the implemented `java.collections.IList.add(...)`

will). This is especially important in C#, where most of the time types are auto-inferred through the `var` keyword, but it is also important in JAVA, when variables are declared with too specific types (e.g., `ArrayList l = …` instead of `List l = …`); this is perfectly valid from the language perspective, but considered bad style and from the point of view of machine learning, these references are irrelevant, introduce a lot of noise in the data, and blow up the model size. This argument only holds for released code though. It is not possible to generalize variable types of code under edit, because it would invalidate the stored proposals in the code completion events of enriched event streams. These proposals point to the concrete type and a generalization could even make a captured selection *type incorrect*.

The generalization is even more important for method calls. Consider the invocation of `o.hashCode()`. The `hashCode` method is originally introduced in the `Object` type, but regularly overridden in subtypes. To *execute* the invocation of our example, it is required to know the runtime type of `o`. Using the type system, it is possible to look up the correct method implementation, i.e., the most specific method declaration that may override the generalized definition that is referenced. For our use case, we don't execute the code though and it is preferable to store the reference to the original method that introduced the signature. According to *Liskov's Substitution Principle* [119], derived methods should not change that behavior. Mandrioli and Meyer [128] call this concept the *contract*, which must not be violated in subclasses. By storing a reference to the first declaration, we can significantly reduce the number of unique method calls in our model and massively reduce noise for any machine-learning algorithm. Please note that, even if the LSP is violated, this replacement would not change the runtime behavior. It is still possible to select the most specific override based on the runtime type of the corresponding object.

Overall, we decided to generalize all type elements that can be overridden (i.e., events, methods, and properties) and to leave the type of variables unchanged. The implementation of such a variable generalization is possible with the information available in the meta model though and could be provided as a reusable component, for example, to preprocess released code from a repository. The names of elements that are declared in a captured SST, e.g., a method declaration, are not generalized and the concrete name is used instead. In case the declaration overrides existing elements, references to these original elements are stored in the *type shape*.

Facilitate Analyses One of the core goals of our intermediate representation is making it easier to reason about source code and supporting static analysis by design, so the intermediate representation should maximizes static analyzability. For example, opposed to code in repositories, source code under edit is very often incomplete or contains invalid parts. Thus, when transforming source code into the representation, these parts must be handled. We argue that this burden should not be left to the researchers that only work with the snapshot, because this issue is easier to solve in-IDE, where all surrounding information is still available and offending parts are easy to identify. We reduce these cases to *unknown expressions* to simplify later analyses.

We take several steps to reduce the complexity of source code and to facilitate static analyses. We make implicit information explicit and avoid optionals by design (e.g., optional `this` references are made explicit), normalize the source code and unfold nested expressions to new intermediate variables, remove syntactic sugar of very specific constructs (e.g., property initializer are reduced to an initialization and regular assignments), and unify the source code to some extent to reduce developer specific styles. These measures will be discussed with more detail in Section 5.3.

There is a clear tradeoff between a representation that is close to textual source-code, which allows analyzing changes on the character level, and a more semantic representation that highlights differences in the AST. Our focus is on semantic changes so we favor the latter. While the resulting representation is still very valuable for research on source-code evolution, properties of the textual representation might get lost that would be subject of studies. Future work should re-consider this design decision and should try to extend the intermediate representation such that also analysis of source-code evolution is supported to the full extent.

Data Scarcity We do not want to capture more information than necessary for two reasons. First, the less information we capture in our design, the fewer information need to be understood and processed by researchers that use our representation. Second, capturing too many details can easily become a threat to privacy, e.g., comments that contain author names or hard-coded credentials in source code, and we do not want to loose participants that are afraid to leak data that we do not need anyway.

While being supported by our design, literals present a big threat to the scalability requirement and at the same time they often do not carry much information. In the direction of data scarcity, we decided to follow related work [170] and to limit the captured literals to a set of special and very common literals (i.e., null, booleans, [-1,3], 0.0, and +/-1.0) and abstract away the rest to remove noise. This decision affects only a few approaches, for example approaches those that rely on comments (e.g., [60, 154]), but it solves a conflict between two requirements.

In the same way, we decided to drop the requirement of having links in the IR that point to the public location (e.g., [92, 106]). For one, this is a rare requirement and hard to realize when most intermediate states of in-IDE snapshots are not available online. In addition, we argue that a fully-qualified type reference provides enough information to find it through an internet search, if required for debugging purposes.

We also drop parts of the original AST that are not required by typical static analyses of RSSE approaches. This includes whitespace and other formatting details like parentheses, also visibility modifiers are not included.

Leverage in-IDE information The transformation to the intermediate representation can leverage more than just the information that is available in the source code. The corresponding process information is helpful to make sense of source-code evolution data and many details are preserved in our *enriched event streams* that are only available in the moment the code is edited. The fact that the source code is being transformed in-IDE also provides several other advantages.

First, some information is easier to extract from the editing context of the developer than to reconstruct after the fact. Type resolution is an example that has already been elaborated earlier. Getting unknown projects to compile is hard, but it is safe to assume that the in-IDE project environment of the developer is compilable, because this is needed for work. While our survey has revealed several approaches that might be able to recover fully-qualified type information after the fact (e.g. ZBINDER [180], BAKER [236], or through *Partial Program Analysis* [37]), these techniques are only heuristics and it is preferable to store correct facts when available.

Second, some issues are easier to solve in-IDE than after the fact solely based on the intermediate representation. The example that has been discussed before was that invalid source code parts could be easily identified in the AST that is provided in-IDE.

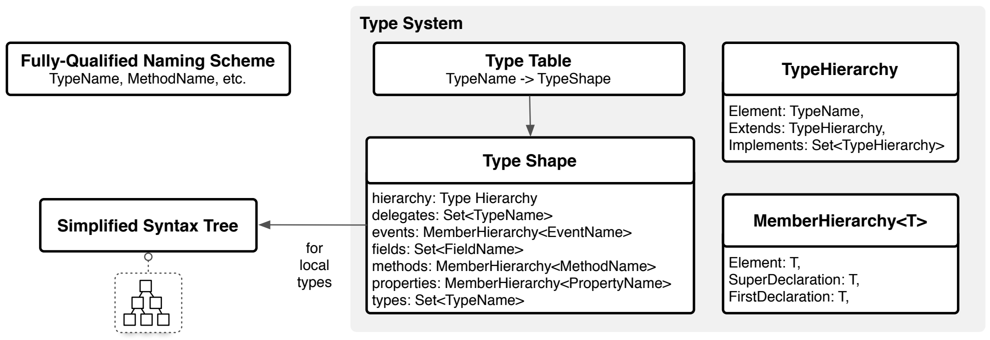


Figure 4.1: Design Overview for the Meta Model

We argue that situations like this should be solved in-IDE instead of serializing them and postponing the handling. The less corner cases are stored in the persisted data, the easier it is to work with the data later.

Third, some process information is closely tied to the source code editor and should be stored in the intermediate representation instead of the *enriched event stream*, e.g., the edit location or information about tool usages like code completion.

Overall, we emphasize again that additional information available in the IDE has to be leverage. In our design, we try to capture existing information, especially if it is hard to recover later, use existing mechanism for the resolution of issues, and try to preserve transient information that is only available in the editing process.

4.1.3 Design Overview

The previous subsections have presented the requirements that we have identified in our literature review. Our discussion has revealed that no appropriate representations exist so far. We needed a new meta model that is tailored to our use case to succeed in our studies. After further elaborating our design goals and considerations, we depict the final design of our meta model in Figure 4.1. It consists of three parts.

Naming Scheme for Code Elements The central part of our meta model is a naming scheme for fully-qualify references to types and type elements that allows us to unambiguously refer to source-code elements in object-oriented programs. For example, a method reference does not only contain the (simple) name, but also its declaring type, its parameter list (incl. types and parameter names), and so forth.

Type System Recovering typing information is expensive after the fact. We provide a lightweight infrastructure for type resolution that is based on *type shapes*, an abstraction for the structure and hierarchy of a type. It can be looked up for any type in our *type table* using a fully-qualified type name.

Simplified Syntax Trees At the lowest level, we capture a source-code representation of type declarations in a *simplified syntax tree*. The tree starts on the level of a type declaration and contains all declared members. We also store the contents of their bodies and go down to the expression level.

These three parts build our meta model that fulfills the requirements and that allows us to conduct the experiments and studies we have sketched before. To be useful, tools

```
// basic non-terminals
Id = ... // an arbitrary string
Num = ... // a positive integer value
Bool = ‘true’ | ‘false’

// types
Type = ‘?’ | <PrimitiveType> | <RegularType> | <TypeParameter> | <DelegateType>
PrimitiveType = <PrimitiveTypeBase> (<ArrayPart>)?
PrimitiveTypeBase = ‘p:void’ | ‘p:bool’ | ‘p:int’ | ...
ArrayPart = ‘[’ (‘,’)∗ ‘]’
RegularType = (<TypeQualifier> ‘:’)? <ResolvedType> (<ArrayPart>)? ‘,’ <Assembly>
TypeQualifier = ‘e’ | ‘i’ | ‘s’ // enum, interface, struct
ResolvedType = (<Namespace>)? <TypeName> (‘+’ <TypeName>)∗
TypeName = <Id> (<GenericPart>)?
TypeParameter = <Id>
DelegateType = ‘d:’ <Method> (<ArrayPart>)?

// organization
Namespace = (<Id> ‘.’)+
Assembly = <Id> (‘,’ <AssemblyVersion>)?
AssemblyVersion = <Num> ‘.’ <Num> ‘.’ <Num> ‘.’ <Num>

// generics
GenericPart = ‘‘’ <Num> (<ArrayPart>)? ‘[’ <GenericParam> (‘,’ <GenericParam>)∗ ‘]’
                        GenericParam = ‘[’ <TypeParameter> (‘->’ <Type>)? ‘]’
```

Figure 4.2: Name Grammar (Types)

need to exist that facilitate working with the meta model like pre-built components for recurring static analysis tasks. We decided to clearly separate the model from the platform in which it is used. Therefore, we will dedicate the rest of this chapter to the design and implementation of our meta model and we will postpone the discussion of the tooling we have built around it to Chapter 7.

4.2 Naming Scheme for Code Elements

When describing source code or referring to its elements, it is always required to use abstractions and to carefully reduce the concrete source code to a representation that only contains relevant information. On the other hand, it has to be possible to map these reduced representation back to source code. Source code is a very expressive construct in which also structure and nesting is relevant and an appropriate representation must be able to express these complex relations and locations. We created a grammar for such an encoding that is shown in Figure 4.2 and Figure 4.3.

The syntax of the naming scheme is driven by the idea to fully-qualify all identifiers and to encode all typing-related information about a code element in its name. The name representation preserves fully-qualified typing information for types and members. While the notation is inspired by the *extended Backus-Naur-Form* (eBNF), we did not intend the grammar to be usable for the automatic creation of a parser, but to make it easy for the reader to understand the data representation. We use “(...)” to group information and denote multiplicities in the commonly used way, i.e., “(...)?” is 0 or 1, “(...)∗” is 0 or more, and “(...)+” is 1 or more.

```
// member names
Member = ('static ')? '[' <Type> '] [' <Type> '].' <Id>

Event = <Member>
Field = <Member>
Method = <Member> <GenericPart> <ParamList>
Property = ('get ')? ('set ')? <Member> <ParamList>

ParamList = '(' (<Param> (',' <Param>)∗)? ')'
Param = (<ParamModifier>)? '[' <Type> ']' <Id>
ParamModifier = 'opt ' | 'out ' | 'params ' | 'ref ' | 'this '

// lambda names
Lambda = '[' <Type> ']' <ParamList>
```

Figure 4.3: Name Grammar (Members)

Full Qualification Each reference to the language model (e.g., a return type of a method) is stored in a *fully-qualified* form that, in addition to the namespace-qualified type, also contains the originating framework and its version (the *assembly* in `C#` terminology). An example of such an assembly-qualified name is "`a.b.C, foo.dll, 1.0`", where `a.b.C` is the type's namespace-qualified name, `foo.dll` is the assembly it belongs to (e.g., a library or framework), and `1.0` is the version of the assembly. Project-local types do not have a version, so we simply omit it, as in "`x.y.Z, MyProject`".

Various Types All examples so far have pointed to classes. Apart from classes, the naming scheme supports multiple kinds of types that are distinguished by prefix. Consider the type "`i:data.IList, collections, 1.2.3.4`". It refers to the `IList` type defined in the `data` namespace and the prefix `i:` denotes that this is an interface type. When a type is defined in a dependency (i.e., a referenced assembly), then both the name of this dependency (in this case `collections`) and the version is used. If the referenced type is instead defined in the current solution, then the name of enclosing project is used and the version is not set (e.g., "`T,P`": type `T` in project `P`). The notation can also encode array types and delegate types. Apart from classes and interfaces, we also support primitive types, enums, structs, arrays, delegate types, and generic type parameters. The concrete syntax varies and depends on the kind.

Generic Type Parameters We also support types with generic type parameters. Consider the type "`T'2[[G1], [G2->T2,P]],P`", which has "`'2`" generic type parameters, of which `G1` is not bound and `G2` is bound to "`T2,P`". Additionally, we also support nested classes. The example "`n.T+NT,P`" refers to the class `NT` nested in `T`.

Object-oriented Elements The grammar is not restricted to type references, it also defines a scheme to refer to other code elements as shown in Figure 4.3. We introduce a syntax for member names and lambda names, but the following examples are restricted to method names for brevity. A method is represented as "`[RT,P] [DT,P].M()`" in our notation. This refers to a method `M` that is defined in the type "`DT,P`", returns an object of type "`RT,P`", and has no parameters. Method parameters are listed between the parentheses: `…M([T3,P] p)`. Like all class members, methods can include the `static` modifier. They can also contain additional modifiers for parameters.

Meta-Model in Practice The information that is defined in the grammar can be stored as a `string`. In practice, we will provide an additional layer on top of this `string` that parses the individual parts. Our implementation will have an interface that is inspired by the reflection APIs of C# and Java and will provides access to the contained information of a given name. For example, for a method name, it is possible to request the return type or iterate over the parameters of the method signature.

Grammar 2.0 We have use the presented grammar in all datasets of this thesis and have successfully applied it in all our experiments. It has proven useful, but it grew naturally with its requirements. The parsing logic became very complex and the required maintenance effort increased significantly. In a major refactoring, we moved the grammar to the parser-generator framework ANTLR[25]. This significantly reduced the maintenance effort, because we could extract the partial information from a parse tree rather than through string operations. In addition, ANTLR grammars are supported in both C# and Java, so all the heavy-lifting of the parsing could be outsourced.

In this course, we fixed minor limitations of the current grammar, e.g., the prefixing that marks the kind of a type (i.e., "`i:`" is an interface) does not work for nested classes. We solved this by moving the prefix to the <TypeName> (e.g., "`i:n.I`" became "`n.i:I`"), which also works for nested classes (e.g., "`n.i:I1+i:I2`" describes the interface `I2` that is nested in the interface `I1`).

The grammar design also had the minor flaw that the array part was scattered over <PrimitiveType>, <RegularType>, <GenericPart>, and <DelegateType>. In addition, it was required to parse the number of "`,`" to identify the array dimensionality. We solved this by removing <ArrayPart> and introducing a new kind of <Type>, namely <ArrayType>. It is defined as "'arr(' <Num> '):' <Type>", which also allows us now to distinguish multi-dimensional arrays from ragged arrays (e.g., `int[,]` versus `int[][]`).

We extended the infrastructure around the naming classes and provided an upgrade path from the first to the second version. While the new version is stable and backed by an extensive test suite, we did not find the time yet to include it in the master release of the project, which requires adapting all affected instrumentations and upgrading our datasets. For this reason, we prefer not to discuss the new grammar in this section, but to stick to the old one as long as it is used in all our work.

4.3 Type System

Sophisticated static analyses do not only work with the program sources, they also require a type system to request information that go beyond the analyzed source code. Capturing complete snapshots of the type system is not possible and our incremental data collection challenges this requirement. It is necessary to trade off consistency and completeness of the type system with the feasibility of the tracking. We propose to completely avoid capturing the type system and to rely on API information instead. We optimistically assume that relevant information is defined in a public API, which can be recovered after the fact, or in a local file that is edited eventually and therefore also captured. We only loose detailed typing information for unedited local types.

We will introduce a reduced type system now that can be used to look up typing information consists of two parts, *type shapes* and a *type table*. A *type shape* is a minimal representation of the inheritance hierarchy and the structural elements declared in a type. The *type table* is a dictionary in which the type shape of a type

// type shape

TypeShape = {th:<TypeHierarchy>, nestedTypes:[<Type>]^1, delegates:[<DelegateType>]^1,
event:[<EventHierarchy>]^1, fields:[<Field>]^1, methods:[<MethodHierarchy>]^1,
properties:[<PropertyHierarchy>]^1 }

TypeHierarchy = {elem:<Type>, extends:(<TypeHierarchy>)?, implements:[<TypeHierarchy>]^1}

EventHierarchy = {elem:<Event>, super:(<Event>)?, first:(<Event>)?}

MethodHierarchy = {elem:<Method>, super:(<Method>)?, first:(<Method>)?}

PropertyHierarchy = {elem:<Property>, super:(<Property>)?, first:(<Property>)?}

Figure 4.4: Type Shape

can be looked up through its fully-qualified name. We will elaborate both parts and will then discuss how we fill our reduced type system with typing information.

Type Shape A <TypeShape> contains all information about the structure and the hierarchy of a type, the corresponding grammar is shown in Figure 4.4. This includes references to all extended classes and implemented interfaces, but also more detailed information about the elements that are declared in the type, i.e., delegates, events, fields, method, properties, and nested types. All information stored in the <TypeShape> is fully-qualified.

Declared events, methods, and properties of a type can be overridden in derived classes and we store information about their hierarchy in the *type shape*. For example, a <MethodHierachy> stores the name of a declared method, but also captures the information if the declaration overrides a previous definition and -if so- which one. A method might be overridden multiple times in a given type hierarchy. We store the name of the "super" method and the name of the "first" method, i.e., the one that originally introduced the signature. Storing this information provides the required information to perform a reference generalization (see page 73) after the fact. For example, when a method of an SST is analyzed, it is possible to replace the concrete name of the enclosing method with an generalized reference to a method that was defined in an assembly. Declared delegates and fields cannot be overridden in subtypes and we only store a [<DelegateType>]^1 and a [<Field>]^1.

For types declared in a local project, the *type shape* also stores the SST representation. This is not possible for types that originate in an assembly, because they are distributed in a binary form and we do not have access to the source code.

Type Table The second part of our reduced type system is the *type table*. It is a simple dictionary that contains type shapes for all types that can be resolved. The type shapes are requested with the fully-qualified name of a type. We follow the strategy that the type table is not captured in the snapshots and that it is recovered after the fact. We perform two different operations to fill it with data.

Precomputed Dataset References to API types can be restored through their fully-qualified names. The names are stable and contain versioning information, which makes it possible to request the corresponding assembly from public releases. We achieve this through a bulk download of released APIs, e.g., through their public releases on NuGet. Iterate over all declared types in an assembly, we extract the corresponding *type shapes* and store them in our *type table* for later use.

Keeping Track of Changes When capturing developer interactions, snapshots for local types are taken whenever the developer edits them. Only minimal information is captured though and typing information from other types than the edited one is not preserved. No consistent snapshot of all declared local types exists, but it is possible to recover the edited types after the fact. Every source-code snapshot that is contained in an event of the enriched event stream will be added to the table, potentially overriding a previous version. Unedited types cannot be recovered, but we optimistically assume that related files will be edited together. It is also necessary to track the reversals of changes to make sure that reverted changes are not kept in the *type table*.

As a result of both strategies, it is only possible to look up types of API that have already been analyzed or that are declared locally and have been edited. Reducing the type system to this incomplete subset is unsound, but it preserves the most relevant information and was a necessary tradeoff to enable our incremental data collection.

Usage Example We will exercise a simple analysis task now to illustrate a potential usage of the type table. Let us assume we want to find all subtypes of `IList` that override `GetHashCode`. To answer this question, we iterate over all *type shapes* that exist in the *type table*. For each entry, we first check, whether `IList` occurs in the hierarchy of the base class or in the hierarchy of implemented interfaces. If this is the case, we can request all declared methods from the type shape and check if a method `GetHashCode` exists that overrides and existing definition, i.e., `super` or `first` is set. Should the iteration find such a method, we have found a usage in which we have been interested in, and would store it for later processing.

Redundancy An obvious limitation of our reduction strategy for the preservation of typing information is the high redundancy. For example, instead of storing the value type of a field reference in the naming scheme, a simple member name (e.g., `_f`) and the fully-qualified declaring type suffice to look up the value type in the type system. Another example is the hierarchy information that we capture for declared members. Instead of storing the super and first definition, we could also identify these details through recursive look-ups using the base class. However, both examples require a complete type system, which is no valid assumption for in-IDE source-code snapshots. There is a trade-off between storing only the minimal information that is required for a sound recovery and storing redundant information that also provide some value if used with an incomplete type system. We decided that we only capture information about the type under edit for space reasons and accept redundancy to provide enough information for a *reference generalization* (see page 73).

4.4 Simplified Syntax Trees

An intermediate representation for source code represents the final part of our meta model. We defined *simplified syntax trees* (SST), a new tree-based intermediate representation for source code that enables our studies on source-code evolution and that facilitates the creation of complex static analysis for RSSE approaches. The model will also be used in enriched event streams to track the source code under edit.

We have considered two main styles for our representation of source code: ASTs and three-address representations. In the former, the representation is close to source code and best reflects the view of the developer. In the latter, it is easier to write static analyses, because the complexity of the language is reduced to simple operations

```
// sst
TypeDecl = {enclosingType:<Type>, partialClassId:(<Id>)?, delegates:[<DelegateDecl>]^1,
   events:[<EventDecl>]^1, fields:[<FieldDecl>]^1, methods:[<MethodDecl>]^1,
   properties:[<PropertyDecl>]^1}
DelegateDecl = {name:<DelegateType>}
EventDecl = {name:<Event>}
FieldDecl = {name:<Field>}
MethodDecl = {name:<Method>, body:[<Statement>]*, isEntryPoint:<Bool>}
PropertyDecl = {name:<Property>, get:[<Statement>]*, set:[<Statement>]*}
```

(a) Simplified Syntax Tree

```
// references
Reference = <AssignableRef> | <MethodRef> | <EventRef>
AssignableRef = <VariableRef> | <IndexAccessRef> | <FieldRef> | <PropertyRef> |
   <UnknownRef>

EventRef = {ref:<VariableRef>, name:<Event>}
FieldRef = {ref:<VariableRef>, name:<Field>}
IndexAccessRef = {expr:<IndexAccessExpr>}
MethodRef = {ref:<VariableRef>, name:<Method>}
PropertyRef = {ref:<VariableRef>, name:<Property>}
UnknownRef = {}
VariableRef = {id:<Id>} // an unknown variable reference has the id "?"
```

(b) References

Figure 4.5: Structural Information in SSTs

on registers and jumps with labels. Our design combines the advantages of both styles. It stays very close to the AST of the original source code and preserves most syntactic elements found in the source code, such as declarations, invocations, or control structures. At the same time, it provide some advantages of a three-address representation. For example, we decided to leave out information that is not typically used when reasoning about the source code such as white space and comments, which further reduces the size of the representation. In addition, we normalize the captured code and flatten complex nested expression, which simplifies the representation and also unifies over alternative styles of writing. An advantage that goes beyond both styles is our decision to avoid implicit information and to make everything explicit. We make SST self-contained by using our naming scheme for references to all types and type members, which allows a later resolution of these references. Storing consistent snapshots also makes the representation robust against event deletions in the enriched event stream that would break, for example, an incremental representation.

We present the SST grammar in Figure 4.5, 4.6, and 4.7 and use the naming scheme as a building block in the grammar. For brevity, we slightly adapt the eBNF grammar notation of the JSON-like format: instead of using terminals for the curly braces, we use the construct {a:<A>} to represent an object that has a property a of type A. The construct [<A>]* stands for a list of <A>, [<A>]^1 for a set of <A>.

Structural Information The SST grammar is designed to capture information about a single type declaration. The root element is a <TypeDecl>, for which we store the fully-qualified name of the enclosing type. `C#` allows the declaration of partial classes, so we also include another qualifier and capture the file name of a partial class as an additional identifier to distinguish the different parts of the type. In addition, the root node is used to store all relevant structural parts of the syntax tree. We store all member declarations (i.e., event, fields, methods, and properties) and also capture the bodies of all declarations that define one. The names of all members are stored in the fully-qualified naming scheme, which also stores several modifiers, e.g., static keywords for type elements. A <TypeDecl> also contains all delegate declarations, but we do not step down into nested types and store them in separate SSTs.

Fully-Qualified References When transforming source code to SSTs, we replace every explicit or implicit type reference by the corresponding fully-qualified name. This means that we also store the respective fully-qualified names for the declaring type, any parameter types, and the value or return type of every reference expression.

All member references include a reference to the target variable and a reference to the statically referenced element. For static references, the naming scheme provides the information that the reference is static and the corresponding variable reference is unknown. We introduced variable references as an abstraction to avoid simple strings, also reserved keywords like `this` or `value` are modeled as variable references.

Method Bodies We capture the most common language constructs in the methods that fulfill our requirement to *support static analyses* (Section 4.1). We include <VariableDecl> and <Assignment> statements to capture data flow operations. The syntactic sugar of `C#` that is used to add or removing event listeners (i.e., `+=` and `-=`) are usually also modeled as assignments, however, we decided to simplify the analysis by avoiding overloaded operators and created the special statement type <EventSubscriptionStmt> to capture these operations. Some expressions often stand alone, e.g., method invocations, without an assignment. To allow this syntax, we provide an <ExpressionStmt> that can be used for wrapping and allowing them as standalone statements. We capture several statements that affect the control flow, namely, operators to control loop execution (<BreakStmt>, <ContinueStmt>) or for method exit (<ReturnStmt>). We also support jump operations with <GotoStmt> and <LabelStmt>.

In addition to these statements, the SST specification includes several <Block> types that affect control flow, namely loops (<DoLoop>, <ForEachLoop>, <ForLoop>, and <WhileLoop>), case control (<IfElseBlock> and its syntactic extension, the <SwitchBlock>), as well as statements for the handling of exceptions, both their creation (<ThrowStmt>) and their handling (<TryBlock>). We support all three catch types in <CatchBlock> that are defined in the `C#` language.

The remaining blocks supported in SST are <LockBlock> that is used for concurrent programs, <UsingBlock> that automatically frees the claimed resources by automatically calling the `dispose method` of the provided object, and <UncheckedBlock> that is used to disable various runtime exceptions, e.g., number overflows. While `C#` supports both a block variant and an expression variant, we reduce both cases to the block variant. While we also capture <UnsafeBlock>, we do not capture the body of the block, as it allows C-like constructs (e.g., pointers) that we do not want to consider.

In case a program that should be represented in SSTs contains unsupported language constructs, an <UnknownStmt> can be used to mark these unsupported parts.

Expressions We support three kind of expressions <SimpleExpr>, <AssignableExpr>, and <LoopHeaderExpr>. <SimpleExpr> are the most basic expressions that are easy to analyze and to understand. We distinguish constants (<ConstExpr>), references of

// statements
Statement = <Block> | <Assignment> | <BreakStmt> | <ContinueStmt> |
  <EventSubscriptionStmt> | <ExpressionStmt> | <GotoStmt> | <LabelStmt> | <ReturnStmt> |
  <ThrowStmt> | <UnknownStmt> | <VariableDecl>
Assignment = {ref:<AssignableRef>, expr:<AssignableExpr>}
BreakStmt = {}
ContinueStmt = {}
EventSubscriptionStmt = {ref:<AssignableRef>, op:<EventSubscriptionOp>,
  expr:<AssignableExpr>}
EventSubscriptionOp = 'Add' | 'Remove'
ExpressionStmt = {expr:<AssignableExpr>}
GotoStmt = {label:<Id>}
LabelStmt = {label:<Id>, stmt:<Statement>}
ReturnStmt = {expr:<SimpleExpr>, isVoid:<Bool>}
ThrowStmt = {ref:<VariableRef>, isReThrow:<Bool>}
UnknownStmt = {}
VariableDecl = {id:<VariableRef>}

(a) Statements

// blocks
Block = <DoLoop> | <ForEachLoop> | <ForLoop> | <IfElseBlock> | <LockBlock> |
  <SwitchBlock> | <TryBlock> | <UncheckedBlock> | <UnsafeBlock> | <UsingBlock> |
  <WhileLoop>
CaseBlock = {label:<SimpleExpr>, body:[<Statement>]*}
CatchBlock = {kind:<CatchBlockKind>, param:<Parameter>, body:[<Statement>]*}
CatchBlockKind = 'Default' | 'Unnamed' | 'General'
DoLoop = {cond:<LoopHeaderExpr>, body:[<Statement>]*}
ForEachLoop = {decl:<VariableDecl>, loopedRef:<VariableRef>, body:[<Statement>]*}
ForLoop = {init:[<Statement>]*, cond:<LoopHeaderExpr>, step:[<Statement>]*,
  body:[<Statement>]*}
IfElseBlock = {cond:<VariableRef>, then:[<Statement>]*, else:[<Statement>]*}
LockBlock = {ref:<VariableRef>, body:[<Statement>]*}
SwitchBlock = {ref:<VariableRef>, sections:[<CaseBlock>]^{1}, defaultSection:[<Statement>]*}
TryBlock = {body:[<Statement>]*, catchBlocks:<CatchBlock>, finally:[<Statement>]*}
UncheckedBlock = {body:[<Statement>]*}
UnsafeBlock = {}
UsingBlock = {ref:<VariableRef>, body:[<Statement>]*}
WhileLoop = {cond:<LoopHeaderExpr>, body:[<Statement>]*}

(b) Blocks

Figure 4.6: Structural Information in SSTs

any kind (<ReferenceExpr>), and <UnknownExpr>. All expression types that go beyond this are modeled as <AssignableExpr> that can stand on the right hand side of an assignment. This, of course, can be <SimpleExpr>, but we also support several more complex expressions. We provide <UnaryExpr> and <BinaryExpr> to capture composed expressions, typically arithmetic or boolean operations. <IfElseExpr> adds support for inline case distinction on the expression level. Complex type handling is enabled by the <TypeCheckExpr> ("`is`") and <CastExpr>. We support both optimistic casting (`o as Foo`) and the traditional variant (`(Foo) o`) that can raise runtime exceptions. We added <IndexAccessExpr> to support arrays. The <InvocationExpr> is used to represent

```
// expressions
Expression = <SimpleExpr> | <AssignableExpr> | <LoopHeaderExpr>

SimpleExpr = <ConstExpr> | <ReferenceExpr> | <UnknownExpr>
ConstExpr = {value:<Id>}
ReferenceExpr = {ref:<Reference>}
UnknownExpr = '?'

AssignableExpr = <SimpleExpr> | <BinaryExpr> | <CastExpr> | <CompletionExpr> |
  <IfElseExpr> | <IndexAccessExpr> | <InvocationExpr> | <LambdaExpr> | <TypeCheckExpr>
  | <UnaryExpr>
BinaryExpr = {left:<SimpleExpr>, op:<BinaryOp>, right:<SimpleExpr>}
BinaryOp = 'Unknown' | 'LessThan' | 'LessThanOrEqual' | 'Equal' | 'GreaterThanOrEqual' |
  'GreaterThan' | 'NotEqual' | 'And' | 'Or' | 'Plus' | 'Minus' | 'Multiply' | 'Divide' | 'Modulo' |
  'BitwiseAnd' | 'BitwiseOr' | 'BitwiseXor' | 'ShiftLeft' | 'ShiftRight'
CastExpr = {targetTyoe:<Type>, op:<CastOp>, ref:<VariableRef>}
CastOp = 'Unknown' | 'Cast' | 'SafeCast'
CompletionExpr = {typeRef:(<Type>)?, varRef:(<VariableRef>)?, token:<Id>}
IfElseExpr = {cond:<SimpleExpr>, then:<SimpleExpr>, else:<SimpleExpr>}
IndexAccessExpr = {ref:<VariableRef>, indices:[<SimpleExpr>]*}
InvocationExpr = {ref:<VariableRef>, method:<Method>, parameters:[<SimpleExpr>]*}
LambdaExpr = {name:<Lambda>, body:[<Statement>]*}
TypeCheckExpr = {ref:<VariableRef>, type:<Type>}
UnaryExpr = {operator:<UnaryOp>, operand:<SimpleExpr>} UnaryOp = 'Unknown' | 'Not' |
  'PreIncrement' | 'PostIncrement' | 'PreDecrement' | 'PostDecrement' | 'Plus' | 'Minus' |
  'Complement' |

LoopHeaderExpr = <SimpleExpr> | <LoopHeaderBlockExpr>
LoopHeaderBlockExpr = {body:[<Statement>]*}
```

Figure 4.7: Expressions

method calls; for static calls, the variable reference points to an unknown variable. Constructor calls are also modeled as <InvocationExpr>. As `C#`'s object initializer expressions increase the complexity of the analysis of an initialization expression, we did not model them in the grammar. A transformation for `C#` must handle affected constructor calls and flatten out the nested initialization logic. <LambdaExpr> can be used to capture anonymous functions. We capture the defined signature in the name and store its body in the SST node.

We skipped the discussion of the expression types <CompletionExpr> and <LoopHeaderExpr> at this point, because they cannot be mapped to existing concepts in object-oriented programming languages. Both will be properly introduced in the following paragraphs though, in which we will justify their existence.

Unification SSTs enforce some unification of the source code that is performed in the transformation step that creates an SST. Consider, the example `if(isX())...`. Some developers want to assign `isX()` to a variable first, while others will not. The IR enforces the former style by only allowing <SimpleExpr> in conditions, i.e., <ConstantExpr> or <ReferenceExpr>, other expressions have to be assigned to an artificial variable first that is created during the transformation. For simple expressions this works even in loops, but it is not generally possible, because the loop condition is evaluated multiple times (e.g., consider `while(isX())...`). The condition could be included both before the loop and at the end of the loop, but this would introduce duplication, which is especially bad when the condition is more complex. SSTs address this problem by introducing the concept of <LoopHeaderExpr>, an expression that is allowed in a loop header. While <SimpleExpr> are directly allowed in this location, more complex expressions have to be normalized. The condition can be normalized into a <LoopHeaderBlockExpr>, which introduces a body such that it can store multiple statements.

Embed Process Information One of the things we were interested in early on is understanding how developer use code completion tools. Therefore, we added the new expression type in addition to the regular code elements defined in the `C#` language, the <CompletionExpr>. It is used as a marker for the edit location and to store information about code completion events. For example, we store the object reference on which the code completion was triggered. When an SST is created without having the developer edit the file (e.g., by transforming a piece of code outside of the IDE), this additional element is never included, because no interaction takes place.

Including this information as a language element ensures consistent source code transformations both inside and outside of the IDE and saves the effort of maintaining two variants. For example, this makes it easy to integrate any recommender system that relies on source code into the IDE, because it can count on receiving the same input whether it is working on in-IDE source code or source code from repositories.

Support for Other Object-Oriented Languages The previous description was specific to `C#`, but the introduced abstractions make it possible to represent other object-oriented languages too. The naming convention is applicable and also the representation of source code in SSTs. It is necessary though to think about appropriate mappings and reduction of syntactic sugar specific to a language. We will illustrate this idea by discussing some considerations for the representation of JAVA in SSTs.

Java shares many concepts with SSTs (`C#`) and many differences are just terminology and can be directly mapped. The basic structure of a type is compatible to SSTs, with the exception that some concepts like <Property>, <Event>, or <DelegateType> do no exist and will never be set in SSTs that are created from JAVA sources. Several basic statements, e.g., <VariableDeclarationStmt>, <Assignment>, or <ExpressionStmt>, correspond to the SST constructs. Other elements have to be mapped to equivalent constructs first, e.g., `synchronized` blocks to <LockBlock>, try with resources to <UsingBlock>, or *varargs* parameters to the `params` keyword of `C#`.

Each language also brings its own peculiarities though that might not all be representable in SSTs. For example, in the case of JAVA, we cannot preserve the concepts of *checked exceptions* and special modifiers like `synchronized`, because our naming scheme that is used to identify method signatures does not support them. In addition, JAVA supports some special constructs like `assert` statements or named versions of the <BreakStmt> and <ContinueStmt> that do not have an equivalent in `C#`. However, we argue that these special cases are less important for RSSE approaches.

So far, SSTs are just a transformed version of the original source code and do not have their own semantics. The representation can be used to store multiple languages and related work has shown that structural refactorings can be provided in a language agnostic way [51, 241]. SSTs also allow the creation of general components that just work on the AST (e.g., source-code differencing). We are less optimistic though that also more specific analyses or transformations can be provided in a language independent manner without defining semantics. Even for languages like JAVA and `C#`

that are very close in syntax, it is easy to break source-code when semantics are not considered, for example, because of differing scoping rules for variable names.

SSTs are not special in this regard and they are as prone to this problem as other meta-models. We highly doubt the benefit of a general language-agnostic platform for the concrete use case of writing static analyses and reusable program transformations. These must carefully consider semantic differences between languages. There is a big chance or future improvement of SSTs, since defining a semantic solves this problem.

4.5 Limitations

We designed SSTs with several use cases in mind, but we have focused on *source-code-based recommendation systems* [139] that are concerned with API usage and that do not rely on additional input (e.g. VCS, bug tracker, mailing list, etc.). For example, method call recommenders, snippet recommenders, or code search problems. The design of SSTs has some limitations, which we will discuss in the following.

The vast majority of related work (Section 2.1.2) extracts feature vectors from the structured context of the editing position. However, an increasing number of publications use text-mining techniques to solve the same challenges. It is straightforward to create a tokenization of SSTs or even to render plain code from it on which text-mining approaches can then be easily applied. However, SSTs are a reduced version of the original source code that only contains relevant information, so these approaches are affected by the reduction and the normalization. Future work should analyze these effects. In addition, it should be analyzed, if text based approaches could benefit from applying components of our framework such as loop normalization or inlining.

We currently do not capture literals of predefined types such as strings, integers, or booleans. SSTs can store these values, but we only capture the most common cases (e.g., `true`, `1.0`, `-1`) and leave out the rest. We did that intentionally to save storage space and also to avoid unnecessary noise such as irrelevant strings contained in `System.out`, as most RSSE approaches do leverage this information. However, specific APIs such as the *Java Cryptography API* rely on string parameters, but since such APIs may not be very common, we opted for reducing noise. Technically, it is very easy to change the implementation and capture everything. Future work should try to find heuristics to decide which parameters to capture and which to ignore.

SSTs represent source code in a normalized form by avoiding the nesting or chaining of expressions. This design decision was driven by the desire to simplify analysis tasks. Some approaches work on the original AST, changing the structure of the tree might impact their performance. We mark the artificial intermediate variables, so inverting the normalization is possible. In future work, we will consider providing the normalization as a reusable component to better separate the snapshot creation from the SST transformation.

SSTs in their current form maximize the analyzability and the transformation removes erroneous parts from the original syntax tree, e.g., parsing or typing errors. However, future work might be interested in these invalid parts, e.g., studies on source-code evolution. It should be analyzed in future work whether SST nodes could wrap these invalid parts. These wrapped nodes can easily be ignored by static analyses, but could be consumed by approaches that are interested in these details.

Right now, the source code representation, the tracking, and the transformation that does the normalization are heavily interwoven. While the result facilitates static analyses, some studies on source code evolution might miss details that were removed in the process. Future work should analyze, if it would be better to split the transformation and the normalization. The tracking could be realized with a representation that is closer to the host-language, but that preserves typing information. The normalization could be an additional component that works on top of this representation rather than directly on source code.

The source-code representation had the goal to capture process information as part of the source-code to make it easier to align changes with recorded interactions. We did one step into this direction and included a new expression type in SSTs, completion expressions, which capture details about incomplete code that is written and used with invocations of the code completion. However, this kind of input is only one *anchor* to the code that is used for tool invocations and others should be captured too. For example, many refactoring tools require highlighting an expression (e.g., *assign to variable*), a single statement (e.g., *surround with try-catch*), or a group of statements (e.g., *extract method*) with the textual highlighting tool. Future work should search for ways to integrate this information into SSTs.

SSTs store typing information, but it is impossible to store the entire language model in every snapshot. Therefore, we restrict the captured information to the immediately relevant parts. For example, a snapshot does not include pointers to all methods that are overridden by a method declaration, but only to the super method and to the first declaration that introduced the specific signature. Even though we miss information that might be required by some approach, we argue that the effect of this tradeoff is rather small. Project-specific references do not carry any reusable information and researchers are usually interested in information about reusable APIs. Our solution presents a reasonable tradeoff for this case, because the fully-qualified type information contained in SSTs allow looking up further information about public APIs in our type index after the fact. Only project-specific information is lost.

Chapter Summary

Source code is an important artifact to study and an appropriate meta model can support this task significantly. We want to capture fine-grained code evolution in our enriched event stream to enable work on research questions that involve source code. A simple textual snapshot of a source code file would pose many analysis challenges and would also loose a lot of information that cannot be recovered, most importantly typing information. No existing intermediate representation could solve this.

We have designed and implemented a new meta model for object oriented source code in this chapter that can be used to capture both snapshots of source code under development and source code found in repositories. We have extensively elaborated on our requirements and the design process and came up with a solution that consists of three parts. We have created a fully-qualified naming scheme that can be used to refer to types and type references, we have designed a new intermediate representation for source code called simplified syntax trees that preserves types, and we have designed a strategy to preserve relevant information from the type system. Combined, the three parts allow capturing incremental and self-contained snapshots of evolving source code and of source code found in repositories. The final section of this chapter has discussed limitations of our meta model that we are aware of.

Part III:

Instantiating In-IDE Development Models

This part will introduce the tooling that we have created to instantiate our meta models. We will start by introducing the infrastructure we have build around them. We will then present our interaction tracker FEEDBAG++ that we use to track developers, present its extensible design and additional consideration about privacy or incentives for participation. After that, we will present the C$\boxed{\text{A}}$RET platform, that provides composable analyses and reusable transformations for our meta models to facilitate static analyses and empirical studies on the development process.

5 Infrastructure

A stable infrastructure is required to enable the creation of general components the can be reused in research on RSSE. In this chapter, we will introduce the basic infrastructure that we have created to instantiate the in-IDE models and to enable working with them. We will start by presenting how we implemented the meta models of enriched event streams and simplified syntax trees (Section 5.1) and how we provide a serialization infrastructure to allow persistence and data exchange (Section 5.2). We will then introduce the reusable source-code transformation that we provide to transform source-code into SSTs (Section 5.3). After this, we will discuss the in-IDE infrastructure that we have created for tools and other researchers in Section 5.4. Finally, we will introduce the service platform that provides the central aggregation point for a data collection in a distributed field study (Section 5.5).

5.1 Meta-Model Bindings

The previous part has introduced meta models for in-IDE software development, *enriched event streams*, and for source code, *simplified syntax trees*. We provide an implementation of both models to make them usable in practice. Lacking a formal specification, we decided to focus on one stable implementation that we can use as the reference. Migrations to other languages must ensure consistent behavior. Our main use case was VISUAL STUDIO development, so naturally we did the reference implementation of the binding in the `C#` programming language.

We have directly adapted the designs that were presented in Chapter 3 and Chapter 4 and have implemented the classes for the different *event types* of enriched event streams, the classes that represent the *code snapshots*, and the *naming schemes* for IDE components and code elements. The implementations for the first two directly correlate with the UML design presented in the previous part. While the names are represented as plain strings, which could directly be used in a data structure, we decided to provide a more elaborate implementation in our code base that hides the string under a parsing layer, which provides access to the semantic information. Therefore, we have defined the semantic information through a hierarchy of interfaces that provide access, for example, we provide a `ReturnType` property to access the returned type from a method name. If the model is migrated to a new language, then it is required to implement these interfaces and consequently also the parser.

In our implementation, we follow the definition of a data structure by Martin [132]. In contrast to an *object*, which should hide its data and only expose methods to operate on the it, a *data structure* should completely expose its data and avoid complex functions. To support this, we realized all classes in a way such that they only hold references to secondary data structures, if their contained information is relevant for the containing class. We also restricted methods on all classes to simple accessor methods (e.g., getters or boolean checks).

Our implementation is thoroughly tested with an extensive test suite. We provide a significant set of unit tests for the data structures and especially for the parsing. A design decision that is not reflected in the class diagram is the guideline that any implemented data structure should provide meaningful `GetHashCode` and `Equals` methods. This greatly improves not only the testability of the infrastructure for the data structure (e.g., the serialization), but also for the generators of the events. In addition, we provide a meaningful `ToString` that makes it is easy to debug programs that make use of the data structures and we encourage migrators to do the same.

As a proof of concept for language migration, we decided to provide the same binding support in JAVA too. First, we had to create the class hierarchy for the code elements, equivalent to the model in C#. Second, we had to implement the parsing of the naming scheme. To ensure consistency with the reference implementation in C#, we have implemented a test generator in C# that uses all existing C# test cases to generate the test suite for JAVA. This test generation proved to be a simple and maintainable way to ensure consistent parsing in both languages. Finally, we successfully migrated all data structures and created a full implementation of all data structures that are related to enriched event streams and simplified syntax trees.

Overall, our implemented bindings to the data structures are ready to use for other researchers. Both the C# and the JAVA implementation are in a stable state and we have used both in our own research already. This experience has proven for us that a migration of the bindings to a new language is easy. It turned out to be a simple engineering task and did not reveal any major issues in the design.

5.2 Data Serialization

We designed the meta models of enriched event streams and simplified syntax trees in a way that that makes it easy to reuse the data schema. To make them applicable in practice, it is necessary to design the serialization format, in which the data is published, to allow storage or exchange of data.

Design Goals We tried to create a solution that is *easy* to work with and did not want to dictate how developers integrate our infrastructure into their applications. To this end, we tried to provide an infrastructure that is as little restrictive as possible by following the following main ideas.

Basic Technologies The data persistence should only use commonly used technologies to avoid potential incompatibilities in new programming environments. It should be possible to work with the dataset by using built-in features of modern programming languages or by relying on very common dependencies.

Simple Organization The data should be organized in a way that avoids unnecessary overhead. We want to be able to distribute datasets by download and it should not be necessary to setup a complex environment to use the data. Following a basic tutorial should be all that get's you started.

No Query API We want to avoid a custom layer of infrastructure that is required to query the data. We don't want to enforce a specific API, instead, the solution should facilitate using common language features to interact with the data.

Extensibility Given the extensible nature of the meta models, also the serialization should follow this principle. The addition of new event types and the maintenance effort, e.g., for custom serialization providers, should not require much effort.

These goals have affected all our design decisions. In the following, we will go through all steps of the implementation. We will first present our considerations regarding the serialization format. We will then go through the details of our reference implementation. Lastly, we will provide a proof-of-concept that shows that the infrastructure can be moved to other programming languages.

Technical Representation Many programming languages are used in practice and it is desirable that a dataset can be used in all of them. It is unrealistic to expect that the publisher of a dataset provides support for all possible languages. The better alternative is to design the data schema in a way that makes it easy to support new languages and serialize it in a way that is as accessible as possible. Users of the dataset can then provide additional bindings when needed.

Several standard serialization solutions exist. We considered plain text serialization into XML and JSON, binary-encoding (e.g., Google Protocol Buffers[35]), or serialization into SQL databases with *object-relational mappers* (ORM). We analyzed the maintenance effort, the performance of reading/writing, the memory overhead, and the practicality for our use case. All solutions had some advantages over the others, but none was generally superior and we ended up using a file-based JSON serialization. Compared to other approaches, the serialization is larger and working with it is relatively slow. However, it is schema-less, which made maintenance in the initial development easier, human-readable, therefore easier to debug, and widely supported in programming languages, which improves its accessibility. Overall, we found the drawbacks to be an acceptable limitation when compared to the advantages.

In our evaluation phase, we identified some general rules of thumb that facilitate serializability, regardless of the concrete solution that is picked.

- Classes must be instantiable with a constructor that has no arguments.
- Inheritance should only be used carefully.
- It is encouraged to limit the used types to those that are easy to serialize. This includes primitive numbers (e.g., `Integer`, `Double`), `Boolean`, enumerations, `String`, built-in types (e.g., `DateTime`), or other classes that follow these rules of thumb.

We made sure that our bindings follow these rules to make using existing serialization solutions straightforward. If necessary, following these rules also makes it easier to switch to other representations later on. For example, users that require an easier management of the data can migrate it to an SQL database or the ones that require increased performance or reduced size can switch to a binary representation.

File-System Layout In our use case, we will always store multiple objects at once, either `IDEEvents` in case of interaction data or `Contexts` for repository data. The decision for a file-based serialization makes it necessary to define how the objects are stored on the file system level. We will serialize each object into a JSON representation and store this string in a file. However, instead of storing many small files on the hard drive, we aggregate all files into a single .zip archive. This significantly reduces the required logical space on the hard-drive when compared to an uncompressed storage of many small files. When reading, To read the data again, it is easy to open the .zip and iterate of the contained files, deserializing one by one on the fly, without the need to actually extract all files at once.

In addition to the payload that is serialized from the object, we use the file path of the .zip archive to encode certain meta-data. As a result, we expect a specific directory structure that depends on the dataset.

Interaction Data We define by convention that all events of the same user will be contained in a single archive. This allows using the file path as a user id.

Repository Data Working spaces in C# are organized by *solutions* that represent a collection of projects. Each solution is defined in a .sln file that contains all configuration options. The transformation of a solution results in a set of *Contexts*, which should be serialized into an archive that has the same relative path to the zip root as the solution file to the repository root. This results in a mimicked directory structure in both folders and allows finding the corresponding solution for a .zip archive easily.

It is also possible in both cases to simply search for all contained .zip archives in the folder. The contained files in the found archives can be read as JSON serialized `IDEEvents` (interaction data) or `Contexts` (repository data). While the actual path of the zip encodes meta-data, it is not required to actually use it.

Reference Implementation We followed the same strategy for our serialization component that we have followed for the bindings. After deciding for the technical representation, we did a reference implementation for the serialization of our meta models. At the moment, the only sources of event streams and simplified syntax trees are implemented in C#. Choosing the same programming language for the reference implementation seemed to be a reasonable choice.

We are using JSON.NET[33] for the JSON serialization, the most popular serialization framework in the NuGet repository. Only minor configuration was necessary, for example, that null values are not being serialized. To allow a deserialization of hierarchies, we enabled the preservation of the concrete runtime type for affected objects. A special field `$type` is added to the JSON representation that contains the fully-qualified type of the serialized object. As these fully-qualified type references were responsible for a significant portion of the serialized JSON string, we have further optimized these type markers and now shorten the fully-qualified type reference to the simple type name for all classes that belong to our meta models.

Our implementation is extensively tested in a regression test suite. We ensure that extensions and changes to data schema or serialization logic do not affect the interpretation of previously serialized contents. We also protect other clients from breaking by ensuring that the serialization format does not change.

Supporting New Languages One of our goals was to make it easy to add serialization support for the meta models to new languages. We decided to provide another deserialization implementation in JAVA to prove that this migration is indeed possible. For the JSON serialization, we picked the very commonly used GSON[34] library. We adapted the configurations options to make them compatible to their JSON.NET equivalents. A consistent naming of types and namespaces in both languages avoids complex mapping strategies for the type information that is stored for hierarchical types. Using simple names for these types made it necessary to apply a simple regular expression in the deserialization that adds the missing parts of the type. Finally, we also had to register a custom (de-)serializer with GSON that uses the parser to instantiate the bindings for the naming related classes.

To summarize, we provide a data serialization that can be used to read and write our bindings for enriched event streams and simplified syntax trees. We initially implemented it for C# and designed it to be simple and extensible. We found migrating the serialization to a new language to be a straightforward implementation task. To help other researchers adopting our data schemas and tools, we provide an extensive documentation on the website of the KaVE project[9] that links concrete examples illustrating how the serialization can be used.

5.3 Reusable Source-Code Transformation

After designing the SST data structures that can represent source code, a transformation was required to transform source code into SSTs. Looking at the grammar, it is obvious that many constructs that are valid in regular programming languages like C# or JAVA have to be adapted to become valid SST construct. The transformation has to transform the input sources into SSTs, which involves reducing some syntactic sugar into valid SST constructs.

Transformation by Example One of the core goals of SSTs was to reduce the complexity of source code to make analyses easier. Our representation is inspired by the language specification of JAVA and C#, but provides several extensions and simplifications. The three most important ideas are: 1) embed fully-qualified typing information, 2) normalize the source code and avoid nesting of expressions, and 3) make any implicit information explicit (like optional `this` references). This has several implications on the transformation, which are best explained with an example.

Figure 5.1a shows a piece of C# code and the corresponding SST when rendered to a source-code like representation (Figure 5.1b). The source-code transformation takes care of creating fully-qualified names for all references to types and type elements, but we omit the full names in this example for brevity. The simplified version makes implicit information visible. For example, the `var` keyword on Line 7 of the original code is replaced by a type reference in the SST and all `this` references are explicitly named. Furthermore, SSTs unify over alternative styles of writing. For example, the nested and chained calls in Line 6 of the original source code are assigned to intermediate variables in Lines 4 to 6 in the SST.

We implemented several transformation steps to follow these principles. The transformation simplifies nested expressions that are more complex than literals or reference expressions by creating artificial intermediate variables to which the formerly nested expressions are assigned. The transformation removes syntactic sugar from the source code and replaces it with the de-sugared variant. For example, C# allows to initialize properties directly as part of a constructor call. Our transformation reduces this initialization to a constructor call, followed by an assignment.

Entry-Point Analysis We formalized the requirement earlier that reusable methods of a type should be marked as *entry points* through which the control flow can enter the class. Additionally, we want to maximize the number of method declarations that are stored in the SST, while only including those that are relevant for code reuse, i.e., those that are either entry points or that are transitively called by entry points.

```
o.A();

if (o.IsX()) {


   o.B(C()).D();
   var e = E();
   e.t□
}
```

(a) Original C# Source Code

```
o.A();
bool $0 = o.IsX();
if ($0) {
   int $2 = this.C();
   string $1 = o.B($2);
   $1.D();
   int e = this.E();
   completion(e, t)
}
```

(b) Simplified Syntax Tree

Figure 5.1: Transformation of a Source Code Example

```
var candidates = CurType.MethodDecls.Where(md => !md.IsAbstract
      && !md.IsPrivate && !md.IsInternal)

var entryPoints  = candidates.Where(md => md.IsDeclaredInParentHierarchy);
CurType.ConstructorDecls.Where(cd => !cd.IsPrivate && !cd.IsInternal)
      .foreach(cd => entryPoints.Add(cd))

var analyzed = new Set();

entryPoints.foreach(ep => AnalyzeTransitiveCallsIn(ep);
foreach (var epc in candidates.Where(c => !analyzed.Contains(c))) {
    entryPoints.Add(epc); // optimistic add
    AnalyzeTransitiveCallsIn(epc);
}

var includedInSST = entryPoints ++ analyzed; // union

function AnalyzeTransitiveCallsIn(md) {
   analyzed.Add(md);

   md.ContainedInvocations.foreach(inv => {
      if(inv.DeclaredInOtherType) return;
      if (analyzed.Contains(inv)) {
         var isSimpleRecursion = inv.Equals(md);
         var isOnlyCandidate = entryPoints.Contains(inv)
               && !inv.IsDeclaredInParentHierarchy
         if (isOnlyCandidate && !isSimpleRecursion) {
            entryPoints.Remove(md); // false optimism
         }
      } else {
         AnalyzeTransitiveCallsIn(inv);
      }
   })
}
```

(a) Pseudo Code for Entry Point Identification

The intention was to start analyses at these entry points and to allow a full coverage of the reusable part of the analyzed source code. We implemented an analysis that identifies entry points and transitive declarations and sketch it in Figure 5.2a.

The algorithms starts by identifying all entry point candidates of the current type in Line 1. Abstract, private, or internal method declarations either have an empty body (`abstract`) or cannot be accessed from outside of the class (`private`) or assembly (`internal`). These declarations are project specific or do not provide reusable context information, thus, we filter them from the set of candidates. From the remaining declarations, we mark all those as definitive entry points that are either originally defined in the parent hierarchy (Line 4) or that are constructors (Line 5&6). All remaining method declarations are considered entry point candidates at this point.

The algorithm now starts to follow the control flow within all definitive entry points to find transitive invocations of methods declared in the current class (Line 10). We remember analyzed declarations to prevent an uncontrolled recursion in case of recursive method calls (Line 19). The algorithm checks all invocations that are included in the current declaration for three cases. First, invocations of methods that are declared in different type are irrelevant for the analysis and can be ignored (Line 22) Second, if a method is already analyzed (Line 23), it does not need to be re-analyzed, but recursion needs to be handled. In these cases, we want to allow methods with simple recursion, i.e., when a method is calling itself, to be entry points, because these cases are easy to detect in later analyses on the SST. However, we remove the entry point flag from co-recursive methods that were optimistically added as entry points, but that were already analyzed (Line 28). This basic handling breaks if two definitive entry points are co-recursive, but we decided to ignore these cases, because we only found very few occurrences in our dataset. Third, the algorithm steps down into invocations of un-analyzed method that are declared in the current class (Line 31).

Once the transitive calls of all definitive entry points are collected, the algorithm iterates over the remaining unanalyzed candidates (Line 11) and starts to optimistically lifting them to being entry points (Line 12). Common examples of this case are `public` and `static` methods, which are typically important for reuse without implementing an interface. The algorithm then also follows the control flow within these optimistic entry points to find transitive invocations of other local methods.

The algorithm will produce a set of methods that will be included in the SST (`includedInSST`) and a set of methods that will be marked as entry points (`entryPoints`).

Implementation of the Transformation SSTs support JAVA and C#, but only the C# transformation is fully implemented so far. We built a basic JAVA transformation as a proof-of-concept, but it is not yet stable and we did not make it available to others.

Our C# transformation is based on the RESHARPER[4] and it processes all information that RESHARPER's parser is able to retrieve from the source code. It can generate SSTs from the source code of VISUAL STUDIO solutions and is available as a service for other RESHARPER plugins. We have integrated our implementation into two different transformation settings. We provide a non-IDE *bulk transformation* that can transform a complete set of C# solutions to SSTs at once, as well as an *in-IDE transformation* that can transform local source code files, e.g., to capture the source code under edit from VISUAL STUDIO's editor after each change.

The transformation has to handle different challenges in both settings. In the first case, since source code in an editor may be invalid (e.g., missing parts or incorrect syntax), the transformation has to be resilient against broken source code. In addition, the current edit location must be captured from the IDE and included in the SST. In the second case, more than one type needs to be transformed. All type declarations of the solutions need to be identified and transformed to SSTs. We have implemented our transformation to be applicable in both cases. This does not only reduce maintenance cost to a single implementation, but also ensures consistent SST transformations.

Reuse The implementation of the transformation can easily reused by others and we foresee two main use cases: dataset creation and pre-processing for tools.

Dataset Creation Other researchers might want to reuse our *bulk-transformation* to transform their own source code into SSTs. This could be useful, for example, to create a dataset that covers APIs that are not covered by existing datasets. Since our implementation can analyze arbitrary solutions, researchers just have to identify the interesting cases and collect them in a folder, the bulk transformation will find all contained solutions and will transform all contained types into SSTs.

Pre-Processing for Tools Our *in-IDE transformation* is available as a service to other Visual Studio plugins and is useful if the current code under edit should be analyzed. The transformation can then be integrating as a pre-processing step and instead of directly analyzing the source code under edit, it is possible to work on SSTs. This solves many lower-level challenges and recurring code analysis tasks and enables the tool designer to concentrate on the more sophisticated analyses on top.

Limitations Our transformation tool is built on top of the ReSharper SDK. We use it to open the solutions and for the automatic dependency resolution. Unfortunately, it is restricted to Visual Studio solutions and it does not support all possible `C#` project types. This prevents analyses of solutions that make use of the most recent .NET version or Windows Phone projects.

We transform many language constructs from the ReSharper AST into SSTs, but supporting the complete language specification of `C#` was beyond the scope of our immediate goals. We prioritized the constructs and focused on supporting the most common ones, while ignoring less-commonly used ones for the time being. For example, we do not transform the syntactic sugar of textual `LINQ` expressions and we currently ignore the contents of `unsafe` blocks. Unsupported constructs will be included as unknown expressions or unknown statements. We continue the work on the transformation and plan to support missing language features with every new release.

SSTs, by design, enforce the expansion of nested expressions in the source code. For example, it is not possible to nest a method invocation as a parameter of another method invocation.While the normalization of these cases makes it easier to analyze the code, the current transformation only contains a trivial normalization component that simply assigns all nested expressions to new temporary variables. This simple approach might change the semantics of the code, for example, it breaks short-circuit evaluation, because all sub expressions in a boolean expression are assigned to intermediate variables and are, therefore, always evaluated.[1] We acknowledge that this transformation is unsound, but it is not caused by a conceptual limitation of SSTs, but by a limitation of our current implementation that could be fixed by improving and extending the transformation with more engineering effort. In addition, this limitation only affects specific analyses that rely on this information, e.g., data-flow analyses that detect missing null checks.

[1] Consider the boolean expression `e || f()` that the variable reference `e` and the function `f`. In `C#`, the function will only be executed, if `e` is false. The execution will be skipped, otherwise.

5.4 In-IDE Infrastructure for Tools and Research

Having a standardized representation for source code has general benefits, the most obvious in our case is the ability to create an infrastructure that facilitates building shared tools. The goal of this thesis is mainly to facilitate research in this area, and we anticipate two use cases, in which such an infrastructure can help other researchers.

In-IDE Prototyping Ultimately, research results and new software development tools have to be integrated into an IDE to prove the practical value for developers. Every researcher that pursues this integration is confronted with the rough world of a real development environment though. All the corners and bumpiness that are typically ignored in research (e.g., analyzing non-compiling code, testability of IDE components, ambiguous code representation) suddenly need to be handled.

We provide an abstraction layer on top of ReSharper that solves many of these challenges. Our source-code transformation is integrated in this layer and provides access to the SST of the current file under edit. We use this layer ourselves to create the snapshots that are captured in our event stream, but the implementation can be reused by others. It makes it easier to write static analyses, to integrate new ideas, and to prototype new systems based on SSTs.

Coplien and Beck [33] define three steps in the lifecycle of a program: first the developer should make it work, then it should be refactored to make it right, and finally it should be optimized to make it fast. For research prototypes, it is usually possible to stop after the first step, because this state is sufficient to evaluate the new tool. The subsequent steps are pure engineering tasks.

Offline Research Once the usefulness of a new kind of tool is proven in practice, the next task for research is a continuous improvement of the tool, e.g., by improving or exchanging the underlying algorithms or through fine-tuning parameters. This kind of work is usually done in an experimental setup using a benchmark that enables a comparison. Such benchmarks need to be established for each specific kind of tool.

Through building approaches on top of SSTs, it is possible to achieve a clear separation of the general and the approach-specific data representation. This provides the means to create reusable components and makes it easier to share datasets, to reproduce results, and to reuse existing implementations. All of this facilitates the creation of a reusable experimental setup and reusable benchmarks.

Overall, the representation provides the required infrastructure that allows conducting research on RSSE topics. The value of such an infrastructure increases with the number of participants. Once people start using the infrastructure and provide reusable components, synergy effect appear: It becomes easier to compare new algorithms to existing approaches, reuse established techniques in novel approaches, and also the design and implementation of novel kinds of tools becomes easier as SSTs provide information that is not tailored to a specific kind of tool.

5.5 Enabling a Distributed Data Collection

It is not enough to provide the client side that captures the interactions and manages the event generation, it also requires a substantial amount of infrastructure that enables the data collection on a large scale and tooling to preprocess the data into a usable form to provide a dataset that can be used by researchers.

Tool Distribution To make the installation of FeedBaG++ as easy as possible, we publish our releases in the ReSharper gallery, the official repository for extensions to ReSharper[7]. The repository is preconfigured in every ReSharper installation so all that is required for the installation is searching for the tool in the extension manager.

To reduce the required overhead for releases, we have our build server automatically publish all successful builds of the *master* branch of our repository that pass all tests. To avoid overly frequent releases and to have time for intensive manual testing before the release, we develop on feature branches and aggregate several features on a *development* branch, before merging the changes back to master.

We also provide the means to create pre-release versions of FeedBaG++. These pre-releases can be used by customizers, e.g., other researchers, that want to add new functionality to the tracker and want to ship these version with their own clients. We enable the creation of local releases that can be distributed in a local gallery.

Data Collection Each participant collects data locally, but eventually it is uploaded to a server on which all feedback gets aggregated. For this, we needed to provide a server that is publicly available so it can accept all uploaded feedback. Being publicly reachable over the internet makes our server a potential target for hackers that send malicious data instead of uploading the .zip archives that are generated by FeedBaG++. Common guidelines for web services recommend to validate the contents of the uploads, e.g., by validating the schema of the uploaded data.[42] In our case, the server accepts file uploads, which prevents a direct parsing of the message and requires a more elaborate validation of the file and its contents. Our implementation ensures that the file is a valid .zip archive, it then extracts all contents and tries to parse each file as JSON. All files that cannot be parsed are ignored. Afterwards, the JSON is re-serialized and added to a new output .zip archive. This approach assures that the stored .zip archives can be used without hesitation.

Post-processing The uploaded .zip files that contain all shared events are stored on the server. The data is stored in a raw form and cannot be directly used in analyses. Before it can be published or used by researchers, it is necessary to treat the data. We have implemented some preprocessing tools that execute the following tasks.

In such a complex setup, it is possible that data gets scrambled or that the serialization on the client broke the JSON file. Such invalid events should be removed from the dataset. In the same way, it is possible that clients upload events twice, because an upload failed and the client had to retry it. Those repeated uploads lead to duplication of the data that can be detected and, as a result, should be removed.

On the server, the data is organized by upload. For a researcher, this organization is meaningless and the uploaded events should be sorted first. Our implemented preprocessing does two things. First it sorts all events by user, such that all events of the same user are contained in a single .zip archive. Second, all events are ordered chronologically, such that subsequent events can never happen in reversed order, an assumption that does not hold in the raw data, for technical reasons.

A last step in the post-processing applies several fixes for broken information in the data structure. For example, in the course of the project, we identified several cases, for which the name encoding of code elements was incorrect. Of course, such a bug must be fixed in the released code first, to prevent further generation of invalid data, but sometimes the bugs can be fixed after the fact. In this case, repairing the representation in the post-processing can prevent an invalidation of the old data.

All parts of the infrastructure that are required to deploy the interaction tracker are publicly available. Researchers can deploy their own server, point their clients to their own server address, and collect a separate dataset. By using our post-processing tooling, the data can be brought to a form that is ready to be used in further analyses.

Chapter Summary

This chapter has introduced the basic infrastructure that we have created to enable the creation of general components around enriched event streams and simplified syntax trees. At its core, this infrastructure consists of the implementation of the meta models for enriched event streams and simplified syntax trees and a serialization component for both. In addition, we have presented the reusable transformation that we use to transform source code to simplified syntax trees, our in-IDE infrastructure that we have created to facilitate tool creation and to support other researchers. Finally, we have introduced the service platform that we have build to enable a distributed collection of interaction data.

6 Interaction Tracking in the Development Environment

To facilitate empirical studies on developers, standardized datasets and better tools for the studies are required. We introduced a general data representation in the last chapters that can be used to store the information. In addition, to create a dataset in this representation, an interaction tracker must be implemented that can be deployed with participants to capture event data from actual IDE usage.

A tool like this should be provided for a mainstream IDE and programming language to achieve results that are representative for the current work practice in the field. Originally, we considered various IDEs, but made the decision for VISUAL STUDIO and C# as a result of an industrial collaboration. When the project started, we considered two alternatives for the implementation: Project Roslyn and ReSharper.

RESHARPER is a very popular extension for VISUAL STUDIO that adds many convenient development tools and analyses. It is extensible with custom plugins and contains an abstraction layer tat makes it possible to support multiple VISUAL STUDIO versions without spending additional effort.

PROJECT ROSLYN is an initiative by Microsoft to open the internal APIs and allow access to the language model from within VISUAL STUDIO. Microsoft has also recently open-sourced their compiler and runtime of C#, making it a very good candidate for realizing this thesis. However, the technologies were not yet matured when this project started, and RESHARPER guaranteed future support, so we decided to base the tooling on this technology to reduce risk. For simplicity, when we talk about VISUAL STUDIO in the rest of the thesis, we always mean the combination of VISUAL STUDIO and RESHARPER, because our interaction tracker needs RESHARPER to work.

To the best of our knowledge the only other tool that captures information comparable to enriched event streams is EPICEA by Dias et all [44, 46] that captures a source-code change history and invocations of refactoring commands from Smalltalk developers. However, it stops at code changes. Other related works stop at capturing information at even higher level and typically limit the events to process information (e.g., [156]), or to a very specific subset of the IDE usage (e.g., [13]). We had the goal to create a tool that supersedes all previous tools in this area. We wanted to capture a large portion of the development process in a mainstream IDE, because research currently lacks such information. The industrial interest in our work on VISUAL STUDIO has demonstrated the practical usefulness of the endeavor.

In this chapter, we will present FEEDBAG++, a general-purpose interaction tracker for VISUAL STUDIO that captures developers' interactions with their IDE. FEEDBAG++ is publicly available and can be installed as a plugin for RESHARPER. The tool represents

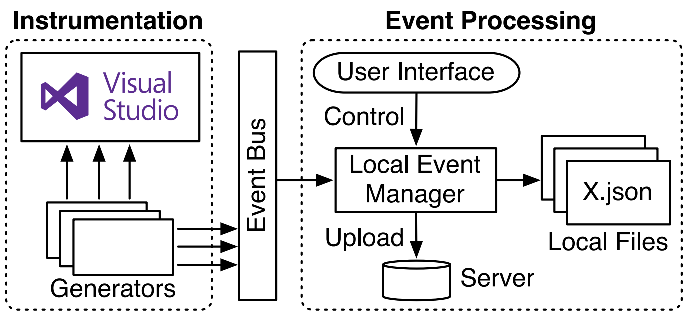


Figure 6.1: Workflow of Event Generation in FEEDBAG++

a general solution for capturing interaction data and is designed to cater for different use cases. It can be deployed in controlled experiments in a minimalistic release that just contains the instrumentation and stores all captured data locally, but we also provide extensions for other settings (e.g., field studies), and provide tooling for these scenarios. For example, infrastructure for reviewing and sharing collected data.

Other options exist. If used in a more restrictive setting (e.g., in industrial collaborations) it might get necessary to enable additional privacy options like additional anonymization that scrambles sensitive information. If the study takes place in a setting that depends on voluntary participation (e.g., in field studies that involve open source developers), the researcher might want to offer additional incentives for participation like an intelligent code completion. All these extensions are supported by FEEDBAG++ and can be configured as needed.

In the remainder of this chapter, we will cover all parts of the project that are related to the implementation of the interaction tracker. We will start by providing details about the instrumentation and the event generation. We will then present the infrastructure we provide for the data collection, both in terms of distribution and event tooling. Afterwards, we will talk about design decisions of the implementation that are related to privacy.

6.1 Event Generation

Once installed, our interaction tracker runs transparently in the background. It monitors the developers' actions, captures enriched events, and does not require any explicit interaction from the developer. The capabilities of the interaction tracker evolved over time. In its first iteration, the tool was called FEEDBAG and captured only command invocations, rather than enriched event streams [3, 4]. Over the course of the project, the tracker evolved and we added support for all event types that were presented in Chapter 3. It was renamed to FEEDBAG++ to reflect the significant update.

This section will provide details about the design and the implementation of the event generation. The data is collected locally and users are asked to regularly upload the collected data to a server.

6.1.1 Design

We designed a workflow for the creation and management of the event stream in FeedBaG++, which is illustrated in Figure 6.1. This workflow consists of three parts: instrumentation, event processing, and the event bus.

Instrumentation The instrumentation of the IDE represents the part in which the actual events are captured. Visual Studio and ReSharper both offer APIs that can be used to access information about the current state of the IDE. The researcher performing the instrumentation needs to identify the information they are looking for and accordingly decide on which part of the APIs is relevant to use.

Event Processing The event processing part is responsible for the local management of events. More specifically, it is responsible for the local storage and the upload of the collected event stream. It also contains a user interface that provides the possibility to the user to interact with FeedBaG++.

Event Bus The event bus decouples the two other parts of the workflow. Events are published by the instrumentation and consumed by the event processing, without having references to each other. The bus is implemented asynchronously, which help processing the events without blocking the user interface.

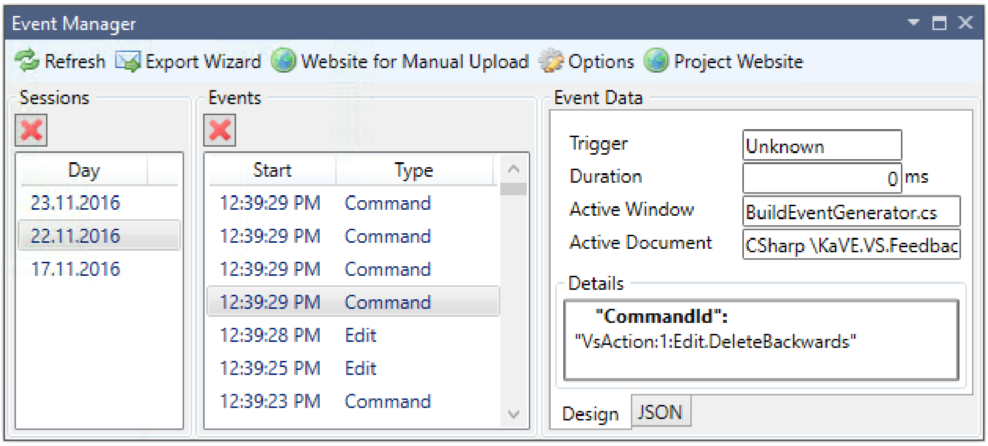
In the following, we will discuss the individual components of these parts and explain how they play together in the data collection.

Generators To allow FeedBaG++ to receive and process any collected data from the instrumentation, it is necessary to implement a new *generator* that is automatically instantiated every time Visual Studio is launched. The first step is to register the generator as a listener for relevant activities (e.g., menu clicks) and react accordingly when they occur. Different kinds of generators exist that obtain their information from different sources. We will cover the instrumentation in more detail in the following subsection 6.1.2. In a final step, an event has to be initialized that can store the captured information, which is then published on the *Event Bus* for further processing.

FeedBaG++ already captures all events presented in Chapter 3, i.e., all invoked commands, activity of the developer (e.g., mouse movement), usage of IntelliSense, build events, debugger events, actions in a document (e.g., open or save), source code edits, usage of the search tool, IDE state information (e.g., startup), solution and file management (e.g., add a new project), window events (e.g., moves), version control, navigation (e.g., ctrl-click), system events (e.g., screen lock), and test runs.

Local Event Management The *Local Event Manager* consumes all events that are published on the event bus. Each incoming event will be serialized as JSON and stored in files on the local drive of the developer's machine. While collecting new information requires updating the instrumentation, there is no need to update the event processing part to handle any newly created event types in the enriched event stream, as long as they extend the base event class.

The local event manager also provides the interface for the user interface that allows reviewing the collected data. It is possible to request events for specific days and to issue deletions. If the user decides to share the collected data, the local event manager executes this task and uploads the events to the configured server address.



Day
23.11.2016
22.11.2016
17.11.2016

Start	Type
12:39:29 PM	Command
12:39:29 PM	Command
12:39:29 PM	Command
12:39:29 PM	Command
12:39:28 PM	Edit
12:39:25 PM	Edit
12:39:23 PM	Command

Trigger: Unknown
Duration: 0 ms
Active Window: BuildEventGenerator.cs
Active Document: CSharp \KaVE.VS.Feedbac

Details

"CommandId":
"VsAction:1:Edit.DeleteBackwards"

Figure 6.2: Event Manager UI

User Interface The main tool we provide users to interact with FEEDBAG++ is the *Event Manager* that is shown in Figure 6.2. This tool provides access to all actions that are offered by the interaction tracker, i.e., configuring the tool, initiating uploads, or accessing websites that provide further information. Its most important functionality of the tool is the option to review the collected events though.

In the review panel, the data is organized in three levels. On the highest level, events are grouped by day to indicate several programming sessions. The events can be filtered to specific days by selecting entries in this list. As a result, the event list in the middle will contain a list contains the time at which the events were created and the corresponding type of each event. On this level, each event can be selected to review all details that have been captured. We provide two views on the contained information: a *design view* that contains the information in a polished form that is easier to read and a *JSON view* that shows the raw data that is stored on the hard drive. While it is impossible to change any information in an event with the event manager, participants can remove events or even complete days if they are not willing to share specific information.

We do not assume that our users keep the *Event Manager* window open the whole time. So, in addition, FEEDBAG++ will regularly remind users to share their collected events or to review them in the *Event Manager*.

Upload Whenever users decide to share their locally collected events, they can initiate an upload to a central server. To make the upload more efficient for our participants, we aggregate uploaded events and compress the data before the upload, to make it smaller and the upload faster.

In the beginning, we included all available events in a single archive that was uploaded at once. However, this approach did not scale well, as the files grew to multiple hundred megabytes in size. Participants on slow connections had to wait up to several hours for their upload to finish. In the end, we split up the upload into smaller parts and only uploaded 1,000 events at once. This change significantly reduced the required upload times per file and the rate of failed or aborted uploads.

While events will be uploaded to the KaVE server by default, the server address can be freely configured, which allows other researchers to collect their own datasets.

Maturity Overall, the implementation of the client side of the interaction tracker was a big challenge. The nature of the interaction tracker makes is different to the typical research prototype. The main difference is the maturity of the implementation of such a system that places a completely different set of requirements on the implementation.

Typical software that is build for research is only executed by the researcher that runs the experiment, who is willing to repeat experiments after the program crashes or who accepts any inconvenience introduced by the tooling. This is completely different for FeedBaG++ that is used by many participants in a distributed setting. These participants use the tool during regular work phases and the data collection should be as non-invasive as possible. Any crash or inconvenience they experience because of the tool will lead to a bias in the collected data, or, in a field study, them uninstalling the interaction tracker and us loosing a participant. Because of this, we had to assure that the quality and stability of our implementation far exceeds those of typical research prototypes.

To achieve this increased quality of the code in different ways. From the very beginning of the project, we put an emphasis on test-driven development of features. In total, we created 6,557 Tests. Many of the generators had to be heavily intertwined with the Visual Studio runtime though and could only be tested manually, but we still achieve a statement coverage of 75%. All tests are automatically executed on our build server, after each commit.

Especially in the beginning of the project, the two main contributors of the project made extensive use of pair-programming sessions and code reviews to ensure an agreement on coding practices and to transfer knowledge. Once common guidelines and practices have been identified, this practice got less important during the project and the extensive reviews were only conducted with students and research assistants.

6.1.2 Implementing Generators

A large part of the interaction tracker implementation is the instrumentation logic that intercepts many in-IDE events to generate the data. In this section, we will provide more information about their implementations.

API Based Most instrumentations rely on the exposed APIs of Visual Studio and ReSharper, or any other API available in Visual Studio's runtime that offers relevant information. During the start-up, FeedBaG++ registers the generators with those APIs to get notified about relevant events. Once the event occurs, the generator will emit an event to the event stream that contains all relevant context information.

Non-IDE Events Some instrumentations were more complex than a simple hook into exposed APIs and had to be solved by intercepting events that happen outside of the IDE. The two most sophisticated examples are version control events and events that generate snapshots (i.e., code completions and edits).

Out of the magnitude of possible version control systems, we picked Git as the most popular system today. Powerful Git integrations exist for Visual Studio, but many developers prefer the command-line util or other external tools for the version control. As a result, version control events can happen outside of the IDE, even while the IDE is not running. To capture this kind of information, we realized the instrumentation by searching for the folder in which Git stores its meta data and -if existent- by listening for changes to it on the file system level. Git has a built-in safety mechanism that prevents data loss, called the *reflog*, which contains all changes to the tree. We leverage the existence of this information and extract the executed activities from it. We remember the last activity that was reported in a version control event and accumulate all changes we have seen since when creating a new event. The beauty of this solution is that it captures both the in-IDE and non-IDE GIT actions at once.

Code Snaphots Code completion events (and edit events for that matter) are the second example of a complex instrumentation. While the APIs of VISUAL STUDIO and RESHARPER allow registering for edits and invocations of the code completion, the biggest part of the events are the snapshots of the editor. We had to access the language model to get access to the source code in a type-resolved form, and apply our transformation (see Section 5.3) to create the snapshot in our intermediate representation. For edit events, this is not always possible due to technical limitations (i.e., the parsing of the document in question is not always done at the point in time at which the event is created). For code completion events, we hook into the invocation of the code-completion tool INTELLISENSE. Creating the snapshot works more reliably here, as the document is already parsed when the tool is invoked. Together, the snapshots contained in both edit events and code completion events create a fine-grained history of the source code that is currently shown in the editor.

Extensions While we already capture the wide range of events introduced in Chapter 3, researchers may be interested in capturing additional events from the IDE. Given the workflow and design introduced in the last section, the event stream can be extended in two ways: First, it is possible to add new event types that define their own specific context. Second, the context of existing event types can be extended with additional specific information. To illustrate both ways, we discuss how we added an extension to FEEDBAG++ that captures navigation information. More specifically, we wanted to know when developers use *control-click* to navigate the code base.

The first step is to create a new event type. We created a specialized event and added a generator that intercepts any control-clicks in the editor window. Each time such a navigation takes place, the generator would instantiate an event, fill in the basic information (e.g., the time), and publish the event. Capturing this event type allowed us to analyze how often this kind of navigation is used in practice. We soon realized that, to understand the data, more context information about the navigation is required. Now that we already had the event type in place, we addressed this by extending the existing generator to capture the additional data and by adding the more specific context information to the event type, namely the *fully-qualified name* of both the current location and the target location to the event.

6.2 Privacy

Researcher are interested to capture as many details about the development process as possible as this helps in later analyses. This usually does not pose problems for controlled experiments, because participants expect to be observed. However, it clearly affects field studies, in which developers are naturally working on arbitrary projects and might even use their computer for private tasks. Being too eager in capturing information in such a context increases the skepticism of the participants and might even keep some of them from participating at all. The privacy aspect of such a data collection has to be carefully considered and it is required to critically reflect on every information whether its collection is necessary.

The idea of thesis motivated an industry project that was executed in collaboration with the IT department of a large German company. The collaboration involved a data collection from several employees of the industry partner. To collect this data, we also had to request permission from the privacy council of the company and had to make our data collection compliant to German privacy laws, We had to reason about data scarcity, account for adequate anonymization of the data, and to provide means to the participant to review and accept all data they share. Even for the open-source developers that participated voluntarily, we needed to make sure that they are aware of the data they provide and we needed to prevent accidental data uploads.

In this section, we will cover our considerations and actions regarding privacy. We will introduce the degree of anonymity we can guarantee by design and introduce additional options we offer for our users to further increase their anonymity. We then present how we capture anonymous user profiles that allow us to create demographic statistics. Finally, we will present how the provided feedback "flows" in our system.

6.2.1 Anonymization

Many information that developers leave in their interaction trace are generic, but some information are very personal and contain information about the individual (e.g., absolute file names may contain the user name). The intention was not to spy on our users, but to gain insights into how the average developer uses an IDE.

A study by Sheth et al. [220] about privacy concerns across North America, Europe, and Asia has revealed that privacy concerns are universally important. According to the work, users and developers perceive both content and interactions as protection-worthy. Even though the study has shown that some users would give up privacy for more functionality, this would introduce a bias in our data collection that we try to avoid. As we capture data in both categories, our only option is to provide anonymization mechanisms to convince skeptical users.

In addition, some developers are less willing to share information than others. We catered for the needs of several potential users types and provided a generally high level of anonymity, but also allowed several additional anonymization options.

Development Sessions Research typically aggregates related events in *sessions* [200], short phases of work, in which the developer was working on a task. To support this concept, we add a *session ID* that is added to every event to indicate the session, in which the event was created. The *session ID* is a random identifier that is automatically generated on IDE start and valid as long as the IDE is open. This makes it possible to differentiate cases, in which developers are working in two separate VISUAL STUDIO instances in parallel, e.g., when working on two different solutions. This had to be handled, because the events of both IDEs are mixed in the recorded stream.

Originally, we also allowed participants to remove the session ID on upload if they are not willing to share this information. This made the data consolidation on the server significantly harder. We removed it in the course of the project, because we only saw few people making use of this feature.

Mutable Event Stream The data collection happens transparent in the background and we do not allow the participant to edit details of captured events. However, we allow participants to delete events from the event stream for two reasons: First, we cannot prevent users to delete events on the file system level. Second, we definitively want to prevent users from regularly resetting the whole installation, just because they accidentally collected information they do not want to share, because this would also affect information we would be receiving, otherwise. The worst case would be that users stop participating and that they uninstall FeedBaG++, a decision that we definitively want to avoid.

If developers are unwilling to share a specific period of their time. This can happen for multiple reasons. For example, a developer might evaluate a new library, which results in some ugly "hacking". The developer might feel uncomfortable sharing these interactions with somebody else. Another example is a developer that accidentally works on a piece of source code that should stay non-disclosed while having FeedBaG++ installed. The developer wants to remove this period from the event stream to avoid any leak of private data.

Allowing deletions is a decision that was motivated by the main usage scenario of a field study. It might be an interesting option to disable this option in other specialized versions of FeedBaG++ that could be used in controlled experiments. In this case, it might even be a valid option to completely prohibit the access to the event manager.

Content Anonymization The event streams holds context data that describes the event. Some of this context data could potentially be sensitive. For example, any absolute file reference might contain the user name. We want to prevent capturing sensitive information in the event stream and we pursued two strategies. First, we apply the principle of data scarcity and only store the necessary information.[20] Related to the file name example, this means that we only store relative file names, relative to the solution file. This way, only information that would be checked into a repository are captured. Second, some context information about the event is optional, at least to some extent, and we provide additional anonymization options to our users for these content information. For example, events store their creation time and duration. We provide the option to remove this information from the event. This still allows us to capture helpful information like code-snapshots through the events, even though we do no longer know when they happened.

Another example relates to our source code snapshots. Part of these snapshots is the naming scheme that is used to point to the object oriented code elements. Most information points to reusable information, i.e., names of interfaces or base methods, but some names point to elements that are defined in the local project. These local names are included to keep a consistent view on the code base, but they do not provide reusable information as they cannot be found in any other code base and would be ignored in for example pattern identification algorithms. If users are not willing to share these local names, e.e., because they do not want to share details about their code base, they can anonymize local parts of the naming scheme. This means that all parts that do not point to local information are hashed.

The hashing is done in a smart way that keeps the naming scheme intact, but that replaces the sensitive parts with the hash. For example, consider the (simplified) field name "`[p:int] [T, MyProject]._f`" that points to the field `_f` with the predefined type `int`. It is declared in the local type `T` declared in `MyProject`. This field name contains two local parts: a) the defining type of the field (`T`) is a local type (defined in `MyProject`) b) the field is called `_f`. Both information are specific to the local implementation and do not provide reusable information. On the other hand, the type of the field (`int`) is defined in the core library, so we do not hash it. As a result, the hashed version of the previous name is "`[int] [#(T), #(MyProject)], #(_f)`".[1]

In summary, we provide several anonymization options for the captured events. Some are incorporated into the general design, others are provided as optional anonymization steps that can be activated separately. Overall, we don't know how many people removed events from their streams, but we can identify, how many of them activated anonymization options and we see that only a minority of our participants actually enabled anonymization options. Even though this could be interpreted as a sign that the provided anonymization options were unnecessary, we are confident that they help to increase the trust that participants place in our system. We did not spent time with further investigation of the anonymization. While it seems to be an interesting task for future work to optimize the number of participants while reducing the ratio of anonymized content, this was out of scope for this work.

6.2.2 User Profiles

We have introduced user profiles to make to ease the analysis of the event stream and to collect some demographic information about our users. User Profiles and their contents have already been discussed as part of the event stream. We have presented how we realized the user profiles in Section 3.2.6. In this section, we will discuss the effect of user profiles on privacy and explain how we use the data. User profiles store two kinds of information: *demographic information* about the user and a *profile id*. Both will be discussed separately in the following.

The *demographic information* we ask our users to provide covers personal areas like education, current occupancy, or programming skills. We explicitly ask our users to provide the information during the first upload process, but we make it clear that providing the data is optional. We provide predefined answers to all questions, and users always have the option to select "I don't want to answer" in all categories. While the demographic information are personal information about the user, it is anonymously captured and does not allow us to trace the feedback back to an individual. We keep the number of personal questions short and only ask about information that are directly related to the analysis of the collected data. Our questions are in line with other questionnaires in the area.

As part of a user profile, we also introduced a *profile id*, which goes beyond the concept of sessions that had been discussed before. To understand why we need them, it makes sense to reflect on the project's history. In the beginning we did not have user profiles. When analyzing collected data, we had the problem that our anonymization concept made it very hard to reason about the participants. For example, it was not possible to judge the diversity of our participants. We could not even tell, how many participants shared data with us. This information is crucial for statistical test though and we needed a way to group related events of the same user. To solve this, we introduced some heuristics to group related uploads. The technique was simple, we just compiled a list of all *session ids* that were contained in each uploaded archive. These lists typically overlap in uploads from the same development machine, so we could merge several subsequent uploads. However, several cases could not be handled

[1] To increase readability, we indicate the hashing with "`#(...)`". In the implementation, we would calculate the `MD5` hash for the value and would use this hash in the naming instead.

by this approach. For one, it is impossible to group uploads by a same users, if she works on more than one machine, because no overlap will exist ever. Second, a reset of the installation or a removal of either the first or the last day of the collected data breaks the grouping mechanism we loose the connection. To overcome this limitation, we have introduced user profiles and assign a randomly generated *profile id* to each user during the first upload process. We decided to make this id freely editable, both to increase trust and to make it easily possible to use the same *profile id* on multiple machines or when migrating to a new one. We also allow users to completely avoid it, if they do not want to associate the captured activities with a user profile.

The *profile id* can be used as an identifying property in the merging algorithm of the archives, in addition to the existing *session ids*. It enables the handling of multiple critical user actions like resetting the plugin or removing all collected feedback. Whether users provide the information and distribute them among their machines is up to them though. Should users decide not to have a *profile id* or not to synchronize it between multiple machines, the concept does not provide a benefit and it suffers from the same limitations of the system before. However, we clearly state during the upload that provided data can only be removed, if a user profile id is provided. Users should have a natural interest in creating a *profile id*, noting it down, and using it consistently. In our uploads, we see that most participants provide this information.

The only way to overcome the identification problem and to guarantee the identity of a participant is to have a central service that is used for authentication. We would require users to *login* to the service before they can upload data. However, the required implementation effort and especially the added inconvenience for first time users who would first need to *register* with the service let us decide against this idea. Future work should revisit this idea though as a centralized login would have additional advantages like the opportunity to build a community around the data collections. It enables personalized dashboards with usage statistics like CODEALIKE[22] or to add gamification elements in the form of leaderboards to the data collection. Applications like this would provide an excellent opportunity to attract new developers and to further increase the number of participants in the data collection.

6.2.3 Event Creation and Flow

The data is automatically collected as soon as FEEDBAG++ is installed. This however represents only a small part of the workflow that is involved in the flow of the events across the whole system from the event creation until their storage in a database. In this section, we will present the whole data flow that is depicted in Figure 6.3 and discuss the details of the upload process.

The developers are the core element of the whole process. Their use of VISUAL STUDIO is tracked by FEEDBAG++ and generates the event stream. All generated data is stored locally in plain text files on the hard drive of the developer. The files contain a JSON encoded representation of the events and are fully accessible by the developer.

We have created the *Event Manager* to make the data collection as transparent for the participants as possible (see Figure 6.2 for a screenshot). It allows a review of the collected data and a selection of data that should not be shared. To this end, it provides an improved visualization of the event contents in a specialized form, which is more comprehensible than just reading the file contents. The *Event Manager* is also the tool that participants use to share their data. A popup regularly reminds the participants to upload the collected data. Once the upload process is started, two

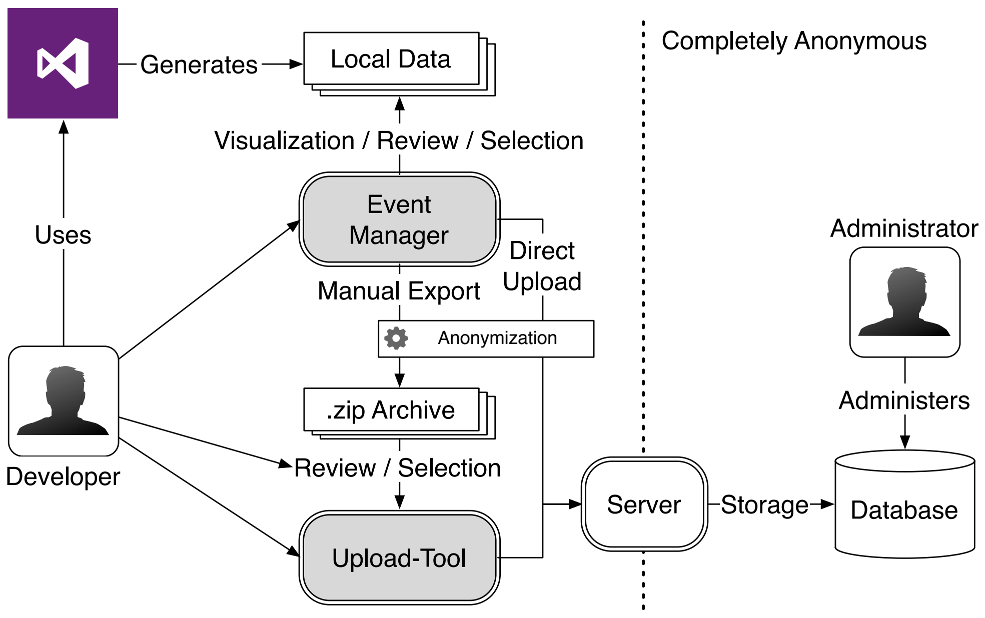


Figure 6.3: Flow and Accessibility of the Generated Event Stream

ways exist in which the participant can proceed: *direct upload* and *manual export*.

The *direct upload* is the upload option that is most convenient for the user. The captured events are compiled in an archive, compressed, and uploaded to the central server through a web service that accepts file uploads. No further interaction is required by the user the upload window can even be send to the background. To increase the reliability of the upload, the archive is transparently split into several small uploads that are processed sequentially.

The *manual export*, on the other hand, requires several manual steps from the developer, but it provides the possibility to review the exported data, before it is uploaded to the server. Once selected, all captured events are compiled into a single .zip archive that developers are asked to save on their machines. From there, it is possible to extract the contents and review all contents that are to be uploaded. It is also possible to remove files from the .zip file that should not be shared. Once this manual review process is finished, developers can access an upload page on our server that allows uploading the formerly exported archive.

An important part of the upload process is the optional data anonymization. The IDE instrumentation stores all captured information in the local files and does not consider any anonymization settings. The application of the anonymization is part of the export and is the same for both export options. Once the events are exported or uploaded, they are deleted locally and developers can no longer access them. The server stores the uploads in a temporary database, to which only the administrators of the KaVE project have access. The administrators regularly extract snapshots of this database that are publicly released on the website of the KaVE project.

One challenge in the process was to ensure that participants understand the upload concept and realize when data transitions from their own machine to the server, including the loss of access to the data that comes with it. In addition, we had to make sure that people understand how we intend to use the data and agree to this. More specifically, we had to prevent accidental uploads by users that contain information that are not to be shared. To solve this, we introduced a disclaimer in the sharing process that clearly states the intended use and the limited accessibility. In the disclaimer, we clearly state that we will publish the dataset, use it in scientific publications, and that provided feedback is irrevocable. Developers have to acknowledge this disclaimer, before they can proceed to sharing the collected data. We integrated the disclaimer both in the sharing interface of the event manager and on the website for the manual upload on our server, to ensure that we do not receive any uploads from developers that did not accept the terms.

However, the legal state of the data collection is still unclear. Participants that provide data are distributed over the whole world, it is not clear if this disclaimer is legally binding and complies with the law. It is also not clear which law is applicable, German law or the local law of the participant. Related to this, it is also not yet defined under which license the participant provides the exported data. The disclaimer is the only regulation how the data can be used, so far. These legal questions are essential when establishing a shared dataset, to prevent any future issues and ensure stability. Related work suffered from privacy issues.[43] In contrast to this, our users explicitly provide the data themselves and acknowledge the disclaimer, so we believe that our solution is safe. However, the legal foundation of the data collection was out of scope for this work and still has to be thoroughly investigated in the future.

In summary, it is clear that the data captured by the interaction tracker is highly sensitive. We use anonymization to avoid capturing information that is not important for the analysis of the development process or allow the removal of optional information. To keep the dataset analyzable, we introduced user profiles to capture demographic information in an anonymous way. We also designed the upload workflow in a way that prevents accidental uploads and informs the participants about the intended use of their feedback, before the data changes its ownership.

However, it is impossible to guarantee complete anonymity when so many details about the development process are being captured. Apart form information that might leak because of bugs in the information, evil parties might start to combine the dataset with other external services to match the captured data with actual user profiles in social platforms like GitHub. This is something we cannot control and only prohibit in the license of the datasets.

6.3 Adding Incentives for Participation

FeedBaG++ does not provide a value per-se. Depending on the scenario in which it should be deployed, it is necessary to convince people to actually install the plugin and participate. The measures also depend on the group of people that should participated

Trust A core requirement is that people trust that the tool behaves orderly and according to a scientific conduct. They have to believe that is not trying to spy on them or get access to private information. We solve this by making the design and implementation of the interaction tracker as transparent as possible. The implementation is released as open-source and we provide an informative website that explains both th data that is being collected and the reasoning why we need to collect it.

In addition, by providing tooling such as the event manager, we allow users to review the events they generate, which both facilitates an exploratory understanding of the project and shows that we are not hiding anything. An additional way to earn trust can be achieved indirectly by discussing the results of our work in scientific publications, which underlines the professionalism of our research.

Of course, some developers do not want to participate categorically or cannot participate legally. Some of them were just very skeptical and it is hard to change their minds. Others had to keep their work results non-disclosed. We provide anonymization options for the source code to cover this case, but if the anonymization does not provide enough safety, it was impossible for them to participate.

Awareness To gain users for our tool, it was required to make people aware of the project. Some people believe in the research goals of the project and participate voluntarily the moment they hear about it. This group was the most welcome for us, as all we had to do was to get them to know about the tool and make the installation as easy as possible.

To spread the work about our tool and to attract users, we did a lot of advertisement. One way was to regularly news about the project over social media channels and to discuss the project in personal discussions with related researchers. We also placed advertisements in a magazine that was handed out at a conference to other researchers in the field (i.e., the MSR yearbook) and indirectly advertised the project through the references to FEEDBAG++ that we mentioned in scientific publications.

Added Value Depending on the intended scenario, in which FEEDBAG++ should be deployed, participants have to be recruited that install the interaction tracker and share their usage data. Assuming that people are open-minded, share the research interest, are willing to install the tracker, and that the tracker is stable and non-intrusive in their development activities, many would still avoid the installation as it does not provide any personal benefit. To win this group of participants, it is necessary to provide additional value to users as an incentive to install the tracker.

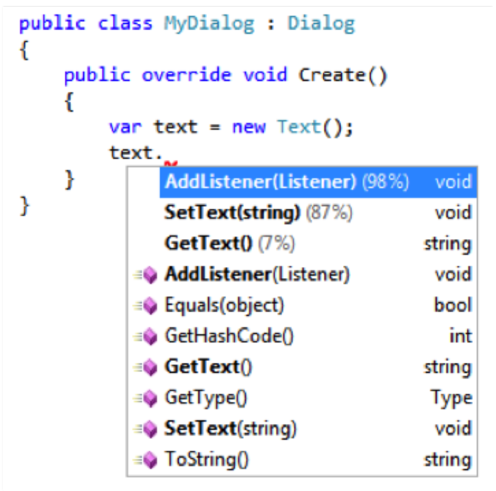
In our discussions with interested developers, we discovered that this argument was by far the most common. Many people would be willing to capture and share interaction data, if the tool would provide any benefit in their daily work. As a result, we spend a considerable amount of time designing and implementing tools that would help us to convince these developers.

In the remainder of this chapter, we will focus on this aspect and will present several approaches we have implemented that provide value to our users.

6.3.1 Intelligent Code Completion

Our first approach of adding value to the tracker was leveraging our own research results. We built an intelligent code completion system that can predict missing method calls on variables [192], even before we started working on FEEDBAG++. Figure 6.4 shows a screenshot of the tool. The obvious idea was to re-implement the system using the new platform and tooling that is introduced in this work and integrate it into FEEDBAG++ to provide additional value for our participants.

For the migration of the tool, several steps had to be implemented. Conceptually, we had to *mine a repository* to learn about *correct API usage*. We then had to preserve the gained knowledge in *models* and create an recommender engine that could use



```
public class MyDialog : Dialog
{
    public override void Create()
    {
        var text = new Text();
        text.
    }
}
```

```
AddListener(Listener) (98%)      void
SetText(string) (87%)            void
GetText() (7%)                   string
AddListener(Listener)            void
Equals(object)                   bool
GetHashCode()                    int
GetText()                        string
GetType()                        Type
SetText(string)                  void
ToString()                       string
```

Figure 6.4: Example of an Intelligent Method Completion

the models to infer meaningful proposals to the developers while working on source code. Both the mining and the querying during development made it also necessary to implement sophisticated *static analyses*. The complete re-implementation is described in Section 10.1.

We integrated the recommender into FEEDBAG++ and created a set of models that are available for developers. Once the developers install these models, they will get intelligent recommendations in their IDE, whenever they invoke code completion on a type that is included in the models.

Offering such a system certainly makes installing FEEDBAG++ more appealing to developers, but providing an intelligent code completion had some effect that we had to consider. The first immediate effect was related to the instrumentation that captures detailed context information about the usage of the code completion. After introducing intelligent code completion, we had to extend this instrumentation such that the captured proposals distinguish regular and intelligent proposals. Capturing this additional information allows interesting analyses of interactions with intelligent proposals, once the system is rolled out on a large scale. However, for now, we were interested in capturing interactions with the code completion that reflect regular usage. Having intelligent code completion introduces a strong bias on the selection that we wanted to avoid.

It is possible to avoid the bias while still offering added value through intelligent completions by randomly disabling the intelligent completion. This strategy would allow to capture both unbiased feedback about the APIs that are not being supported by the recommender and a detailed view on how developers interact with intelligent proposals. The only requirement is that enough users participate and share data.

We considered several options on how the randomization could be implemented in a smart way. A pure randomization is not recommended as the behavior would change in between invocations, but tools should behave consistently and follow the expectations of the user. In addition, depending on the implementation, a pure randomization could easily be tricked by quickly aborting and re-invoking the recommender, a behavior that is undesirable from our point-of-view. The better option was to randomly select some types, for which the completion does not work, while still offering full support for the rest. To guarantee a selection of types that is stable for a specific user, we based the randomization on the profile id. This approach produced the interesting anecdote, that we wanted to provide a "tutorial" on the website that everybody can follow. We had to whitelist the type used in the tutorial to make sure that everybody will receive recommendations for it.

6.3.2 Gamify the Development Process

A relatively new approach to increase motivation, engagement, and dedication of users is *gamification*. While the concept and the underlying ideas have been used before to design interactions [24, 126], Deterding et al. were the first to attempt a scientific definition of the term [43] and define it as "the use of game design elements in non-game contexts" with the goal of a long term effect on user engagement.

Gamification has various forms and mechanics. Typical elements are reward systems (experience, points, money), progress indication, achievements, tutorials, leaderboards, and many others. These elements are introduced to the users with the goal to motivate and engage them, to teach them, or to affect their behavior.

Gamification is heavily used in modern video games, where another form of reward mechanism is established. Players can earn achievements, which often mark specific milestones in the game that are indicators for progress. However, many are unrelated to the direct gameplay and often resemble tasks that require much effort, a high skill, or they force the player to play games in a specific -sometimes unusual- way that might not resemble their regular play style. While the main motivation of the game publishers to introduce such systems is to extend the time players play a game after they have finished the main game, one could also look at them as guidelines that can affect the behavior of players. To earn these achievements, is often required to critically reflect the play style and learn about the game mechanics. This effect is even increased by making the list of earned achievements publicly available in user profiles that allow comparing the gained achievements with other gamers. All these self-reflective effects are also desired in non-gaming contexts.

Previous work has already shown the positive effect of gamification on various processes, like engagement of university students [58], motivation of tourists [256], or knowledge acquisition [155]. More recent work also focused on the analysis of gamification in the domain of software engineering. Snipes et al. analyzed the effect of gamification on the behavior of developers [230]. They provided tools that monitor the developer and that grant points for good behavior (e.g., using shortcuts instead of mouse clicks) or for using good coding practices. These points serve as experience points and participants level up by gathering them. A leaderboard exists that allows comparing the personal score with colleagues. Snipes et al. could show that developers are generally interested in such a gamified system and that rewarding good behavior can influence their behavior.

Another example of gamification in software engineering is CodeAlike[22], an online service that generates personal dash boards and statistics about the IDE usage. The goal of the service is to help developers to better judge their own *performance*. They have created a metric that summarizes the performance of the participant, which is calculated not only through the time participated (e.g., the number of participated days), but also includes several variables that describe the *focus* of the developer, like the length of the work periods or the number of interruptions. Codealike also maintains a global leaderboard, in which developers can compare their performance ranking with each other. The idea is that the peer-pressure in the global leaderboard encourages developers to get more productive by increasing their focus.

For our interaction tracker, we have implemented a solution that combines both previous works. We build an achievement system[2] that consumes events from our enriched event stream directly in-IDE to calculate and maintain statistics about the observed events. A screenshot of the system is shown in Figure 6.5. The user interface shows both the date of accomplishment for all earned achievements, as well as the progress in unearned achievements.

In the current form, the system is designed for a local achievement tracking. More specifically, all statistics are updated *on-the-fly* while the events are generated in the IDE. One part of our achievement system is an overview of the aggregated statistics that are captured, as shown in Figure 6.6. The statistics can be reset, but they cannot be regenerated, as the generated events might have been uploaded in the meantime, which removes them from the access of the developer.

We support different types of achievements and took inspiration from achievement systems in games, like the extensive achievement system used in the very popular hack-n-slay game *Diablo 3* [42]. The achievements can be grouped into several categories.

Simple The basic achievements only have a single criterion that needs to be fulfilled to earn the achievement. Examples of this are “Has used tool X”, “Has worked after 10pm”, or “Did a commit without compiling first”.

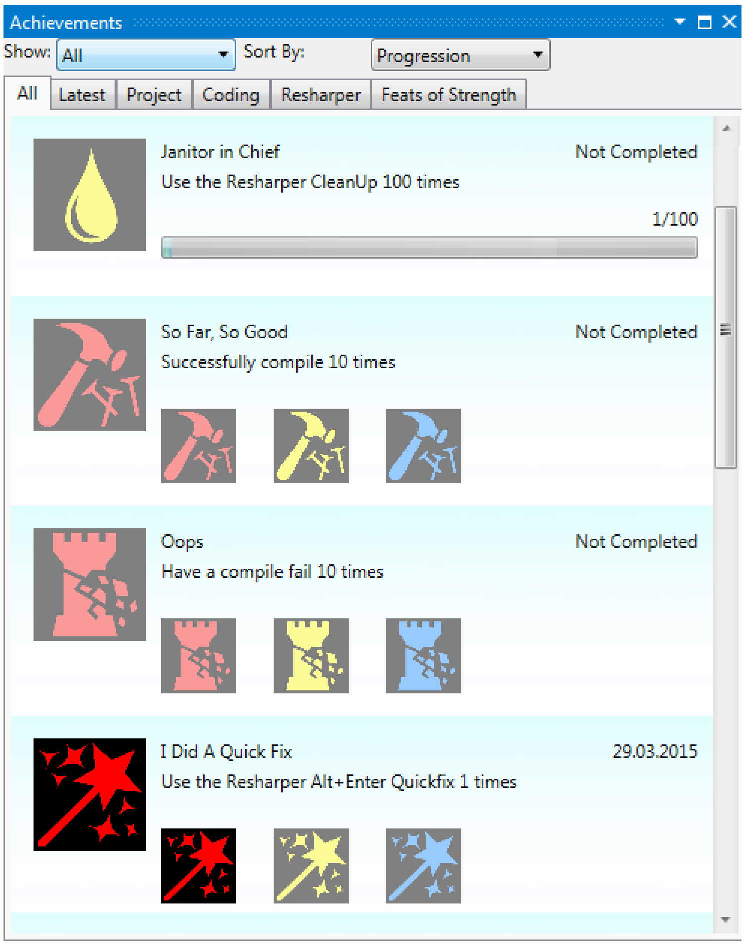
Accumulative Achievements in this category accumulate repeated interactions that are observed in the event stream. The accumulation can accumulate counts or durations. Examples of this category are “Has used the auto-formatter X times”, “has spent Ymin in the code completion window”, “broke the build Z times”.

Staged Some of the *accumulative* achievement have several stages. The different stages are earned separately, but as they are related, the user interface lists them together. A simple example of this category is “Has successfully compiled 1/10/100 times”, while the distinct steps can be earned separately.

All existing achievements of the achievement system can be assigned to these categories. Support for new achievements can be added easily by implementing a new achievement tracker that consumes the event stream. However, extensions will not be discussed here as they are out of scope for this work.

The current implementation only works locally. As a result, the gamification element of the system is only based on encouraging the participants to earn the achievements that we have designed. However, no conceptual limitation exists that prevents us from extending the system to an online platform, which would add the social component. These aforementioned statistics could also be calculated on the server and used to create dashboards and rankings that are similar to Codealike. This would add the peer-pressure of the comparison with colleagues and friends, and leaderboards. A trivial implementation of such a social platform would simply be based on the configured user profiles, but this could easily be rigged. The introduction of such a system would easily justify the introduction of a regular email-based registration system that would be accepted by the participant.

[2] This project was implemented in a *Bachelorpraktikum* by the students Markus Zimmermann, Mattis Kämmerer, Felix Weirich, and Sebastian Kemper.



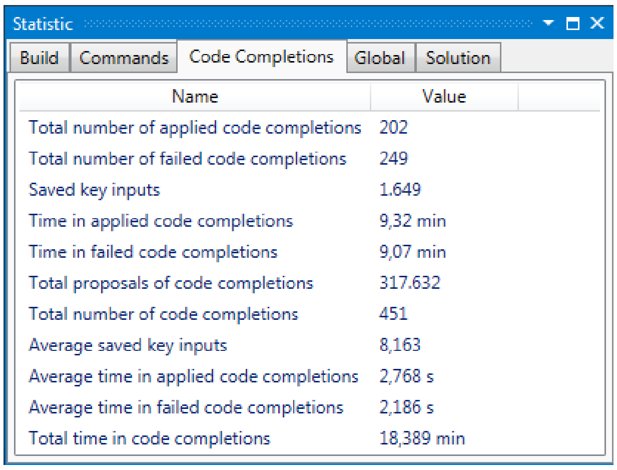
Janitor in Chief
Not Completed
Use the Resharper CleanUp 100 times
1/100

So Far, So Good
Not Completed
Successfully compile 10 times

Oops
Not Completed
Have a compile fail 10 times

I Did A Quick Fix
29.03.2015
Use the Resharper Alt+Enter Quickfix 1 times

Figure 6.5: User Interface of the Achievement System



Name	Value
Total number of applied code completions	202
Total number of failed code completions	249
Saved key inputs	1.649
Time in applied code completions	9,32 min
Time in failed code completions	9,07 min
Total proposals of code completions	317.632
Total number of code completions	451
Average saved key inputs	8,163
Average time in applied code completions	2,768 s
Average time in failed code completions	2,186 s
Total time in code completions	18,389 min

Figure 6.6: Overview of the Maintained Achievement Statistics

6.4 Behavioral Bias

Empirical studies always have to cope with confounding factors, but sometimes the experiment itself is the confounding factor. In this section, we will discuss the effect of having an interaction tracker installed in the IDE.

A famous study at *Hawthorne Works* analyzed the productivity of workers with respect to changes on their work environment (i.e., lighting, work hours, and break times). In the study, an improved performance could be measured after any change, but also that the effect diminishes after some time. Surprisingly the researcher could measure the same positive effect after reverting the change. Many related works analyzed this effect, which is known in the literature as the *Hawthorne effect* [112, 138].

The general consensus is that the observed effects were the result of change, not of the concrete measure itself. The interpretation of the phenomenon differs though and several theories exist that explain why the performance was improved. The original study concluded that the measured performance improved in the study, because workers felt appreciated by management paying attention to their working conditions [134]. Another obvious explanation of the improved performance is a fear of personal consequences, i.e., workers that might get fired in case of a bad performance [175]. Follow up work suggested that the increased productivity could be explained through *novelty effects* in the environment that counteract boredom and increase the attention of workers [30]. Others argue that the participants try to please the experimenter by providing seemingly helpful feedback or by behaving as they think is expected [232].

While the working environment is different between industry workers and software engineers, the general findings of this study can be projected to any empirical study that observes developers. Observed behavior in experiments is always biased, because participants know that they are being monitored. This also affects studies that are conducted with the interaction tracker FeedBaG++ that is introduced in this work.

Effect is Introduced by the Setup The severity of the bias effect depends less on the data collection strategy rather than on the experimental setup. Let's consider two scenarios for the argumentation: a typical short-term controlled experiment and a long-term data collection in a field study.

In the typical controlled experiment subjects are recruited from the direct environment of the researcher, e.g., students or colleagues, and they work on a specific task using a new tool. In a different scenario, the experiment is conducted as a field study. Participants are acquired over social media channels, they work in their regular environment, participate over several weeks, and they do not use a specific tool.

FeedBaG++ can be use in both scenarios to collect data about the participants. The theory says that both experiments are biased, but the extend differs in both cases. In the first scenario, it is very likely, that the observed behavior is biased because of the aforementioned reasons: participants know the researcher and might want to help, they are working with a novel technology and are excited to try it, and they are well aware of being monitored. In the second scenario, the aforementioned effects are likely weakened. Subjects don't know the researcher, their working environment does not change, and while being aware of the monitoring, the duration of the data collection makes it less likely that developers change their behavior.

No novelty effect In addition, both experiments can use the same release of FeedBaG++ for the data collection. In the first scenario, the data collection can be completely transparent as even the upload and review functionality could be disabled. In the second scenario, the basic interaction tracker does not introduce any added value that is novel and that could stimulate excitement. And even if the tracking and upload itself introduces an effect, previous work found that *novelty effects* decay after 8 weeks [30]. We would assume that this effect manifests much earlier, as we do not introduce constant changes to the environment, apart from a daily popup.

No Fear of Consequences In its basic form, we would argue that an experiment with FeedBaG++ is less prone to effects than the *Hawthorne studies*. We argue that the participation on experiments is voluntary, in contrast to the inner-company experiments at Hawthorne, so we do not think that subjects fear personal consequences. In addition, the collected data is anonymous and cannot be traced to an individual.

Added Value Adds Bias This section introduced an intelligent code completion engine and an achievement system that add value to the interaction tracker to attract more users. Unfortunately, both not only add value, but also have a significant effect on the measured data.

Intelligent code completion adds new proposals to the code completion and by reranking and proposing seemingly intelligent proposals heavily influences the decision of the user. The effect might be mitigated to some extend with our strategy of randomized disabling, but a bias cannot be avoided completely.

Gamification and achievement systems naturally influence the behavior of the participant. This is why they are introduced in the first place. Even if the achievement system would be disabled and the system would show just the numbers, there would be an effect, as people will try to game the system and they would try to optimize the numbers we show them.

Most importantly, both represent new tools in the IDE that provide novel functionality. This creates the *novelty effects* that were absent so far. Naturally, users will explore these new tools and try to integrate them in their workflows, which heavily influences the data collection.

In summary, we estimate that the bias introduced in an experiment by using a basic FeedBaG++ version is small. There are no novel features, users can feel safe and anonymous, and the minimal intrusiveness of the data collector lets participants forget about the tooling. FeedBaG++ provides several components that can be enabled on demand, which have an effect on the data collection though. The best configuration depends on the experimental goal. If a huge bias on code completion is irrelevant for an experiment, because it only analyzes command invocations, then the added value might pay off in more participants. If the same experiment would introduce the achievement system, it would smudges the statistics about command usages in the IDE, and it should better stay disabled. Overall, it is always necessary to consider the introduced tradeoff and decide per scenario.

Chapter Summary

This chapter presented several parts that relate to the implementation of the general-purpose interaction tracker FeedBaG++. After motivating the decision to implement the tracker for the development environment Visual Studio, we presented details of our implemented instrumentation. We have then introduced our infrastructure that we built to distribute the tool and the tool we use for the data collection and pre-processing. Afterwards, we discussed several considerations regarding privacy of our participants. We presented considerations about our endeavors to recruit new participants. Finally, we present the implementation from another point of view and discus how it could be used as a platform for development tools.

7 Platform for Composable Analyses and Reusable Transformations

The overall theme of this thesis is to provide better means for research on Recommendation Systems in Software Engineering (RSSE). We specifically focus on source-code-based recommendation systems [139], which are concerned with API usage. This includes, for example, method-call recommenders, snippet recommenders, and code-search engines. The research area has reached a certain level of maturity, while still offering many new research opportunities [205]. It is, therefore, time to reflect on the current practices and to devise more systematic ways for moving forward.

A recent study on replicating results of prior work in the area revealed that many existing approaches do not make their tools and datasets publicly available [206]. This makes the replication and comparison of results harder and also prevents new techniques from easily building on existing ones. From our own experience and from reviewing the state of the art, we have identified two main reasons for this phenomenon: First, there is limited support for comparability of the results due to lack of shared datasets used to train and evaluate different RSSE. Also the lack of a standardized format to share such data hinders this further. Second, there is no support for reusing components that realize repetitive tasks that typically occur in the preprocessing of the input (e.g., finding a set of compilable projects) or during static analyses of source code (e.g., performing a points-to analysis). These tasks are time-consuming and have to be repeated for specific kinds of RSSE, all while not being the novel conceptual contributions of the new recommender system.

In this thesis, we tackle the first point by introducing the data structures *enriched event streams* and *simplified syntax trees* as a means to standardize the representation and to enables the assembly of reusable datasets. However, when proposing a new representation, reusable components are needed, to make it practical. It is crucial that tools are easily applicable and that they are not bound to a specific research question. The standardized representation represents a common infrastructure that builds the foundation for the solution to the second point, the creation of reusable and shared tools. We will motivate this point in the following and will elaborate our endeavors towards a solution.

Typical Workflow Let us assume that a new software engineering task that may benefit from a recommender system has been identified. The toolsmith responsible for designing this recommender must first think of and design a new technique that is appropriate for the problem at hand. This also includes identifying the input data

Input Preparation

1. Collect a set of projects to analyze.
2. Make them compilable.

Static Analysis

3. Configure a static-analysis framework, e.g., WALA.
4. Resolve ambiguities and implicit information (e.g., optional `this` references)
5. Perform "standard" tasks (e.g., a points-to analysis or data-flow tracking)
6. Extract the required input data (e.g., methods called on a particular object).

Tool Building

7. Process the analysis output (e.g., filter corner cases).
8. Build the RSSE's underlying logic (e.g., using machine-learning).

Evaluation

9. Collect a suitable dataset for the evaluation.
10. Evaluate the RSSE's effectiveness.

Figure 7.1: Typical Workflow When Creating RSSE

required to realize the technique. Once these prerequisites are fulfilled, the next step is creating a pipeline that can be used to instantiate and evaluate the new approach. This high-level task consists of several steps that are illustrated in Figure 7.1.

To get the process started, it is necessary to collect an input dataset – the set of software projects to analyze. Further, one of the most challenging tasks is to write a static analysis that extracts the necessary information from the source code. The next step in the workflow is to build the actual tool that provides the value to the developer. In a final step, the value of the tool must be proven in an evaluation. All these tasks contain repetitive steps and provide chances for reuse. In the following, we will elaborate on these chances and on what prevents it right now.

Input Preparation As a first step in the workflow of creating RSSE, researchers need to collect a set of projects, which they can use to train their tool. Several open datasets exist (e.g., [214, 215, 239]), yet, we have seen in our literature review that works on RSSE usually tend to use their own datasets. We can only speculate about the reasons for this and we assume that researchers start with creating their proof-of-concept implementations that are trained on an initial selection of projects that is convenient for them to use. Once their tools works, we further assume that they only focus on their experiments, because integrating an existing dataset requires additional effort and there are usually no incentives to do so.

Preparing new datasets over and over is tedious though and involves a lot of manual effort. The biggest challenges when collecting a dataset is making the projects compile. Most projects that are hosted in public repositories like GITHUB rely on external dependencies. However, some of these dependencies might not be available anymore or have to be installed manually because of licensing issues This often prevents a fully-automated collection of a large-scale input dataset.

Overall, the current situation leads to additional effort for researchers over an over again and also limits comparability between approaches. We see a great chance for improvement by introducing a dataset that is easy to access and integrate such that researchers are motivated in using it right away instead of retrofitting it to their needs.

Static Analysis Once the input dataset is available, the next step is to write a static analysis that extracts knowledge from the source code. Existing frameworks like Soot[5] exist that provide generic components for reuse (e.g., points-to analyses), but these generic parts are usually intertwined with the specific parts of the analyses.

To get started, the analysis framework has to be configured (e.g., selection of a specific points-to analysis technique) and this configuration is typically tied to the approach. An experiment that compares different RSSE should rely on equal configuration for all approaches though (e.g., the same level of field sensitivity in data flow analysis), which can not be guaranteed in these highly-integrated analyses. It gets even worse, if only a binary release is available and no source code. Differing configurations are a confounding factor to experiments and should be eliminated.

We see a great opportunity to improve reusability of analysis parts and for higher quality of these analyses, by further increasing the modularization and composability of these parts. However, this can only be encouraged by design principles and facilitated by a framework through ease of use, but it cannot be enforced.

Tool Building The next step is to build the actual tool, which highly depends on the concrete approach. Typically, approaches integrate existing libraries for statistics or machine learning to build models (e.g., TensorFlow[30] or Mahout[32]) and reuse their existing implementations of common algorithms. Apart from this, we don't see many chances for providing any prebuilt components that could be reused in the building process that would provide any value that goes beyond the current state.

We see a big chance for reusing infrastructure though. A platform could provide generic preprocessing steps (e.g., partitioning of the input dataset for cross folding or selecting subsets) or could provide infrastructure for persisting intermediate results that are used in later steps of the pipeline. This further emphasizes the need of shared data representations and encourages a clear separation of task implementations.

Evaluation The last part in this workflow is also probably the hardest, because designing a proper evaluation is challenging. We saw two main trends for the evaluation of tools. 1) Approaches conduct user studies and measure the time to solution or the correctness of a task result. While these experiments certainly provide valuable insights, they cannot be easily repeated, yet alone compared to other tools after the fact, which is why we ignore them at this point. 2) The second trend is to use a benchmark to measure properties of the tool (e.g., its prediction quality).

For this to work, optimally, a dataset is required that can be used for benchmarking. In practice, authors usually create query scenarios for their tool based on some part of the input and evaluate their tool for example in a cross folding strategy. One challenge with this in a comparative evaluation is that the query construction typically differs between approaches, which makes multiple tools hard to compare, because the cross folding needs to be done before the query creation (i.e., for the static analysis already).

This approach does not work at all, if different approaches rely on different input data, e.g., complete snapshots of released source code (e.g., [262]) versus change information taken from a programming history (e.g. [167]). Comparative evaluation also don't get along that well with binary releases of tools, because these mix all parts of the RSSE into a single execution, which makes them hard to integrate.

We strongly believe that, among all four parts of the workflow, the evaluation part offers the greatest potential for reuse. Having an environment that facilitates evaluation through approach-independent datasets opens up the possibility to create standardized benchmarks, the ultimate tool for increased comparability of RSSE. Once a benchmark exists for a specific type of RSSE, it can be reused for all future approaches that are compatible, i.e., produce the same kind of proposals from the same kind of queries. A common data representation is key for this, as the individual approaches need to be able to process each benchmark scenario individually.

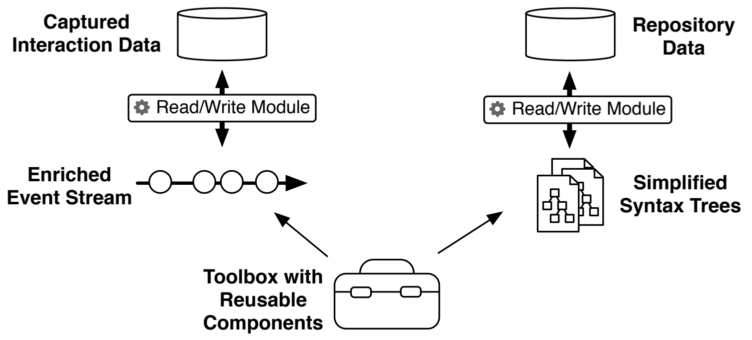
Overall, it becomes clear that many of the above steps are of a generic nature and should be reusable. Only steps 6-8 are specific to a new approach. Even though the evaluation part also depends on the tool that is build, it only has to be designed once and can be reused later by other researchers that propose a *compatible* tool.

There are of course tools and analyses that fit into one or more of the above steps, e.g., existing datasets, static analysis frameworks, or libraries for common machine learning tasks. However, building the complete recommendation system includes taking bits and pieces from each tool and patching things up together to achieve the overall goal. In other words, there is no structured way of re-using existing tools or methodologies, which leads to repetition. Previous work found that approaches do not simply use existing algorithms, but combine their own analyses and transformations with established processing steps [204]. The entire approach is usually implemented as a single process, where all these parts intertwine. No isolated implementations of the individual processing steps exist that could be reused in a new approach.

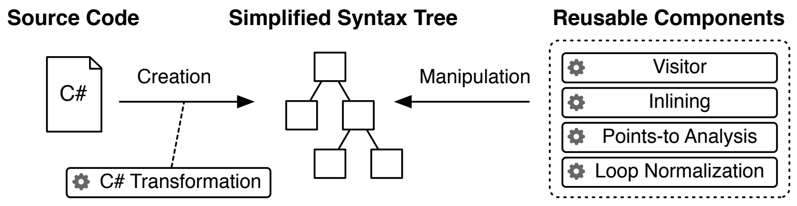
All these factors pose a threat for the work in our field. We argue that to increase reuse, reduce potential errors, and improve comparability, we should separate input preparation, static analysis, and tool building into isolated implementations. Reused tools likely become mature and stable over time, but new implementations always bear the risk of new bugs. Furthermore, we should separate generic analysis tasks to enable toolsmiths to focus on their original contributions. We envision a platform that provides easy access to the above steps and argue that this paves the road to advancing the state of the art over the current "island development".

Our Solution We present C$\boxed{\text{A}}$RET, a platform that provides *Composable Analyses and Reusable Transformations* to serve as a common ground for research on RSSE by providing reusable components that solve recurring tasks. The overall structure of the platform is sketched in Figure 7.2. It is based on enriched event streams and simplified syntax trees, which cater for the specific needs of typical RSSE. C$\boxed{\text{A}}$RET provides reusable components on top of SSTs that facilitate the construction of RSSE enabling researchers to focus their efforts on developing components that are specific to their recommendation problem at hand. It is designed to cater for the requirements of a wide range of source-based recommender systems that we have collected through a literature survey. We will show in Part IV, how we have derived a benchmark for method-call recommender and how (re-)creating existing recommenders on top of our platform can be achieved. Furthermore, C$\boxed{\text{A}}$RET is not specific to a special kind of RSSE and its generic components allow the creation of future RSSE.

C$\boxed{\text{A}}$RET offers the following three core features (1) It can directly work with our published datasets. (2) It features reusable components on top of SSTs that help researchers built RSSE, for example, generic analyses, transformations, and preprocessing steps. (3) It provides support when building an evaluation pipeline for RSSE. In the remainder of the section, we will elaborate on these features through which C$\boxed{\text{A}}$RET saves development effort, reduces the risk of error, and improves comparability.



(a) Overview over the Available Infrastructure



(b) Reusable Components for Simplified Syntax Trees

Figure 7.2: Overview Over The KaVE Platform

7.1 Working with Datasets

On the highest level of abstraction, C☒RET provides the necessary infrastructure to work with our datasets of released source and interaction data. We decided against storing the data in a database, because this adds another level of technology on top. Instead, we decided to use a file storage that is based on common technologies, i.e., JSON serialization and Zip compression, to make the datasets as accessible as possible and easy to share. The C☒RET persistence layer allows to conveniently read and write the data stored in these files. We provide bindings that allow to work with both datasets in JAVA and C#, but the available options depends on the data source. Please note that this implementation is not specific to any dataset and can be reused by other researchers. The technical details were discussed in Section 5.2.

Interaction Data The data source that makes the C☒RET platform unique is the interaction data that was captured from developers by the FEEDBAG++ interaction tracker. We store the captured data in enriched event streams, which is a simple data structure that can be easily serialized. No complex logic is required to read or write the JSON. When designing the data structure, we mainly selected built-in data structures of the programming languages to facilitate using languages features to work with the collected data. For example, time or duration related fields use built-in date classes. This allows to process the events with functional constructs like maps or filters.

We provide several convenience functions that can be used when reading the interaction data. For example, we support to split the events by user and by programming session. In addition, we provide some general event stream filters that can be used to filter noise from the stream like obvious duplication that is caused by running Visual Studio and ReSharper in parallel. These functions were needed when we have worked with the interaction data in our own experiments and we decided to implement the components in a reusable way that makes them valuable to others.

Released Source Code We store all source code that we capture in the form of simplified syntax trees. While SSTs can be part of the enriched event streams (e.g., in edit events), we provide a second dataset that is made from released source code taken from GitHub. Providing both datasets in the same representation supports the idea to create generic components that can be applied in both cases.

The dataset of released source code consists of one .zip file per analyzed C# *solution*. We provide components to reconstruct the GitHub user and the repository name from the file path, which is enabled by the file organization. When opened, all analyzed types of a repository are deserialized one by one. We provide general components to make it easy to work with the data. For example, extensible components that can be used to navigate SSTs, a component for pretty-printing, and tools that make SST easy to debug, e.g., when writing static analyses.

Overall, the file-based persistence lowers the barrier for others to access the data and the C$\boxed{\text{A}}$RET platform further simplifies it. In the very beginning, we have used a persistence strategy that was based on a database, but we quickly realized that this quickly brings typical database engines to their limits. The file-based persistence and the persistence layer of C$\boxed{\text{A}}$RET have proven their practicality in the majority of our own experiments that we will present later in Part IV. We provide tutorials on how to work with the two datasets[12] to allow others the same smooth experience.

7.2 Reusable Components

We have introduced simplified syntax trees as the required means both to make collecting a fine-grained source-code history possible and to provide a common data representation that is tailored to the creation of RSSE. In addition to the basic infrastructure that is provided by C$\boxed{\text{A}}$RET, we provide several reusable components in our platform that support working with SSTs. We separate different processing steps into specialized components that are shown in Figure 7.2b.

In addition to the source code transformation component that creates SST from C#, our extensive literature review has revealed several recurring analysis challenges in typical RSSE approaches. In this section, we will introduce our reusable tooling for the analysis and manipulation of SSTs that solves the most common ones. These reusable components can serve as combinable building blocks that facilitate static analyses of typical RSSE approaches. We provide these tools in Java to make them compatible to other research tools that are typically written in Java. In the following, we will provide more details for each provided component.

Source-Code Transformation The most important component is the transformation that creates SST from C# source code. The component can be used to capture SSTs from files under edit in Visual Studio or to transform a given C# solution to SSTs. Two make it applicable and easily reusable by others, we provide two integrations. The technical details about the C# implementation were already presented in Section 5.3.

In-IDE The transformation is available as a service for all ReSharper plugins to transform source code that is contained in the current solution. This component transforms the current enclosing type into its SST representation and is, for example, used by FeedBaG++ to capture the current file under edit on change.

Bulk Transformation The second integration that we have provided for the C# transformation is a bulk transformation that can transform complete C# solutions. In contrast to the in-IDE integration, all available types are identified and are then transformed to their SST representation. We have used this component to create the dataset of released source code from GitHub repositories.

We have used both integrations, but they are also valuable to and reusable by others. Other researchers can reuse the in-IDE integration in their IDE plugins. Use cases for this are either the capturing of the current state of a file or a preprocessing of a type that makes its analysis easier. The bulk transformation can be reused by others to create separate datasets, this might be necessary, if unreleased source code should be analyzed or simply because the researcher is interested in a different set of projects.

Visitor The "gang of four" book [67] laid the foundation for modern software engineering by defining several programming patterns that provide generic solutions to recurring programming problems. The visitor pattern is one of these patterns and it achieves a separation of a data structure from algorithms that operate on it. This allows to add functionality to a closed hierarchy of elements without changing the data structure.

Our implementation of SSTs implements the visitor pattern, which makes traversing the SST tree structure very convenient. More importantly, it opens up the opportunity to implement new static analyses on top of SSTs without altering the data structure. The visitor pattern has also been used in other representations of abstract syntax trees, e.g., by the Eclipse JDT or by the AST representation used in ReSharper.

All reusable components for analysis or transformation that are provided by us are realized as visitor implementations, including the following components that are discussed in the remainder of this section.

Points-to When studying source code, it is often the case that a researcher is interested in the objects that are referenced or addressed in a specific statement. Our literature review has revealed several RSSE that perform a points-to analysis to identify method calls that are invoked on distinct instances of types. Our goal was to provide a points-to analysis that can be easily integrated into analyses of such approaches.

The C$\boxed{\text{A}}$RET platform provides points-to analyses for SSTs that can be used to identify the abstract location to which variable references point to during the execution of a program. We have provided four implementations:[1] two basic implementations that identify the object instances by enclosing method, type, or variable name and which do not perform any flow analysis, a Steensgard-style unification analysis [233], and an implementation that is based on constraint inclusion [209].

The points-to analyses can be easily applied to SSTs and can then be asked for the abstract object behind a variable name. We provide a collection of examples that explain how to integrate the points-to information into an analysis.[12]

[1] Created by Simon Reuß as part of his Master thesis [196].

Inlining It is a common guideline and a good coding practice to keep method bodies short and to identify coherent building blocks that can be outsourced to a helper function [132]. Unfortunately, from the point of view of a researcher that wants follows the control flow to identify related statements (e.g., method calls), this makes it harder to write a static analysis and requires to track the control flow into called methods. Other analyses simply stop at the intra-procedural level and do not consider statements that happen outside the current method scope, which reduces the quality of the results while making the static analysis much easier.

We provide a inlining component[2] in C$\boxed{\text{A}}$RET that represents a tradeoff between both approaches and that makes tracking analyses superfluous. When applied to an SST, method calls to private helper methods (non-entrypoints) will be inlined into the calling method; only the entry points of an SST remain in the end. This allows to restrict a static analysis to an intra-procedural scope, but still get some benefits of inter-procedural analyses. After the inlining, tracking is unnecessary, since all related code is now in the same place.

We decided to restrict the currently implemented inlining approach to class boundaries, but it is a simple engineering task to extend this to inline other local classes as well. We think that this reflects the usage of APIs best, especially when considering the notion of entry points to API implementations (see *reference generalization*).

The current implementation does not inline recursive methods, as this requires special handling to prevent stack overflows. Note that the special handling is necessary for any tracking analysis and is not a peculiarity of our transformation. A simple approach to solve this limitation is to limit the inlining depth for recursive calls.

Loop Normalization Many code elements can be used to express a loop in a program, e.g., `while`, `for`, `foreach`. Depending on their preference or knowledge of the programming language, different developers might select different loop constructs when implementing the same piece of code. While the same semantics are being implemented, it gets much harder to find patterns in the resulting code, as the structure of the resulting source code is less similar because of these differences.

Static analyses in the context of security avoid this problem by design. The three-address representation that they use for the analysis (e.g., JAVA byte code) does not contain these control structures anymore, but reduces them to a combination of labels and jumps. This unifies the syntax and requires analyses to handle fewer cases. However, this approach is no solution to our pattern-mining problem. The goal is to learn patterns that are close to source code to make them easy to understand by humans and to re-integrate them in RSSE in the IDE. It is not guaranteed that labels and jumps contained in a learned pattern can be easily mapped back to source code, therefore, we prefer a solution that stays closer to source code.

We followed a different approach that is more appropriate in our use case. We provide a reusable component that normalizes all loop constructs into a `while` loop, with the goal of unifying source code written by different developers.[3] The benefit of this solution is that the representation stays close to source code, while reducing the overhead for the analysis. Future work could integrate additional normalization strategies, e.g., rewriting boolean expressions to contain only `and` and `not` operations. This is less interesting from the perspective of static analysis on API usage though.

[2] Created by Jonas Schlitzer as part of his Bachelor thesis [216].
[3] This functionality was implemented by Carina Oberle in a lab.

Future Ideas We have focused on the most important transformations and analyses so far, because our limited resources forced us to prioritize. However, also less frequently used components might still be valuable for future approaches and we have ideas for other components that would open up new possibilities.

Slicing A tool that we have seen less frequently used in the reviewed approaches was program *slicing*, i.e., reducing a piece of code to these lines on which another selected line of code depends. We do not currently provide such a module, since we have only seen it twice in the literature that we have reviewed, but we think that this would be a useful tool when C$\boxed{\text{A}}$RET is extended to pattern mining.

Jimple Export Many tools, especially in the context of security, use very sophisticated static analyses. These analyses have to be very precise and are often conducted in mature static analysis frameworks like Soot.[5] These frameworks are very mature, but at the same time require great knowledge about static analysis to be useable. We think that it is an interesting idea to allow researchers to select the most appropriate framework for their work and to provide the opportunity to decide between the world of simple and optimistic analyses and sophisticated and precise analyses. This would be enabled by an export component that generates Jimple code, i.e., the intermediate representation of Soot, from SSTs, which could be further analyzed by Soot.

The real strength of the C$\boxed{\text{A}}$RET platform is that it encourages and makes it easy to contribute more reusable components. We are optimistic that in the future, other researchers will contribute further analyses and transformations, which makes the platform even more attractive for future research.

7.3 Evaluation Pipeline

One of our early motivations when building the C$\boxed{\text{A}}$RET platform was to improve the evaluation of a specific RSSE that we built. We quickly realized that the potential is much greater: the combination of released source code with the captured interaction data provides a unique opportunity to build evaluation pipelines for a wide range of systems. We have used the platform to develop various evaluations for method-call recommenders (see Chapter 10), but the same is possible for other kinds of RSSE as well. In the following, we will go through several reusable components that we provide in C$\boxed{\text{A}}$RET to facilitate the creation of evaluation pipelines or benchmarks.

Collection of Evaluation Metrics

A recurring task in the evaluation of a recommender system is the calculation of the metrics that are used to estimate the quality of an RSSE. In our own research, the most important ones have been precision and recall and their geometric mean, the F1 value. C$\boxed{\text{A}}$RET contains reusable components that allow calculating these values between sets. Other metrics exist that are typically used in RSSE evaluations, for example *accuracy*, *top-k accuracy*, or *area-under-curve*. However, they have not been relevant in our evaluations so far and, therefore, we did not implement them yet. Adding them is only a simple engineering task and future work could provide them or additional metric implementations.

Especially in artificial evaluations, it is sometimes the case that several queries are generated for one scenario. A common problem is that a bias is introduced in the evaluation through some scenarios that produce more queries than others. We provide a reusable component that can merge queries by scenario.

Once the results for all queries in an evaluation have been calculated, they need to be stored. We provide data structures that can be used to store large numbers of double values and that provide several statistical calculations for the added values. This includes basic calculations like average and mean, but also more complex measures like quantiles or combined metrics like a box plot. There are different definitions of a box plot, ours includes the mean, the first and the third quartile, as well as the 0.05 and the 0.95 quantiles. We also support the export of the double values to a text file that allows to run complex statistical tests using for example the statistics software R or to create plots in tools spreadsheet software like EXCEL.

Data Templates for Reusable Evaluations

C$\boxed{A}$RET builds the foundation for the creation of reusable evaluation benchmarks. A benchmark is always specific to some kind of RSSE, e.g., a system that can be queried with an SST to propose missing method calls, and it is not possible to compare incompatible systems. The core of each benchmark is a dataset that is used to create the evaluation queries and that defines the expectations.

The two datasets of C$\boxed{A}$RET (i.e., collected interaction data and released source-code) provide a unique opportunity to cater for a wide range of evaluation scenarios. Depending on the research question or the design of an evaluation, different kind of data is required and we provide the necessary means to cater for many scenarios. Our platform provides utilities that can generate the following data templates.

Static View on Releases Many works are based on released source code. They use the static view on the latest release in their analyses, assuming that this represents the most complete and most reliable state of the source code. Datasets of released source code are used in several papers, most likely because they are easily accessible.

To build a valid evaluation around such a dataset, it is necessary to *cross fold* the data. Meaning, to split it up into a training and a validation set and repeating this split in several iterations until all dataset items have been used for validation. Cross folding avoids to use an item at the same time for training and for validation, which can prevent overfitting, while avoiding bias introduced by a specific partitioning.

C$\boxed{A}$RET provides a reusable component that can create the individual folds of a cross-fold evaluation. We integrated this component in our own evaluation for a method-call recommender, but it is not tailored to a specific kind of RSSE and can be reused for other benchmarks as well.

Program History Some approaches are based on the program history and extract change information. While this is also available in public repositories, this typical commit history is usually very coarse grained, because developers tend to commit sparsely. We have solved this issue in our dataset of interaction data that contains the fine-grained history of edited source code. We also store save actions and commits to the version control system, which create the opportunity to compare all three granularities (i.e., on change, on save, and on commit).

The interaction history contains all edit files in the order in which they have been worked on. They are not sorted by file and edits are mixed with all other development events, which makes it hard to extract a consistent edit history. We provide utilities on top of the interaction data to extract *Edit Streaks* (i.e., subsequent versions of the same file from the same developer) and to create *Micro Commits* (i.e., tuples of two versions of the same file) to facilitate the analysis of program histories. We have used these components in the evaluation of artificial evaluations (see Section 11.2).

Developer Interaction The final and arguably the most realistic way of evaluating an RSSE is to use real interaction data as ground truth. The assumption is that developers use their tools deliberately and that they don't just randomly select items in a tool. If this assumption holds, it is possible to preserve realistic query scenarios for the recommenders by recording the context in which they have used a tool and by storing the proposal that has been selected as the expected outcome.

Our dataset captures such detailed information about the usage of the code completion tool. No special tooling is necessary to extract this kind of information from our dataset. The corresponding data structure can directly be used, because it was tailored to capture realistic query cases for method-call completion systems.

We did not use the dataset in our published evaluation, because the amount of available data has not been big enough yet. However, the dataset has grown significantly in size and such an evaluation is possible by now. We will design such a benchmark for method-call completion systems in Section 11.3. The dataset can also be used to evaluate other kinds of recommenders, as long as the ground truth can be found in the data (e.g., completion for static types, parameter completion).

Overall, there is not a single best way for RSSE evaluations, but there is plenty of chances to create an invalid one. To make avoiding an invalidation as easy as possible, we provide some predefined data templates in C$\boxed{\text{A}}$RET that can be used for evaluation. The selection of the most appropriate data template depends on the use case and cannot be recommended generally.

We have used the dataset and the components in our own evaluations of a method call recommender, However, the data templates are not specific to this use case an can be reused in other settings too. More details on how we have used these components in our own experiments will be provided in Part IV.

Scalable Experiments

An evaluation can quickly run into scalability issues when working with large datasets. The typical scalability problem is concerned with technical limitations of an approach that prevent it from handling large input sizes. However, evaluations often introduce another bottleneck that is far worse: the explosion of evaluation parameters.

The evaluation of an RSSE often resembles an optimization problem: out of a set of possible features, weights, and parameters, the ones need to be selected that perform best. The problem is that each configuration flag doubles the size of the search space and continuous values like integer let it explode. This problem can be solved to some extend by reducing the search space to a meaningful set of combinations. However, the sheer amount of different scenarios makes evaluations slow.

To solve this problem in a generic way, we have implemented a lightweight variant of the map reduce algorithm [40] in JAVA. Our basic implementation avoids the complex setup of existing solutions like APACHE HADOOP[31] by relying on conventions to simplify the system. A sketch of the platform is shown in Figure 7.3. The core idea is to have a central server that slices down the work into smaller jobs (Step 1). Multiple workers

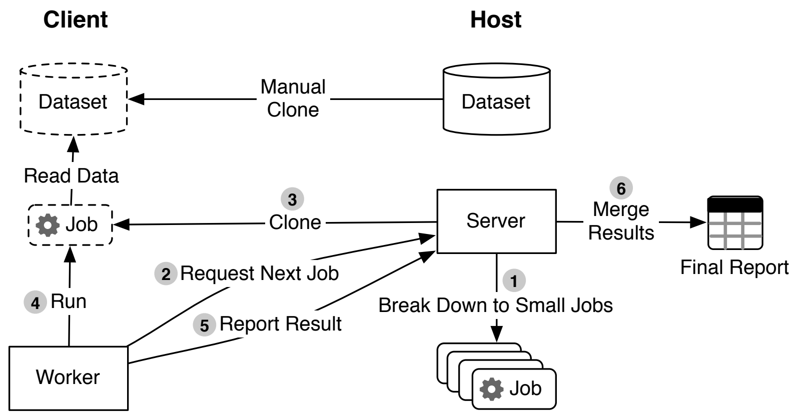


Figure 7.3: Leight-weight Map Reduce

can be run that request jobs over the network via *remote method invocations* (Step 2). The requested job is transferred, but it is not used as a transparent proxy reference, but the instance is cloned on the worker machine (Step 3). The worker now runs the job (Step 4) and reports successful results or error messages back to the server (Step 5), which merges all parts of the original problem back together to create the report for the evaluation task (Step 6).

The clients are operated in an evaluation agnostic way. The framework is designed such that only the server knows how the tasks are sliced and put back together, the clients only request and run generic task objects. The only requirement is that the dataset is cloned for each worker that is run and to distribute the current packaging of the executed program. The server will detecting outdated clients to avoid incompatible results, so while the data has to be synced only once, the worker program needs to be updated on every change.

We have successfully used this lightweight parallelization infrastructure to scale an excessive evaluation of our method-call recommender. The nature of our evaluation allowed us to achieve an almost linear speed-up with up to 26 workers, which resulted in a significant reduction of the runtime from days to hours. Researchers that want to adapt the system have to find a strategy to slice down their evaluation into subtasks that can be run in parallel. If this is possible, then the system is easily reusable in other evaluations through its evaluation-agnostic design.

Chapter Summary

This chapter presented a modular platform, C$\boxed{\text{A}}$RET, that facilitates studying the development process and that provides reusable components for the development of recommendation systems in software engineering. C$\boxed{\text{A}}$RET provides reusable components for generic static analysis tasks, such as points-to analysis and tracking analysis, which can be used as building blocks in new approaches. We believe that C$\boxed{\text{A}}$RET can facilitate the development of future RSSE by relieving tool smiths from implementing recurring source-code analysis tasks and helping them create shareable data sets. This also leads to better comparability of recommender approaches. We have used the platform in our experiments and will report about the results later in Part IV.

Part IV:
Applications & Evaluations

The C$\boxed{\text{A}}$RET platform is a collection of several components that are strongly connected and it is hard to evaluate these components individually. Instead, we decided to evaluate the platform as a whole by showing the applicability to several use cases and the usefulness of the proposed data structures and the platform.

To this end, we will first provide data to work with and compile both a dataset of released code and a dataset of interactions collected in a field study. We will then present several applications as case studies that stress the various components.

8 Compiling Datasets and Instantiating the Platform

The C$\boxed{\text{A}}$RET platform is tailored to the needs of studies on the development process and of the creation and evaluation of recommendation systems for software engineering. The only requirement for actually using it in experiments is to have data. To this end, we compiled two different kinds of datasets that complement each other and that can be used with the platform: a dataset of released source-code found on GitHub and a dataset of developer interactions that we have collected in a large and long-running field study. Both datasets are published at the project website and regularly updated.

Additionally, most research in the field focuses on source code and developers from the Java ecosystem and we have observed that no much research addresses C#. According to the TIOBE[28] index, C# is among the most commonly used programming languages though, so we believe that it is important to close this gap.

The dataset was created with the tools and representations that are being presented in this thesis. This enabled the creation of datasets that can easily be shared and reused, which facilitates the design of repeatable experiments and performance comparisons of alternative approaches.

8.1 Released Code

A multitude of source-code-based recommendation systems for software engineering (RSSE) exist and a selection has been discussed in the related work (Section 2.1.2). Examples include systems for code completion [195], code search [262], or snippet mining [169]. A recurring challenge for new RSSE is to find suitable projects that can be used as input for their static analyses that extract information. The recent rise of platforms like GitHub, Bitbucket, or SourceForge make it easy to find vast amounts of source code that can be used for the above research. However, open-source repositories cannot be directly used in analyses, because some effort is required in order to prepare them, e.g., resolving dependencies and making them compile. Additionally, it is necessary to think of how the results of a batch analysis of many repositories are stored such that it is easy to access them later in a structured way. In short, using and analyzing a large number of open-source projects is usually non-trivial.

This section introduces a dataset that contains the source code of 360 C# repositories in a simplified form and which has been published before [189]. Instead of spending time on assembling a dataset and making all sources compile, researchers can simply use our curated dataset for their research. Reusing an existing dataset has the added benefit of improving comparability and reproducibility of the results. The dataset is primarily meant for use in research on source-based RSSE and to answer questions that target correct usage of *application programming interfaces* (API), but it could also be used in related areas, e.g., anomaly detection.

Dataset Creation

To create the dataset, we have been reusing the tooling and the concepts presented in this thesis to transform source code from GitHub. The focus has been to create a dataset of released source code that illustrates usages of various APIs.

Data Representation The dataset is provided in the form of SST that was introduced in Chapter 4. The tool that we built to perform the transformation is based on ReSharper, a very common plugin for Visual Studio. In ReSharper, it is possible to open many different C# project types. A C# repository contains one or more *solutions*, a C# specific construct that represents a collection of several related *projects*. We have implemented a runner that finds all solutions that are contained in a folder or one of its recursive subfolders and opens them one by one. It processes each type that is declared in the solution, traverses the abstract syntax tree and performs our transformation. Our tool is open source[9] and can be used by others to transform additional solutions.

Repository Selection Our selection strategy for the repositories had a focus on API usage relevant for RSSE. In the beginning, we started by manually finding repositories by randomly going through popular GitHub C# repositories. Based on the brief repository description, we skipped repositories that themselves are used for library or framework development, because we believe that code that is more geared towards end-user applications would represent a more typical usage of an API. To capture mature repositories, we skipped repositories with less than 10 commits or with less than 3 committers. After becoming confident in how to find good candidates for our dataset, we used the GitHub API to automatically find a larger amount of repositories. The number of available repositories is huge, so we restricted the search to a specific set of API types that we have observed in the interaction data to make both datasets complement each other. In this search, we also included GitHub's star rating of the repositories to further reduce the number and to exclude low-quality repositories. We have transformed all contained solutions of the remaining repositories and provide a checkout script for these repositories together with the dataset.[10]

The selected repositories cover a wide variety of project types, including exemplary repositories that mainly contain tutorials for different parts of an API, small applications, and large-scale projects with many committers. The project domains in use are also quite diverse. For example, our selection includes web service clients, applications for machine learning, games, and applications with a graphical user interface.

Data Organization For easier distribution, we have combined all pieces of the dataset into a single downloadable archive. The archive contains one folder per analyzed repository. Each folder contains .zip files, one for each analyzed solution of the repository. The .zip file contains one file for each type declaration found in a project of the solution. These files contain a serialized representation of the corresponding SST in the *Javascript object notation* (JSON). Our naming scheme makes it possible to reconstruct the project structure from the SST and our file organization of the .zip files makes it easy to identify the GitHub repository from which this data was created, as well as the path to the solution file within the repository.

Statistics Our dataset[1] is created from a set of 360 GitHub repositories from 309 users that contain 2857 solutions. We transformed the source code found in each project into SST that represent more than $68.6M$ lines of code[2] extracted from $805K$ type declarations ($677K$ top level, $127K$ nested). The type declarations represent 652K classes, out of which $516K$ extend a base class or implement an interface. Over all classes, we find 3.7M method declarations, 784K out of these override an existing method or implement an interface. We find more than $14M$ invocation expressions in all method bodies of the dataset. $5.5M$ invocations refer to one of the $244K$ unique methods that are defined in a referenced assembly.

Overall, the dataset contains example usages for 2134 unique assemblies. 361 of these assemblies are used in at least 5 solutions (186 in 10, 111 in 20, 80 in 30). The 45 assemblies that are used in more than 50 solutions represent built-in assemblies of the .Net framework and very common assemblies, like various test frameworks (e.g., NUnit, xUnit, Microsoft Unit Test Framework) and the serialization library Json.NET.

Extending the Dataset

We foresee multiple ways in which the dataset could be extended or further improved. The extensions come in two flavors: some increase the amount of data that is extracted and stored, the other alternative extract new kinds of data from the same input.

An obvious way to extend the dataset is to use more repositories. This could include a larger number of repositories from GitHub, as well as from other platforms such as Bitbucket or SourceForge. We did not explicitly aim for creating a dataset that is representative for different factors such as project size, complexity, domain, etc. Thus, future work could introduce a more structured way of selecting the included repositories and apply a measure to quantify the representativeness of the dataset similar to that sketched in our previous work [187].

The dataset could be extended by supporting more code elements in the IR. Examples of this include visibility modifiers, comments, and attributes. In the same way, the transformation could be extended to cover more cases (e.g., syntactic sugar of C#'s textual Language-integrated Query (LINQ) expressions) or features from more recent C# language specifications that are not yet represented in SST (e.g., auto-property initializers).

Future approaches might also want to consider more information from the type system than what is captured in our naming scheme and in the type shapes. For example, also capturing the type shapes of all extended classes instead of just capturing it from the type under edit. While the IR could be used to store the additional information, it is necessary to extend the transformation in order to extract the required information from the source code.

Limitations

Our current selection of repositories covers a wide variety of applications and project sizes, but the repositories are only selected from GitHub. While we don't expect to find

[1] As of this writing, the most recent version of the dataset has been released on April 28, 2017.
[2] We are referring to the normalized version of the source code, e.g., comments are excluded.

conceptually different kinds of projects hosted on other hosting sites like BITBUCKET or SOURCEFORGE, we do not include any closed-sourced projects in the dataset. We expect several large open-source projects that are driven by large companies are representative for this kind of software, but future work should validate this assumption.

We do not include any meta data about the repository in the dataset that makes it possible to select a representative subset of repositories. As a result, we cannot provide any representativeness guarantees for the dataset, e.g., it might be that the selected repositories are biased towards a specific domain or contain more research projects than industrial open-source projects. However, we believe that this does not limit the use of the dataset unless the target research problem explicitly requires an equal distribution of different domains, project size, project ages, etc. In this case, integrating external services like OPEN HUB[29] could provide additional meta data that might help to improve the categorization.

Our selection strategy can introduce a potential overlap of contained type declarations in repositories. This can happen in two scenarios: 1) To support multiple versions of VISUAL STUDIO, repositories can have multiple solutions tailored to specific releases. These solutions can share references to the same physical files, which leads to a file overlap *within* a repository. 2) A common way of including dependencies is including them as nested checkouts in the repository. Our checkout strategy would clone these nested repositories as well and would analyze them separately. The same could happen for clones of a repository (a *fork* in GITHUB terminology). While we did our best to manually avoid such a case, we cannot be sure of its absence. In any way, we don't think that the influence of both overlap cases is severe, because they are easy to detect (through the fully-qualified type naming and the file organization) and to circumvent, should an overlap conflict with the research problem.

While our automated selection approach helped us scale the dataset to a reasonable size, it might be that the selection mechanism includes suboptimal repositories, e.g., *forks*, dummy projects, or obsolete source code. We included the star rating and selected only active repositories to mitigate this and also reviewed the selected data manually to identify and remove these unwanted repositories. However, it might be that we missed some that should not be included.

8.2 Developer Interactions

More and more works propose interesting tools and many struggle to study the effects of their tools on developers in a realistic setting. Recurring challenges in this regard are collecting statistics about the development process to prove the existence of a problem or finding valid ground truth for the automated and repeatable evaluation of a technique. Researchers that are willing to collect this data in experiments are confronted with a vast overhead. Specialized tooling needs to be implemented that can capture the relevant data, the experimental setup should guarantee a representative setting, and it is required that enough people participate in the experiment.

We wanted to close this gap and collect a dataset that can be used by us and other researchers. It should allow to focus on an actual research question related to the development process instead of spending time finding users, conducting an experiment, and collecting data. We hope that publishing such a dataset facilitates research in this area and also increases reproducibility of the results.

Dataset Creation

The goal of our dataset was to collect information on in-IDE activities of average developers. Our datasets provide general information about IDE usage and development process, but also many specialized information like test events. We focus on the use of intelligent code completion and also capture a fine-grained source-code history, to allow using our captured data as input or ground truth for research on RSSE.

In a first step, we collected a preliminary dataset during an industry collaboration using FeedBaG [3], the predecessor of FeedBaG++. It was used in the experiments of Section 9.2, but it did not yet capture the full extent of enriched event streams that we have presented in Chapter 3. The deployment with the industry partner prevented us from publishing the dataset, but this first iteration helped us to identify chances for improvement. After incorporating the *lessons-learned* in our tooling, we decided to repeat the data collection in a public setting to build a second, publishable dataset to enable and facilitate further research in this area. We considered several ways to collect such a dataset and decided to focus on a field-study to maximize the number of participants and to cover a broad range of work settings.

Study Setup Our interaction tracker FeedBaG++ that is used in the experiment offers several configuration options that allow enabling features of the system. Out of the various configuration options of FeedBaG++, some increase the trust in the system at the expense of consistency guarantees related to the collected events (i.e., anonymization, possibility of inspection and deletion) and others provide value to the users (i.e. intelligent code completion) or introduce gamification elements to the IDE. The configuration we use in the field study is a tradeoff between trust, consistency, the expected number of participants, and the degree to which the behavior of the participants is altered by the system.

In the end, we decided to maximize trust and the capturing of unbiased interactions by only enabling the option to anonymize the collected data and to allow editing of all infrastructural options in FeedBaG++. We deactivated both the intelligent code completion, and the achievement system for now. Both tools might increase the interest in our tool and lead to an increased user base, we were afraid though that the participants try to optimize their scores or are influenced by the recommender. This would strongly affect the observed behavior and we expect that the introduced bias would outweigh the effect we try to measure. Once these incentives are left out, we don't think that a data collection that is done transparently in the background would create a significant bias.

Please note that all of the aforementioned tooling and even advanced tooling for downloading and management of models was implemented, we did just not release it yet. We have only shipped one model for a single type that can be installed by everybody, which serves as a proof-of-concept. Once, the user base of FeedBaG++ has grown and a sufficient amount of data is collected, a bigger set of models should be published and the corresponding tooling should be released.

Acquisition of Participants The biggest challenge in a field study is to find participants, especially after our decision to disable all parts of FeedBaG++ that provide value to the developer. We advertised the field study directly in social media channels and through advertising efforts (e.g., MSR year book), and indirectly through referring to the project in our papers. Our invitation to use FeedBaG++ and to participate in the data collection was open to any interested developer, which means we did not target a specific population of users. However, our assumption is that a high number of random participants and long observation times provide a rich data set with a variety of projects and participant backgrounds. We added a voluntary questionnaire to learn about participants' backgrounds.

To make installation as easy as possible, we released the tool in the *ReSharper gallery*, the standard way to make extensions available to the public. The extension can be installed by all interested developers directly from within their IDE using the ReSharper extension manager. Updates to ReSharper are free for existing users, so over the course of the project, it was necessary to update the tool several times to adapt to breaking changes introduced by the ReSharper SDK, to avoid loosing our user base. Since its original release, FEEDBAG++ has been downloaded more than 1,200 times as of now and is regularly listed among the "Top 100" ReSharper extensions.[8]

Data Organization and Representation In a preprocessing step that we conduct before releasing the dataset, we combine all events uploaded by the same user and sort the events chronologically. We encourage users to use the same identifier across machines to allow us to group activities of the same user. As a result, active phases recorded on multiple concurrent IDE instances may overlap in the event stream. In this case, the *session id* that is stored in each event can be used if a split into programming session is required, but we do not split these phases by default.

The events are stored in plain text files that correspond to the *Javascript Object Notation* (JSON) of the event data structures that we have introduced in Chapter 3. All events of the same users are compressed in an archive, and all archives are again combined in a single downloadable archive for easier distribution. The data contained in the archive can directly be opened and processed with the tooling of C$\boxed{\text{A}}$RET.

Statistics So far,[3] we have received $11M$ events that have been uploaded by 81[4] developers. Out of these developers, 43 come from industry, *three* are researchers, *five* are students, and *six* are hobby programmers. *24* participants did not provide this (optional) information about their position. Asked about their education level, 21 participant answered that they hold a Master degree or higher that is related to computer science, 17 hold a Bachelor degree. One participant reported that he was trained in programming, 15 participants reported that they do not hold any related degree or that they learned programming on their own. *26* participants did not provide this (optional) information about their education.

The submissions cover an aggregated $1,527$ days and were collected over *eleven* months, but not all developers participated the whole time. On average, each developer provided $136K$ events (median $54K$) that have been collected over 10 days (median 18.9 days) and that represent 185 hours of active work (median 48 hours). In total, the dataset aggregates 15K hours of development work.

In our own work, we were most interested in the usage of code completion and test execution, for which we have provided the most advanced instrumentations. The dataset contains detailed data about 200K usages of the code completion and $3.6K$ test executions. An average user provides $2.5K$ usages of the code completion (median 640) and 44 test executions. A median of zero test executions suggests that at least half of our users do not test.

[3] As of this writing, the most recent version of the dataset has been released on March 1, 2017.
[4] These shared at least 2,500 events. We filtered out all developers that have contributed less.

Extending the Dataset

The dataset that we provide is a very valuable first step, but it does not represent the ultimate dataset that can be used to answer any future research question. We identified several options on how to extend the dataset to further improve its usefulness and to increase its applicability.

Users A direct way to extend the dataset is to increase the number of recorded activities. The data collection in our field-study is an ongoing effort and we see dozens of downloads per months. Our statistics show that typical users provide data that covers about two weeks. The dataset could be extended by either motivating the participants to stick longer or by mobilizing more users to install the tool.

Also related to users, it would be valuable to capture more demographics about users. The experiments that we will presenter later in this thesis provide a first indication that more in-depth information about the role of the user with respect to the project for which the data is collected would be helpful. As the user might have different roles in different projects or might switch roles during the day, a novel way of requesting this information from the user is required that is tied to a project rather than tying it to the user profile.

Setup We decided to collect the reference dataset in a field-study-like setup. However, similar data could be collected in a different setting, for example, in a more controlled setup. There, the data collected by FEEDBAG++ could be complemented with qualitative data collected during or after the experiment.

Please note that the tooling required to create additional datasets is publicly available and that the server to which the data is uploaded can be configured in the tool. Other researchers can easily reuse FEEDBAG++ and C$\boxed{A}$RET to capture different scenarios. Ultimately, the research community would come up with a set of datasets with differing foci that complement each other.

Enriched Event Stream We tailored the generators to activities that we have identified as relevant after our literature review, but we cannot foresee requirements of future research. If future work is interested in additional context information of other activities, these can be added through additional generators. As FEEDBAG++ is open-source, other researchers can extend FEEDBAG++ and the capabilities of enriched event streams to their own needs, if necessary, and extend the amount of data collected. The field study is an ongoing endeavor so once the contributed extensions are accepted in FEEDBAG++ the extended data will be available eventually.

Limitations

We tailored our data collection to works on RSSE and collected the data from released source code and interaction data. Even though we have carefully designed the data collection in both cases, we had to accept several limitations in our dataset.

We decided to strip FEEDBAG++ down to the bare data collection and not to provide any value to our participants to avoid capturing biased data. This decision makes it hard to find volunteers that provide interaction data and limits the participants to a group of curious people or to those who believe in the research. We collect demographic information to quantify the effect, but can by no means claim that the group of participants is representative for typical developers. For example, the nature of the data collection prevents capturing activities of developers that work on closed source. However, even though we expected in the beginning that our participants mainly consist of students and other researchers, we found to our own surprise that more than 50% of the participants are professional developers, making them the strongest group in the dataset. We don't expect them to program differently in their side project, which significantly increases our trust in the representativeness of our dataset. In addition, we published our tooling so everybody can collect new datasets that complement ours to further improve the diversity of the data.

As elaborated before, we do not capture much context information about the participants so far, e.g., their role in the project they are editing, which could help in deeper analyses of the data. However, we assume that different roles would also use their IDE differently, e.g., project managers would not code as excessively as for example the software engineers, and expect that it should be possible to identify roles at least heuristically. In addition, the main focus of the current collection was to capture correct usages of APIs and we don't expect a negative impact on this goal.

We capture incorrect data from inexperienced users or test data from user who just played around with the tracker for a bit, which reduces the quality of the collected data. We have to means to mitigate this. First, we ask participants for a self-estimation of their skills, which helps to identify novice programmers. Second, while it is true that the dataset contains "playing around" data, we saw that most people participate in the data collection for quite a long time (i.e., on average around two weeks). We expect that the amount of affected data is negligible in relation to the complete collected data and that it can be safely ignored.

8.3 How To Use

Both datasets that are described in this section are available on the website of the KAVE project.[10] After downloading and extracting the corresponding archive files, using the data is straightforward. We provide bindings in both JAVA and C# that can be used to reading and writing it. Additionally, we also provide utility functions that make processing it easy such that dataset users do not need to organize the file structure themselves. Extensive code examples explain how to read the dataset, access information, work with SSTs, or how to perform various processing steps.[12]

Previous Use We have been using the datasets in various applications of our own research, including the measurement of the time budget of industrial developers or to answer questions about source-code evolution. The dataset and SSTs are tailored to the needs of source-based recommendation systems. Our dataset as well as future data sets encoded in our new IR serve as standardized input to build and evaluate RSSE. We have used the dataset ourselves to build recommendation systems for software engineering and to build more realistic evaluations of these systems. The existence of the datasets and of the tooling around it helped us to succeed in these endeavors. You will find more information about our own use cases in Part IV.

Future Use The data contained in the dataset is very general and is not limited to research on RSSE, we foresee future researchers using it to answer various research questions. For example, in source-based research that builds intelligent completion engines, e.g., call or snippet completion, code search engines, and anomaly detectors, or in research that also involves the software development process, e.g., navigation patterns. The most interesting opportunity arises in research questions that span over both areas, e.g., analyzing how developers refactor test code, how to identify tasks through observing activities to decide how to find coherent change sets in the source code, or to use the interactions of developers as ground-truth for novel recommendation systems.

Chapter Summary

Datasets are required to make the C$\boxed{\text{A}}$RET platform useable. In this chapter, we have presented our efforts to create such datasets. We collected data about development activities in a large field-study and compiled a collection of released source code taken from GitHub as a source to train source-based recommendation systems. Both datasets can be used together and complement each other. We have successfully used the datasets in our own research and foresee many more use cases, in which the datasets represents a valuable contribution to the research community.

9 Studies on the Development Process

To further improve Integrated Development Environments (IDEs), we need to understand how developers typically spend their time in an IDE, which tools they actually use, and how they write source code. Enriched event streams capture developer interactions and in-IDE activities with many context details, e.g., a fine-grained history of source-code changes. This allows to use the enriched events in such studies. To evaluate their applicability to this research area and to study the usefulness of C$\boxed{\text{A}}$RET, this chapter will perform studies with two different perspectives and report on our experience: Section 9.1 will answer questions that are concerned with source-code evolution, Section 9.2 has a focus on analyzing the interactions of developers.

9.1 Source-Code Evolution

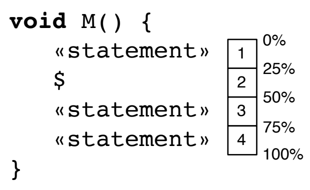
Another area of research that is relevant to this thesis is source-code evolution. We have conducted two experiments for which our platform and the collected data is applicable to analyze the applicability to this domain and to identify potential issues.

9.1.1 How do Developers Typically Write Code?

Assumptions about the users of a SE recommender system influence researchers' decisions about the best way to evaluate their tools. Some evaluations assume that code is developed linearly (e.g., [22, 169]), respectively non-linearly (e.g., [192, 195]). Since there is no empirical evaluation of how close such alternatives are to reality, the choice can even be made based on implementation convenience. Thus, the first question to address is in which location of the source code do developers trigger code completion. Since code completion is automatically triggered in VISUAL STUDIO, this serves as a proxy to determining code editing patterns. Given the captured IDE interactions in our dataset, we can easily analyze where code completion is typically invoked.

All recorded interactions contain information about the source code and the exact location in the code were the tool was triggered. We define the method size n as the total number of its statements. A completion request is treated as an artificial statement in the method body. By traversing the method body, we can determine the position p_{cc} of the completion statement in the list of statements $s_1, ..., s_n$. We remember its location by storing the coordinates (p_{cc}, n).

For each method size, we create histograms that show the frequency of code completion requests per location. However, method sizes vary significantly and adding



```
void M() {
    «statement»
    $
    «statement»
    «statement»
}
```

Figure 9.1: Trigger Location for Code Completion

frequencies, even relative ones, of locations in different method sizes does not yield meaningful results. Hence, to combine histograms of varying sizes, we fix an arbitrary size and interpolate all completion locations in other method sizes to it. Each observation point is fairly split across any relevant available histogram bins. We choose to projet all methods to a size of 25 lines to make them comparable. This choice is completely arbitrary ad has no effect on the interpretation of the results, it only affects the number (and therefore the width) of the histogram bins.

Creating a Histogram for Trigger Locations

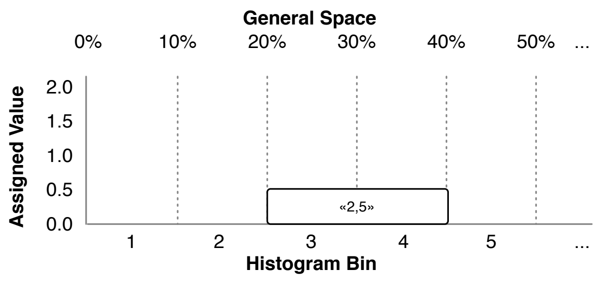
In the following, we will describe our algorithm to merge histograms for different method sizes. We use the following terminology: a *bin* is a specific range in the histogram (one column). The method size is the number of contained statements in that method. The trigger of a completion (`$`) is a special statement and it is part of the size calculation. The location of a completion trigger is given by the tuple (*index of trigger statement*, *size of method*). Consider the example illustrated in Figure 9.1, in which the location of the completion trigger is «2,4». Storing the exact location makes it impossible to compare it to other methods with differing sizes (e.g., «3,7»), so it has to be normalized. We identified three alternatives to achieve this and will go through them in the following, assuming a normalization to 10 bins.

Alternative 1: "Single Assignment" The most simple alternative is an assignment to a single target bin, which is identified through a division. For example, the trigger location «1,3» would be divided and rounded to the closest bin ($1/3 = 0.33.. \approx 3/10$), which would count the trigger for BIN_3. While being easy to calculate, this counting alternative is biased towards specific bins and can never be equally distributed.

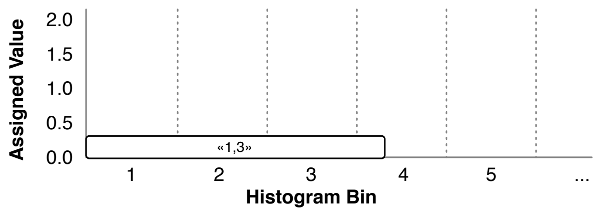
- A trigger in an empty method («1,1») is always counted in BIN_{10}.
- «□,2» are only counted in BIN_5 («1,2») or BIN_{10} («2,2»).
- «□,3» are only counted in BIN_3 («1,3»), BIN_7 («2,3»), or BIN_{10} («3,3»).
- ... (until the method size is equal to the number of available bins)

Methods that are shorter than the number of available bins will always introduce a bias, because the distribution does not cover all bins. As a result, the plots will contain "jumps" between the bins, which make them incomprehensible.

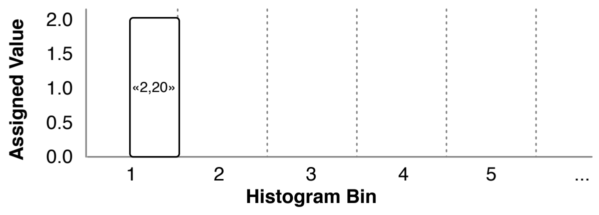
Alternative 2: "Multi Assignment" The second alternative that we have considered is to count triggers in all (normalized) bins that have an overlap with the (theoretical) bins of the different method size. For example, assume the trigger location «1,3». We would calculate the covered area in a general bin space from 0-100%, which would be 0-33% for the example. We would project this area onto the actual ten normalized



(a) «2,5»



(b) «1,3»



(c) «2,20»

Figure 9.2: Exemplary Trigger Locations for Code Completion

bins. Once rounded, the projection covers BIN$_{1\text{-}3}$, so we would count a trigger in each of them. This solves the distribution problem, but counts each trigger multiple times.

- A trigger in an empty method («1,1») is counted in BIN$_{1\text{-}10}$ (10 triggers).
- «□,2» are counted in BIN$_{1\text{-}5}$ («1,2») or BIN$_{6\text{-}10}$ («2,2»), which would store 5 triggers.
- «□,3» are counted in BIN$_{1\text{-}3}$ («1,3»), BIN$_{4\text{-}7}$ («2,3»), or BIN$_{8\text{-}10}$ («3,3»), which would store 3 triggers (or 4, depending on the case).
- ... (until the method size is equal to the number of available bins)

The resulting plots do no longer contain jumps, but the result is still biased towards methods that are shorter than the number of available bins, because these cases will always be counted multiple times. While we have no reasons to believe that developers work differently in small or large methods, this counting scheme weights shorter methods higher, resulting in a higher effect on the outcome.

Alternative 3: "Fair Distribution" The final alternative that we have considered is also based on the idea to project the trigger location from the general space onto the normalized bins. However, covered bins do no longer just count triggers, but accumulate a floating-point number. Instead of counting boolean events, we treat each trigger as 1.0 and distribute it among the covered bins, while respecting the covered area. Let us clarify this by considering several trigger locations shown in Figure 9.2, in which

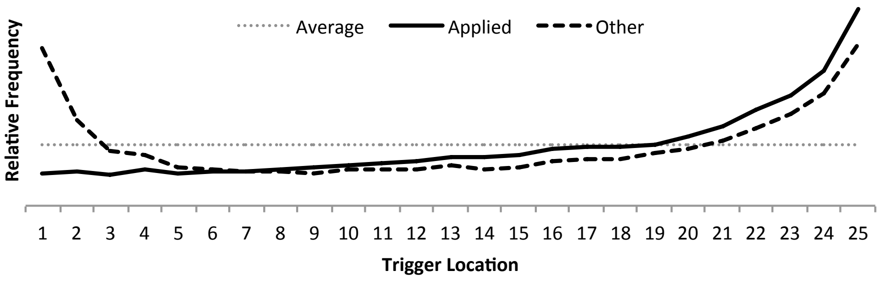


Figure 9.3: Typical Tigger Location of the Code Completion Tool

all blocks represent one trigger. The width of the trigger represents the covered area in the method, the height differs, because the area of the block always equals 1.0.

- A trigger at location «2,5» covers the area 20-40% in the general bin space. We calculate the overlap of this area with the normalized bins and distribute the 1.0 accordingly. In this example, we would assign 0.5 to both BIN_2 and BIN_3 (Figure 9.2a).
- A trigger at location «1,3» covers the area 0-33% in the general bin space, resulting in an assignment of values to $\text{BIN}_{1\text{-}4}$. However, BIN_4 is only partially covered (30-33%). We distribute fairly and assign 0.3 to $\text{BIN}_{1\text{-}3}$ and 0.1 to BIN_4 (Figure 9.2b).
- A trigger at location «2,20» covers the area 5-10% in the general bin space. This area is completely included in BIN_1, so we would assign the full 1.0 (Figure 9.2c).

The counting ensures that each bin contains a "fair" frequency that reflects how often it was covered, summing up all bins results in the total number of completions.

We considered all alternatives and concluded that a *fair distribution* is the best alternative, for two reasons. First, similar to *single assignment*, the size of the method does not affect its influence on the global histogram. Each trigger of code completion is only counted once, as opposed to *multi assignment*. Second, it avoids the "small-method bias" towards bins that would be favored by *single assignment*, e.g., BIN_5 and BIN_{10} in the case of «□,2» completions.

To allow a comparison of different plots (e.g., all completion triggers versus only triggers that have been applied), we normalize the bins by dividing their counts by the total number of completions (#bin/#total). The y-axis is no longer a frequency, but a ratio. This does not change the form of the plot, but allows a direct comparison of different plot types, even though their calculated frequencies differ significantly.

Analyzing Actual Usage Data

We have used the fine-grained program history that we have collected in our dataset of interaction data to study where developers write source code. We use the trigger location as a proxy for the actual edit location, which is a valid replacement as code completion is triggered automatically by VISUAL STUDIO. The dataset contains 199,787 code snapshots, out of which 72,007 contain a trigger location. Using these, we are able to create a histogram of the edit location within edited methods. In our experiment, we have normalized all methods to 25 statements, meaning the histogram has 25 bins.

The combined results are presented in Figure 9.3. The x-axis shows the code location, in which the code completion was triggered. Given the method size of 25, a value of 3 means it has been triggered at the third line of the method. The y-axis shows the fraction of completions that were invoked in the corresponding location. The curve is cumulative, i.e., the area under the curve is 100%.

We include three curves in the plot, the baseline, applied proposals, and all completion triggers. The baseline assumes that developers work in all parts of a method equally and, therefore, that the completion triggers are equally distributed. We normalized the the methods length to 25 lines, so the baseline assumes that each bin is selected in 4% of all completion triggers. The curve plotted with a solid line represents code completion triggers, for which the developer explicitly selected one of the proposals, the dotted line represents all code completion triggers. Differences between the two curves arise in two scenarios: either the developer cancels the completion or the developer keeps typing without choosing any of the proposed methods, which refines the proposal list as the developer types, triggering additional completions.

Figure 9.3 shows that completion triggers are distributed over the complete length of an average method with a clear tendency to trigger completion towards the end of the method, which supports the linearity assumption of several related works on source code. However, it can be observed that a large fraction of the code completion triggers that happen at the beginning of a method (locations 1-3) are not applied, which could suggest non-linear development. One interpretation is that developers use the code completion tool to explore available methods without the intention of applying the proposals or that they keep refining the proposal, because they know in advance what to select. These results encourage future work to further investigate this question, maybe by differentiating the average location between manually and automatically triggered code completion triggers.

9.1.2 Source-Code Differencing

One typical application of works on source-code evolution is the differencing between two programs. A typical application is the user interface of version control systems, in which two versions of the same file can be compared . The evolved parts are visualized in a way that makes it easy for developers to understand the changes. For us, having a working differencing solution in C$\boxed{A}$RET is also a step towards providing a snippet completion recommender, as proposals also need to be integrated into existing source code, which is a special case of differencing.

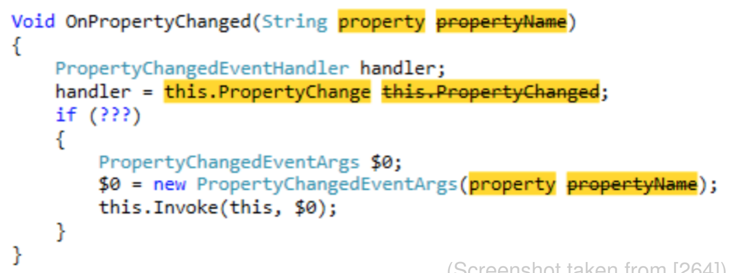
Originally, source-code differencing was text-based (e.g., [158]), but more recent differencing algorithms are based on tree structures (e.g., general data [28], AST [59]). We were interested in an analysis of how well SSTs are suited as the basis for works on source-code differencing. To this end, we have answered two research questions.[1]

- Can source-code be differenced on the basis of SSTs?
- Is it possible to use domain-knowledge to improve the visualize of semantic changes (e.g., wrapping statement in conditional; moving a statement)?

In addition, we were also interested in getting first hand experience in applying SSTs to real problems to learn about possible limitations of the representation.

Improved Edit Script We extended the existing ChangeDistiller algorithm [59], that extracts an *edit script* of two programs. The script contains all required changes to get from the first program to the second, using only *insert*, *delete*, *change*, or *move*

[1] This research was conducted by Zimmermann [264] in his Bachelor thesis.



```
Void OnPropertyChanged(String property ~~propertyName~~)
{
    PropertyChangedEventHandler handler;
    handler = this.PropertyChange ~~this.PropertyChanged~~;
    if (???)
    {
        PropertyChangedEventArgs $0;
        $0 = new PropertyChangedEventArgs(property ~~propertyName~~);
        this.Invoke(this, $0);
    }
}
```

(Screenshot taken from [264])

Figure 9.4: Screenshot of a Novel Diff Tool With Improved Change Detection

operations. The algorithm to create this script consists of two steps. First, all nodes of the two programs are being matched to find unchanged parts, then the required operations are identified that reflect the change. We have introduced several heuristics to improve the matching of nodes through applying domain-specific heuristics, e.g., to recognize renaming of methods through detecting equal method bodies. In addition, we introduce some transformations to the resulting edit script, e.g., better recognition of moves instead of multiple insert/delete combinations, to make it easier for developers to understand the semantics of the difference.

Part of the work was the implementation of an actual diff viewer that visualizes the edit script, a screenshot of the resulting tool is shown in Figure 9.4. The main goal was to create a proof-of-concept system available for an empirical study, but we also tested several visualization alternatives for move and change operations.

Results The work has shown that SSTs are indeed a solid foundation for source-code differencing. It is possible to create an edit script directly on the representation without an additional intermediate state. Integrating the technique into assistance tools seems to be possible. Through our own experience when working with SSTs, we learned that there are some properties that could be improved in future work.

Big Nodes SSTs simplify the AST by favoring big nodes. For example, in contrast to regular ASTs, in which a method declaration would typically have separate nodes for return type, method name, and its signature, SSTs aggregates this information in a single node and stores the information using our naming scheme. While storing this information in a single string make it convenient to store and to work with when traversing the AST, however, it makes it harder to match nodes of two trees meaningfully. Traditional differencing approaches work better with smaller and simpler nodes and breaking down several big SST nodes would simplify the matching.

Missing Blocks Another simplification that is introduced by SSTs is that nested blocks are not modeled as SST nodes, but as list properties of the containing node. It turned out that this makes it harder for the algorithm to differentiate child nodes, if multiple nested blocks exist (e.g. an `if` statement has then and else blocks).

Normalization The SST transformation normalize the source code by removing invalid or incomplete parts and by factoring out nested expression statements to new artificial variables. Our results show that the normalization makes it harder to match nodes. Both the existence of several unknown expressions and the stable names for the artificial variables affect the matching and, as a result, make the creation of a comprehensible visualization harder.

Overall, the results are very promising. Using SSTs for source-code differencing works well, and being able to directly use SSTs in algorithms for source-code differencing is a first step towards code-snippet completion. Future work should try to use edit scripts for integrating code-snippet proposals into existing source code.

However, we also identified chances for further improvement in future work. The matching would get easier if the differencing step either reverts the normalization or if an unnormalized SST would be used in the first place. Future work could try to separate transformation and normalization of SSTs to preserve both the benefits for the static analysis gained through the normalization and the benefits for source-code differencing gained from a representation that is closer to the actual sources.

Section Summary

In this section, we have analyzed the applicability of SSTs to research on source-code evolution. We presented two experiments, one analyzed the edit location of developers in the source code, the other studied the fitness of SSTs for source-code differencing tasks. The results of this section show that the current state of the C$\boxed{A}$RET platform and the captured datasets provide a solid foundation for research in this area.

9.2 Developer interactions

The second part of this chapter is concerned with studying developer interactions with their Integrated Development Environments (IDEs) as a means to evaluate the data that is captured and preserved in enriched event streams [4].

IDEs are very popular among software developers since they provide support for many of their daily development or maintenance tasks. Modern IDEs provide integrated debuggers, automated refactorings, assistance tools like code completion, and even integrated version control. To further improve IDEs, we need to understand how developers typically spend their time in an IDE and which tools they actually use.

Previous studies investigated developers' use of IDEs [13, 148, 156]. Those studies looked at different IDEs (ECLIPSE and PHARO) and had a mix of academic, professional, and free-time developers. To provide better understanding of how IDEs are used, more large-scale studies with various settings and different IDEs are needed. In this section, we provide such a study that examines how developers use an IDE in an industrial setting. We focus on an IDE different from those used in previous studies, namely Microsoft's VISUAL STUDIO IDE (VS).

We deploy FEEDBAG (the predecessor of FEEDBAG++) at the software-development department of an industry partner to study how industrial C# developers use VISUAL STUDIO. The department has more than 400 developers that write software in C# and we have collected more than 3.5 million interaction events over a total of 6,300 work hours.[2] We transform the captured events into high-level activities such as development, navigation, IDE configuration, and project management to identify how much time developers spend on each activity. Additionally, we analyze their usage of the tools offered by the IDE. Specifically, the data we collect allows us to answer:

RQ1 How do developer spend their in-IDE time?

[2] Please note that we had to delete the collected raw event data after the collaboration ended. Therefore, we could not combine data of the industry study with our field study to update results.

RQ2 Which IDE tools do developers use and how frequently?

By comparing the answers to RQ1 and RQ2 to those from previous studies on other IDEs, we additionally answer:

RQ3 How does IDE usage differ between IDEs?

From our observations in answering these research questions, we come up with ideas for next-generation IDEs.

To the best of our knowledge, we present the first large-scale study with professional C# developers using Visual Studio. Our analysis shows that our participants spend almost 30% of their in-IDE time on code editing and execution and another 22% navigating documents and source code. They spend little time on other activities, like IDE configuration or project management. Also developers are often inactive for short intervals (< 5 min) while using Visual Studio. We find that the code completion is by far the most frequently used assistance-tool, followed by the build system, the debugger, code search and navigation tools, the quick-fix, and version control. In contrast, unit-testing tools are rarely used. We compare our findings to other IDE-user studies and infer a set of actionable outcomes to drive future research on IDEs.

Related Work

Throughout this section, we compare our setup and results to relevant studies that have also looked at developers' activities and tool usage. In this section, we introduce such related studies and compare their overall goals to ours.

General Work Habits Perry et al. [177] conducted one of the earlier investigations into how developers spend their time, using time cards and an observation study. While we also want to understand how developers spend their time, we focus on how they spend their time *within* the IDE. Since we do not physically observe developers, we cannot (and do not aim to) come to conclusions about the activities they conduct outside the IDE (e.g., sending emails or talking to co-workers).

More recent studies include that by González et al. [74] who looked at how developers multi-task. They introduced the notion of working spheres. LaToza et al. [113] studied developers' typical tools, activities, and practices based on two surveys and eleven interviews. Their main goal was to investigate how developers understand code and keep track of the information they need. Singer et al. [226] also studied software practices of software engineers through questionnaires and developer shadowing. They mainly focused on activity switches and not on time duration of any of these activities. While all these studies aim to understand how developers get their tasks done, and often what they spend their time on, none of them instrumented the developer's working environment to precisely capture the interactions taking place.

IDE Usage The studies closest to ours are Murphy et al.'s study [156] on the usage of the Eclipse IDE for Java and Minelli et al.'s study [148] on the usage of the Pharo IDE for Smalltalk. Both groups of researchers instrumented their respective IDEs to track developers' activities. Our work expands this space of knowledge by a study on the usage of Visual Studio for C#. This provides an interesting point of comparison between the results. A key difference between our work and both studies is that we focus on professional developers from industry and do not use any open-source or student developers as participants. A large-scale study by Beller et al. [13] reports on developers' usage of ECLIPSE with respect to unit testing. They analyzed how much time developers spent on editing test and production code. We also compare our findings to theirs, where applicable. Kersten and Murphy [102] also study IDE usage, but focus on how their proposed tool, MYLYN, affects developers. Snipes et al. [230] study how gamification impacts developers' usage of VISUAL STUDIO. Similar to FEEDBAG, their tool, BLAZE, logs IDE interactions. Additionally, it provides developers with feedback about their IDE usage. We explicitly avoid this to capture the status quo of how developers use VISUAL STUDIO. Snipes et al. [229] recently presented a practical guide for IDE-usage studies. We support the value of such guides, although, unfortunately, as it was published only after we conducted our study.

Assistance Tools Other studies specifically look at how developers use static analysis tools. For example, Johnson et al. [98] investigated developers' perception of static analysis tools and the reasons they might avoid using them. Similarly, Ayewah et al. [6] used online surveys and questionnaires to understand how developers use FINDBUGS. Both studies included industrial participants. While we use different research methodologies and do not focus on static analysis for bug detection, our findings about tool usage align with the findings of both studies. However, our work does not investigate why developers use certain tools versus others.

9.2.1 Investigating IDE Usage

We use the data collected by FEEDBAG to investigate developers' activities as well as their assistance-tool usage on a typical day. To prepare both analyses, we split the events in our dataset by developer day. From the resulting developer days, we identify both activity intervals and tool usages.

Identifying Developer Days Interaction data of developers is sensitive data. We intentionally designed FEEDBAG not to capture any information that identifies individual developers. Instead, we use a session Id to identify events created by one developer during one calendar day. Then, we group sessions that were sent to our server in a single upload to identify *developers*. We find that developers regularly work past midnight, but never between 2 A.M. and 5 A.M. Therefore, we do all subsequent analysis on *developer days*, which span from the first event after 3 A.M. to the end of last event before 3 A.M. the next calendar day. We refer to the length of a developer day as the *work time*.

Approximating Participants For reporting purposes, we estimate the number of participants from the data FEEDBAG generates. Since we do not track individual participants, we cannot report a definitive number, but determine an upper and a lower bound instead. When a participant uploads multiple sessions in one bundle, we know they originate from the same developer. When she uploads them in separate bundles, we have to assume they belong to different developers. Hence, when we identify developers based on simultaneously uploaded sessions, we might count multiple developers that actually correspond to the same participant. Therefore, the number of developers based on this strategy is an upper bound to the number of participants. To determine a lower bound, we merge as many identified developers as possible, as long as no events from their corresponding developer days overlap. The number of remaining developers presents a lower bound to the number of study participants.

Identifying Activity Intervals We derive *activities* from the low-level events captured by FeedBaG (e.g. mouse clicks and key presses that occur within the IDE). To analyze how developers spend their time, we want an unambiguous mapping from events to activities. The first and second author created such a taxonomy of activities following an open-coding approach. First, they each separately created a mapping from events to at least one activity they defined ad hoc. During the process, they considered the event category, the target window for window switching, and the command Id for command executions. Second, they merged their mappings by joining all activities either of them assigned to each event. Third, they removed duplicated activities, unifying activity names where necessary. Fourth, they found a single, more abstract activity for each event that was mapped to multiple activities and assigned this new activity also to every event previously assigned to either one of the more specific activities. They iterated through the last step until each event was mapped to a single activity. The complete mapping scheme can be found on our artifact page. At the end of this process we had the following taxonomy of activities:

Code Editing & Execution Includes editing of documents; using automated refactorings, code generation, or find & replace; adding, renaming, and removing files, projects, or solutions; building projects; and using the debugger.

Navigation Includes opening and closing documents; using searches, both textual and code specific (e.g., find usages); using arrow or position keys; and using bookmarks.

IDE Configuration Includes opening, closing, or moving around windows; changing window settings, selecting filters in view, or configuring columns shown in a table; and opening dialogs (e.g., IDE options or file properties).

Project Management Includes managing issues, tasks, or requirements; and working with source control.

Other Includes all remaining interactions, which we could not group into any larger meaningful activities (e.g., activating FeedBaG-related windows, the Visual Studio command shell, the Tips & Tricks window, or the start page).

Inactivity Denotes phases where the IDE has focus, but no interaction occurs. We do not consider mouse movement as an interaction. While it may indicate activity, such as reading code, previous work has shown that developers perform isolated mouse movements for less than 4% of their time [148]. Therefore, we expect the impact on the time budget to be small. We also note that our notion of activity and inactivity does not equal nor necessarily correlate with developers working or not working. Since we only capture interaction within the IDE, we cannot account for other work activities, such as meetings, phone calls, discussion, and the like. It is neither our claim nor our goal to report on how much developers work.

We implemented a framework that groups events by developer day, orders them by timestamp, and iterates over them. The algorithm for computing activity intervals from the resulting event streams is shown in Figure 9.5. For each event, we either create a new interval, if none is running (Line 4), stop the current interval, if the user left the IDE (Line 7), or do both, if the current activity has changed (Lines 9 and 10). Intervals time out, when no event occurs for some time. Each new event cancels the active timeout (Line 2) and sets a new one, which ends 15 seconds after the event ends (Line 12). If this timeout is reached, we end the current interval and start an inactivity interval (Lines 15 and 16). If the developer leaves the IDE open overnight,

```
1  onEvent(Event e):
2    cancel timeout
3
4    if (no currentInterval)
5      openInterval at start(e) with activity(e)
6    else if (activity(e) == "Left IDE")
7      close currentInterval at end(e)
8    else if (activity(e) != currentActivity)
9      close currentInterval at end(e)
10     openInterval at start(e) with activity(e)
11
12   timeout 15 seconds after end(e)
13
14 onTimeout(Timestamp t):
15   close currentInterval at t
16   startInterval at t with "Inactivity"
17
18 onEndOfDeveloperDay:
19   if (exists currentInterval)
20     remove currentInterval
```

Figure 9.5: Activity-Interval Detection

```
onStartOfDeveloperDay:
  lastTool = "None"

onEvent(CommandEvent ce):
  if (lastTool != tool(ce)
        && tool(ce) != "IDE Core")
    increment usages of tool(ce)
    lastTool = tool(ce)
```

Figure 9.6: Tool-Usage Detection

the developer day ends with a running Inactivity interval. In this case, we delete that interval (Line 20), which leads to the same activity stream we would get if she had closed the IDE directly after her last activity.

Finally, to compute a developer's time budget, we sum the durations of intervals per activity and for Inactivity.

Identifying Assistance-Tool Usage In VISUAL STUDIO, assistance tools are used via commands invocation. Our interaction events tell us which commands developer use. Before we analyze tool usages, we reduce noise in the respective set of command Ids:

Simple Keystrokes We remove editing keystrokes, such as the arrow keys, enter, backspace, or delete, because -similar to character strokes during typing, which are also excluded- they do not represent special command behavior.

Equivalents We find that commands are not consistently reused throughout VISUAL STUDIO, e.g., selecting `Close` from the file menu has a different command Id than closing the document using a key binding or via the x-button on the top of the document. Fortunately, such commands often result in a specific sequence of other events. For example, closing a document triggers a document-close event and also a window-close event. We derive a mapping of equivalent commands by analyzing sequences of commands that follow one another within up to $100ms$ and group all such micro-sequences with a common suffix. The mapping was manually reviewed and then used to reduce equivalent commands Ids to a single one. Murphy et al. [156] encountered similar problems when analyzing command usage in ECLIPSE. They also manually created a mapping between equivalent command Ids. We support their plea to IDE developers to consistently use command Ids to simplify analytical work.

Duplicates Some interactions trigger multiple commands, because extensions like R# install own commands for the same interaction. Both the original VISUAL STUDIO command and the R#-equivalent appear in our statistics. We mine commands that co-occur within 100ms, assuming that it is unlikely for a developer to actually invoke multiple commands in such a short time. From the results, we manually create a mapping of command pairs and use it to filter duplicates.

Usage Frequencies After cleaning up the collected event data, we identify tool usage frequencies in three steps. First, we manually create an exhaustive mapping from commands to tools. The mapping is based on our understanding of the command's functionality, usually obvious from its name. There are two special tool categories in the mapping: `IDE core` and `Misc`. The first includes commands that constitute IDE core functionality, such as copy and paste or file open, close, and save. The second includes commands that we could not identify, because the command gave no clue as to which tool it supports.

Second, we traverse the event streams of our developer days to compute tool usages as shown in Figure 9.6. At the beginning of each developer day, we reset the last tool used (Line 2). We then process all command events and increment a tool count the first time the developer switches to it from a previous tool (Line 7). We ignore all interactions with `IDE Core`.

Third, we rank the assistance tools by the average number of usages per developer day. Note that this ranking favors tools whose usages span longer time periods and encompass multiple invocations of related commands. For example, while using code completion takes a single command and finishes almost instantaneously, using the debugger often takes several minutes and involves various commands. This increases the chances of the developer using other tools during a debugging session. If the developer starts the debugger, steps a few times, then uses a search to look up some code, and afterwards continues to step, we would count two usages of the debugger (even though it is technically the same debugging session). We believe that this calculation methodology reflects the actual impact a tool has on the developer day and we accept this imprecision.

Industrial Case Study

In this section, we describe our industrial case study. We specifically describe how we used FeedBaG to collect data about professional developers' use of Visual Studio. For privacy reasons, we cannot name our partner so we will refer to them as *ACME* throughout the rest of the thesis.

ACME develops tax and accounting-related software as well as in-house software for 50 years. It employs more than 1,600 developers, out of which more than 400 write programs in C# and use ReSharper. Development projects span from small training examples to core-business applications.

Incremental Rollout To make sure that FeedBaG works properly in ACME's settings, we deployed it in multiple steps.

Customized Development We developed the tool in close collaboration with a single developer from ACME. Therefore, we got early feedback and ensured that technical requirements are met, according to ACMEs environment.

Pilot Study When we deemed the tool production-ready, two volunteers from ACME installed FeedBaG as pilot users. Our goal was to ensure correct functionality of FeedBaG in many different use cases and also to convince the management that the study does not interfere with the regular tasks of the developers. The pilot phase lasted about 2 months, during which we identified and fixed minor bugs.

Company-wide Study The successful pilot study confirmed that FeedBaG was stable and ready for the deployment in production. The management permitted a large-scale rollout. We prepared extensive documentation to inform developers about the project and to motivate why we track their IDE interactions. We also released FeedBaG as open-source software so that developers can check themselves that no personal information is stored. Additionally, we provided the Event Manager (see Section 6.1.1) to give them full control over their data.

We then sent a request for participation to the 400 ReSharper users at ACME. In this email, we introduced the project and provided instructions. To encourage participation, we promised them a method-call recommender for their code completion, similar to our previous work on Java [192], but specifically trained for their in-house frameworks. We made clear that we would provide the recommender to all developers, independent of a contribution to our study. Participants did not receive any other benefits. All participants installed FeedBaG voluntarily and otherwise followed their regular work schedule. They were not assigned to any special tasks for the study.

We continually posted project updates and intermediate results on the mailing list throughout the whole study. Additionally, we attended community events of developer groups at ACME (e.g., Clean Code Community and Software Craftsmanship Community) to introduce FeedBaG.

Collected Event Dataset We tracked IDE interactions of developers for about six months, from mid January to mid July 2015. For the first two months, we had only our pilot users. In the middle of March, we started recruiting participants, which quickly raised their numbers to between 27 and 84 (see Section 9.2.1). The resulting dataset encompasses 3,505,858 events, amounting to over 6,355 hours of work time. From this time, participants spent about 2,103 hours inside Visual Studio. On average, they worked 7 hours and 12 minutes per day. This suggests that multiple full-time developers participated in our study.

Not being able to publish our industrial dataset was an early motivation for us to collect a second dataset from students and open source developers in our field study that is public and reusable by other researchers. However, since this dataset was still small at that time, we wanted to avoid an uneven mix of both datasets. As a result, we will only report our statistics for the industrial dataset in this chapter.

Statistics on Identified Developers While we base our analysis on developer days to be precise, we show statistics about the identified developers in Table 9.1. The first column shows an identifier for the developer. The second and third columns show the number of events and days from her. The fourth column shows her total work time. The fifth column shows the time spent in Visual Studio, i.e., the work time minus the time any application other than Visual Studio had the application focus. The sixth column shows the active interaction time, i.e., the in-IDE time minus the time in which we record no interactions with the IDE. The last column shows the average work time per day. The full table is available on our artifact page.

Statistics on Tool Commands From all developer days, we recorded usages of 2,493 different commands. After noise reduction, 1,346 unique commands remain. In a manual inspection, we found no further equivalents or duplicates.

Table 9.1: Statistics on Identified Developers, Sorted by Active Time

Dev.	# Events	# Days	Work Time	In-IDE Time	Active Time	Avg. Daily Work Time
			in hours			
D01	198,272	40	344:56	159:40	89:45	8:37
D02	311,293	38	286:18	117:37	75:47	7:32
D03	119,323	17	169:19	70:26	63:08	9:57
D04	150,542	23	196:21	83:06	59:40	8:32
D05	140,438	23	203:38	81:04	55:58	8:51
D06	67,444	18	151:12	74:03	52:44	8:24
D07	294,388	17	131:26	62:21	40:41	7:43
D08	102,696	12	121:11	78:24	40:32	10:05
D09	85,839	20	130:34	51:44	35:51	6:31
D10	153,369	23	181:14	52:52	34:37	7:52
D11	99,351	43	297:32	56:24	34:22	6:55
D12	55,903	32	241:10	47:47	33:40	7:32
...						
D82	345	1	2:16	0:14	0:10	2:16
D83	434	1	16:28	0:20	0:08	16:28
D84	41	2	2:33	0:08	0:02	1:16
Overall	3,505,858	881	6355:15	2103:25	1302:09	7:12

Analyzing the Time Budget of Developers

We analyze two aspects of developers' use of VISUAL STUDIO: their *time budget* (i.e., how much time they spend on each activity) and their *tool usage* (i.e., frequency of use of IDE tools). We report on both aspects and also compare to similar studies performed on different IDEs or in non-industrial settings.

For all subsequent consideration, we exclude developer days with less than 30 minutes of activity time. This leaves us with 588 developer days, accumulating 5021 hours of work time and 1255 hours of active IDE interaction. For these developer days, the average daily work time is 8 hours and 32 minutes. Figure 9.7 shows the average time budget of all such developer days. Figure 9.7a shows the overall time budget, while Figure 9.7b zooms in on the in-IDE time. Subsequently, when we talk about *a developer*, we refer to the average developer represented by this time budget.

The remainder of this section proceeds as follows: First, we present a high-level overview of a developer's working day. Second, we discuss her activities in the IDE. Third, we present a detailed analysis of her IDE-tool usage.

A Developer's Work Day Figure 9.7 shows a high-level overview of a developer's working day. We differentiate between time spend outside the IDE (a) and time spend in-IDE (b) and discuss both parts of the plots individually.

Outside the IDE A developer spends 39.8% (3h24m) of her daily work time with the application focus away from the IDE. Since FEEDBAG does not track interactions during this time, we cannot differentiate between times where she does not interact

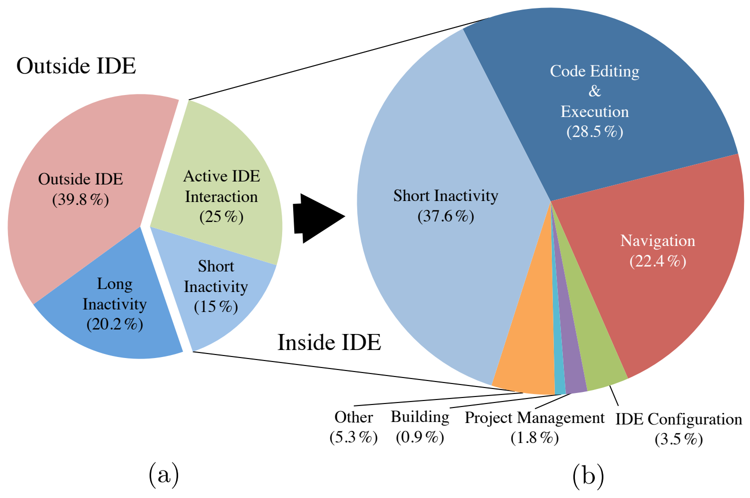


Figure 9.7: A Developer's Time Budget

with her machine from those in which she uses other applications (e.g., a browser or email client). A study by LaToza et al. [113] reports that developers indeed use many applications besides their IDE.

> Observation 1: Developers spend a considerable amount of time outside the IDE, potentially using external tools for their work.

Inactivity For a total of 35.2% (3h) of her day, a developer is within the IDE, but does not interact with it (sum of long inactivity and short inactivity in Figure 9.7a). We see two kinds of (causes for) inactivities:

1. The developer is interrupted in her work, e.g., by a phone call, a colleague, a meeting, or the lunch break. Note that the application focus does not change, if the workstation is locked or the screensaver activates, which is why times during which the developer is actual away may appear as in-IDE inactivity in our statistics. Developers face many such (unplanned) interruptions of their work [141, 177].

2. The developer stops interaction to, e.g., read code, think, or take a sip of coffee. In such cases, we record inactivity, but the developer incurs no context switch.

Note that at the time of this study, FeedBaG did not track mouse movements and scrolling. Such information might help to partially separate code reading and maybe even thinking (assuming occasional mouse movement) from actual inactivity. However, Minelli at al. [148] report that isolated mouse movements make up for only 3.5% of a developer's in-IDE time, indicating that this does not affect our time budget much.

> Observation 2: Developers spent a third of their in-IDE time not interacting with the IDE.

Since we cannot directly determine the kind of inactivity from our events, we heuristically separate inactivity intervals by their duration, reasoning that longer inactivity is more likely to be an actual interruption of a developer's work in the IDE. We separate *short inactivity* from *long inactivity* using a threshold t. Figure 9.8

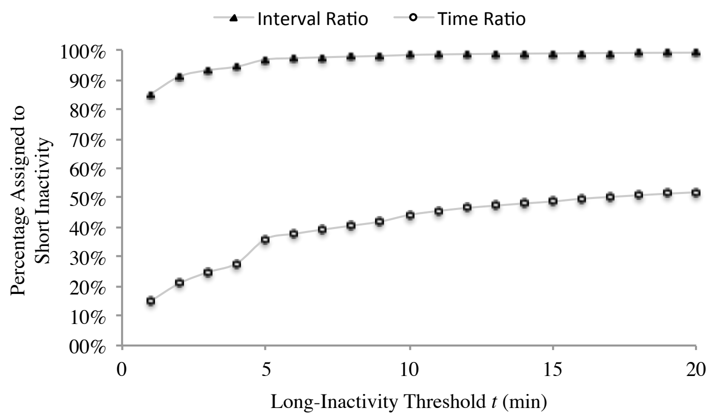


Figure 9.8: The Effect of the Threshold t on the Split of Inactivity

shows the effect of t on the percentage of inactivities and the total inactivity time that would be considered as short inactivity. We see a relatively small number of inactivities that last for more than a couple of minutes. In fact, 85% of all inactivities are shorter than 1 minute, while 97% are shorter than 5 minutes. At the same time, we see that the inactivities below 1 minute make up for only 15% of the total inactivity time and even those below 5 minutes for only 36%. In both figures, we observe the start of a saturation effect at about $t = 5$ minutes. This means developers who are inactive for 5 minutes are more likely to be inactive for even longer. With this threshold, we find a strong negative correlation between long-inactivity time and away time (Pearson's $r = -0.71, p = .01$), supporting our theory that long inactivity indicates that the developer left her machine. Therefore, we subsequently consider all inactivities of more than 5 minutes as long inactivities and exclude them from the in-IDE time. Interestingly, Minelli et al. [148] chose the same threshold to determine when a developer became idle. However, they do not report how much long-inactivity time they observe or how they determined their threshold.

During her day, a developer has 153 short inactivities, with an average duration of about 30s, and 4 long inactivities, with an average duration of about 23:18min.

Observation 3: Developers have many very short inactivities and very few long, break-like inactivities.

Active IDE Interaction Removing the time spent outside the IDE and both long and short inactivities leaves 25% (2h08m) of *active-interaction time* within the IDE. Kersten and Murphy [102] report a similar daily interaction time of about 2 hours from a diary study with six senior IBM developers. For our participants, this time is fragmented into 158 continuous-interaction periods (i.e., periods with no inactivity whatsoever) with an average duration of about 49 seconds.

What a Developer Does in Visual Studio Figure 9.7b zooms in on how a developer spends her in-IDE time. Subsequent statistics consider this time only.

Short Inactivity About 38% of a developer's in-IDE time consists of short inactivities. Recall that short inactivities are those with a duration of up to 5 minutes. The average duration of such inactivities is 30 seconds. A study by Minelli et al. [148] reports that

66% of their participants' in-IDE time is short inactivities. Possible causes for this big difference between their and our results include the work settings (industrial developer at work vs. Open Source developers in their free time), the IDEs (VISUAL STUDIO vs. PHARO), and the programming languages (C# vs. SMALLTALK).

> Observation 4: The total short-inactivity time for VISUAL STUDIO users varies significantly from that of PHARO users.

Code Editing & Execution C# developers in VISUAL STUDIO spend 28.5% of their time on code editing and execution. Minelli et al. [148] report that SMALLTALK developers in PHARO spend a mere 5.8% of their time on comparable activities, while Beller et al. [13] report that JAVA developers in ECLIPSE already spend 30.5% of their time on only editing. We imagine that reasons for these huge differences include the programming language (C# vs. SMALLTALK vs. JAVA), the study participants (e.g., experienced programmers vs. novices), the project lifecycle stage (e.g., development vs. maintenance), and the IDE UI concept (perspectives vs. floating windows).

> Observation 5: The reported times for code editing and execution varies significantly between PHARO, ECLIPSE, and VISUAL STUDIO users.

Interestingly, there is a strong correlation between the time for editing and execution and the average duration of continuous-interaction periods per developer day (Pearson's $r = .69, p = .01$). This indicates that developers who are less frequently interrupted spend more time on editing and execution or that focused developers are less likely to get interrupted. There is no significant correlation between the average duration of continuous-interaction periods and the time spent on any other activity.

> Observation 6: The average continuous-interaction time and the time for code editing and execution are strongly correlated.

Navigation For about 22.4% of her time, a developer navigates the code base. Surprisingly, we find only a weak correlation between navigation and code editing and execution (Pearson's $r = .2, p = .01$). This suggests that navigation is not necessarily a means to reach code with the intention of editing or executing it. A developer might navigate a lot without editing much and edit much without navigating a lot. On the other hand, it is interesting to see that there is a strong correlation between navigation and the number of short inactivities (Pearson's $r = .9, p = .01$). We see two possible explanations for this: Either inactivities happen when the developer becomes unsure about some property of the codebase, which she then navigates to look up, or while navigating the codebase she regularly stops to read and understand. Both alternatives suggest that the amount of navigation may correlate with the need for code understanding.

> Observation 7: The amount of navigation is a likely indicator of the need for code understanding.

IDE Configuration A developer spends 3.5% of her in-IDE time on configuring VISUAL STUDIO. In comparison, Minelli et al. [148] report that their participants spend almost 15% of in-IDE time fiddling with Pharo's UI. We speculate that this large difference might, in part, be caused by the different UI concepts of VISUAL STUDIO and Pharo. In Pharo, windows can be arranged independently and may overlap. In VISUAL STUDIO, windows are embedded into areas of the main window (perspective) and resizing one window automatically adjusts other windows such that they never overlap. Moreover, VISUAL STUDIO maintains and automatically switches between separate perspectives for debug and design mode, saving the need for rearrangement. The additional freedom provided by Pharo may result in developers spending more time on UI fiddling and configuration, to reach the perfect setup and layout for the (current) needs.

If we look at identified developers, the averages for VISUAL STUDIO configuration range from 0.4% to 21.9%. In comparison, Minelli et al. [148] report that their participants spent from 4% to as much as 30% of their time fiddling with the UI. A reason for the consistent high variance could be different experience of developers with the IDE.

Observation 8: The time spent on IDE configuration varies significantly between VISUAL STUDIO and PHARO as well as between individual developers of each IDE.

Project Management A developer spends only 1.8% of her time on project management. We know that ACME mandates Microsoft's Team Foundation Server (TFS) as the task management and versioning system. TFS is fully integrated into VISUAL STUDIO. Nevertheless, developers could choose to use IDE-external tools, like a standalone TFS client, over the integration, e.g., because of its specialized interface.

Observation 9: Developers spend little time using the IDE-integrated tools for project-management.

Building A developer spends 0.9% of her time waiting for builds. Note that this includes only the time from the start of the build to the developer's next interaction. We find that the total build times are about four times larger. This supports our intuition that developers continue working during builds.

Observation 10: Developers continue working while builds run in the background.

Other Activities Only 5.3% of a developer's in-IDE time is spent outside of the above activities. To determine our high-level overview of a developer's activities, we consider this fraction small enough to spend no more effort on assigning these interactions to our activities or come up with new ones.

How a Developer Uses Visual Studio's Tools Before we analyze tool usage in VISUAL STUDIO, we look at the usage of individual commands. Table 9.2 show the top 10 commands by absolute usage frequency. We find that our top 10 commands for VISUAL STUDIO are very similar to the top 10 Murphy et al. [156] report for ECLIPSE. The commands printed in bold font in Table 9.2 appear in both lists. Apart from these, their top 10 includes only simple keystrokes, which our noise reduction filters. However, these keystrokes are also among the most-frequent commands in our unfiltered list.

Next, we analyze how developers use VISUAL STUDIO's assistance tools. Table 9.3 shows the top 10 tools. Note that this list does not contain VISUAL STUDIO's code editor, since we consider it a core feature of the IDE. This might be different for IDEs with different editing concepts. Subsequently, we discuss those tools we can make interesting observations about. We present usage frequencies as tuples (D, I), with D being the percentage of developer days the tool is used on and I being the average number of interactions per developer day.

Code Completion The code completion is by far the most frequently used tool in VISUAL STUDIO $(87\%, 78.8)$. Note, however, that VISUAL STUDIO's code completion opens automatically, whenever the developer starts typing. This usage frequency, therefore, represents an upper bound to the number of explicit triggers. Developers possibly ignore the suggestions provided by this automatic tool. To examine this more closely,

Table 9.2: Top 10 Commands, Sorted by Absolute Usage Frequency

Rank	Command	Frequency	Usage
1	**CodeCompletion**	63,885	19.7%
2	**Debug.StepOver**	41,294	12.8%
3	**Edit.Paste**	25,840	8,0%
4	Debug.Start	23,563	7,3%
5	**Edit.Copy**	17,558	5,4%
6	**File.SaveSelectedItem**	14,417	4,5%
7	Window.NextDocumentWindowNav	7,223	2,2%
8	QueryResultsMaxRows	7,220	2,2%
9	AltEnter	6,649	2,1%
10	Build.BuildSolution	5,586	1,7%
...			

Table 9.3: Top 10 Assistance Tools, Sorted by Usages per Dev. Day

Rank	Assistance-Tool	Dev. Days with Usage	Usages per Dev. Day	Usages per Usage Day
1	Code Completion	511 (87%)	78.8	90.5
2	Build System	546 (93%)	13.4	14.4
3	Debugger	520 (89%)	12.8	14.4
4	Textual Search	521 (89%)	7.7	8.6
5	AltEnter	403 (69%)	7.7	11.2
6	Code Search	486 (83%)	7.1	8.6
7	Version Control	521 (89%)	6.4	7.2
8	(Un)Comment Code	308 (52%)	2.3	4.5
9	Unit Testing	158 (27%)	1.9	7.0
10	Data Tools	50 (9%)	1.7	20.3

we compute the frequency of code completion usages where the developer selected a proposal. This gives us a usage frequency of $(80\%, 69.5)$, which still ranks code completion as the most-frequently-used tool. When we consider only the manual invocations of code completion, the usage frequency is $(54\%, 3.7)$, which would still rank the tool 7th. This matches the results of Murphy et al. [156] who report that ECLIPSE's Content Assist is used about as frequently as standard editing commands like copy and paste. Since ECLIPSE's content assist opens only when the developer explicitly invokes it or when she types a dot, we believe that their numbers more accurately reflect the intentional usages. Unfortunately, we cannot compare usage frequency between IDEs, because Murphy et al. report only a relative frequency to other commands.

Observation 11: Code completion is the most frequently used tool.

Debugger The debugger is the third most frequently used tool in VISUAL STUDIO $(89\%, 12.8)$. This is in line with the observations of Murphy et al. [156] who report `Debug.Step` among the ten most-frequently-used commands in ECLIPSE. In contrast, Meyer et al. [141] find industrial developers to only debug 3.9% of their time.

Searches We find that both textual search $(89\%, 7.7)$ and code search $(83\%, 7.1)$ are very frequently used. Code search groups specialized searches, e.g., for usages or declarations. Singer et al. [226] make the same observation.

Observation 12: Searches are frequently used to navigate the codebase.

Quick Fix ReSharpers quick-fix mechanism AltEnter is the 5th-frequently used tool $(69\%, 7.7)$. AltEnter offers a context-specific set of *simple* refactorings when the developer presses Alt + Enter. The three most often applied refactorings are `Organize Imports`, which removes unnecessary imports, `Change Name Fix`, which changes a name to match a naming convention, and *Use Var Fix*, which replaces a type name by `C#`'s `var` keyword. This aligns with the findings of Johnson et al. [98] who report that developers adore quick fixes that automatically resolve code or style problems.

Observation 13: Developers very frequently use R#'s quick-fix tool.

Version Control A developer frequently interacts with the version control via Visual Studio's TFS integration $(89\%, 6.4)$. Moreover, she uses version control consistently, i.e., on 89% of all developer days. The only tool she uses more consistently is the build system. Since a developer spends little time on project management (O9), we conclude that the TFS integration supports her effectively in her regular usages.

Observation 14: Developers interact with the integrated version control multiple times on almost every day.

Unit Testing Testing is considered one of the main activities accompanying software development [20]. We find that the unit-testing component of ReSharper is the 9th most frequently used tool $(27\%, 1.9)$. However, it is used on little more than a fourth of all developer days. This aligns with the findings of Beller et al. [13] who report that the majority of participants in their study do not actively practice unit testing.

A caveat to this finding is that some of the developers at ACME use the NCrunch[26] suite, an automated concurrent testing tool that automatically detects and runs tests that exercise code changes. Thus, requires no explicit interaction once set up. The test results are shown in the editor, when a test class is open, and in a dedicated window. To estimate the number of NCrunch users, we count how many identified developers either interact with an NCrunch results window or execute an NCrunch configuration command. We identify 9 such developers with occurrences on 21 developer days. We deduce that the number of developers actively using NCrunch is small.

Towards Next-Generation IDEs

The observations we make during our study highlight interesting questions about IDE design. In this section, we discuss research opportunities that can answer these questions and present possible ways to investigate them further.

To Integrate or Not to Integrate? Developers spend a considerable amount of time outside of the IDE (O1). It is likely that, during this time, they use IDE-external tools for their work. Indeed, LaToza et al. [113] report that developers frequently use external tools. With several plugins available for most modern IDEs, many toolchains are now integrated into the IDE. However, it is unclear when such integration is effective and why certain tools are (not) used.

In their survey, LaToza et al. [113] find that tool usage often correlates with developer preferences. We know, however, that there are other factors that influence toolchains, like ACME's policy to use TFS. In this case, we observed that developers use VISUAL STUDIO's integrated client (O14). We assume that the reason for this is the efficiency of the integrated solution (O9). That is, both policy and efficiency may be criteria for choosing a particular (integrated) tool.

To find out more about the criteria that guide tool choices, we could extend FEEDBAG to capture the window name of the currently focused application and whether or not interactions occur. We could then generate a personalized survey to ask developers for their reasons to use the observed tools and their satisfaction in using them. Such a study would give valuable insight into which tools developers use and why. It would also help in identifying problems with developers' toolchains, and whether additional integrated IDE support can overcome them.

From the tools VISUAL STUDIO already provides, developers very frequently use code completion (O11) and quick fixes (O13). Both tools are provided through a comparatively simple-to-use dropdown that offers context-sensitive proposals. Nevertheless, we find that code completion is used much more frequently and on many more developer days than quick fix. One major difference we see between the tools is that code completion automatically opens on writing, while the quick fix requires an invocation by the developer. This aligns with the findings of Johnson et al. [98] that automation and workflow integration of tools are major factors for tool adoption. It obviously does not make sense for all tools, e.g. for automated refactorings, to open automatically. However, a profound understanding of how integrated-tool presentation impacts adoption would be valuable for IDE designers.

Why Are Developers Inactive? Developers spend a third of their in-IDE time not interacting with the IDE (O2). Besides a few longer breaks, we observe that many short inactivities (O3) heavily fragment their activities. Although we cannot say whether such fragmentation is problematic, we find that the duration of continuous activity strongly correlates with time spent on code editing and execution (O6). Thus, eliminating the reasons for short inactivities might increase developer productivity.

Minelli et al. [148] assume that short inactivities occur when the developer has to understand code. Our data and intuition tell us there are additional reasons, like when she is waiting for a test run to finish. Since we only track IDE interactions, we cannot generally identify why a developer becomes inactive. However, we believe that we can derive inactivity reasons, to some extend, from the activities surrounding the inactivity. A first experiment, analyzing which individual commands frequently precede short inactivities, did not lead to interesting findings. However, mining for larger interaction patterns might. This would help to refine our understanding of the time developers spend on program understanding. It could also help identify when developers often wait for the IDE to finish some task, thereby guiding IDE designers towards bottlenecks.

A common approach to mitigate the impact of long-running tasks, such as builds or static analyses, on developers is to run them in the background. This strategy seems successful since we find, for example, that developers do not wait for build runs (O10). However, previous studies show that expensive background computations often slow down a developer's work in the IDE or make her digress [98, 141]. To investigate such impacts, we would, first, identify which interactions trigger long-running background tasks and, second, analyze which kinds of interactions happen while such tasks are running. We could then identify developers' reactions to certain tasks, like switching away from the IDE, or the impact of tasks, like slowing down interactions.

What Causes Usage Differences between IDEs? General aspects of IDE usage, such as the time spent on code editing and execution, on IDE configuration, or in short inactivities vary significantly between different IDEs (O5, O8, and O4). To enable the understanding of how project nature and developers influence these aspects, we incorporated a respective questionnaire into the new version of FEEDBAG. We also hypothesize that the difference in the time spent on IDE configuration between Minelli et al.'s study [148] and ours might be partially caused by the different UI concept of VISUAL STUDIO and the Pharo IDE. Exploration of such usability indicators could guide IDE designers in creating future IDEs.

Why Do Developer Navigate the Codebase? The frequent usage of search tools (O12) indicates that developers often need to navigate the codebase. Due to the strong correlation between navigation time and the number of short inactivities, we hypothesize that the amount of navigation may be an indicator for a developer's need for code understanding (O7). It would be interesting to explore this hypothesis with additional datasets, especially from other IDEs, to encourage better support for code understanding, e.g., using documentation miners [154] or example providers [181].

Should IDEs Distinguish Developer Persona? The amount of time spent on IDE configuration varies significantly between developers (O8). A reason for this may be a developer's familiarity with the IDE. It would be interesting to see if future studies on various IDEs confirm such a correlation. If so, then IDE designers could use configuration time as a metric to provide specific support to new IDE users. Another reason for the observed variation in IDE configuration time might be the kind of tasks a developer performs, e.g., testing versus feature development. This information could also be used to personalize IDEs for different developer roles.

9.2.2 How Do Developer Test?

In contrast to the high-level details that have been discussed so far in the chapter, we were also interested in testing the extensibility of our solution to new research questions and the ability to better make use of context information in experiments.

Previous work by Beller et al. [13] analyzed how JAVA developers test. They applied WATCHDOG in the JAVA IDEs ECLIPSE and INTELLIJ. The experiments in their paper were based on intervals of several activities (i.e., IDE open, active periods of the developer, reading and typing in a file, test execution). The original release of our interaction tracker, FEEDBAG, has captured any commands initiated by the developer so all of these activities were already included in our event stream. However, for test executions, we have only captured that an execution was initiated, but no further details about the individual tests. The enriched event streams captured by the extended release, FEEDBAG++, provide an opportunity to extend the study to VISUAL STUDIO though. In the extended release, we added an instrumentation of the RESHARPER test runner. It captures the names of each executed test, as well as duration and result of the run. We then designed a *test event* data structure to store the relevant information.

A technical difference is that FEEDBAG++ captures (and uploads) a fine-grained event stream, whereas WATCHDOG lifts this stream to intervals on the client side and only uploads the resulting intervals. Intervals capture when and for how long an activity took place. We implemented an offline conversion in C$\boxed{A}$RET from enriched event streams to the intervals described in their paper to make our enriched event

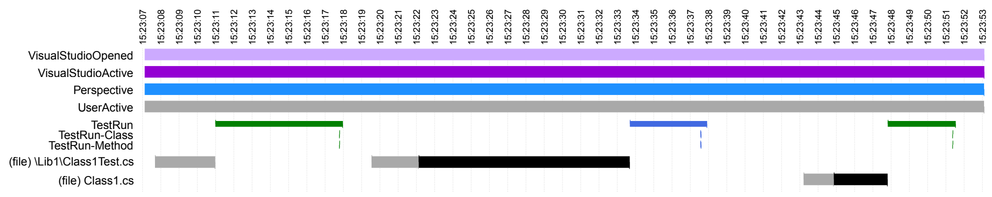


Figure 9.9: Visualization of Testing Activities in the Interval Debugger

stream compatible with WatchDog. The original authors confirmed that the created intervals are fully compatible and useable to run the experiments in their pipeline.

The logic behind maintaining the intervals is complex and depends on various events that occur in the IDE. We have created an extensive test suite as a means to clearly communicate the expectations with the WatchDog team. However, the events captured in real deployments do not always arrive in a deterministic order and it is hard to automatically write unit tests for all cases. To allow debugging the interval export and to enable early spotting of errors, we have implemented a visualization tool for the interval creation. You will find a screenshot of the visualization in Figure 9.9.

Please note, that testing related events have not yet been collected in the industrial FeedBaG deployments. However, the dataset collected in the field study contains them, which allowed us to extract testing information for Visual Studio that we could provide to the WatchDog team. While this project is still an on-going collaboration, it provides an indication of the research possibilities that enriched event streams open up. It shows that having FeedBaG++ made it easy to extend WatchDog to a new IDE. It also shows that enriched event streams already contain a wide range of context information and that new generators can be added to capture more.

Chapter Summary

In this chapter, we have analyzed the applicability of C$\boxed{\text{A}}$RET and its data structures in studies on the development process. To this end, we have conducted studies that were concerned with source-code evolution and developer interactions.

First, we have studied the typical edit location of developers in a method and have created a new tool for source-code differencing. We could show that SSTs are a solid source-code representation for these tasks and the first-hand experience also revealed opportunities for further improvement.

Second, we have considered both high-level and very detailed information about the development process. The very broad range of activities that are covered by enriched event streams allowed us to analyze the time budget of developers. The specialized context information that are captured in addition can be used to study more in-depth questions, e.g., we extended an existing study of testing behavior to Visual Studio.

Overall, it is to say that the enriched event stream dataset presents an excellent opportunity for research on the development process. We see grand potential for future studies that go far beyond our initial experiments.

10 Building Recommendation Systems for Software Engineering

Recommendation systems are an integral part of modern Integrated Development Environments (IDEs). They reduce the amount of typing required, thus accelerating coding, and are often used by developers as a quick reference for the *Application Programming Interface* (API), because they show which fields and methods can be used in a certain context. The context is typically determined by the static type of the variable on which the developer triggers the completion. However, using the static type of the variable as the only criteria for determining the developer's context may produce countless recommendations and reduces the effectiveness of the system.

Over the years, many approaches have been proposed to support developers in various activities, e.g., call completion [195], snippet recommender [169], code search [262], or anomaly detection [152]. All proposed approaches have in common that they apply static analyses on the edited code to infer their proposals.

One of our main goals when designing SSTs was to allow the ability of building extensive static analyses on top of them and to support re-occurring analysis tasks with reusable components from C$\boxed{\text{A}}$RET. We will discuss our own experience with our platform in this chapter to evaluate the extent to which we have achieved these goals. Our main interest is evaluating whether the SST representation is rich enough to re-implement existing RSSE and how well C$\boxed{\text{A}}$RET supports the development with reusable components. We are going to study these questions in two steps.

First, we will present an in depth explanation of one line of research that has created three different kind of recommenders. While they are now shipped with C$\boxed{\text{A}}$RET, we do not count them as a core component of C$\boxed{\text{A}}$RET, as they are specific to method-call recommendation. However, they are now reusable by others and provide a great example that the platform indeed works as the research-oriented base for which it was designed. Second, we will sketch the creation of other existing approaches for which we did not had the resources for an actual implementation. We plan to do this as future work and only provide *cookbook recipes* that introduce the necessary steps for re-creation at a technical level to show the possibility using C$\boxed{\text{A}}$RET.

10.1 Recommendations Based on Object Usages

In a first use case that allowed us to test the capabilities of SSTs and provided first experience in applying C$\boxed{\text{A}}$RET in a real development setting, we have built three

```
class MyDialog extends Dialog {
  Text t;
  @Override
  void createDialogArea(Composite parent) {
    t = new Text(parent);
    t.□ // code completion invoked here
  }
}
```

Identifier	Type	Definition	Call Site	Kind
`parent`	`Composite`	Parameter	`Text.<init>(Composite)`	Parameter
`text`	`Text`	New	`Text.<init>(Composite)`	Receiver

Figure 10.1: Example of a Code Completion and Extracted Information

RSSE. All approaches rely on *object usages* [22], a representation that describes all the context information relating to a single *object* within the boundaries of an enclosing method. For example, the enclosing method declaration as the *method context* or which methods were invoked on the object, among other detailed information.

While the approaches share the same input representation, they were development independently. Bruch et al. [22] introduced the *Best Matching Neighbor* (BMN) approach. We built an extension [192], *Pattern-based Bayesian Networks* (PBN), that significantly improves scalability through clustering the input data. Originally, our extension was built in separation and it made us experience the need for a shared infrastructure first hand. Once we had built C$\boxed{\text{A}}$RET, we re-implemented both BMN and PBN on top of it. Later, we further improved the internal clustering approach through *Boolean Matrix Factorization* (BMF) in a specialized variant of PBN.

In this section, we will go through the history of this line of research. We will start with a detailed explanation of the static analysis that extracts object usages from source code in Section 10.1.1. After that, we will introduce the three approaches BMN (Section 10.1.2), PBN (Section 10.1.3), and PBN_{BMF} (Section 10.1.4), which all are available as implementations in the C$\boxed{\text{A}}$RET platform.

10.1.1 Static Analysis

To learn how specific types of an API are typically used, we implemented a static analysis that extracts *object usages* from source code. The term object usage refers to an abstract representation of how an instance of an API type is used in example code and consists of two pieces of information: (1) all methods that are invoked on that instance and (2) the *context* in which the object usage was observed. The first captures the method calls that should be later proposed in the intelligent code completion. The second captures all information about the surrounding source code, i.e., the enclosing method and class. This idea is based on prior work [22], but it was not clearly described in the resulting publication. Our work contains a detailed description of the static analysis to allow other researchers to build their own implementations. Additionally, we extract more information than the original publication.

Reusable Context Information The goal of the static analysis is to extract as much *re-usable* context information from the source code as possible. In Object-oriented Programming Languages, there are two ways to make functionality reusable and to distribute it to others: *libraries* and *frameworks*. Libraries provide useful helpers that can be reused anywhere in your code. For example, the *Java Database Connectivity* (JDBC) API is an example of a library API. In such a context, an intelligent code completion engine can learn API protocols or which methods are usually invoked together. However, the surrounding source code usually refers to user-specifics, i.e., the names are unique and will never be re-used by other developers. This does not provide any information relevant for intelligent code completion. Contrary, a framework (e.g., Swing, the JAVA UI widget toolkit) is an example of the *inversion of control principle* [131]. The framework authors decided for well-designed extension points and provide base classes and interfaces as the mean for extension. Custom extensions extend or implement these points and use other building bricks of the framework. Because of that, the extensions of a framework contain useful pointers to overridden methods that can also be observed in source code of other developers. While intelligent code completion systems can also be provided for libraries, we focus on frameworks, because the surrounding code contains more context information and the intelligent code completion can provide more specific proposals.

Consider the example from Figure 10.1: the user-specified `MyDialog` extends the framework class `Dialog`. Learning how objects are usually used in a context referencing `MyDialog` does not provide shareable knowledge, because other developers will most likely name their user-specific subclasses different than `MyDialog`. The extension point that was intended by the author of the framework was `Dialog` so we would reference this as the enclosing type. The enclosing method is even more concrete and follows the same pattern. Instead of pointing to `MyDialog.createDialogArea`, the analysis extracts `Dialog.createDialog` as the enclosing method. By going up in the class hierarchy as much as possible, we increase the likelihood that others use the same classes. This is valid because, according to the *Liskov Subsitution Principle* (LSP), the contract of all subclasses must not break the contract of the super class [131].

In addition to the information about the enclosing method, we further extend the notion of a *context* in this work and extract more information than in the original publication. We also capture the enclosing type context, all method invocations to which an object usage was passed to as parameter, and information about the definition of an object.

Entry Points and Tracking We assume that the typical usage of a type is context dependent, therefore, we collect object usages separately for each context. Thereby, each public or protected method is considered as a single context and is used as an entry point for the analysis. Private methods do not form a context on their own, because they were created by the author of the concrete class, do not belong to the abstraction expressed in the base class or interface, and do not carry any reusable information. A call graph is computed for each entry point method p_{entry}, in which all method invocations are pruned that leave the enclosing class. Additionally, all exception-handling edges are pruned from the intra-procedural control flow graph.

An object usage is created for every distinct object instance used in the scope of p_{entry}. We use the call graph to track the object instances inter-procedurally in the class. The tracking stops on calls leaving the current class (e.g., calls to methods of other classes) or on calls in the current class that are either entry points or defined in

```
public class A extends S {
  private B b = new B();
  @Override
  public void entry1() {
    b.m1();
    helper();
    C c = fromS();
    c.entry2(b);
  }
  private void helper() {
    b.m2();
  }
}
```

```
public class C {
  public void entry2(B b) {
    b.m3();
    entry3();
  }
  protected void entry3() {
    D d = new D();
    try {
      d.m4();
    } catch(Exception e) {
      d.m5();
    }
  }
}
```

Entry Point	Type	Class Context	Method Context	Definition	Call Site	Call Site Kind
`A.entry1`	`B`	`S`	`S.entry1`	Field	`B.m1()`	Receiver
					`B.m2()`	Receiver
					`C.entry2(B)`	Parameter
`A.entry1`	`C`	`S`	`S.entry1`	Return	`C.entry2(B)`	Receiver
`A.entry1`	`S`	`S`	`S.entry1`	This	`S.fromS()`	Receiver
`C.entry2`	`B`	`Object`	`C.entry2`	Parameter	`B.m3()`	Receiver
`C.entry3`	`D`	`Object`	`C.entry3`	New	`D.m4()`	Receiver

Figure 10.2: Object Usages Extracted From Various Entry Points

a super class. In case we find a call to a private method, we step down in this method and *track* all objects of the current scope in the private method.

Example We illustrate our static analysis in Figure 10.2. When starting at method `A.entry1()`, the analysis stores the method calls `B.m1()` and `B.m2()` on field `b` as well as the call to method `C.entry2(B)`, for which `b` is used as an actual parameter. The call to method `B.m2()` is stored, because the private method `A.helper()` is called from within the entry point `entry1()` and `b` is tracked in it. `b` is not tracked in method `C.entry2(B)` (i.e., `B.m3()` is not stored) - even though it is called from within `A.entry1()` - because it is declared in another class than p_{entry}. Instead of tracking `b` to this method, the static analysis stores the information that `b` is passed as an actual parameter to it. `C.entry2(B)` is another entry point for the static analysis. A separate object usage is created that extracts information about the usage of type `B` in the context `C.entry2(B)`.

The interpretation of method invocations on `this` depends on the place of definition of the target method. Consider the call to `A.helper` in `A.entry()`. It is defined in the same class and it is no entry point, so the analysis steps down into the method and tracks all objects in it. In contrast to that, the call to `S.fromS()` is not tracked, because it is defined in another class. Objects are never tracked into calls to other entry points, independently of the defining class (same class or other class).

Data structure The following list describes all properties that are collected for an *object usage*. The properties *Parameter Call Sites*, *Class Context*, and *Definitions* are introduced in this work. All other properties were previously used and are just included to complete the description.

Type The type name of a variable or, if discernible, the more specific type of an object instance. For example, if a `String` is assigned to a variable of type `Object` then `String` is stored as the type of this variable, if this is always the case.

Call Sites: All call sites connected with the instance. These can be of two kinds:

Receiver Call Sites are method calls invoked on the object instance. The statically linked method is stored. In the example in Figure 10.2, these are the methods `m1()` and `m2()` for variable `b` starting from entry point `entry1()`.

Parameter Call Sites are stored if the object instance is passed as the actual parameter to another method. The invoked method is looked up in the type hierarchy and the first declaration is stored, which can be an interface or an abstract base class to store the most reusable context reference. In the enclosing method `entry1()`, the method call `entry2(B)` is an example of a parameter call site for variable `b`. The argument index at which the object was passed is stored as well (omitted for brevity in the example).

Class Context The direct super type of the class enclosing p_{entry}. In the example, the class context is type `S` for `entry1()`, and `Object` for `entry2(B)` and `entry3()`.

Method Context The first declaration of p_{entry} in the type hierarchy, i.e., the method declaration of a super type that is overridden by p_{entry} or the method declaration in an interface that is implemented by p_{entry}. In the example, `A.entry1()` overrides `S.entry1()`, therefore, `S.entry1()` is used as method context. If no method is overridden then the method itself is used as the context.

Definition Information about how the object instance became available in p_{entry}. We distinguish five different definition kinds, each carrying specific information:

New The instance is created by calling a constructor[1]; the specific constructor is stored for this kind of definition. In the example, the object usage extracted for variable `d` in `entry3()` is defined by a call to a constructor. Note that field `b` of class `A` is not recognized as having a *new* definition, because the constructor is not called as part of the considered entry point.

Parameter The instance is a formal parameter of p_{entry}; the parameter index is stored. In the example, the object usage extracted for `b` in `entry2(B)` is defined by a parameter.

Return The instance is returned by a method call; the name of the invoked method is stored. In the example, the object usage extracted for `c` is returned by a method call.

Field This definition kind denotes that the reference to the object was received by accessing a class field. We store the fully-qualified name of this field. In the example, the variable `b` used in `entry1()` is recognized as field.) If the field is

[1] In [22] constructor calls were treated as method calls. With the introduction of definition kinds as part of context information in this work this is changed.

	`Dialog.` `createDialogArea`	`ModifyListener.` `modifyText`	`<init>`	`setText`	`getText`
Usage 1	1	0	1	0	0
Usage 2	1	0	1	1	0
Usage 3	0	1	0	1	1
Query	1	0	1	?	?
Proposal				50%	0%

Figure 10.3: Proposal Inference for BMN

initialized in p_{entry}, e.g. assigned by a constructor call, then we refer to this definition instead, assuming it to be more precise.

This Denotes that the object usage was the receiver of the call to p_{entry}. The same rules as for the enclosing methods apply here: only calls to methods defined in any super type are collected. `S.fromS()` is an example of this definition in `A.entry1()`. However, the method `A.helper()` is not included, because it is defined in the same class as the entry point.

The static analysis always captures *fully-qualified* references to types, fields, and methods. A fully-qualified reference to a type includes the package name and the type name. A fully-qualified field name includes the fully-qualified declaring type, as well as the name and the fully-qualified type of the field. For a method it is the fully-qualified type of the declaring type, the method name, the fully-qualified return type, and fully-qualified types of parameters.

Implementation We have two implementations for this system. The original approach was implemented to analyze arbitrary JAVA bytecode using WALA[6], the static-analysis toolkit for JAVA bytecode of IBM. Some context information is extracted from the class hierarchies so all types in the example code need to be fully-resolvable. We use the 1-CFA [221, 222, 223] implementation that is available in WALA for the points-to analysis to track object instances inter-procedurally. Note that the order of call sites cannot be retained because the 1-CFA implementation is flow insensitive. This is not an issue in our case since we already store call sites as unordered sets.

To analyze the applicability of SSTs, we have rebuilt the system in C🄰RET using the dataset of released source code. The embedded typing information of the SST representation made it easy to get started with the dataset, because there is no need to compile it. The revised static analysis was implemented as a visitor on SSTs. We made use of the points-to analysis provided by C🄰RET to identify potential object instances and collected the required information while traversing the syntax tree.

10.1.2 Best Matching Neighbor

The *Best Matching Neighbor* (BMN) algorithm [22] is inspired by the *k-Nearest-Neighbor* (kNN) algorithm [35] and leverages domain knowledge specific to object usages. BMN represents each object usage as a binary vector, in which each dimension represents an observed *method context* or a *receiver call site*. In detail, each binary vector representing an object usage contains a “1” for each context information and call site that applies for the object usage at the corresponding dimension, and "0" otherwise. The model for each object type is a matrix that consists of multiple object usages, i.e., binary vectors. Each column in the matrix represents a context information or a call site. Each row represents an object usage found by the static analysis. For illustration, the matrix for the imaginary type `Text` is shown in the upper part of Figure 10.3. *Usage 1* is equivalent to the object usage of the `Text` widget listed in Figure 10.1.

Queries are partial object usages and treated similarly. A query is represented as a vector with the same dimensions as the vectors of object usages in the model. Each observed context information and all call sites already connected to the variable for which the completion is invoked are marked with a "1" in that vector. Context information that does not match is marked with a "0". All receiver call sites, which are not contained in the query, are potential method proposals, this is illustrated by marking them with "?". If a developer triggers code completion at the position illustrated in Figure 10.1, the query shown in the lower part of Figure 10.3 is generated.

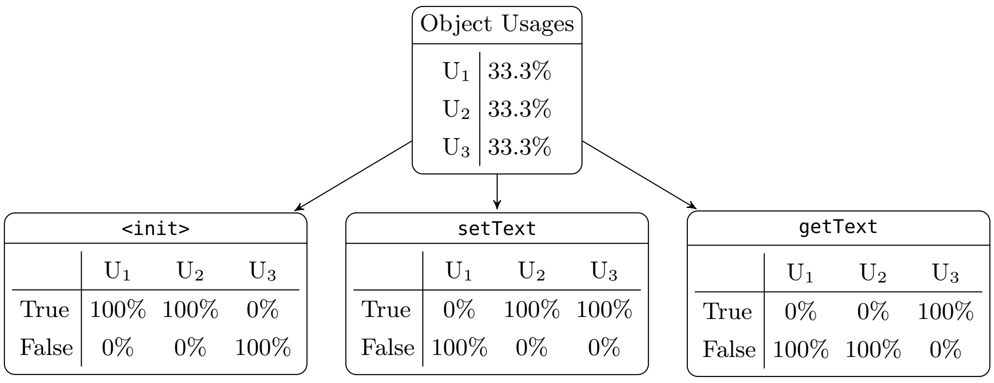
The nearest neighbors are determined by calculating the distance between the object usages and the query. The Euclidean distance is used as distance measure, whereby only dimensions containing a "1" or "0" in the query vector are considered. However, receiver call sites that do not exist in the query are not included in the calculation, because it cannot be decided if they are missing on purpose or if they should be proposed. The *nearest neighbors* are those *object usages* with the smallest distance to the query. Unlike kNN, all neighbors that share the smallest distance to the query are selected, not only the k nearest neighbors. In our example, *Usage 1* and *Usage 2* have both distance 0 and *Usage 3* has distance $\sqrt{3}$, thus the former two are nearest neighbors.

The nearest neighbors are considered for the second step of computing proposals. For each potential method proposal, the frequency of this method is determined in the set of nearest neighbors. The probability is computed by dividing it by the total number of nearest neighbors. In the running example in Figure 10.3, the call to `setText` is contained in one out of the two *nearest neighbors* of the query. Therefore, the call is recommended with a probability of 50%. The call `getText` is not contained in any nearest neighbor, the probability is 0%. The call sites are passed to the completion system as proposals for the developer, ordered by probability.

BMN was re-implemented for this work to be usable in the evaluation. We optimized it for inference speed of proposals and for model size. For example, we introduced an additional `counter` column in the matrix. Instead of inserting multiple occurrences of the same row, we increase the counter. In addition, we extended the model by more context information, represented as additional columns in the model matrix. Originally, only the method context and receiver call sites were included and we added support for the class context, definition, and parameter call sites to obtain a fair comparison of `BMN` and `PBN`. The inclusion in the table is configurable for all context information. We use this to evaluate if the extra context makes a difference.

10.1.3 Pattern-based Bayesian Networks

This section introduces a novel approach for intelligent code completion called Pattern-based Bayesian Network (PBN) [192]. Bayesian networks are directed acyclic graphs that consist of *nodes* and *edges*. A node represents a *random variable* that has at least two *states*, which again have probabilities. The states are complete for each



Object Usages	
U$_1$	33.3%
U$_2$	33.3%
U$_3$	33.3%

`<init>`

	U$_1$	U$_2$	U$_3$
True	100%	100%	0%
False	0%	0%	100%

`setText`

	U$_1$	U$_2$	U$_3$
True	0%	100%	100%
False	100%	0%	0%

`getText`

	U$_1$	U$_2$	U$_3$
True	0%	0%	100%
False	100%	100%	0%

Figure 10.4: Conditional Probabilities in a Bayesian Network

random variable and the sum of their probabilities is always 1.0. Nodes can be connected by directed edges that indicate that the probabilities of the target node are conditioned by the state of the source node. A Bayesian network can be used to answer probabilistic queries about the modeled random variables. If the state of one variable is observed, this knowledge can be used to infer the updated probabilities of other variables. This has already been used in other areas, for example to rank web searches [27] or for recommendations in social networks [257].

Using Bayesian Networks as Code Completion Models PBN uses a Bayesian network to identify how likely specific method calls are, given that context information and potentially even other method calls have already been observed. The idea is to describe the probability of a method call conditioned by a specific object usage. We apply Bayes' theorem to answer the reverse question: how likely is a specific object usage, given that a method call is already present in the current code. This information can then be used to infer how likely other yet not present method calls are.

Figure 10.4 shows the Bayesian network for the example from Figure 10.3. The *Object Usages* node has the states U$_1$, U$_2$ and U$_3$, representing *Usage 1* to *3* of Figure 10.3. We have observed three object usages and each of them exactly once, thus each has a probability of 33%. The remaining three nodes represent the method calls `<init>`, `setText`, and `getText`. The edge from the *Object Usages* node to the method call nodes indicate that the probabilities of the calls are conditioned by the object usage. The states of each method call node are *True* and *False*, which represents whether the method call appears in an object usage or not. For example, the call `<init>` is present in *Usage 1*, but neither are `setText` nor `getText`. Therefore, the conditional probabilities for the different methods are:

$$P(\texttt{<init>}|\mathrm{U}_1) = 100\% \qquad P(\texttt{setText}|\mathrm{U}_1) = 0\% \qquad P(\texttt{getText}|\mathrm{U}_1) = 0\%$$

Please note that the correct notation is $P(\texttt{<init>} = true|\mathrm{U}_1)$, but the check for the state is omitted for brevity. Although this kind of data is easily extractable from example object usages, we want to answer a different kind of question in the use case of code completion. It can be observed, for example, that the constructor `<init>` is called on an instance of type `Text`. The developer wants to know which method call is missing. Therefore, the probabilities of all method calls are calculated in such a case and method calls with a high probability are proposed as missing. Hence, if the method `setText` is one of the possible calls, we want to calculate the probability $P(\texttt{setText}|\texttt{<init>})$.

For the following equations, which show the calculations to answer the above question, we will need the probability of $P(\texttt{<init>})$. It is defined as the sum of the joint probabilities of `<init>` and each usage U_i:

$$\begin{aligned} P(\texttt{<init>}) &= \sum_{i=1}^{3} P(\texttt{<init>}, \mathrm{U}_i) \\ &= 0.333 + 0.333 + 0 \\ &= 0.667 \end{aligned}$$

The probability of the method call `setText`, given that `<init>` was called before:

$$P(\texttt{setText}|\texttt{<init>}) = \sum_{i=1}^{3} P(\texttt{setText}, \mathrm{U}_i|\texttt{<init>})$$

Assuming the independence of all methods, Bayes' theorem can be applied to calculate the probability: [2]

$$\begin{aligned} P(\texttt{setText}, \mathrm{U}_1|\texttt{<init>}) &= P(\texttt{setText}|\mathrm{U}_1) \cdot P(\mathrm{U}_1|\texttt{<init>}) \\ &= \frac{P(\texttt{setText}|\mathrm{U}_1) \cdot P(\texttt{<init>}|\mathrm{U}_1) \cdot P(\mathrm{U}_1)}{P(\texttt{<init>})} \\ &= \frac{0 \cdot 1 \cdot 0.333}{0.667} = 0 \end{aligned}$$

The calculations of $P(\texttt{setText}, \mathrm{U}_2|\texttt{<init>})$ and $P(\texttt{setText}, \mathrm{U}_3|\texttt{<init>})$ are similar:

$$\begin{aligned} P(\texttt{setText}, \mathrm{U}_2|\texttt{<init>}) &= \frac{P(\texttt{setText}|\mathrm{U}_2) \cdot P(\texttt{<init>}|\mathrm{U}_2) \cdot P(\mathrm{U}_2)}{P(\texttt{<init>})} \\ &= \frac{1 \cdot 1 \cdot 0.333}{0.667} = 0.5 \end{aligned}$$

$$\begin{aligned} P(\texttt{setText}, \mathrm{U}_3|\texttt{<init>}) &= \frac{P(\texttt{setText}|\mathrm{U}_3) \cdot P(\texttt{<init>}|\mathrm{U}_3) \cdot P(\mathrm{U}_3)}{P(\texttt{<init>})} \\ &= \frac{1 \cdot 0 \cdot 0.333}{0.667} = 0 \end{aligned}$$

By combining the intermediate results, $P(\texttt{setText}|\texttt{<init>})$ can be calculated:

$$\begin{aligned} P(\texttt{setText}|\texttt{<init>}) &= \sum_{i=1}^{3} P(\texttt{setText}, \mathrm{U}_i|\texttt{<init>}) \\ &= 0 + 0.5 + 0 \\ &= 0.5 \end{aligned}$$

The interpretation of this result is that if `<init>` is observed for an object instance, a call to the method `setText` has a probability of 50%. The same calculations can

[2] Even though the methods might not be independent, previous work has shown that no direct correlation exists between the accuracy and the degree of feature dependencies [198]. We will show in our experiments that the accuracy is comparable to existing techniques.

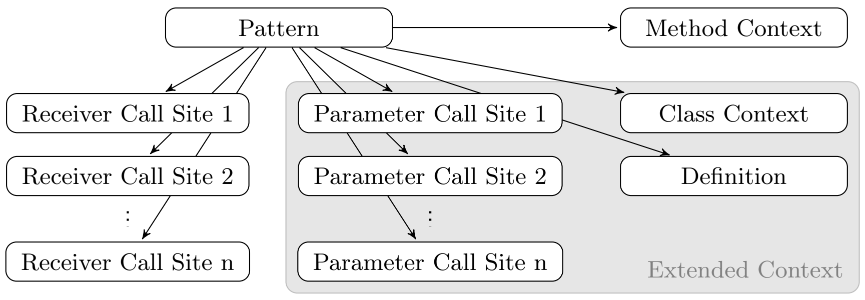


Figure 10.5: Structural Representation of the Bayesian Network used in PBN

be done to reason over `getText`. Generally, all states of context information and all calls present in the query are used as evidence in the Bayesian network. Accordingly, the probabilities of all remaining receiver calls are inferred with respect to these observations. These calls are collected in a proposal list that is ordered by probability.

If the query contains a combination of features never observed in the training data, proposals might be incomputable. To avoid such cases, we implemented *add-delta smoothing* in our learning algorithm [29]. A small delta of 0.000001 is added to all probabilities and their sum is normalized to 1.0 afterwards.

Adding Context Information to the Network Structure So far, the Bayesian network presented in the example above does not include context information. To add the context information, the model is extended by adding more nodes that are conditioned by the object usage node. For example, the method context is modeled by a node that contains one state per observed method context. In contrast to call site nodes that only have two states, *true* and *false*, a state in the method context node corresponds to a method name and the number of states is not limited. Assume that the node contains M method contexts and, as it is conditioned by the pattern node with N patterns, the probability table of this node contains $M \times N$ values. Each value describes how the corresponding method was observed, given a specific pattern. Other context information nodes are added in the same way.

The complete Bayesian network that will be used in experiments is illustrated in Figure 10.5. All nodes for call sites contain only two states, the nodes of the method context, the class context, and the definitions contain multiple states. Note the distinction between *receiver call sites* and *parameter call sites*.

Additionally, we changed the name of the root node from *Object Usages* to the more generic name *Pattern*, because the states of this node do not necessarily map exactly to observed object usages. In fact, we collapse all object usages that have the same receiver call sites into a single *pattern* state. Therefore, context information previously having conditional probabilities of 0% or 100% will now be represented by their frequency in the set of collapsed object usages. For example, consider the case of two object usages for which both the method calls `m1()` and `m2()` were invoked. Additionally, assume that one usage was observed in method context `C1` the other in `C2`. Both will be represented in a single state P of the pattern node, because they refer to the same combination of method calls. As the context was different for both, both method contexts `m1()` and `m2()` each have a probability of 50% given the pattern P. The probability of all states in the pattern node is calculated by normalizing their frequency.

Introducing Clustering for Improved Learning The number of detected patterns has a major impact on the size of the resulting model. Each detected pattern creates a new state in the pattern node so its size grows accordingly. Additionally, the number of stored values in all other nodes also depends on the number of patterns, because all nodes are conditioned by the pattern node. Each conditioned node needs to store an amount of values equal to the product of its own states and patterns. Therefore, reducing the number of states in the pattern node has a huge positive impact on the model size. We propose to use a clustering technique to reduce the number of detected patterns. Information may be lost in this process, because multiple *similar* object usages are merged and the quality of recommendations could be affected. On the other hand, clustering ensures the scalability of our approach. It is necessary to find a reasonable trade-off between model size and proposal quality.

We implemented a learning algorithm that is inspired by Canopy Clustering [136] to detect patterns for the PBN model. Similar to Canopy, a random object usage is chosen from the set of all object usages for a specific type. This object usage becomes a cluster center. Each object usage that is closer to this cluster center than a specific threshold is assigned to the cluster and is removed from the set of object usages still to be assigned to clusters. The algorithm proceeds until all object usages are assigned to clusters. Each cluster becomes a state of the pattern node in the resulting Bayesian network. The probability of the pattern state is the number of object usages in that cluster divided by the total amount of object usages. Each value, i.e., call site or context that was set in any object usage belonging to the cluster, gets a conditioned probability reflecting the frequency of the respective value in the cluster.

To determine the distance between two object usages, the cosine similarity [235] is used, which is also a common choice in the research area of information retrieval. There, vector representations have similar characteristics as in the context of representing object usages: they are typically sparse and high dimensional. Cosine similarity can deal with such vector characteristics well [218]. It is defined as the angle between two *vectors* v_1 and v_2:

$$d_{cosine} = 1 - \frac{v_1 \cdot v_2}{|v_1| \cdot |v_2|}$$

It has a helpful property for distance calculation between object usages: If vectors differ, their distance gets smaller with the number of (set) dimensions they have in common. For example, the distance between two vectors that differ by one call without having another call in common is bigger than the distance between two vectors that differ by one call but that have one or more calls in common. Note, that 1.0 is the maximum distance calculated by cosine similarity. A geometrical interpretation of this distance is that two vectors are orthogonal and no dimension is set for both.

Although this clustering approach is very simple, our experiments show reasonable results. The algorithm is fast and can handle huge amounts of data. Also, it implicitly solves the question of how many patterns are to be found, which for many clustering algorithms must be defined upfront. Additionally, the distance threshold can directly control the trade-off between prediction quality and model size. For example, the minimum threshold 0.0 results in practically no clustering at all. The higher the value chosen, the more information will be lost and the smaller the size of the model will become. In the following sections, we will encode the concrete threshold used for a PBN instance in the name, e.g., a PBN_{10} instance uses a distance threshold of 0.10. By using a threshold of PBN_{100}, all usages are merged into a single pattern. In that

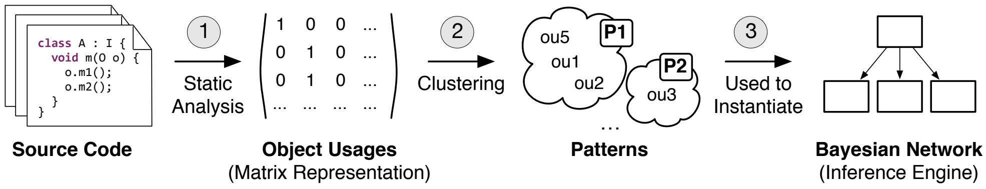


Figure 10.6: Pipeline

case, the model degenerates to a simple call-frequency based model in which context information has no influence on the proposals anymore.

10.1.4 Boolean Matrix Factorization

The PBN recommender is designed as an extensible inference engine for method completion. The complete picture of the PBN workflow to get from the source code to a recommender model that can be used in the PBN inference engine is shown in Figure 10.6 and consists of three steps. First, in a *static analysis* the object usage representation is extracted from the source code. Second, a *clustering* algorithm extracts abstract usage patterns from the object usages. Third, these patterns are used to *instantiate* a Bayesian network that builds the underlying model for the PBN inference engine. The second step is the most interesting one for this section, because the concrete algorithm that is used for the clustering is extensible.

The original implementation has used a variant of the canopy-clustering algorithm, which simply groups similar data points. While this already reduces the dimensionality of the data significantly, it does not introduce any generalization that can abstract from corner cases and, therefore, is prone to noise in the data. We have replaced this basic clustering approach with a more sophisticated alternative to prove the extensibility of the pipeline and to get first experience with integrating related work [26].

We chose to exchange the clustering with a *Boolean Matrix Factorization* (BMF) technique [110, 145]. Traditionally used to decrease the dimensionality of large matrices by decomposing them into two smaller factor matrices, we use it as a more sophisticated clustering approach for the PBN pipeline that is able to cope with noisy data. The resulting PBN_{BMF} recommender reuses most of the infrastructure of the PBN recommender, with the exception of the clustering. It is also based on object usages and returns mined patterns in the same format as the original clustering. The goal of this section is to discuss the extensibility of PBN and its adaptability to new algorithms; we will not discuss the details of the BMF clustering itself.

Through the integration of the improved clustering, we were able to show that the quality of the clustering can indeed be improved significantly. BMF detects fewer patterns while keeping the prediction quality on par with the previous clustering. Obviously, the BMF clustering is better suited for abstracting the data and is less prone to noise. As a result, both the memory consumption and the required time for inference is reduced and the recommender shows better scalability properties, opening up the opportunity to further increase the amount of input data. However, the current implementation of the algorithm suffers from the excessive requirements of computing power, which makes the approach in its current form less applicable in practice.

We have exchanged the clustering as an evaluation for the extensibility of the existing pipeline. The scalability issue that we have observed did not reveal a limitation of the PBN pipeline, but was caused by the implementation choice to use MATLAB for the matrix factorization. Having access to only a single license prevented any parallelization that goes beyond the available cores in the experimental machine. In addition to this, we found two limitations of the pipeline that we could both solve generically.

First, the pipeline did not allow for *out-of-process* execution of tools, it was expected that all subtasks are implemented in JAVA and can be run from within the JVM. We solved this by splitting up the execution of the pipeline and stop the pipeline after a new *pre-clustering* state to allow for external processing. The new state can be used, for example, to serialize intermediate results that are then processed externally. Once the pipeline continues, it starts in the clustering step. In case of BMF, we store the object usages in the pre-clustering step, run Matlab offline for the factorization, and just read the factorization results in the continued pipeline execution.

Second, the original pipeline design made use of configuration data structures to control mining and evaluation. This has the advantage that the pipeline can be configured in a type-safe way. However, while the evaluation options are stable (because the evaluation setup does not change), the mining options are specific to the concrete technique. The necessity to extend the options for every new technique clearly breaks the *Open-Closed-Principle* [142]. We decided to encode specific options in a `string` that has to be parsed individually by each technique, sacrificing guaranteed type-safety of the configuration options for generality.

Once these two minor limitations have been fixed, we could easily rerun previous experiments designed for the PBN recommender using the new PBN_{BMF} variant. Overall, the PBN inference engine has proven to be a solid ground for building recommendation engines that are based on object usages. The lessons learned from integrating a different clustering technique were an important first step for us in the design of a reusable benchmark that can be reused by arbitrary method call recommenders.

Section Summary

In this section, we presented one line of research that has built three RSSE on the basis of object usages. We discussed the static analysis that extracts object usages from source code. In addition, we have introduced three RSSE that we built on top of object usages. The first hand experience in creating the RSSE approaches allowed us further insights into the requirements towards a shared platform and significantly affected the design of C🅰RET. Our re-implementation on top of C🅰RET has shown that the creation of a common pipeline for different recommender approaches is feasible. In addition to the idea of standardizing the infrastructure, we have also developed the idea to separate the model creation and the evaluation part to allow more realistic results. We combined both ideas and were able to create a general benchmark that facilitates the creation and evaluation of method-call recommenders. We will present details about this benchmark in Chapter 11.

10.2 Sketching the Creation of Additional Approaches

To show the general applicability of the C🅰RET platform and our datasets when creating source-based recommendation systems for software engineering, we will sketch now, how several existing systems could be re-implemented. Providing a sketch that is fully compatible to an existing technique proved to be hard. Typically, no sources are available and the descriptions in the original publications are naturally reduced and very dense, because of the limited space. To mitigate the risk for misinterpretation, we have intensively discussed our sketches with our peers that know the corresponding works. In our description, we will focus on how the approaches and their static analyses can be realized in C$\boxed{\text{A}}$RET. We will not go into the details about how the specific approaches build up their recommender models, because this second step is unrelated to C$\boxed{\text{A}}$RET and therefore not relevant at this point.

File-based Recommender Early work by Mccarey et al. [137] build a recommendation system that can predict API elements that should be added to a given class. They apply collaborative filtering and content-based filtering to predict this information. Their data model treats classes as users and API elements that are used within a specific class as the predicted items. This information is straight-forward to extract from an SST, which represents a single class. All referenced type elements (i.e., methods, fields, or properties) can be extracted in one traversal of the syntax tree. The elements can easily reduced to API references, because our naming schema can distinguish elements that are defined in an assembly from those that are defined in a local project. An evaluation will follow the original approach presented in the paper, but it can be simplified through using our dataset of released source-code. The dataset will be partitioned into a training set and a validation set, e.g., in a cross-fold evaluation. The evaluation will learn a recommender from the training data and will measure how well it can predict the API elements that are used in the validation set.

Structure-based Code Search Holmes et al. [88] have build the code search engine STRATHCONA that uses structural information of the current coding context like extended class, implemented interface, or enclosing method to find relevant examples in a repository. When a query is triggered, this information will be extracted for the current class under edit and will be send to a server. The server matches the structure to a large set of structures extracted a repository to find relevant examples that are structurally similar. The source code of the matching location is then being proposed in the IDE. SST are applicable for this approach: both the required structural information and the source code contained in the method bodies can be extracted from our dataset of released source code. The original evaluation has used a manual approach to decide how relevant the proposed snippets are. Our in-IDE infrastructure makes it easy to access the same information also in the IDE and to re-build STRATHCONA. We expect that we can even improve over the quality of the original approach through the *reference generalization* (see page 4.1.2) that is enabled through the additional information about the type hierarchy captured in *type shapes*. While a re-build allows a replication of the original study, our platform provides the unique opportunity to evaluate the approach in an automated fashion. The fine-grained history of source code that is captured in enriched event streams could be used to evaluate whether the recommendation system is able in the beginning of a coding session to find source-code examples that are similar to the code state at the end of the coding session.

Syntax-Based Recommendation Heinemann et al. [83] provide an approach that uses syntactical tokens preceding a method call as the context to query a method call recommender. The approach is based on a very light-weight static analysis: the data extraction can be implemented by collecting the required information in a simple AST traversal without performing any advanced data-flow or control-flow analysis.

It is required to extract method invocations and their preceding syntax tokens, which can be implemented in C\boxed{A}RET using a simple visitor. While traversing the syntax tree, the visitor has to tokenize each visited element as described in the paper. For example, keywords are wrapped in brackets (e.g., `<new>`) or method names are split and stemmed (e.g., `getOpenedFile` → {get, open, file}). Every time the visitor visits a method invocation an occurrence is stored for later experiments. The extracted data contains both the fully qualified method name that should be predicted from the recommender, together with the *n* preceding tokens. Having a large number of extracted occurrences, it is possible to both learn a model and use the occurrences for the validation, e.g., through cross validation. In the evaluation, the preceding tokens are used to query the recommender and the prediction quality can be measured by comparing the proposals to the observed method name.

Text-based Recommendation Even though our main focus is source-based recommender systems that make use of the structural information in the code, C$\boxed{A}$RET can still be used for purely text-based recommender systems. For example, we can implement Hindle et al.'s text-based recommendation system [87], that only relies on the tokenization of the source code, as follows. Since the underlying technique only needs the source code tokens, these can be created by implementing a visitor that collects all tokens in the IR. Note, however, that since our IR is a simplified abstraction, some details such as comments or visibility modifiers are lost. This does not pose a problem for the usability of C$\boxed{A}$RET since we intentionally designed it for techniques that make use of code structures and context information.

Sequence-based Recommendation Raychev et al. [195] propose SLANG that can synthesize method completions at every location in a program. They solve this problem by modeling a program as a set of histories that represent unique sequences of method traversals. Missing method calls (or the trigger of a code completion for that matter) create holes in these sequences, which will be filled by the synthesis.

With C$\boxed{A}$RET, it is possible to reproduce the static analysis of this work that creates the preprocessed data that is used to train a language model. Using our dataset of released source-code as input improves replicability compared to the original work, in which the source code used for the experiments was not defined. The intra-procedural static analysis that extracts the sequences of method calls required for learning the language model can be extracted by traversing the syntax trees of the method bodies stored in SSTs. All control-structures are preserved in SSTs so the extraction logic corresponds to the description in the original work and is implemented through tracking forks and joins of the control flow. The original work applies a Steensgaard-style points-to analysis to further differentiate between different object instances, a reusable component that is also available in C$\boxed{A}$RET's toolbox. We expect that applying a *reference generalization* (see page 4.1.2) that is enabled through the additional information stored in *type shapes* would further improve the approach: reduced data sparseness results in smaller language models and likely in increased prediction quality.

The approach is evaluated on artificially created holes in a complete program, which is also possible by cross-folding our dataset of released source code. However, the completion events in our captured interaction also contain source code with holes, which could be used instead. We are convinced that these provide a more realistic ground truth than artificially created holes.

All previous approaches are already fully supported by C$\boxed{\text{A}}$RET and their reimplementation is straightforward. We found other approaches, which rely on additional components for analysis or transformation that could be provided by C$\boxed{\text{A}}$RET. This shows the synergetic effects that are enabled by a common platform. Adding further reusable components makes the platform more attractive and the more researchers actually use it, the more synergies exist through sharing reusable components.

Sequence-Based Code Search Zhong et al. [262] propose MAPO a framework for mining API usage patterns that can then be used in a code search engine. The patterns are identified using frequent subsequence mining on sequences of API methods that are extracted in a static analysis that heavily relies on control flow. While control structures are not stored in the model, they are considered when the sequences of constructor calls, static and non- static calls, as well as casts are being extracted. The analysis iterates over all public method declarations found in a repository and extracts sequences of reusable API references. Method invocations that point to declarations in a local project are not being captured though and the control flow is tracked into these methods instead, which results in an inter-class analysis for all local classes. Finally, the authors maintain a mapping between learned patterns and real examples that were observed in source code that can be inspected by developers, in case a specific pattern seems interesting to them. SSTs preserve all details about the source code, so it is possible to follow the same approach.

Our meta model for object-oriented source code contains all relevant information to rebuild this approach and the approach can be re-implemented with our platform. It is required to implement a visitor that traverses all method declarations of an SST and that collects all sequences of method invocations in the method bodies. The control structures are preserved in SSTs, so it is straightforward to handle control points at which the current sequence has to be split. For example, the control flow either enters the `then` or the `else` block of an `if`, never both. Our naming scheme allows distinguishing method references to reusable API elements from local references. C$\boxed{\text{A}}$RET's toolbox provides an inlining transformation that significantly simplifies the static analysis. The required inter-procedural tracking is made superfluous by inlining the called methods. However, C$\boxed{\text{A}}$RET currently only supports the inlining of methods within a class, but extending the existing component to a real inter-class scenario is a matter of engineering effort, rather than any conceptual limitation. As long as the analyzed SST is extracted from a repository, it is guaranteed that all declarations to these local references can be resolved. This cannot be guaranteed for SSTs that were captured in-IDE, but an approach could fall back in such cases to the intra-class inlining that already exists. Extending the inlining component to the inter-class level is excellent example for another reusable component in C$\boxed{\text{A}}$RET that would provide value for future use cases of the platform.

The authors use example code found in the official documentation of a framework in the evaluation of the paper. They type-in the programming context and the first method call of the example and measure how well the tool is able to find snippets similar to the full example. The evaluation could be replicated in the same way, however, using C$\boxed{\text{A}}$RET, it might also be possible to infer automatic evaluation scenarios from the captured source-code history, as already elaborated for Strathcona.

Change-Based Recommendation Nguyen et al. [167] propose ApiRec, a change-based recommender that exploits the repetitiveness of changes to source code. They extract change information from two GitHub corpi that feature a long history of changes and that have been used in research before. For each change they extract the set of atomic changes that forms the full change transaction. In addition, they capture the preceding tokens of an edit location as the code context. Both information are used to learn association rules that describe likely co-changes.

The C$\boxed{A}$RET platform provides appropriate means to re-implement this approach using the fine-grained change history captured in enriched event streams. The authors call a commit history fine-grained, but Negara et al. [166] have shown that a it is not representative for actual development. The history captured in our source-code snapshots is much more fine-grained. In fact, it is so fine-grained that each snapshot corresponds to their notion of an atomic change. To aggregate these changes into their transaction concept, it is required to merge multiple changes. This can be achieved through the additional information that is contained in enriched event streams and multiple merge strategies are possible. For example, using fixed time frames, timeouts, or maybe even through identifying abstract tasks from the recorded developer activities. We expect that all approaches would represent the change process more precisely than commits. Once the mergeable snapshots are identified, researchers can follow the proposed approach and apply the GumTree algorithm to extract changes between the start and the end state of a merged change. However, the individually recorded snapshots already conform to their notion of atomic changes and they are easier to match than two snapshots that contain a mix of multiple individual changes. Both tasks, the identification of related changes and the extraction of change information, could be implemented as reusable components in C$\boxed{A}$RET.

The authors follow three strategies to evaluate their recommender. They measure the overall precision of the approach across their dataset. In addition, they analyze the precision within a project history and within a user history. C$\boxed{A}$RET provides the data for all three strategies: the interaction data is organized by user, which makes the third strategy straight forward and merging all available interaction data into the same evaluation corresponds to their first strategy. Our naming scheme makes it possible to identify the enclosing project for each snapshot so also the second strategy can easily be supported by our approach. Moreover, our snapshots contain the edit location of the developer, which makes it easy to extract the preceding source code tokens, something that can only be heuristically solved using a commit history.

Documentation Creation Moreno et al. [154] propose the tool Muse that can find example usages for any given API method. Users interested in getting support for APIs of a specific library must provide the source code of this library, if available its JavaDoc, and clients of the library that use the APIs. If a developer then requests examples for a method m, occurrences of this method are looked-up in all clients. An intra-procedural backwards slice is extracted for each usage to create minimal snippets that illustrate how the specified method is used. The authors apply clone detection to group similar examples and identify the most representative one with a ranking mechanism. If available, JavaDoc is added to extend the example with a description.

To make the approach work using C$\boxed{A}$RET, it is necessary to implement the backward slicing, which relies on a program-dependence-graph. All code elements are contained in SSTs required to build such a task, namely method calls, order, variable names, assignments, and control structures. Given that the scope of the slice is also limited to the current method, this task is a straightforward engineering task. In the next step, clone detection is applied to identify duplicates. The authors use Simian[46] to realize this. We could reuse the pretty-printing components of C$\boxed{A}$RET to create valid Java or C# code and apply the same tool for the clone detection.

Our source-code meta model does not contain comments, so we cannot enrich the resulting snippets with JavaDoc. However, rebuilding the approach using C$\boxed{A}$RET has several benefits. First, our naming scheme enables to unambiguous references to specific API methods. We can even distinguish different versions of the same API, which is relevant if an API release introduces breaking changes. Second, our dataset of released source-code provides an excellent data source for the approach, as it contains examples for many different APIs that are easy to find using their names. Third, the original implementation requires the source code of the target API to set up the approach. This is not necessary in our case, because our naming schema already provides all required information and we do not need to compile the sources.

In addition to the RSSE that we have sketched in this section, we have considered many other source-based recommender systems during the design of SSTs such as other method call recommenders (i.e., [2, 22, 83, 137, 195, 261]), snippet recommender (i.e., [87, 127, 169, 171, 240]), and tools for code search (i.e., [90, 92, 154, 262]), documentation (i.e., [135, 143, 144, 263]), and anomaly detection (i.e., [117, 152, 184]). These were subject of our requirements analysis, so we are confident that a re-implementation is possible. Our survey has covered a wide range of applications, so we expect that C$\boxed{A}$RET is also applicable in future approaches. However, we might have misunderstood certain requirements and cannot predict the future. It remains the subject of future work to actually implement these further approaches to really prove C$\boxed{A}$RET's applicability.

Chapter Summary

This chapter, we have build and sketched recommendation systems as another use case to show the usefulness of SSTs and C$\boxed{A}$RET. In its two parts, we have first presented a line of research in which we introduced a sophisticated static analysis to extract object usages, from which we then derived three RSSE approaches. This exercise has proven that it is possible to create sophisticated analyses and that using C$\boxed{A}$RET helps. Especially the extensible pipeline has proven helpful in the process. Creating such a reusable pipeline to facilitate the comparison of different approaches was an early motivation of this thesis. The second part contains sketches that explain how other existing RSSE approaches could be realized using our infrastructure. While the required effort to implement the described approaches prevent us from creating working systems, our sketches prove that the platform is not tailored to the needs of object-usage-based approaches and that it can be applied in a wider context.

11 Analysis and Improvement of Current Evaluation Techniques

Research in the area of Recommendation Systems for Software Engineering (RSSE) regularly produces exciting ideas on how to automatically support developers with their daily coding and maintenance tasks. Examples include recommending the next syntactic token [168], links to related code snippets [154], and which windows to close [149]. Extensive evaluations provide (quantitative) evidence that such tools are accurate and valuable. However, conducting such evaluations is often challenging.

We have been interested in understanding the evaluation process of RSSE to identify the current challenges and to further improve the state of the art. In our work, we have approached the topic from three directions, that we will describe in the remainder of this chapter.

Holistic Picture We will analyze current evaluation practices. In addition to the typical evaluation that is focused on prediction quality, we will introduce two additional dimensions of evaluation, model size and inference speed, and analyze the tradeoffs between all three dimensions in a more holistic evaluation approach (Section 11.1).

Realism A realistic ground truth is usually not readily available and evaluations typically rely on creating artificial queries from stable code found in version control systems. We will evaluate the level of realism that can be achieved by such an artificial strategy when compared to a realistic evaluation (Section 11.2).

Common Benchmark So far, designing an evaluation for method call recommender is a recurring task for researchers. We will design a reusable benchmark that enables comparative evaluations of such systems and that facilitates reusability, replicability and comparability of experimental results (Section 11.3).

In the following, we will have a closer look at the evaluation of RSSE with a focus on method call recommenders. We will go through the three dimensions that we have identified before and will present a detailed discussion of our experiments and of our contributions to evaluations in this area.

11.1 Extending the Scope of Evaluations

The evaluation of previous recommender systems is usually focused on comparing prediction quality of different algorithms, performance or mode sizes are only mentioned as a side note (e.g., in [22, 117, 261]. However, RSSE are designed for the use by humans on typical developer machines with limited resources and (empirical) evaluations of recommender systems should take this into account. In other words, event though two approaches can predict method calls equally well, they might impose very different requirements on their environment. We believe that these additional requirements should be discussed in a fair evaluation. Another property that has rarely been evaluated for RSSE so far is the scalability of these properties with respect to the amount of training data that is used.

We will conduct an experiment to understanding the effect of these different properties. To this end, we will conduct an evaluation that, in addition to prediction quality, takes two additional properties into account: (1) time needed to compute proposals and (2) size of the models that are used to make the proposals. We will analyze the tradeoff between these properties and also analyze scalability of the RSSE.

As a use case for our experiment, we will compare the results of different method call recommenders. For simplicity, we will reuse the two object-usage-based recommender engines that have been discussed in the last section, BMN and PBN. Both approaches can be configured to use a different set of features found in the input data, in addition, the training phase of PBN can be controlled by several parameters, which enables the creation of several different configurations that can be used interchangeably. We will instantiate several recommender configurations and will evaluate their differences.

11.1.1 Methodology

Reusing BMN and PBN has several advantages: both recommenders use the same input data that is generated by the same static analysis, have the same interface to the outside, and create the same kind of proposals.

We will learn a recommendation system for the SWT framework[40] in our experiment, the open-source UI toolkit used in the Eclipse IDE. We will apply our static analysis that extracts object usages to the update site of the Eclipse "Kepler" release,[41] the main source of plug-ins for all Eclipse developers. We learn from and test against SWT example code take from this snapshot. We think it is safe to assume that the API usage in such a release has reached a stable state: the contained source code is no longer under development, changes are only introduced to fix bugs or to implement new features. On September 30, 2013, we took a snapshot of the update site and extracted about 190,000 object usages for different SWT widget types from the 3,186 contained plug-ins. `org.eclipse.swt.widgets.Button` is the type for which we extracted the highest number of object usages (47,000).

Evaluating the Prediction Quality

The recommendation systems we create for the experiment use separate models for each API type and can predict missing method calls in a given object usage. We evaluate each type separately with a 10-fold cross-validation over the object usages that have been extracted for the type as shown in Figure 11.1. The data is partitioned into ten splits that are then combined to create ten folds, whereby each fold consists of one split used as *validation set* and the union of the remaining nine splits as *training set*. For each fold, models are learned from the training set and the validation set is used to query these models. Accordingly, it is guaranteed that no object usage is used for training and validation at the same time.

Our experiments have shown that intra-project comparisons introduce a positive bias to prediction quality. We want to avoid this kind of bias to better reflect development reality, i.e. the intelligent code completion engine is used in a code base

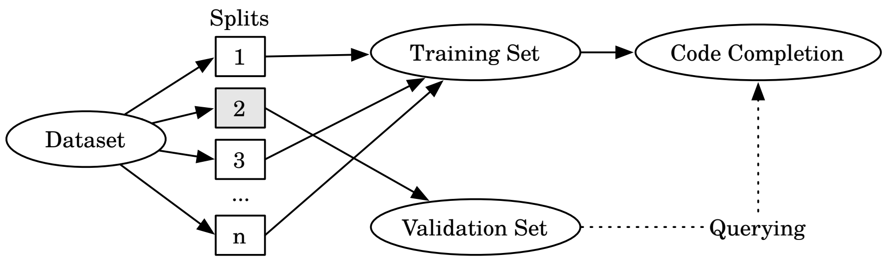


Figure 11.1: One Iteration of an n-Fold Cross Validation

that was not used to learn the models. Therefore, we ensure that all object usages generated from the same project are assigned to the same split. This means that the set of projects used to create queries is disjoint from the set of projects used to built the model. As a result, we can only include types in the evaluation that are used in at least 10 different projects and the sizes of the splits differ slightly, especially if the total number of object usages is small for a specific type.

Each usage from the *validation set* is used to query the model and to measure the results. This approach is illustrated in Figure 11.2. A *query* is created by removing information from a complete *usage*. The resulting incomplete usage is used as a *query*, and the removed information constitutes the *expectation*. When a completion is requested by passing a query, the recommender engine returns a list of *proposals* in descending order of the proposals confidence value. We follow previous work and filter all proposals with a confidence value lower than 30% [22]. The *proposals* are compared to the *expectation* and the F_1 measure is calculated to quantify the quality of a given proposal. F_1 is the harmonic mean of the two measures *precision* and *recall*:

$$precision = \frac{\#hits}{\#proposed} \qquad recall = \frac{\#hits}{\#expected} \qquad F_1 = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

Call sites are stored in an *unordered set*, no information is available about the original call sequence. We use different removal strategies for both receiver and parameter call sites to create queries. Other context information (e.g., the enclosing method) is not removed, because we think that they always exist in the context of a framework.

We make use of two different strategies to remove call sites from the set. The *No Calls* strategy removes all call sites and therefore creates exactly one query for each usage. This mimics a developer that starts to work in a new method and triggers code completion to get a first impression of what to do. The resulting queries are denoted as 0|M queries, where M is the number of receiver calls contained in the original object usage (i.e., a 0|3 query contains no calls from the three calls that are contained in the original object usage). The *Partial Calls* strategy removes *about half* of the call sites (i.e., 0|1, 1|2, 1|3, 2|4, ...). The resulting queries are denoted as N|M queries, where N is the number of calls contained in the query, and M the number of calls in the original object usage. Both N and M refer to a number of receiver call sites. This strategy simulates developers, who started to do some work in a specific context, but came to a point where they did not know how to continue and trigger code completion for help. There are $\binom{M}{N}$ possibilities to remove calls from the set. Because this number gets impractically large with a growing number of calls in the set, only three N|M queries are randomly selected. We calculate the average result of these queries to

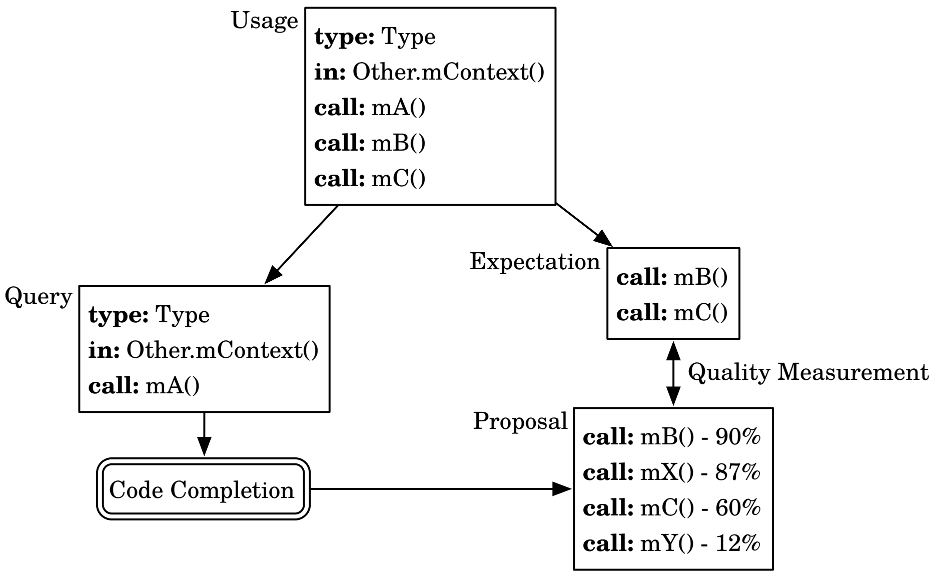


Figure 11.2: Conceptual Illustration of the Evaluation of a Single Proposal

merge the separate results into a single result value and store it as the quality for the originating object usage. Parameter call sites are removed separately with the same strategy. However, we did not include them in the notation, because they only represent context information and are not proposed by the completion system. If not stated otherwise, the *Partial Calls* strategy is used in all experiments of this chapter.

Evaluating the Model Size

It is crucial to take the model size into account, when evaluating RSSE and it is preferable to have smaller models: Models are loaded into the IDE of the developer and their size effectively affects the number of models that can be loaded simultaneously and the time for deserialization from the hard drive is lower. Additionally, models have to be distributed, which is easier for small models.

The model size can be determined empirically by measuring the memory representation in the *Java Runtime Environment* (JRE). However, the result of this approach depends on implementation details of the model, the used JRE, and characteristics of the garbage collection, which might induce considerable noise. We decided to use a more simple approach of calculating the theoretical model size of the raw information that is contained. For BMN, the size is calculated by multiplying the number of rows in the table by the size of each row. For PBN, the total number of stored float values in the Bayesian network is calculated. We calculate the size in Byte for both approaches to create comparable values.

Evaluating the Inference Speed

All computations that depend on context information can be made efficient by precomputing as much as possible and storing this information in the model. The precomputation does not need to be fast, because it can happen offline on a powerful

1. Analysis of the working context in which the completion was triggered
2. Building a query
3. Loading of the appropriate model, used to answer the query
4. Inference of relevant method calls
5. Rendering of the code completion popup

Figure 11.3: Computations Steps to Get from Trigger to Code Completion

server. However, using intelligent code completion makes it necessary to compute several complex steps in the IDE, whenever it is activated, as shown in Figure 11.3.

The perceived duration of the completion task for the user is the sum of all of these steps, but not all of them are evaluated in this chapter: the static analysis of the working-context is the same for BMN and PBN. Therefore, step 1 is not considered. All models are pre-computed and stored in a serialized form. To be usable, they need to be de-serialized again and loaded into memory. Nevertheless, by using caches this loading time can be avoided in most cases. Additionally, I/O increases noise in performance measurements and the loading time itself mostly depends on the framework used for serialization. Therefore, we ignore step 3 as well. The two steps 2 & 5 are trivial compared to other phases and can safely be ignored. Step 4 is of relevance for this chapter. First, it depends on the size of represented data in a model and is critical for scalability. Second, the inference process differs between the models.

The precision of timing information that can be reliably read from the system timer is milliseconds.[27] To increase the precision of the results, we (1) measured the total computation time for all proposals and divided this time by the number of queries and (2) ensured that at least 3,000 proposals are requested per type. The second point mainly addresses the evaluation for types with only a few object usages, because here the typical answer times are smaller than a millisecond. We repeated all experiments that include timing three times and calculated the average to overcome slight deviations caused by potential interfering processes.

Our experiments showed that the just-in-time compilation of modern runtime environments has a significant impact on the performance of the inference. The steady state is reached after thousands of queries for each model. Although this might be unrealistic for a practical usage of the completion system, we decided to repeat all experiments that evaluated inference speed multiple times until the resulting values were stable to create comparable and repeatable results.

It is of paramount importance that predictions are computed quickly, in order not to disturb the workflow of the developer. Based on prior research that analyzed the impact of user interface delays on users [172], we derive two timing constraints that we will use to asses the inference speed of RSSE approaches: (1) a code completion should provide information in less than 100 milliseconds to avoid any noticeable delay in the workflow of the developer, (2) it must answer in less than a second because the current thought of the developer is interrupted, otherwise. These constraints disqualify some recommendation systems for a practical usage, even though they might achieve a higher prediction quality than other approaches.

Experimental Setup

To speed up the experiments, we implemented a map-reduce like computation in the evaluation framework and distributed the computation to multiple worker threads running on different machines. All machines were running JAVA 8.[1] Experiments that include measurement of timings were run locally on one machine in a single-threaded application.[2] We designed the evaluation such that the results are not influenced by disc I/O (e.g., loading all necessary data into memory before starting the evaluation), hence further information about the storage system is omitted.

11.1.2 Analyzing the Impact of Additional Context Information

Section 10.1.1 presented the context information that we collect per object usage. Three kinds of context information - the type of the receiver, the receiver call sites, and the enclosing method definition - were previously used for code completion recommendations. The other three kinds - parameter call sites ($\Box_{+P}$), class context ($\Box_{+C}$), and definitions ($\Box_{+D}$) - are introduced in this work.

In this section, we investigate whether the new context information can be used to improve the quality of the intelligent code completion. The baseline for this experiment are models that do not use the new context information. We compare this to the models that are created with the new context information and exhaustively evaluate all combinations of enabled information classes. First, we activate all information classes separately, then we activate pairs ($\Box_{+DP}$, $\Box_{+CP}$, $\Box_{+CD}$), and in the end, we activate all together ($\Box_{+ALL}$). We are especially interested in first insights about the trade-off between increased model size and prediction quality gain.

We use all available object usages in this experiment that were extracted for types that belong to the `org.eclipse.swt.widgets` package. We average the prediction quality over all queries and average the model size over all models generated for different types. However, it is necessary to be cautious with the interpretation of the model size in this plot, for two reasons: First, a simple average puts an emphasis on types with only few object usages: the available input is not equally distributed among the types, the majority of object usages is extracted for a minority of types. Therefore, the resulting model sizes shown in the plot are biased towards small input sizes. Second, it is not possible to directly compare model sizes for BMN and PBN in this plot. As we will see in a later subsection, their implementation make them scale differently for types with only a few object usages. All the same, we decided to include the model size. Even though it is necessary to read it with caution, it is a first indicator of the impact of a specific information class on the resulting model size. The exact impact of the amount of available input on model sizes is analyzed in a later subsection.

Figure 11.4 shows the experiment results. The horizontal axis of the plot shows different configurations of context; the legend under the plot shows which context information is used in each configuration. The vertical axis is organized in two dimensions: the prediction quality (on the left) is depicted with black bars and the average model size (on the right right) is depicted with smaller white bars. The plot contains results for different BMN and (unclustered) PBN_0 models.

When comparing the configurations with activations of each context information kind in isolation, $\Box_{+D}$ is the context information with the biggest impact (third bars in

[1] Oracle Java(TM) SE Runtime Environment (build 1.8.0_25-b17) with heap settings -Xmx3g
[2] Intel Core i7 with 2.8Ghz and 16GB of RAM with a clock speed of 1,600Mhz

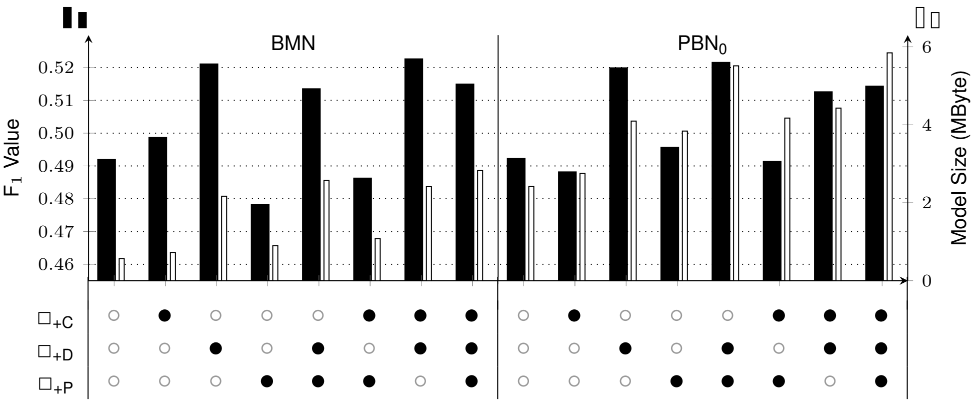


Figure 11.4: Comparison of Context Information (All SWT Widgets)

the plots of both BMN and PBN). Compared to baseline, this configuration increases the F$_1$ value by 0.03, which represents an improvement of 6% for both BMN$_{+D}$ and PBN$_{+D}$. However, this comes at the cost of a model size that is three times bigger for BMN$_{+D}$ and 1.7 times bigger for PBN$_{+D}$. The results are indecisive for both □$_{+C}$ and □$_{+P}$: In case of BMN, the prediction quality is slightly increased for BMN$_{+C}$ and decreased for BMN$_{+P}$, but it is the other way around for PBN$_0$. We consider both deltas irrelevant for the results as they are small (i.e., 0.01). While □$_{+C}$ has only a minor effect on the model size, □$_{+P}$ significantly increases it.

□$_{+D}$ is the dominating context information; the others seem to be negligible.

The results are similar when multiple context information kinds are activated together. We can ignore □$_{+CP}$, because the prediction quality gain is low. In case of BMN$_{+CP}$, it is even lower than the baseline. Both □$_{+CD}$ and □$_{+DP}$ show a comparable prediction quality to □$_{+D}$. However, the introduction of both information classes increases the model size. We conclude that adding them is unnecessary, since they do not increase the prediction quality. For both BMN and PBN the □$_{+ALL}$ configuration leads to worse results. Presumably, the reason for this is that queries become too specific, i.e., different object usages become more similar because they share irrelevant features. However, we did not further analyze this result.

Given the results, we focus on the evaluation of □$_{+D}$ approaches in the remaining experiments. Next, we investigate ways of reducing the PBN model size via clustering and analyze the effect on the prediction quality. The model size cannot be reduced for BMN: BMN$_{+D}$ and PBN$_0$ will be our reference points regarding prediction quality.

11.1.3 Comparing Clustering Configurations

In Subsection 10.1.3 we introduced a clustering technique to reduce the size of learned models. Now, we compare BMN to several instances of the PBN approach that all have different thresholds for the clustering. The distance threshold is used to control how much information is dropped during the clustering. Recall that the threshold is encoded in the name, e.g., a PBN$_{15}$ configuration clusters all data points with a mutual distance of less than 0.15. PBN$_0$ is the unclustered instance. We conduct this experiment to identify distance thresholds that provide useful trade-offs between

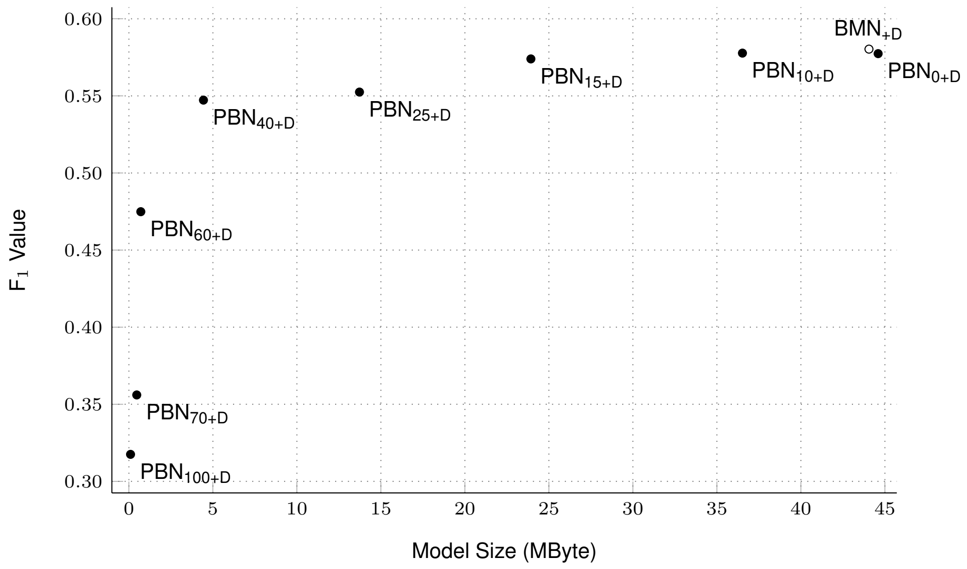


Figure 11.5: Quality and Size of Different Recommenders (About 47,000 Usages for Button)

prediction quality and model size. Model sizes also depend on the amount of training data that is available for a given type. For the remainder of this Subsection, we will focus on `org.eclipse.swt.widgets.Button`, as we were able to extract the highest number of object usages, 47,000, for it.

The results of this experiment are illustrated in the scatter plot in Figure 11.5. The plot contains data points for BMN_{+D} and for several PBN_{+D} variants with different distance thresholds. All points are positioned according to their respective model size in Megabytes and their prediction quality, which is denoted by the F_1 measure. The plot shows that the model size and prediction quality is very close for BMN_{+D} and for the unclustered PBN_{0+D}. The model size can be reduced through clustering by increasing the distance threshold of the PBN approach. The conservative distance threshold used in the PBN_{15+D} configuration significantly reduces the model size with virtually no effect on the prediction quality. If the threshold is further increased, a small decrease in quality can be measured while the model size constantly decreases. The imaginary curve that connects all PBN instances in the plot has an inflection point at PBN_{40+D}. This inflection point seems to be a moderate clustering that represents a good tradeoff between prediction quality and model size.

PBN_{40+D} saves 90% of the model size with an F_1 decrease of only 0.03.

If the threshold is even further increased, the clustering generates fewer and fewer patterns, because all object usages are aggressively merged. This leads to a very fast decrease in prediction quality. The prediction quality of PBN_{60+D} decreases by 0.08 when compared to PBN_{40+D}. However, the model size is comparable to the minimal model size of the PBN_{100+D} approach. The prediction quality drops significantly if the threshold is increased beyond 0.60. These cases can be ignored as the model size is already negligible for PBN_{60+D}.

In summary, the distance threshold can be used to control the tradeoff between model size and prediction quality depending on the use case. The maximum prediction

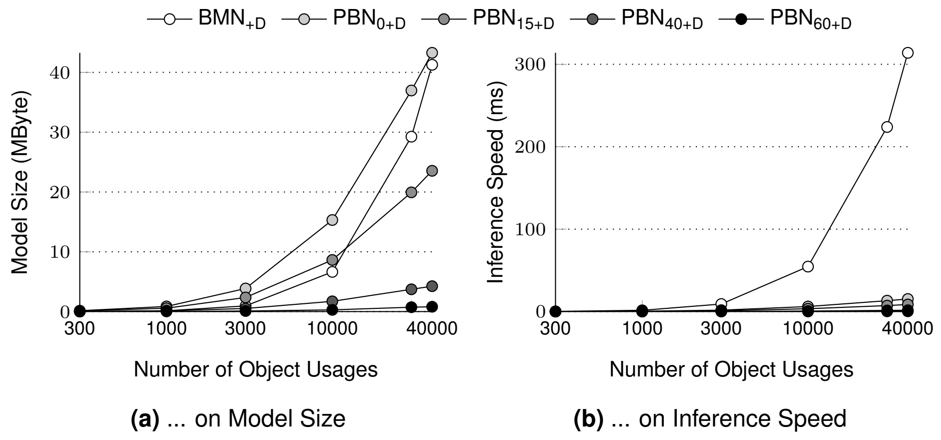


Figure 11.6: Effects of Increasing Number of Object Usages (Button)

quality is provided by BMN_{+D} or from the unclustered PBN_{0+D} instance (there is no measurable difference between them); however, both come with large model sizes. If a small model size is important, then the aggressive clustering of PBN_{60+D} still provides a reasonable prediction quality. For use cases, where both model size and prediction quality are important, PBN_{15+D} or PBN_{40+D} seem to be provide good trade-offs. We use these three thresholds in the remaining experiments to create clustered instances, in addition two the two unclustered variants.

The PBN distance threshold can be used to control the tradeoff between model size and prediction quality.

11.1.4 Scaling the Input to Large Sizes

We now look at the effect of increasing the number of object usages on the prediction quality, model size, and inference speed. We wanted to investigate two different questions: (1) How do different approaches scale from smaller to bigger input sizes? (2) Is it possible to further increase prediction quality by using bigger datasets. We wanted to extrapolate the results to predict saturation effects, i.e., when providing more usages will not increase prediction quality.

The experiment is limited to `org.eclipse.swt.widgets.Button`, because it is the only type for which we have more than 40,000 object usages available. A random subset of all available object usages was used to conduct a cross-fold validation. We started with a minimal set of object usages and exponentially increased the input size in all experiments of this subsection. To get stable results, we ran three iterations for each input size and stored their average result. The previously chosen instances of PBN and BMN are used in this experiment, all include $\square_{+D}$ context information.

Model Size The impact that scaling the input size has on the model size is shown in Figure 11.6a. The input size is shown on the logarithmic horizontal axis and the resulting model size in Megabytes is shown on the vertical axis. The plot shows that the model size for PBN_{0+D} is generally bigger than BMN_{+D}. However, the model size grows faster with an increasing input size for BMN_{+D}. At an input size of 40,000, both have a comparable model size. We could not evaluate larger input sizes, but

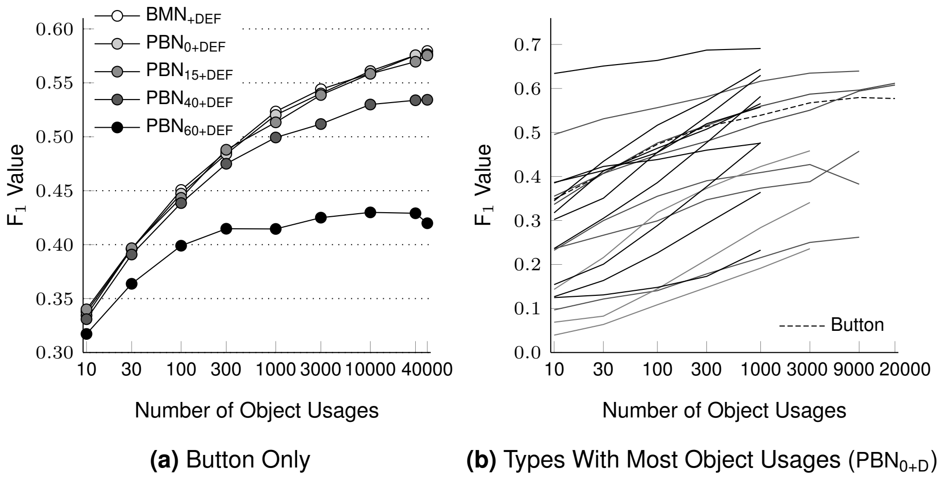


Figure 11.7: Prediction Quality for Increasing Amounts of Object Usages per Type

the plot shows that the model size of BMN_{+D} grows faster. A non-logarithmic plot of the same values, which we omit for space reasons, shows a linear increase for BMN_{+D} and a logarithmic increase for all PBN instances. If we extrapolate the results to more object usages, we expect that BMN_{+D} has bigger models than PBN_{0+D}. This is even more obvious for clustered instances: the break-even for PBN_{15+D} is reached at ~ 10,000 usages, and at even less than 3,000 usages for PBN_{40+D} and PBN_{60+D}.

The model size of PBN scales better than BMN with the input size.

Inference Speed The impact of the scaled input size on the inference speed is shown in Figure 11.6b. The input size is shown on the logarithmic horizontal axis and the resulting inference speed in milliseconds is shown on the vertical axis. The plot shows that inference speed is irrelevant for input sizes less than 10,000. However, starting from 3,000, the slope of the BMN_{+D} plot is much higher. A non-logarithmic plot, which we omit for space reasons, shows a linear increase for BMN and a logarithmic increase for PBN.

Inference speed is significantly higher with PBN and scales better than with BMN.

Section 11.1.1 introduced time limits for the proposal for inference. BMN_{+D} exceeds the imperceivable delay of 100 milliseconds at about 15,000 usages. If the input size is larger than that, a delay is perceivable in the code completion. By extrapolating the results beyond 40,000 object usages, it becomes obvious that a further increased input size soon exceeds the limit of a second. This would interrupt the developer's thought process and would present a disturbance in the work flow.

The inference computation of PBN is significantly faster than this limit. Even the unclustered PBN_{0+D} takes only 15ms to compute the proposals with 40,000 object usages. The inference speed is even higher for the clustered PBN instances. By extrapolating the results, it is obvious that the input size can be significantly increased before any time limit is reached.

Input size can be significantly increased before inference speed is an issue for PBN.

Prediction Quality The last property to analyze was the impact of the scaled input size on the prediction quality. The results are shown in Figure 11.7. In both plots, the input size is shown on the logarithmic horizontal axis, the prediction quality denoted by the F_1 measure is shown in the vertical axis.

Figure 11.7a was created with input of `Button` only to compare the results of different approaches. Unsurprisingly, the plot shows that increasing the input size has a positive impact on prediction quality. $\text{BMN}_{\text{+D}}$ and $\text{PBN}_{\text{0+D}}$ show equal prediction quality and no difference in scaling behavior: the plot is flattening for larger input sizes, even though it has a logarithmic scale. The interpretation of this is two-fold: First, it is possible to see saturation effects starting at about $1,000$ object usages, i.e., every tripling of the input size leads to a smaller increase in prediction quality. This is a promising result, because for most types there is not so much input available. Second, even though the gain in prediction quality constantly decreases, it is still possible to further increase it by using more input. By extrapolating the results for larger input sizes, it seems that the boundary for `Button` is an F_1 value of $0.6 - 0.65$.

First saturation effects can be observed with an input of $1,000$ object usages, use more input to maximize prediction quality.

The plot also contains the results of the three clustered `PBN` instances. Even though we have seen in the previous experiments that $\text{PBN}_{\text{15+D}}$ scales significantly better with input size than the unclustered $\text{PBN}_{\text{0+D}}$, the prediction quality is exactly the same. If the clustering is more relaxed, a negative impact on the prediction quality can be seen. For $\text{PBN}_{\text{40+D}}$, there seems to be a gap in prediction quality if more than 300 usages are used. For $\text{PBN}_{\text{60+D}}$, the prediction quality seems to saturate between $300 - 1,000$ usages. Both results suggest that there might be potential to improve the clustering approach, which we want to address in future work.

Using $\text{PBN}_{\text{15+D}}$ for large input sizes preserves a reasonable prediction speed and model size, without negative effect on prediction quality.

We further analyzed whether these results for `Button` also hold for other types. Therefore, we conducted the same experiment for all types of SWT for which we could extract at least $1,000$ object usages. We used $\text{PBN}_{\text{0+D}}$ for the comparison. The results are shown in Figure 11.7b. The result for `Button`, which has already been shown in previous Figures, is shown as the dashed line. Lines that are not continued to the end of the plot belong to types for which we do not have enough object usages. The plot is not meant to present quantitative results of how well models for these types work, but to qualitatively illustrate general trends of saturation effects. The plot shows that recommender systems do not work equally well for all types: Some already start on a high F_1 level, others barely reach an F_1 level of 0.2 with models learned from $3,000$ usages. However, all plots roughly point into the same direction, which means that their scaling with the input size is comparable and that the previous findings are also valid for all of them.

Lessons Learned The experiments have shown that it is possible to increase the prediction quality even further by using more than $40,000$ usages as input. However, the number of input values that lead to further improvement grows exponentially. Such large input data sets make the evaluation very time consuming, though, because with a n-fold cross validation every usage is used as a query once. In total, we had an input data set of over $190,000$ usages. Even for fast code completion configurations,

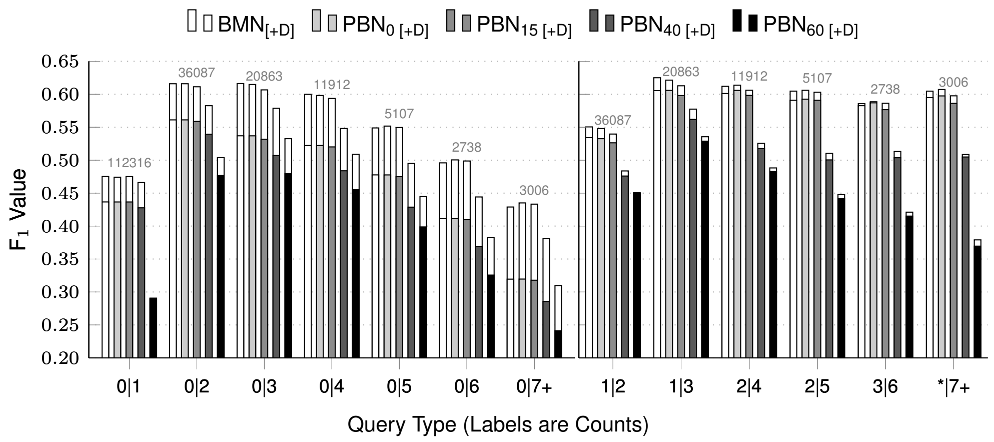


Figure 11.8: F$_1$ Values for 0|M and N|M Queries (All SWT Widgets)

which infer the proposals in less than 100ms, computing all queries takes about 5.5 hours. All experiments exercised more than one configuration, including very slow ones that need more than a second for inferring proposals, therefore, a complete evaluation run took more than 7 days for some experimental configurations.

We have shown that for our data set the prediction quality is already very close to the expected maximum with an input size of $10,000$ to $30,000$ usages. Even though we used all available input in our experiments, we postulate that using only a subset is sufficient for reasonable evaluation results, as long as this subset is large enough. For future evaluations, a faster and nevertheless valid approach may be to run experiments multiple times with randomly selected subsets and to average the results. This would speed-up the experiments significantly without invalidating the results.

11.1.5 Revisiting Prediction Quality

All experiments in prior sections were focussed on very general questions regarding configuration and scaling. We are also interested in a better understanding of the prediction quality of the recommender. This last series of experiments analyzes the impact of the new context information in different scenarios. We identify scenarios that greatly benefit from the added context information, but we also point to examples in which the added context information does not make a difference.

Different Query Types We compare the results for different types of queries. We use two kinds of queries for the experiment: *No Calls* queries that do not contain any calls (i.e., 0|M) and *Partial Calls* queries that preserve about half of the calls from the original usage (i.e., N|M). As motivated in the description of the methodology in Section 11.1.1, these query types represent two different use cases.

The plot in Figure 11.8 shows the results for the different recommender instances. The filled bars represent the result for instances that do not consider the additional context information. All bars have a white extension on top that represents the increase in prediction quality gained by considering $\Box_{+D}$. The rightmost category in both parts of the plot contains the aggregated results of all queries that contained 7 or more method calls and therefore did not fit into another category (i.e. 0|7+ contains

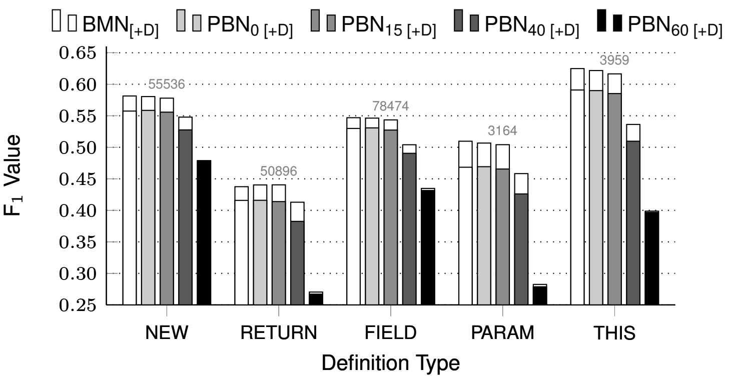


Figure 11.9: Results for Different Definition Sites (All SWT Widgets)

0|7, 0|8, ...; *|7+ contains 3|7, 4|8, 4|9, ...). The small grey number over the bar groups denotes the amount of queries that fall into this category.

Let us first consider only the filled bars of the approaches without additional context information. We find that 0|1 seems to be a special type of query for two reasons. First, it occurs about three times as often as 0|2 but the prediction quality is lower. Second, compared to the other query types and to PBN_0, the impact of the clustering on prediction quality is notably different for PBN_{60} (i.e., 0.15 vs. ~ 0.08). All approaches show similar characteristics: they have a jump in prediction quality from 0|1 to 0|2, but the prediction quality constantly decreases for 0|2+ queries. However, the decrease is uniform for all approaches.

Prediction quality stays roughly on the same level across all N|M categories. Only PBN_{60+D} exhibits a decreasing F_1 value for increasing M values. Both plots suggest that calls seem to be a very important piece of context information. Additionally, calls seem hard to predict when no other calls are given already, the F_1 value of 0|M is lower across all approaches.

Patterns that contain many calls are hard to predict if queries contain no calls.

The white bar on top of all results represents the delta in prediction quality introduced by considering $\square_{+D}$. The plot shows that additional context information generally increases prediction quality. However, the increment is so small for some combinations that the white box is invisible. A special case is the 0|M queries, for which we could measure a big delta. Apparently, the additional context information is especially helpful in situations where no calls are included. The delta seems to be comparable between all other query types and between all approaches, the differences seem to be negligible. It only seems to be a bit bigger for 0|7+ queries. The benefit of the additional context information for N|M queries is smaller, but present.

The greatest benefit of using $\square_{+D}$ is gained for queries that do not contain any calls.

Different Definitions To get a better understanding about the scenarios in which the recommender systems performs well, we conducted an experiment that evaluated the different kinds of definition information discussed in Section 10.1.1 separately. We generated queries with the *Partial Calls* strategy for the available object usages of all SWT types. The resulting F_1 value is the average over all queries.

The results of this experiment are shown in Figure 11.9. Each bar group in the plot represents a definition kind, denoted in the horizontal axis. Each single bar reflects the results for a specific approach. The filled part of the bar is the result if no additional context information is used, while the white part on top represents the delta introduced by $\square_{+D}$. The small numbers attached to the bars denote how many queries are available for this kind of definition, please note the large differences in these numbers. Compared to the other definitions, there are only few usages available for PARAM and THIS. The vertical axis shows the prediction quality as F_1 value.

In general, it is interesting to see that the prediction quality is very different for the various kinds of definition. The best results are observed for queries on object usages with THIS definition, but the number of queries might not be big enough to decide that. Many queries exist on object usages with NEW and FIELD definitions and both show comparable results, even though queries on object usages with NEW definition perform with higher prediction quality. The prediction quality for queries on object usages with PARAM definition is slightly lower than this, but again, there is only a small number of usages available for this definition. The worst results are achieved for queries on object usages with RETURN definition, the prediction quality is ~ 0.15 lower than for NEW definition.

All recommenders in this experiment gained similar improvements from additional context information over all definitions. The only exception is the excessively clustered PBN_{60+D} instance, for which the gain was smaller in some bars than the plot resolution.

The additional context information is equally valuable for all definition kinds.

Section Summary

In this section, we have conducted a series of experiments to evaluate the differences between several configurations of the PBN and BMN recommender. The goal was to extend the scope of evaluations and to show that prediction quality alone is not sufficient to judge the quality of recommenders. Our experiments have shown that it is always necessary to find a good tradeoff between prediction quality, model size, and inference speed that is best for the users of the recommender systems. Accepting minor losses in one dimension often enable huge benefits in the other two dimensions. Our experiments have also shown that future evaluations should always analyze the scalability of the recommender, as it might significantly impact the findings.

The performance function that is used for evaluation has to consider more than one property and the creation should reflect the use case. For example, our experiments only evaluated properties that are relevant for the direct user of the recommender system. We have ignored any computation cost that can be moved to an offline server, but these costs could be important too. If the computation of a single model would take several days, than the practicality of an approach is limited, independently of its potential prediction quality.

11.2 Evaluation of Artificial Evaluations

Conducting realistic evaluations of RSSE is very challenging and often involves humans (e.g., [41, 88, 154, 181]). Unfortunately, conducting such controlled experiments is often infeasible due to the high cost in terms of both time and resources [127]. Other issues include the reproducibility of the experiment or privacy constraints when analyzing developer activities. In addition, controlled experiments are based on selected use cases, which limits the generalizability of the results.

To overcome these challenges, many researchers have resorted to *artificial evaluation strategies* that generate evaluation queries from released code. These strategies can overcome the drawbacks of controlled experiments with real developers. They are easy to conduct and scale well, allowing them to cover different scenarios. However, using them raises the following critical question: How realistic are they? In other words, do artificial evaluations actually reflect real-life usages? Given that the method for creating queries from released code greatly differs between the evaluation strategies we surveyed, it is important to understand how different decisions may influence evaluations. To the best of our knowledge, these questions have not been systematically and empirically investigated so far.

Our interaction tracker, FEEDBAG++, provided us with a fine-grained change history of source code from several developers. This allowed us to understand how the source code evolved over time and which method completions helped the developer to reach the final state. In our experiments, we feed a method-call recommender we built before (see Section 10.1.3) with queries extracted from the captured code changes and compare this realistic evaluation strategy with artificial approaches to answer two research questions:

RQ1: Do artificial queries affect the measured prediction quality of recommenders?

RQ2: Do real queries have properties that are not reflected in artificial queries?

Our results show that artificial evaluations can be misleading, often suggesting a higher prediction quality than what would be achieved in practice. We show that the differences result from ignoring evolving context that is not captured in artificial queries. Our results help toolsmiths make informed decisions about the evaluation strategy best suited for their goals and understand implications of these decisions.

In the remainder of this section, we will go through the following steps to answer our research questions. We will classify common evaluation strategies seen in related work based on the design decisions they make (Section 11.2.1). Using our dataset of captured developer interactions, we collect a set of real code changes and use this dataset to conduct an extensive experiment. We will first present the experimental setup (Section 11.2.2, 11.2.2, and 11.2.2) and then evaluate the quality of artificial evaluation strategies and tradeoffs in their design space (Section 11.2.3). Finally, we will discuss the immediate implications and findings of our results (Section 11.2.4).

11.2.1 Current Evaluation Techniques

One focus of our review of related RSSE approaches was a survey of current evaluation techniques in Section 2.1.2. Our survey has suggested that artificial evaluations are more popular than real evaluations, which is not surprising, since real evaluations tend to be too expensive and time-consuming to be practical [127]. They usually also involve factors like privacy considerations and convincing developers to use a research prototype. While artificial evaluation strategies do not have the issues mentioned above, it is important to understand how close they are to real evaluations. To investigate this, the goal of this chapter is to answer the two research questions posed in the introduction. Specifically, we evaluate different artificial evaluation techniques that employ different *query generation strategies* and compare them to a realistic

Table 11.1: Classification of existing artificial evaluation strategies. Columns show the identified query scenarios while rows show the selection strategies.

	0|M	**N|M**	**M-1|M**
Linear	[22, 192]	[22, 83, 169, 262]	[195, 261]
Random		[79, 127, 192, 195]	

evaluation. Our survey identified two factors that differentiate the artificial query generation strategies: the query scenario and the selection strategy.

Query Scenario Any piece of code has *context information*. This includes structural context information such as the surrounding method or class, as well as information such as which methods have been called and in which order. The *query scenario* describes *how much context from the final code is kept* in the query. Most query-generation approaches focus on the target information they want to predict, e.g., a method-call recommender focuses on the methods called in a particular context, while a parameter recommender focuses on the parameters of a particular call. Given a final code snapshot of `M` items of target context information, the query-generation approaches range from removing all of this context (*0|M*) to "leave one out" (*M-1|M*), with shades in between, where the target context is partly preserved (*N|M*).

0|M In this case, all target context information is removed from the final state of the code, resulting in a minimal query. If the approach depends on specific information, e.g., the type of the variable on which code completion was triggered, this information is preserved. This strategy mimics the situation where developers are just starting to write code and may not know where to start. Creating such queries is straightforward since there are no ambiguities in what goes into a query.

N|M In this case, parts of the existing code is preserved. This mimics the typical development scenario, where developers implement some parts of a method, but potentially miss details for which they need the recommender's help.

M-1|M In this case, only one piece of information is removed. This mimics the case of developers who already implemented most functionality but only miss one part.

Selection Strategy The *selection strategy* is the second differentiating factor, which determines *how partial information is selected* from the final context. It answers the question: Given a complete piece of code from a repository, which parts of it should be removed for querying? Several approaches assume a linear development of source code and remove the later parts of a method. Other approaches perform a random selection of the context or even repeat the random selection multiple times to cover different parts of the existing code.

Linear Assuming code is developed in a linear fashion greatly simplifies the evaluation and makes its implementation straightforward. However, there is no empirical evidence that developers actually code in this fashion and so it is unclear how realistic this assumption is.

Random For a thorough evaluation, several random sub-selections are made for a complete usage. The results are averaged to get one representative prediction-quality

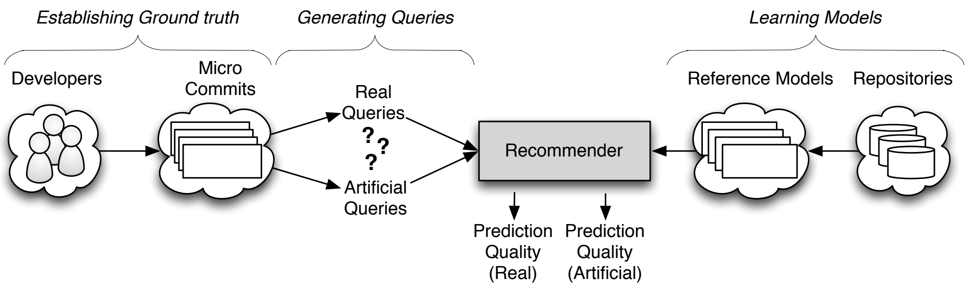


Figure 11.10: Overview of Evaluation-Comparison Strategy

measure. The averaging adds an extra layer to the implementation. Heuristics may be needed to limit the number of selected queries, since the number of sub-selections can be quite large. Randomly selecting parts of the existing code to remove might also hide corner cases in which a recommender performs particularly well or badly.

Table 11.1 classifies the related work we presented in Section 2.1.2 along the two identified dimensions. Given the variability of artificial evaluation strategies, it is important to understand the impact of different choices, how they reflect real-life developer usage, and how they compare to a real evaluation strategy. Such knowledge is valuable in two ways: First, it helps to judge the validity of evaluation results reported in the literature. Second, it helps researchers make informed decisions about their evaluation strategies and how to interpret their evaluation results.

11.2.2 Overview of Evaluation Setup

Figure 11.10 outlines the overall setup we use to compare the identified evaluation strategies. The idea is composed of two parts. First, we establish a ground truth, from which we generate different types of queries for the identified evaluation strategies. Second, we provide the different queries to a recommender and compare the quality of its proposals as a means to compare the different strategies.

To ensure fairness and comparability, we use the same recommender system for all evaluation strategies. The choice of the particular subject recommender system is less important, because the effect of any weakness or strength of the recommender will be equal across all evaluation strategies. We use the *Pattern-based Bayesian Network* (PBN) method-call recommender [192], because it provided us with a complete open-source evaluation pipeline. The core data structure used in PBN is an *object usage*, which contains information about the context in which the usage of a particular API type was observed (e.g., the enclosing method or the way the object is initialized), as well as the set of methods that have been invoked on that instance of the type.

To build the recommender, we create reference models for various API types (the *Learning Models* step in Figure 11.10). We used an open-source dataset of C# projects as input training data to create the reference models [189]. The dataset was created from 360 open source GitHub repositories and contains usage data for more than 560 unique APIs, each of which includes various types. We filter the dataset to the 407 types that appear in our ground truth and use the collected usages to build reference models for these types.

Establishing the Ground Truth

To ensure a realistic and fair comparison, we first establish the ground truth about what queries developers actually perform and what the code looks like at that point. Thus, we are interested in capturing a development history of how method calls are added to the code under edit over time. This allows us to replay the development using the actual state of the code at query time, i.e., to simulate a controlled experiment with real queries. It also allows a comparison of the results for different recommenders.

The commit history obtained from a project's source control repository has been commonly used to obtain such a development history (e.g., Hassan and Holt [81]). However, it has also been shown that, on average, commits are created on every third day, with a high variance between users and the type of changes [109]. It has also been shown that version-control commits shadow many of the intermediate code changes [166]. This means that the version history found in public repositories of Open Source Software is too coarse grained for our purposes. Instead, we choose to capture more fine-grained code changes directly from the developer's Integrated Development Environment (IDE), similar to the idea previously proposed by Robbes and Lanza [201]. While both Hassan and Holt [81] and Robbes and Lanza [203] used some form of development history to improve recommenders, our work is different in that we use this development history to *compare different evaluation strategies* rather than use it to improve the recommender itself.

Creating the Tooling To get more fine-grained code changes, we want to capture snapshots of the code under edit every time a change occurs. Such information is best captured directly from within the developers' IDE. Additionally, we want to capture interactions of the developers with the IDE's code-completion tool and store which method was selected from the list of proposals, if any. We combine the code snapshot and the timestamp, as well as the optional selection of a method proposal, if available. We call the collection of this information an *enriched micro commit*. We can use these micro commits to create real queries and to replay the recorded development history, including code completion, after the fact.

To collect such information, we extended FEEDBAG++, an open source instrumentation of VISUAL STUDIO that collects interactions of C# developers with their IDE [3]. We extended the instrumentation of the code completion and added a static analysis that extracts context information from the code under edit. Each time code completion is triggered by the developer (or when it pops up automatically), we create a snapshot of the source code under edit. Snapshots are stored in the form of *simplified syntax trees* [189], a lightweight format that also includes typing information and which supports markers for code completion trigger points.

Gathering the Data FEEDBAG++s sources are publicly available, and the tool can be installed from within VISUAL STUDIO. Once a user has the tool installed, their interactions and micro commits are automatically captured. Users can then upload this captured data to our servers at any time through a provided dialog.

We first deployed our modified version of the tracking tool with ACME (cannot be named for privacy reasons) that develops tax and accounting-related software as well as in-house software for 50 years. It employs more than 1,600 developers, out of which more than 400 write programs in C#. Development projects span from small training examples to core-business applications. In addition, we advertised the project in several social media channels and during various conferences to widen our user base.

Table 11.2: Contributed Events per Developer

Id	Type	# Days	# Queries	%
0*	Researcher	89	4888	20.6
1	Student	66	4625	19.5
2	Student	52	3162	13.3
3	Hobby Programmer	48	2096	8.8
4	Student	28	900	3.8
5	Student	32	771	3.2
...				
45	Unknown	4	10	<0.1
...				
55	Professional	3	2	<0.1
total		753	23,746	100.0

Even before our active recruitment efforts, several open-source developers independently installed our tool after seeing it in VISUAL STUDIO's public plug-in repository. We also had several students install FEEDBAG++ while they were developing different systems, including a game, web applications in ASP.NET, or their own Masters thesis project. Finally, the author of this thesis along with student assistants also had FEEDBAG++ installed while they working on the tools used in this paper.

Our final data set, therefore, contains queries from a variety of users and projects, including industrial developers, open-source developers, researchers, and students. The individual participant contributions are listed in Table 11.2. Note that the table is cropped for brevity, but the complete list is available on the artifact page. The asterisk in the table marks the contributions of the thesis author.

In total, we received submissions of captured data from 56 users. Out of these, 27 were industrial developers that provided 13% of our queries. The remaining users (with percentage contributions shown in parenthesis) were 8 students (45.9%), 4 researchers (21.5%), and 2 hobby programmers (9.6%). The remaining 15 users (10.0%) decided not to fill our (optional) profile information. The submissions cover 753 days and span over a period of 13 months, but not all users participated the whole time.

Post-processing the Data There is still a gap between our collected ground truth and the input data required for PBN. The collected micro commits are file-oriented snapshots whose contents reflect a complete type declaration in a file (e.g., a class with all its methods and the corresponding method bodies), while the input data for PBN are object usages. To bridge this gap, we first sort the micro commits by time and declared type. As a result, we get the development history of a file. After this, we extract object usages for all types used in each micro commit. Since a micro commit represents a whole class, we extract object usages for several types in this step. Finally, we merge the resulting usages from all micro commits of the same user, group them by type and by enclosing method, and preserve the order to create a complete usage history.

Figure 11.11 illustrates an example. The file icons depict micro commits, while the squares represent object usages. The character in each square shows the type of that specific object usage; the index is only added for easier reference.

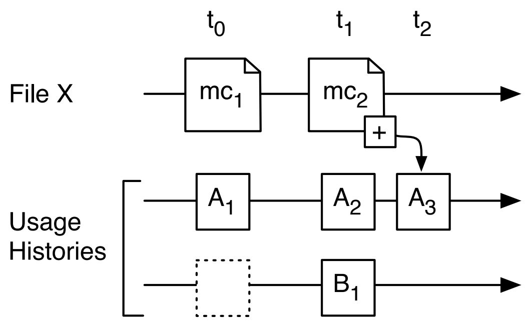


Figure 11.11: From Micro Commits to Usage Histories

Assume that we have created two micro commits for file X that were captured at times t_0 and t_1. For the micro commit at t_1, we captured the information that a specific method was selected from the code completion proposals (depicted by the "+'). Assume that at time t_0, the code only contained usages of type A, while at time t_1, usages of type B were also added. Our usage extraction therefore finds a single object usage for type A in the first micro commit and two object usages for the types A and B in the second one.

In order to use all information that is contained in the usage history, we developed two strategies that transform implicit knowledge into explicit states in the usage history. Given all extracted usages, we can identify types that are used in a specific context, but for which we were not able to extract usages for all micro commits. In the example, we can derive that B was not yet used in the first micro commit, but added to the second one. To make sure that we include such usages that are created *from-scratch* (i.e., usages where we only know the object type and surrounding context but do not have any called methods yet) in our evaluation, we add an empty usage to B's history at time t_0.

Another corner case that needs to be handled is if the last micro commit in a usage history contains a selection. In this case, we will not actually see the effect of the selection in subsequent micro commits, because none exist. In order to preserve the selection for our evaluation, we create an artificial usage in which we merge the usage on which code completion was triggered with the selection result. Referring to the example in Figure 11.11, a method selection took place for the micro commit captured at time t_1. Assuming the completion trigger would have taken place on the object usage of type A and that $t+1$ is the last micro commit in the history, we would create an object usage A_3 at time $t+2$, which contains the selected method too.

After the extraction, our usage history may contain subsequent occurrences of the same object usage. This duplication may be due to several reasons. For example, consider the situation in which a developer invokes code completion, but then cancels it. This would result in two equal usage snapshots in the history. Another example is that the developer uses the types A and B in the same context. She adds multiple method calls to the object of type B, but leaves the object of type A untouched for a while. This would add several unchanged usage snapshots to A's history. We post-process the histories and remove such duplicates.

After applying the described transformations, our final ground truth set that is used in our experiments consists of 6,189 usage histories. On average, each usage history contains 4.6 snapshots, but a few outliers exist with a length of more than 700 usages. We manually inspected these cases and all inspected cases were examples in which the developer spent time in a specific context implementing an algorithm and working with the same type over and over again. Frequent additions and removals of the same methods bloat up the histories for these usages, e.g., adding and removing a log statement. However, outliers with more than 17 steps represent less than 2% of our collected data, so we did not introduce special handling for these cases.

Our collected usage histories cover 407 types used in 5,834 different method contexts. The dataset we used to train the recommender contains a total of 650,340 object usages for these types. Note that PBN models do not contain ordering information. While object usages only contain a *set* of method invocations, our extraction implementation guarantees that the order in which invocations are entered reflects the order in the source code. While this is irrelevant for object usages used in PBN models, it is important for generating order-dependent queries for some of the evaluation strategies we compare.

Generating Queries

We will discuss next how we use our collected ground-truth data set to generate queries that can be used to compare different evaluation strategies. As discussed in Section 11.2.1, artificial evaluations are based on released code found in a repository. This version is treated as the final state of the code and considered correct and complete. Artificial evaluations apply heuristics to approximate past states of this version, which are then used to generate queries. The final state serves as expectations to judge the quality of an RSSE. The evaluation, thereby, measures the recommender's ability to lead the developer from the past states towards that final state.

Since artificial evaluation strategies are much cheaper to apply than real evaluations, we expect that toolsmiths and researchers will continue to use them. The goal of our evaluation is to analyze the different heuristics that are used to approximate queries. We do this by comparing an evaluation based on these heuristics to an evaluation based on real queries, which we obtain from our object-usage histories. This evaluation comparison uncovers qualities and problems of the artificial strategies and we identify guidelines for future evaluations.

Obtaining Real Queries The usage histories from our dataset mimic the real development history and reflect changes to the files under edit in a very fine-grained manner. In terms of evaluating a recommender, a *query* has an input state and an expected output state. We assume that the last snapshot of a history represents the outcome of a development task. We, therefore, use this final snapshot to formulate our expectation on the evaluated recommender's proposals, similar to how artificial evaluations use the code from a repository. However, the difference between an artificial query and a real query lies in which code state is used for the query input.

We extract 23,746 queries from our usage histories by combining pairs from each snapshot in the history with the final state. After filtering 6,218 pure removals (i.e., queries in which calls were removed, but no calls were added) and 10,371 queries that contained equal start and end states, we ended up with 7,157 real queries for the evaluation.

Start

```java
public void M() {
  T t = new T();
  t.m1();
  t.mX();
  t.mY();
}
```

End

```java
public void M() {
  T t = T.Create();
  t.m2();
  t.m1();
}
```

Strategy	Definition Site	Calls in Query
Linear	`T.Create`	`m2`
Random	`T.Create`	`m1` and `m2` in multiple queries
Real*	`T.Create`	`m1`
Real	`new T()`	`m1`, `mX`, `mY`

Figure 11.12: Example of a $3 - 2 + 1$ query case (labeled as 1|2) and the queries created for each strategy (Yellow is change, red is removal, and green is addition).

Generating PBN Queries At this point, we have the set of real queries obtained from the usage history. To compare the results of a real evaluation to an artificial approach, it is necessary to emulate the heuristics that build an artificial start state from the real end state. Recall from Section 11.2.1 that there are two dimensions used to automatically generate artificial queries: *query scenario* (0|M, N|M, and M-1|M) and *selection strategy* (Linear, Random, and Real). To create the artificial queries, we first identify the query scenario for each real query and then apply the different selection strategies on that query to create an artificial one.

Query Scenario We first categorize the collected queries by the type of performed change. We assign each query a label that reflects the number of calls added or removed. A label `n-r+a` means that the query contained `n` calls in the input and that `r` calls were removed, while `a` calls were added, to come to the final state. Consider the query in Figure 11.12 as an example. The start state contains three calls (`m1`, `mX`, and `mY`). For the final state, `mX` and `mY` were removed, while `m2` was added. Thus, this is an example of a `3-2+1` query.

Since PBN can only suggest method-call additions, and not removals, we needed to adapt query labels accordingly. We do so by dropping the removals from the labels used for categorization. For example, for the query in Figure 11.12, even though the removals `mX` and `mY` are used in the real query, we do not include them in the final categorization label. Instead, we label the query as a 1|2 change to indicate that the query already contained one out of the two final calls. We assign the queries to the three query scenarios based on this label.

Selection Strategy Once a query is assigned to a query scenario, we next generate the actual queries for each selection strategy as follows. We use the query in Figure 11.12 to explain the difference between strategies.

Linear In this case, the query is taken only from the end state. The method calls to be included in the query are selected top-down from the list of existing method calls in the end state. For our example query, which was classified as a 1|2 query according to the above query scenario classification, we could technically generate both 0|2 and 1|2 queries. Yet, we only generate a 1|2 query to have the real query to compare it to. The linear approach would select `m2` for the query, because it is the first method call that exists in the end-state code. However, the order of appearance of calls in source code at the end state might not be the order in which they were added during development, as can be seen from the example.

Random The random strategy also selects the information to include in the query from the end state, but instead of selecting method calls linearly, it selects them randomly. In the end state of Figure 11.12, there are only two methods. To generate the 1|2 query using the random strategy, we randomly pick one call and use it in the query. To make sure all scenarios are covered, the approach repeats this random selection until all possible method calls are covered and the results are averaged. In our example, two possible queries will be generated where the first includes only `m1` as input and expects `m2` and the second includes only `m2` as input and expects `m1`. An average of the prediction quality of both results is then taken to reflect the prediction quality of this whole query.

Real Based on the unique opportunity of having detailed development information available in our collected usage histories, we introduce the real selection strategy to reproduce what would actually happen during development or during a controlled experiment with subject developers. We only use the information that would be available to a recommender in a real-life scenario where the query was placed during development. This means that only the start state is used to query the recommender. The fine-grained history reflects the *evolving context* over time and includes all information that do not show in the end state, because they were missing, changed, or got removed in the usage history. For example, the query does not only include method `m1`, but also the removed methods `mX` and `mY`, as well as the original definition site `new T()` that was changed during development to the static call `T.Create()`.

Real Without Noise (Real)* To understand the effect of the *evolving context* in real queries, we add a fourth strategy that we call *real**. The only difference between real and real* is that the latter would not include any evolving context in the query. To create the query, real* uses the context of the end state and selects all methods from the start state that have not been removed during development. In the example, the 1|2 query would include only `m1`, because it existed before. In addition, `T.Create()` would be selected as the definition site. We consider *real** to be an artificial approach, because the selection of the methods for the query and the inclusion of the correct definition site can only happen *after the fact.*

The input of each generated query is used to request proposals from the recommender, which is built from the reference models. We measure the prediction quality by comparing the set of proposals with the expected outcome, which is the set of methods that are missing in the query input, but that exist in the end state. The similarity is calculated through the F1 measure (i.e., the combination of recall and precision). A detailed overview of the number of queries we captured in each query scenario is shown in Table 11.3.

Table 11.3: Available queries for N|M scenarios

N|M	1	2	3	4	5	6+	Σ	
0	4327	703	592	82	13	30	5747	(80.3%)
1	-	741	109	60	3	6	919	(12.8%)
2	-	-	252	52	15	11	330	(4.6%)
3	-	-	-	95	19	6	120	(1.7%)
4	-	-	-	-	19	5	24	(0.3%)
5+	-	-	-	-	-	17	17	(0.2%)
Σ	4327	1444	953	289	69	75	7157	
	(60.5%)	(20.2%)	(13.3%)	(4.0%)	(1.0 %)	(1.0 %)		

11.2.3 Do Artificial Queries Work?

In this section, we empirically compare the different selection strategies described in the previous section. We follow the evaluation comparison strategy that has been outlined in Figure 11.10 and explained in Section 11.2.2 in order to answer the two research questions posed in the introduction.

Do Artificial Queries Affect the Measured Prediction Quality?

To compare the evaluation strategies, we feed our reference recommender with the generated queries. The quality is measured by comparing the proposals to the expected additions available in the end state of the query case. Table 11.4 shows the quality obtained for each selection strategy and corresponding query scenario. We explain these results by going through each query scenario (columns) and comparing the selection strategies (rows). We present the results of each query strategy and provide an interpretation of the findings.

0|M Queries in this category contain no method calls as part of their input. This means that none of the method calls in the expected state appear in the input state. This is the most common case and 80.3% of our data falls into this category. We factored out NEW as a special subset of 0|M that reflects the case in which the developer did not write any code so far. In this case, no information is available in the code context, apart from the enclosing method and the type of the usage. Another special category is 0|1 queries, which can be assigned to both 0|M and M-1|M. We decided to assign it to the 0|M category but show it as a separate column in the table for better examination of the results.

The results in Table 11.4 show that for such queries, no difference exists between the artificial strategies. This is not surprising, because all artificial strategies end up with the same query created from the same end state. One observation is that it seems that the more missing calls exist, the harder it is for the recommender to find them. Another observation is a notable difference in the results of real queries. While the difference is already noticeable for NEW queries, it gets worse for 0|1 and the recommender seems to be unable to process 0|2+ queries (4.9%). The only difference here between real and the artificial strategies is missing or changing context information. While the definition site might change for all 0|M queries, 0|1 and 0|2+ might additionally contain calls that are about to be removed.

Table 11.4: Different query scenarios [F1/%]

	0|M			N|M		
	NEW	0|1	0|2+	1|2	N|3+	**M-1|M**
LINEAR	60.2	38.2	34.0	15.0	45.4	34.8
RANDOM	60.2	38.2	34.0	29.9	50.0	30.9
REAL*	60.2	38.2	34.0	19.1	45.1	24.0
REAL	53.0	15.1	4.9	12.0	43.3	23.7
# Queries	3612 (50.5%)	1445 (20.2%)	690 (9.6%)	741 (10.4%)	294 (4.1%)	375 (5.2%)

N|M Such queries contain some existing calls and reflect the use case in which the developer has already started to write some code and then asks for help. This is the case for 14.5% of the queries in our dataset. We factored out 1|2 queries for the same reasons as the 0|1 queries.

We find that the F1-values for this query scenario differ greatly for 1|2 queries. The random approach reports the highest quality for the recommender (29.9%). The other approaches result in a much lower quality, 15.0% for linear and 19.1% for real*. The real evaluation, which includes *evolving context* in the query, yields the lowest quality with 12.1%). The results for N|3+ differ slightly between the different selection strategies and are around 43-45%. The only approach that sticks out is random, which reports a much higher quality of 50%. Overall, the random results are consistently higher in the N|M query scenario than the other selection strategies. The real evaluation again reports worse quality than the remaining artificial strategies.

M-1|M This is the extreme case of N|M: only the last call in the method is missing, while the remaining methods are given as part of the input. Only 5.2% of the queries in our dataset fall into this category. The results show that the quality of the real and the real* strategy are comparable. However, the reported quality of random is much higher (30.9%) and even exceeded by the linear strategy (34.8%).

The selection of methods in the query is the only difference between the three artificial approaches, yet we see different results. The linear selection strategy reports the highest quality (34.8%), while real* is close to the real result and reports 24.0%. The random approach mixes the different extremal values and reports a quality in between (30.9%). It seems that some missing methods are harder to predict than others and that developers select these methods last.

Interpretation of Results For the 0|M queries, no difference can be seen between the artificial approaches. On the other hand, the real queries perform worse, because the definition site is unknown. The more calls that need to be predicted, the more problematic this seems to be. In addition, a direct comparison between real* and real shows that evolving context in real queries reduces the reported quality. This is true for all query scenarios. We look at this more closely in Section 11.2.3 where we examine the effect of evolving context information.

When compared to the random selection strategy, the linear strategy seems to be better in some cases (e.g., M-1|M), while worse in others (e.g., N|M). The randomized generation of multiple queries and averaging of the results seems to cause a smoothing

Table 11.5: Effects of Evolving Context [F1/%]

	NEW	0	−M	△D	−M+△D
REAL*	60.2	21.4	29.0	38.8	40.9
REAL	53.0	21.4	25.9	24.1	1.9
# Queries	3612	794	1173	249	1329
	(50.5%)	(11.1%)	(16.4%)	(3.5%)	(18.6%)

effect that creates more robust results. We averaged the results over all queries (not shown in the table) to see if this is a general effect. However, we found that all artificial selection strategies achieve results that are comparable to each other (linear 46.6%, random 48.2%, and real* 46.6%).

Although all artificial strategies create queries from the same set of methods found in the end state of a micro commit, they select the methods to include differently. This seems to have a real effect on prediction quality as can be seen for both N|M and M-1|M query scenarios. Such a difference suggests that some methods are harder to predict than others. A possible explanation is that these calls might be used very rarely and the recommender favors common method calls.

To explain this intuition, let us go back to the example in Figure 11.12 and assume that `m2` is not a very typical method, which makes it harder to predict. In the linear case, the recommender is lucky, because it gets `m2` as input and has to predict `m1`. In the random case, two queries are provided to the recommender. It does really well in the easy query (i.e., when the input is `m2`) and really bad in the other one (i.e., when the input is `m1`). Since the results of both queries are averaged, the extreme effect is smoothed out a little. However, for the real* strategy, the recommender gets `m1` and has to predict the hard method `m2`. It, thus, follows that it would have a lower prediction quality.

As opposed to artificial strategies, queries in the real strategy only include the methods that were actually included by the developer first. Since the prediction quality of the real query is even lower, this also suggests that developers add such hard-to-predict methods later to their code rather than earlier. Thus, it seems that for such corner cases, artificial approaches tend to give a higher prediction quality than it would actually be the case in a real setting.

Do Real Queries Have Properties Not Reflected in Artificial Queries?

Released code is a static picture of the development activities. Any intermediate changes or removals cannot be identified by artificial evaluations, because they are part of the development process and do not show up in the released code. Our data set provides a unique opportunity to explore the impact of intermediate code changes and code removals on recommender evaluations. To explore this, we compare the *real* and *real** strategies in more detail. The input to real queries might miss context information or might contain context information that is changed in the end state. For PBN, this includes both method calls that have been removed during development and changes to the context of an object usage. The context is defined by the enclosing method and the definition site of an object. By construction, the enclosing method is fixed for all query generation strategies, because object usages are always bound to a specific method. If the enclosing method is changed, this would count as a different object usage. On the other hand, the definition site, which describes how an object instance was created, may change during development (e.g., see the definition site in Figure 11.12). If there are no removals or changes in the definition site, the query will not contain any artifacts of an *evolving context*, i.e., changed or removed information. Artificial selection strategies are unaware of such changes, and are thus not sidetracked by them. This results in a higher prediction quality than for a real query. We examine the effect of evolving context in more detail in Table 11.5. In addition to queries in an *unchanged context*, the context may evolve in four different ways that are applicable here. Moving method calls to other enclosing methods is a fifth kind. However, distinguishing these moves from removals is outside the scope of our work and not considered here.

*Newly Created Usage (*NEW*)* The most represented category in our dataset is usages that are added from scratch. The values are equal to Table 11.4 and only included for easier comparison. The difference between real and real* here can be explained by a changed definition site.

*Unchanged Context (*0*)* We categorize queries into this category that do no include any changes in the surrounding context. By design, no difference exists for real and real* queries in this category.

Retired Method Calls ($-M$) This category refers to the case in which the query has calls in the input that are removed during development and no longer contained in the end state. For example, this can happen when the developer chooses a more appropriate method call to use.

We find that this is the case for 16.4% of the queries in our dataset. Our results suggest that these extra methods that appear in the input seem to confuse the recommender. The quality decreases from 29% for real* to 25.9% for real.

Changed Definition (ΔD) A small part of our dataset (3.5%) includes queries in which the definition site changes between the input and end state. This particular context change has a big impact on real evaluations. Table 11.5 shows a quality drop from 38.8% for real* to 24.1% for real.

Combination ($-\mathrm{M} + \Delta D$) The second largest category in our dataset are queries that combine both a change in definition site as well as the removal of method calls. It prevents any meaningful proposal in our experiment for the real approach and the quality drops to 1.9%.

We find that real evaluations are sensitive to context changes. While retired method calls have a minor impact on the result of a real evaluation, the change or absence of the definition site leads to a large difference in the result. Such an impact is not covered in the artificial evaluation strategies.

While the specific context information we discussed in this section (removed calls and changed definition sites) may be specific to PBN, other recommender systems that use context information will suffer from the same problem when this information evolves (e.g., changing the implemented interfaces [88], removing method calls on related objects [261], re-ordering call sequences [195], etc.). The general problem is that context information can change during development. Artificial evaluations usually do not mimic context evolution, which results in “unrealistic” quality reports.

11.2.4 Results and Implications

Our results indicate that the evaluation focus is an important factor when determining which query scenario and selection strategy to use. In the following, we will discuss the immediate findings of our results and their implications for future work.

Choice of Query Scenario Depending on the type of users the recommender is meant to support, researchers can decide on which query scenario they use in their evaluation. While 0|M represents green-field projects or novices that request help from the recommender before actually starting to write code, N|M reflects advanced programmers that write something before triggering code completion, or maintenance tasks, in which the programmer typically starts to edit existing methods. The M-1|M query scenario can be used to demonstrate support for corner cases or that even experts that just miss "the last bit" can get valuable support.

Choice of Selection Strategy Our ground truth reflects developers' typical code completion usage. In our dataset, 0|M queries are the common case for which no difference exists for the different selection strategies. On the other hand, we find great differences between the selection strategies for the uncommon cases that include the N|M and M-1|M query scenarios. We observed for these query scenarios that some methods seem to be harder to predict than others. Depending on the methods that are contained in the query, the reported quality may vary. The random selection strategy leads to a smoothing effect that can overcome this effect to some extent, but it leads to a reported quality that is generally higher than for the other artificial approaches.

Overall, by comparing the average results over all queries, we could measure only minor differences between the different artificial selection strategies. We could not find a single artificial evaluation that is more realistic than the others. When compared to the result of real queries, we found that all artificial evaluations consistently report a higher prediction quality.

Considering the Effect of an Evolving Context Missing or changing context information affected many query cases. We found that in real queries, the evolving context has a negative effect on the recommendation quality. The more changed context features the recommender engine takes into consideration, the bigger this negative effect is, because more "confusing" information is passed to the recommender. Researchers should be aware of this effect in their evaluations.

While this does not necessarily invalidate the results of artificial evaluations, it results in more positive results than what would actually be measured from real developers. Existing artificial evaluations remove only the target information (i.e., the method calls). However, to make an artificial evaluation more realistic, toolsmiths have to check their assumptions about the context information used in the query. They have to identify what context information may change in reality and they should mimic that in their automatically generated queries. For example, they should remove additional context information (e.g., definition sites in our case) from the input of a specific fraction of the queries or set it to a random, but valid, value (e.g., set an arbitrary definition site stored in the model for a fraction of the queries).

Further Comparisons We designed our experiments to use a single recommender to compare artificial and real evaluation techniques. Future experiments should compare artificial and real evaluation techniques (using the same dataset) across multiple

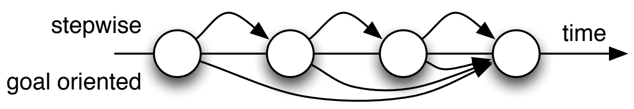


Figure 11.13: Identifying Expected Query Results

recommenders. This allows investigating which recommenders are more resilient to evolving context, for example.

Improving Recommenders We found that 26.2% of the snapshots in our dataset were pure removals of method calls. While it could be the case that these are artifacts of maintenance tasks, we hypothesize that this could also be caused by going through a learning process on how to use a given API to solve a task at hand. These removals are not considered in any RSSE so far, probably because they cannot be observed by statically analyzing source-code repositories. Future work should investigate these removals. It seems that they contain information that could be leveraged to further improve existing approaches or to create a new kind of RSSE that points the developer to methods that should be removed.

Evaluation Style Our dataset consists of a series of source code snapshots as illustrated in Figure 11.13. The evaluation design we used in this chapter follows what we refer to as a *goal oriented* style, in which the validation queries are created from the intermediate states and the proposals are validated on the end state. This follows the intuition that this final state is the desired outcome of a development task and that the recommender should lead the developer towards this outcome. It might be that a recommender that suggests such an end state increases productivity, because it points the developer to methods she must use in the end. However, it might also be that the proposal hinders the learning process, because it might be unexpected at that exact point in time.

We observed in our dataset that the path towards the final state is rarely straight. Therefore, a viable alternative could be to use a *step-wise* evaluation style instead, in which each subsequent change would be evaluated individually. For example, each snapshot could be used as a query and the subsequent snapshot formulates the respective expectations. Such an approach follows the intuition that an RSSE is expected to propose what the developer thought was right at the time of query. Such a tool might support the learning process, which could be beneficial, especially for novice developers. However, experienced developers that know the API in question might be disturbed by incorrect proposals.

It is not clear which of these should recommenders be designed to do and accordingly, which of the evaluation strategies is better or more realistic. To investigate this, we conducted a preliminary experiment that used each usage snapshot as a query and compared the proposals to both the next snapshot and to the final snapshots. We calculated the F1 measure for both scenarios and, over all queries, the stepwise approach resulted in an average of 38.4% and the goal oriented style in an average of 41.4%. A Mann Whitney U test shows that the difference between these two evaluation approaches is statistically significant (p-value=0.001). Therefore, future work should investigate assumptions on the expectations used to evaluate recommender systems. In other words, it is an open question whether to RSSEs should propose the *correct answer* or the *expected answer*. Future should analyze the effect of both.

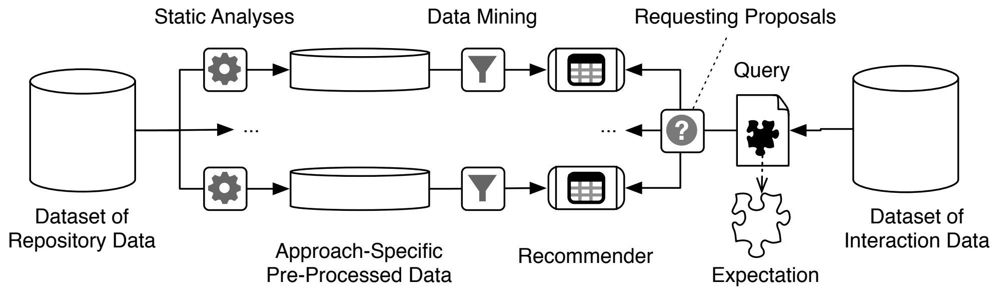

Static Analyses
Data Mining
Requesting Proposals
Query
...
...
Dataset of Repository Data
Approach-Specific Pre-Processed Data
Recommender
Expectation
Dataset of Interaction Data

Figure 11.14: Conceptual Workflow in a Comparative Evaluation

Section Summary

In this section, we surveyed related work to identify the current state of the art of evaluation strategies. We found that many existing approaches are based on artificial evaluations. Our goal was to analyze whether these evaluations reflect real-life usages. We presented a concept for the comparison of different evaluation strategies and collected a ground truth data set that allowed us to conduct the comparison. We analyzed how the results of a real evaluation relate to the artificial results. We showed that artificial approaches report a misleading quality for the evaluation if they do not consider evolving context information and, therefore, provide more information in the query than what would be available in a real scenario. We believe that artificial evaluations can still work if context evolution is carefully emulated in the evaluation.

11.3 A Benchmark for Comparative Evaluations

The last big idea related to the evaluation of RSSE that is introduced in this thesis is that evaluations need to mature to the point where it is easy to compare different approaches. Our experiments for the PBN recommender provided a first step into this direction and included a previous approach, the BMN system, as a baseline. This was a special case though, as both systems rely on the same input data. Finding a solution for the general case is more challenging, because every approaches relies on its specific input data. Establishing a common ground sufficient for the comparison of different approaches that are possibly not even invented yet is a hard task.

This is were our platform come into place that caters for the evaluation of recommendation systems through the combination of a generic data schema, *simplified syntax trees*, two datasets (released code and fine grained source-code history captured in-IDE), and a platform, C$\boxed{A}$RET, that facilitates working with this data. Evaluating various approaches in a comparable setting is made straight-forward. The different approaches to build and evaluate a recommender system all follow a similar workflow [191] that is depicted in Figure 11.14. Most approaches rely on analyzing released source code, which is included to the left-most side of the image. We have created such a dataset from source-code that is hosted on GitHub and provide it for reuse (see Section 8.1). The approaches then typically employ static analyses to extract information from the source code and store it in a pre-processed form that is tailored to the specific approach. From this intermediate data, approaches apply a diverse set of data mining techniques to learn a model that can be used in a recommender.

Once the recommendation system is instantiated from repository data, it can be used in evaluations. To this end, the recorded data from the interaction dataset serves as the ground truth. We pass the observed IntelliSense interactions as a query to the recommendation system and compare the proposals to the recorded selection of the original user to calculate the prediction quality. An evaluation might also request the model size or measure the inference speed to cater for the other dimensions that we have introduced in the previous section about holistic evaluations (see Section 11.1).

Compatible Systems Are Required All recommenders that can be compared in such an evaluation need to have the same *interface*. In our concrete example, the interface requires a query methods that provides a list of method call proposals, when called for an SST that contains a completion trigger point as an anchor. The interface may also contain additional methods that provide the means to request further information from the recommendation system, e.g., the memory consumption of the current model.

Some interfaces might be compatible even though they are not equal, e.g., a recommender that considers order might be applicable in evaluations that do not consider it, but in general, it is not possible to meaningfully integrate a recommendation system that does not follow the required interface of an evaluation. In such cases, a separate evaluation interface has to be developed, e.g., for evaluating snippet recommenders.

Support for Different Paradigms The sketched workflow introduces *static analysis* and *data mining* as the two phases that are necessary to instantiate the recommendation system. However, some approaches might be able to skip one or both of them. For example, an approach might train a recommender without creating intermediate data or it may rely on hard-coded strategies that do not need to be trained at all.

Several alternative flows are possible in the sketched concept. For example, if approaches share preprocessed data (i.e., object usages in case of PBN and BMN), static analysis is only done once and the approaches only have their specific data mining technique. It is possible to integrate those approaches in the workflow by skipping unnecessary phases. Another kind of recommender might not be trained on released code, but through changes observed in the program history (e.g., [167]). These kinds of recommenders can be integrated in the workflow by conducting a cross-fold validation over the recorded changes in the interaction data.

Alternative Evaluation Styles Having both an extended dataset of repository data and the captured interaction data at ones disposal, it is possible to make use of various query styles to accommodate for specific requirement of an evaluation.

The most realistic option to evaluate method call recommenders is using the completion events that were recorded from actual IntelliSense invocations. It is also possible to extract fine-grained change histories of edited source-code from enriched event streams. While this strategy provides an alternative way of evaluating method call recommenders, it also enables a realistic evaluation of other recommendation systems, e.g., snippet recommendation or code search.

Sometimes, it is not possible to base an evaluation on the captured interaction data, for example, when no interaction data is available for a specific API that should be evaluated. In these cases, the evaluation can fall back on an artificial evaluation approach by running a cross-fold validation on the dataset of released code. We have used this technique ourselves for the evaluation conducted in Section 11.1, because we did not have access to any captured interaction data at that time.

Overall, the different contributions of this thesis make it possible to conduct a comparative evaluation. We did not yet conduct such an evaluation though. While we have already been working on several new recommender implementations (Section 10.2), only a few recommendation systems were fully implemented at the time of writing. In addition, the implemented approaches (BMN and PBN) have already been exhaustively evaluated. This section has sketched precisely how the workflow of a comparative evaluation would look like. Realizing the idea of a large scale comparative study with a collection of state-of-the-art recommender systems is one of the very next steps in our future work.

The described benchmark only considered method call recommendations. While we do not see any conceptual limitation that should prevent using the data for other kinds of source-based recommendation systems, e.g., snippet recommenders, we did not detail out the required steps so far and it might be that additional data is required. We would assume that the biggest challenge there is to come up with meaningful metrics to quantify the prediction quality, but that the dataset provides all required information once a metric is found.

The interaction dataset represents the only limitation we see for an evaluation base on the novel benchmark. Despite the significant size of the dataset, the number of participants, the amount of available interaction data, and the coverage of possible APIs are limited. The data collection is still on-going though and we can observe new installation of FEEDBAG++ each day, so this limitation is constantly being mitigated.

Chapter Summary

This chapter was involved with the evaluation of RSSE, and especially concerned about method call recommenders. We have discussed our work in three related directions. We could show in Section 11.1 that the evaluation of RSSE always presents a tradeoff between prediction quality, inference speed, and model size. While existing evaluations often focus on prediction quality, we were able to show that the other two dimensions have a significant effect on the usability of such systems and that they must not be ignored.

Many evaluations rely on artificial data for evaluations to avoid the expensive collection of realistic ground truth. We have analyzed the realism of such evaluations in Section 11.2. Our results show that evolving context information exists in reality that is not captured in the coarse grained commit histories of version control systems. We found that this evolving context has a significant impact on the realism of the reported results, if not considered carefully.

In an attempt to avoid the recurring task of designing evaluations for method call recommenders in the future, we have proposed a novel benchmark for the evaluation of method call recommender in Section 11.3. The lack of available RSSE implementations prevented a usage of the benchmark so far, but we have sketched in details how various evaluation strategies and query styles can be supported. Our benchmark improves the state of the art in the evaluation of method call recommender and facilitates reusability, replicability, and comparability in this area of research.

While the focus of our work was on method call recommenders, the findings of this chapter are of a general nature. We expect that they are not specific to our experiments and that they extend to other types of recommendation system as well.

12 Threats to Validity

This thesis presented several experiments that made use of our implemented tools and collected data. Our conclusions in these experiments rely on the validity of our tools and the methodology for the experiments. Even though we designed the experiments diligently, the validity of the conclusions might be threatened for several reasons. These threats are always related to a specific experiment. However, the work presented in this thesis reuses many parts and some threats exist for all experiments. Instead of including them repeatedly with every experiment, we discuss all threats in one place. We will discuss threats to validity of the following parts of our work.

General Some threats are related to cross-cutting concerns span across several experiments, e.g., threats related to our meta models introduced in Chapters 3 and 4.

Data Collection All experiments use at least one of our datasets. We have compiled a dataset of released source code (Section 8.1) and have collected developer interactions in two studies. Our original data collection in an industrial context (Section 9.2) was later repeated in a field study (Section 8.2). Some experimental threats are caused by these datasets.

Source-Code Evolution We have designed an experiment to understand how developers write source code (Section 9.1.1) and to study the applicability of SSTs for source-code differencing (Section 9.1.2).

Time Budget This category subsumes all our experiments that are based on captured developer interactions. We used this in our study on the time budget of developers (Section 9.2), including our analyses of testing behavior (Section 9.2.2).

Recommender Creation We have build RSSE as a show case for SSTs (Section 10.1) and have sketched the creation of several more (Section 10.2). All threats to validity that we identify for this line of research also apply for the benchmark that we have introduced in Section 11.3.

Holistic Evaluation Our research that was concerned with an extended evaluation of method-call recommenders (Section 11.1).

Meta Evaluation We have conducted a study that evaluates existing evaluation strategies of method-call recommenders (Section 11.2).

In this chapter, we will present the threats to validity that we have identified for these categories, our estimation of risk, and mitigation strategies. To avoid duplication, we will use the category labels for structuring without introducing them again.

12.1 Internal Validity

This section will discuss threats to the internal validity. This relates to factors or effects on the experiments that influence the results, but that are not controlled. Ignoring these factors can change the interpretation of the results and the conclusion.

Data Collection One threat to the statistics we collect is that we do not know the exact number of participants in our collection of interactions. Originally, strict German privacy laws did not allow us to uniquely identify participants in our industrial collaboration. To overcome this, we presented lower and upper bounds to the number of participants in our study on the time budget of developers and perform all our analyses on *developer days*. While we cannot state conclusions about individual developers, we can safely analyze what a typical workday for a developer looks like.

Our follow-up study introduced a questionnaire into FeedBaG++ that requests demographic information from participants and also includes an optional user id. This allowed us to improve the heuristics we use to merge uploads of the same developer, but we still cannot identify developers if the id is not filled or loose connections between uploads when developers change their ids or move to new machines. The only possible mitigation of this threat is to introduce a registration in future work, which cannot be introduced without the side effects discussed in Section 6.2.2.

Also the dataset that we created from GitHub repositories represents a threat to the internal validity of experiments that use it. Our selection of repositories is restricted to those that can be opened by our ReSharper based batch runner. This biased selection might introduce effects in the data that influence experimental results or might blur effects that would become obvious in a more diverse project selection. To mitigate this threat, we selected a large number of repositories and covered a wide range of project domains.

Source-Code Evolution Our study of the edit location in a method is based on the interaction data that we have captured in our field study. Unfortunately, we don't have any further information about the tasks performed by the developers that have installed FeedBaG++. It could be the case that we only tracked novice developers that closely followed tutorials, which would also explain the linear edit strategy that we have observed in the experiment. However, we tracked developers over a long time and most participants classify themselves as professional developers. We acknowledge that our event stream might contain traces of tutorial work, but we think that the influence on the overall result is not significant.

Time Budget Our interaction tracker only tracks activities in-IDE and we don't have any information beyond that. We don't know exactly what affected the actions that we have recorded and the conclusions are subject to interpretation rather than knowledge, a longer inactivity could be a sign for a coffee break or for a discussion with a colleague. In addition, we also don't know if the recorded activities represent a single user or if multiple users were using the machine simultaneously, e.g., in a pair programming session. For these reasons, it is possible that the recorded event stream does not model some effects on the developer activities, which could lead us to false conclusions. This gap cannot be closed through an extension of FeedBaG++ though. If this kind of internal validity is required, the experimenter must complement the data collection with an observational study that captures such details.

It might be the case that FeedBaG++ fails to track some interactions that are important for the recognition of a specific development activity or to recognize an uncontrolled factor on the developer. To mitigate this threat, we performed our own pilot phase of over six months with two of the authors and six students, before publishing the interaction tracker. During this time, we repeatedly analyzed the tracked interactions and improved FeedBaG++ to increase both the correctness and completeness of the tracked event stream. Our experiments were conducted after publishing the interaction tracker and they did no reveal any missing data. The additions that we have implemented for the field study only capture additional events that allow answering new research questions, e.g., we added an instrumentation for test events to be able to study testing behavior of developers.

One threat to our conclusions in the industrial collaboration is that were not able to verify our interpretation of the data with our participants, e.g., through interviews. It is also possible that the conclusions are influenced by developers that behaved differently while using FeedBaG++, e.g., to better present themselves. However, since they were aware that the collected data is completely anonymous and that, legally, their managers cannot access it, we believe that it is very unlikely that their behavior was "staged" over the six months of data gathering.

Recommender Creation We created a recommender to prove the applicability of SSTs and C$\boxed{\text{A}}$RET in the RSSE domain. However, we restricted our re-implementation efforts to a single recommender, PBN, that we have used before. In addition, the risk exists that the underlying static analysis contains bugs that result in invalid results. Such threats would have a ripple effect, as all our follow-up evaluations are based on the PBN system. To reduce the risk, we provide an extensive test suite for the analysis to make sure that we cover a wide range of usage scenarios. We have also build a working IDE integration for the recommender system. While not being rolled out to the general public, we use it to manually check the static analysis and the recommender. The system has been subject of several internal presentations and we are confident that both the underlying static analysis and the recommender work as expected. An additional benefit of SSTs that we would like to point out is that they provide the means that enable writing such an analysis only once and reusing the same static analysis both in an offline setting and in-IDE, which prevents a second implementation of the same analysis that represents a second source of bugs.

While we did not implement other approaches to collect further evidence for the applicability of SSTs, we sketched detailed recipes that describe how to create them step by step. While we followed the original publications very closely, we might have misunderstood the technical descriptions. To mitigate this risk, we have discussed the algorithms and our approach for solution with our peers.

Holistic Evaluation Experiments results heavily depend on the measurements taken on the experimental machine. We might have failed to isolate a factor the influences the measured performance of the system, e.g., a specific kind of `I/O` or a peculiarity of the execution environment. This influence would have affected the various approaches in a similar way though and would not lead to a bias towards one specific configuration. In addition, we ran the experiments multiple times to achieve stable-state performance and designed the measurements in a way that avoids `I/O` during the measurements.

Even though they might not be described in the original work, we gave our best to provide an optimized implementation and we include obvious optimizations for PBN and BMN that improve memory consumption or performance. However, maybe we might have missed other optimizations that significantly influence the experimental results, which would pose a threat to the validity of our conclusions. We are confident that our implementations are representative for the three analyzed properties, as they show the behavior that we have expected a priori, and we don't see other immediately obvious optimizations that would provide other significant improvements.

Meta Evaluation We used an existing recommender system with its existing evaluation pipeline to decrease the possibility of implementation bugs and their effect on the results as much as possible. We implemented only those parts necessary for the new experiments and thoroughly tested them. Our results also depend on the quality of the static analysis of the C# code we use to extract the micro commits. C# is a real-world programming language, and the analyzed programs contain very complex expressions. While we excessively tested the analysis in an extensive test suite, it is possible that we may have missed corner cases.

The prediction quality range reported by PBN for the new dataset is lower than prediction qualities usually obtained in the literature, including our previous experience with PBN on JAVA [192]. We plan to investigate this more qualitatively to find out if this is a factor of language differences (C# versus JAVA) or a factor of the API types included in the ground truth data set. However, our goal is not to promote a particular recommender, but rather to compare different evaluation strategies. Thus, we do not believe that this impacts the validity of our results since any difference between evaluation strategies would still be observed. Additionally, to avoid confounding factors that may affect our comparison, we ensure that we have enough object usages to build reference models for each API type we are interested in. This ensures that all API types have a fair chance of getting good predictions. That said, even if the model for one API type does not have enough data, the effect would be the same across all evaluation strategies, keeping our comparisons fair.

12.2 Construct Validity

The threat of construct validity refers to situations, in which the experimental setup is not fit to collect the intended measures or in which the measured data does not represent valid evidence that can be used as an indicator to derive an answer. This thesis introduces abstractions for source code and developer interactions and collects corresponding datasets. This data is used in all experiments, which poses the main threat for the validity of our results, because the experimental validity depends on the fitness of our representations to reflect reality.

General This thesis introduces abstractions for capturing developer activity and to represent source code. It might be the case that this simplified model does not properly reflect reality. The same threat exists for the tooling that we have provided on top of the representations. Any bug in the transformation into the abstracted form or in any of the reusable components that work on top of it may lead to a misrepresentation of our model that prevents meaningful conclusions.

To prevent this threat, we followed the practice of test-driven development and have expressed many transformation scenarios and our corresponding expectations to make sure that the implementation captures our intuition. Especially for the interaction tracking, we also made heavy use of manual testing in a test deployment on our own machines to assert that the crucial interactions are logged. Still, it might be the case that FEEDBAG++ generators don't capture some interactions, e.g., related to the completion events that are used in multiple of our experiments, that would change our conclusions. If future works misses a new kind of information in the data representation that is required to model their picture of the development process, it is possible to add this missing piece. Both enriched event streams and the generator framework of FEEDBAG++ are extensible.

The SST representation leaves out several information from the original source code, e.g., attributes and comments. The current design of the data structure was motivated by our concrete use case and we ensure compliance with our testing efforts. Novel use cases might depend on information that is currently not modeled in SSTs. We don't think that this is an issue because the extent of the model is defined and deficiencies are easy to recognize to avoid using it in an inappropriate use case. In addition, also the SST representation is extensible in future work.

Data Collection Users of FEEDBAG++ are aware of a data collection, which might introduce a bias. This effect is called Hawthorne effect and can invalidate the results that we derive from the collected data. We have already discussed the implications of this effect in Section 6.4 extensively. Overall, we don't think that the effect will have a significant effect on the collected data. The FEEDBAG++ configuration that we have published does not provide any value to its users. Participants will neither gain an advantage in changing their behavior, nor do they have to fear a drawback when submitting real data. FEEDBAG++ also does not alter the user interface at all, so we don't expect the behavior of our users to be significantly biased.

Source-Code Evolution We have created a novel merging strategy for histograms of different sizes to establish a technique that allows a fair comparison of edit locations in source code. To mitigate the threat that we choose a representation that leads us to invalid conclusions, we have discussed several alternatives for the merging and compared their advantages and drawbacks. In the end, we decided to use a *fair distribution* for the merging, because it represented the best tradeoff. A better visualization might exists for such a problem that we were not aware off. It might also be necessary to represent different edit styles in a more complex model that would allow to derive more precise conclusions than our simple histogram based model.

Time Budget FEEDBAG++ allows participants to delete (parts of) the recorded sessions before uploading it to our servers. In our experiments, we have observed large amounts of inactivity and it is possible that this observation is caused by participants making heavy use of this feature. However, we doubt that developers would go through the hassle of deleting many individual events. It is much more likely that they delete an entire day, should they be unwilling to share information about larger parts of it. Therefore, we assume that the days we received events for are close to complete.

Holistic Evaluation We have conducted an extensive evaluation for PBN using a cross-folding approach over the dataset that we have extracted from an existing repository. The evaluation queries are created artificially by removing facts from the observed examples. At the time we conducted the experiment, we copied a state-of-the-art evaluation strategy, but it was not clear if it is a valid strategy to create artificial queries. In fact, we analyzed this later in a meta experiment that evaluated this evaluation style (Section 11.2). We found that *evolving context* is a strong influence factor in realistic evaluations. Not mimicking it in artificial evaluations does not necessarily invalidate an experimental comparison between recommender systems, but it does affect the perceived prediction quality in a realistic setting.

12.3 External Validity

Threats to the external validity of our experiments affect the generalizability of our findings to other experimental settings.

General Enriched event streams are designed to capture a holistic picture of the development activities that happen in-IDE, but they might miss crucial parts of the development process that are relevant for other studies. We have shown in our experiment on the time budget of developers that the data structure is sufficient to draw a broad picture of the development activities. We have also shown in the study on testing behavior that very detailed questions can be answered too. The latter also proves the extensibility of the design and that it is easy to add detailed events later, should they become relevant.

SSTs represent a simplified version of a program that leaves out several properties of the original source code. Future work might rely on features of the source code that we did not capture so far, rendering SSTs useless for their work. We cannot foresee any future approach, but reviewed a broad selection of related work to identify requirements for our design of SSTs. The resulting representation is close to source code and includes all constructs that are required for extended static analyses.

Data Collection All experiments of this thesis rely on at least one of our datasets. Our datasets may not be representative though, which threatens the generalizability of our experimental results.

We did not know the experience level and roles of the participants in our study at ACME. This might threaten the representativeness, e.g., when most of our participants fall into one experience level. However, we believe that the large number of (measured) participants and the diverse project portfolio of ACME mitigate this threat significantly. The many similarities between the behavior of our participants and behavior reported in other studies [13, 113, 148, 156], make us confident that our participants are representative. In addition, we have added a self-estimation of the skill level for new participants in our field study. Future work should analyze how the experience level affects the collected data.

The dataset of released source code from GitHub is restricted to C# solutions and to project types that can be opened by ReSharper, on which we built our tooling. This filtered selection might reduce the insights to a specific kind of project and might prevent the generalization of the results. However, our dataset contains a large number of solutions. The contained programs belong to various domains and cover very different project sizes. While we cannot measure the representativeness of the dataset, we are confident that it covers a broad variety. In addition, we think that the true value of the dataset is its alignment with the interaction dataset, but this could be interpreted as a bias as well, if the study participants are not representative.

Time Budget We conducted our study on the time budget of developers only in a single company, ACME, and our results may not generalize beyond that. To provide the means for validation of our findings, we conducted a second field study with open-source developers. Additionally, we only track IDE interactions for C# developers in VISUAL STUDIO. It is possible that the observations we make do not generalize to developers using other programming languages or IDEs. However, we believe that our results help in reasoning about differences between languages and tools, and we already compare our results to similar studies on other IDEs [13, 148, 156].

Holistic Evaluation Our holistic evaluation analyzed several properties of RSSE in addition to prediction quality. We found in our experiment that scalability is a large issue for RSSE. However, this issue might be specific to our dataset and to the SWT framework that was targeted in our experiments. The framework is very complex and commonly used in the ECLIPSE repository that we used. The largest part of all available libraries might involve fewer types, a less-complex API, or fewer clients that make use of it. As a result, the scalability problems that we have found in the experiments may not be representative for most libraries and frameworks.

We argue though that even though our dataset was big enough to reach the scalability limits of BMN, its size is far from the sizes of *big data*. Algorithms need to be improved to be able to scale to vast input sizes. While these may not be reached for all frameworks, it is also necessary to provide tool support for very common frameworks, e.g., the core library. In addition, we did not neglect prediction quality over scalability in our experiments, so even if an algorithm does not run into scalability issues for a library, the prediction quality should not be affected.

Meta Evaluation Our results are based on the comparison of the different evaluation strategies for only one recommendation system for single-object patterns (PBN). Our results might not generalize to other recommenders that have a different notion of context, that deal with multi-object patterns, or that propose complete code snippets. Since we use the same recommender across all evaluation strategies, our results are valid and ensure non-biased comparisons. Since this is a limitation of the recommender rather than our ground truth data set, the same comparison can be repeated with additional recommenders. However, this might entail some engineering effort to adapt each recommender to use the ground truth data set. We cover various kinds of developers by collecting data from diverse groups. We have data from professional developers, open-source developers, researchers, and students. However, we do not currently have information about the project types these developers worked on. In the future, we will capture more information about the type of the project (e.g., green field or maintenance) and the role of the developer (e.g., developer, tester, or integrator) to get a better picture about the task the developer was working on.

Chapter Summary

In this chapter, we have discussed the threats to validity for the experiments presented in this thesis. We have grouped them into internal validity, construct validity, and external validity. Each threat has been presented in detail and we have described our mitigation strategies. While this chapter lists several threats to validity, we are confident that our described mitigation strategies have successfully reduced their risks.

Part V:
Conclusion

This thesis has produced several contributions. Previous parts have motivated the problem, have introduced meta models and the infrastructure around them, and have presented in depth, how we have applied them to evaluate their value. This final part will first discuss our insights in how the individual components of this work have facilitated our own applications or the shortcomings that we have identified working with the platform. We will then present an outlook to future work that is enabled by the results of this work, discuss both immediate next steps and more midterm goals, and will end this thesis with a summary.

13 Discussion

This thesis has introduced a complete platform, C$\boxed{\text{A}}$RET, to facilitate studies on the development process. Instead of evaluating the various components separately, we decided to collect first-hand experience and we used them in our own research. After reporting the results in the previous chapters, we will now discuss our insights, the experienced advantages and disadvantages, as well as opportunities for improvement that we have identified. We will go through the various components one by one.

13.1 Enriched Event Stream

We introduced *enriched event streams* to preserve in-IDE development activities. They combine two abstraction levels: the *process-level* captures high-level information about the activities, whereas the *context-level* enriches these events with additional context information, e.g., source code snapshots for change events. Enriching the raw process information provides a more holistic picture of the performed activities.

Enriched event streams have been crucial for the experiments that we have presented in this thesis. The events capture a very broad range of in-IDE activities on the process-level. From the information captured, we were able to gain a broad overview of development activities and we could conduct the study on the time budget of developers (Section 9.2). For several activities (e.g., testing), the captured information at the context-level has allowed studying the details of these activities. For example, our study on the testing behavior of developers (Section 9.2.2). These studies were enabled by our data representation and could not be done before for VISUAL STUDIO.

Even though the data representation covers a wide range of activities, we found it convenient to work with. In addition, we had students working with us that have built an achievement system on top of enriched event streams that is integrated into VISUAL STUDIO (Section 6.3.2). Their positive feedback shows that the events are also usable by non-experts that lack knowledge about the internals of the project.

A special case of context information is the embedded source-code snapshots that are part of edit related events, like code completion events. Through these events, it is possible to reconstruct a fine-grained history of the source-code that is perfectly aligned with the process information. This opens up new research directions that were not possible before. For example, we have successfully used it to create the ground truth for our meta study on the evaluation of artificial evaluations (Section 11.2). We foresee many further uses, such as learning more about debugging activities of developers, analyzing navigation behavior, or understanding details of code evolution.

While both abstraction levels, namely process and context, are general and extensible for future requirements, a conceptually different style of representing the historic information or the abstraction of certain needed information might be required for specific approaches in the future. For example, we capture the events *on change*, *on save*, and *on commit* in our enriched event stream to indicate different edit granularities. However, some approaches extract this information from versioning systems and also make use of the actual commit message to derive further information, for example to classify the change type [194]. Since commit messages were not central to our research, we currently do not capture or preserve them. However, they can be added in the future by extending the corresponding VCS generator.

The current design of the events requires the introduction of a new event class in case a new activity should be captured. This enables type-safe access to the events, but might lead to a proliferation of classes. However, we argue that the alternative of encoding both the event type and its specific context information in a generic event using magic strings is worse. However, we acknowledge that the current implementation requires manual casts from abstract `IDEEvents` to concrete event types. Future work should provide a type-safe conversion function, e.g., through the visitor pattern.

Our first-hand working experience and the feedback from our students say that the events are easy to use. However, we also got feedback from other researchers that would have expected to access the data in an SQL database or basic CSV files. This would have allowed them a processing in scripts, instead of using our provided binding in a general purpose programming language. While we think that our proposed way is the better alternative, future work should think about defining a default SQL schema for the data and about providing automatic exporter for the data to cater for tools like R[37] or Power BI[36] that are commonly used among data scientists.

In summary, the representation of enriched event streams provides the required information to conduct both high-level studies and to look into low-level details that are aligned with the process information. We found several points in our experiments that could be addressed in future work. However, many successful integrations in our own research or in projects of students that worked with us prove the usefulness and especially the low entrance barrier to understanding and analyzing the captured data.

13.2 Simplified Syntax Trees

We have introduced *simplified syntax trees* as a means to facilitate static analyses of RSSE. Additionally, we designed them to make the self-contained snapshots of edited source code in the IDE feasible, without the necessity of storing the whole workspace. SSTs mix a three-address-like representation with an AST that is close to source code, combining static analyzability with readability. The fine-grained editing snapshots that we capture also allow to use SSTs in studies on source-code evolution.

We have conducted several experiments that have been made possible through the SST representation. First and foremost, we have build static analyses on top of SSTs to build RSSE. The C[A]RET platform provides several reusable components for complex analyses (e.g., points-to analysis) and transformations (e.g., inlining transformation), that find sufficient information in SSTs to achieve their goal (see Section 7.2). We have gained a deep understanding of the applicability through implementing a sophisticated static analysis for a line of research that extracts *object usages* from source code to learn correct API usage (Section 10.1). In addition, we have sketched how other works could be replicated on top of SSTs to show the generalizability of the representation (Section 10.2). Through these sketches and because we have based the SST design on a survey of several related RSSE that go beyond method-call completion, we expect that our experience extends to other researchers that work on different kinds of recommender, such as snippet recommenders or code search.

The SST representation features *anchors* that make it possible in enriched event streams to perfectly align process information with changes to the source code. Right now, we provide a completion anchor that marks the current edit location in case IntelliSense is triggered. In addition, we capture how the developer interacted with the pop-up afterwards. This unique combination allowed us to develop a general benchmark for evaluating method-call recommenders that leverages the fact that the exact interaction can be reconstructed from the data (Section 11.3).

This discussion does not need to repeat limitations that we have already discussed in the SST chapter, e.g., that they do not capture all literal values from the source code, because we have accepted these sacrifices in the design as a tradeoff between different goals. However, several points are no direct limitations of the current design, but reflect on the usability and applicability of SSTs as a representation.

Our review of related work in Chapter 2 contained work that are concerned with source code evolution and with the creation of RSSE. We derived our requirements from these works to the best of our understanding from the technical descriptions in the papers. While we discussed the surveyed papers with peers, there is still a chance that we misunderstood some technical description or that our selection of papers might have missed crucial publications. In the same way, it is not granted that SSTs satisfy the requirements of future approaches. This is especially true for works on source code evolution, as the main driver for the SST design was static analyzability. We do not see any limitation of SSTs that prevent such studies though and could even show the applicability of SSTs in our initial experiments on source-code evolution. We still need to prove the applicability to more complex questions in future work.

Based on the identified requirements, SSTs do not include all information of the original source code and we leave out code elements that are typically not used, e.g., modifiers, comments, or attributes. We successfully evaluated the compatibility of SSTs and C$\boxed{\text{A}}$RET by sketching the implementation of several existing RSSE (see Section 10.2), which makes us confident we cover many approaches. It might still be the case that issues arise for other reimplementations that we did not attempt. Given the wide area of RSSE that is covered in our survey and that SSTs are very close to an AST representation, we expect that they are applicable to many more approaches. In addition, the reduction does not limit the generality of the approach. We designed SSTs, C$\boxed{\text{A}}$RET, and the transformation to be extensible, adding support for additional information is just a matter of extending the SST grammar and spending engineering effort to extend the transformation. The impact and tradeoff of different abstraction levels on the recommender techniques should be further analyzed in the future.

SSTs were created as a tradeoff that makes regular in-IDE snapshots of source-code feasible by making them self-contained without the need to store the whole workspace. While the JSON serialization makes it easy to parse SSTs and to store the preserved typing information, the serialized file gets significantly larger than a pure plain-text representation of the same source code. This tradeoff was not further analyzed in this thesis, but future work could try to combine both ideas and capture a text-based source-code snapshot while preserving fully-qualified references for types and type elements. Instead of embedding them in the naming scheme, we could also include all corresponding types in the serialization, e.g., in a partial type table.

While we expect that the combination of source-code and process information will provide a valuable input for more than just our use cases, the completion anchor that we store in SSTs might not be sufficient for all cases though. Some applications, e.g., extract method refactoring, require a different kind of alignment anchors, e.g., which nodes have been highlighted prior to a refactoring invocation. Future work should analyze how SSTs could be extended to support additional anchors.

Source-code changes and process information can be perfectly aligned in enriched event streams, if the SST contains an anchor for the recorded interaction. However, our statistics show that not all edit related events contain such an anchor. This can happen, for example, when the source-code snapshot is taken for an IntelliSense invocation that is triggered within an invalid code snippet. Future work should further analyze the extent and the effect of this limitation of our implementation and improve the handling of cases, in which a meaningful anchor could be added.

We were able to show that using SSTs for source-code differencing worked well. However, our experiments have also shown that several opportunities exist in this context to make the SST representation more applicable. For example, future work could split up the normalization and type preservation into two distinct steps and avoid normalization in source-code differencing tasks. The promising result of this insight is that SSTs can be directly used for source-code differencing, which is a first step towards building snippet recommenders on top of SSTs.

In summary, the SST representation provided all necessary information for the static analyses and transformations that we have performed in our research. Even though we found several limitations that threaten the generality of the representation, we could show in our experiments that the representation is fit for the intended use case. The fact that several of our students have successfully used the representation in labs and theses show that the representation is useful and usable.

13.3 FeedBaG

Our interaction tracker FeedBaG++ provides a practical way of instrumenting Visual Studio to capture development activities. The tracker is built as a framework that can be extended with custom generators that can be used to capture additional data. FeedBaG++ provide a generic infrastructure for the data collection and provides the required components to manage the collected events and upload them to the server system. To increase trust of the user, we have provided several privacy components that can be used to anonymize or redact data that participants are not willing to share. We provide an integration to request basic demographic information from our users. FeedBaG++ features several components that provide value to users as an incentive for participation. Depending on the intended use case, these components can be activated, the default version only contains the data collection components.

The interaction tracker is restricted to in-IDE activities, with only a few exceptions (e.g., system events, VCS events). The extensible design of FeedBaG++ makes it possible to integrate any event source into the in-IDE event stream We have demonstrated this extensibility through adding testing events after the fact to support our study on the testing behavior of developers (Section 9.2.2). We encourage other researchers to contribute generators to FeedBaG++ to further enrich the event stream with more specialized event types.

We have deployed FeedBaG++ to collect data in practice. The deployment took place in multiple iterations, which allowed us to learn from the feedback of the users and from insights of our own research. In the beginning, we have mainly tracked ourselves, but we have soon deployed it with professional software developers in an industry collaboration, and later in a field study that attracted several hundred developers. The successful deployments show several things. First, the industry collaboration proves that our privacy considerations are compatible with German privacy laws and that the implementation is stable to be permitted in a production setting. Second, the number of voluntary participants that have used FeedBaG++ for extended periods without getting value from the tool underlines the importance of the idea. Third, we have also successfully used the tracker in a preliminary attempt of a controlled experiment [176], which proves that it indeed can be adapted to different use cases.

Empirical research on developers is a complex task. FeedBaG++ focuses on the aspect of capturing in-IDE data from the working developer. Depending on the research question, a more holistic picture of the process is required. For example, studies in psychology or applied social studies might need information about the sentiment of the developer or any utterances they express in the experiment. Another example are studies that involve non-IDE tools (e.g., the command line or websites) or data (e.g., biometric information). We cannot differentiate in-activities from absence with the current tracking, e.g., coffee break or having a phone call, because we only see "no activity". While the latter requires additional information sources, it is a fundamental question whether the former kind of information should be integrated into the stream (e.g., the name of the currently active application). We decided that we generally do not cross the IDE boundary in our tracking to avoid privacy issues.

The current implementation listens for command executions in the IDE. Not all executed functionality is accessed through the command pattern, some is just executed programmatically. To catch these cases, we listen to interactions with arbitrary buttons in the UI and store their caption as an approximation for their command id. However, such captions often have a general nature (e.g., "Ok") and do not carry meaning. Future work should analyze whether more expressive commands ids could be created by considering the path to the button in the XML based UI declaration.

The current design of FeedBaG++ that was described throughout the thesis is optimized for a field study, through which data is collected in a distributed setting. Several considerations, e.g., privacy, depend on the intended scenario for the data collection though. For instance, if the tracker is deployed in a controlled experiment as way to simplify the data collection, consistency guarantees (e.g., being able to assign events to a specific participants) outweigh privacy concerns (e.g., removing personal information from the events). In such settings, it is also possible to enrich the tracked interaction data with interviews or with a manual protocol of developer activities to preserve a holistic picture of the development process.

It is an open question whether interactions trackers should be general or specialized. Building specialized tools that are tailored to a specific research question allows reducing details on the client. This helps building trust with users, which increases willingness of participation and ultimately results in higher user counts. A general-purpose data collector on the other hand allows exploring intuitions and to form a research question after having collected the data, at the cost of developers being concerned about their privacy. We think that both kinds of collection have a right to exist. While the general-purpose tool is more appropriate in the exploratory phase of research and more forgiving in terms of problem definition for the data collection, a more specialized tool requires an exact definition of the goal, but is easier to deploy and to maintain. In practice, there might be a transition from one to the other and our interaction tracker provides the means for research and exploration up-front to avoid unnecessary experimentation effort.

Independently of the kind of study that should be conducted, a lot of advertisement is necessary to find participants. In the case of a field study, it is also required to convince people about the value in such a data collection, as they provide insights into their own work practice that go beyond a controlled experiment. This becomes obvious when comparing the download numbers of our plugin to the number of participants that we can identify in our collected data. It seems that our advertisement efforts increased the awareness of the tool and made many people interested, but only a specific subset of developers is actually willing or allowed to take the next step and share details about their development activities. Unfortunately, advertisement also has to stay an ongoing effort. We saw that our download rates and especially the registered uploads went down significantly, after we have reduced our advertisement.

The FeedBaG++ interaction tracker was implemented in a team and the implementation was a major effort that took several years. Unfortunately, the result is not free of maintenance. Regular updates are necessary to keep it up-to-date with its runtime environment. We have to acknowledge that it would be easier to provide the same tool on a Java-based IDE. This would significantly reduce the effort for debugging or for understanding the internals of open-source software, and it may even be possible to extend an existing system like Mylyn. However, even though scientific conclusions should be drawn from a diversified mix of data points, research was very focused on the Java ecosystem over the last decades. We refuse to believe that breaking this cycle and introducing a novel platform in a new environment is a disadvantage.

In summary, FeedBaG++ can track in-IDE activities and is a solid tool for empirical studies on developers. Its configurable feature set makes it applicable in several scenarios. We have deployed it in an industrial collaboration and in a large field study, but it is also valuable for controlled experiments, in which it can be used to simplify the data collection. The general infrastructure allows other researchers to use it and its extensible design allows extending it to the needs of future research, making it a valuable and general tool for research in this area.

13.4 CARET

The C$\boxed{A}$RET platform is the central component of our proposed infrastructure that provides access to the concepts and representations of this thesis. It is a Java based tool suite that provides bindings to our data representations, a tool chain for reading the datasets, and reusable components for various tasks related to studies on developers and RSSE. The C$\boxed{A}$RET platform was designed to facilitate such research and to improve the level of standardization to allow the creation of reusable tools. The result has the potential to be the foundation for future research in this area.

We have used the C$\boxed{A}$RET platform for all research that was presented in this thesis. Most reusable components were created on demand in our own research, but implemented in a way that makes them reusable by others. By now, the platform consists of several components. The part that is universally useful for all C$\boxed{A}$RET adopters is file handling: we have implemented bindings to our data representations and utilities for reading the datasets. We also provide other useful components on top that facilitate working with the data. No sophisticated operations are required for enriched event streams and most of our components focus on RSSE. We provide static analyses and transformations for SSTs (Section 7.2), an evaluation pipeline for method call recommenders (Section 7.3), and several working implementations of RSSE approaches (Section 10.1).

Our experiments have shown that the platform is usable and we found it to be a helpful tool for writing analyses and evaluations. Joined work with several students has shown us that it is easy to get started with C$\boxed{A}$RET. It is necessary to understand the different parts and their connections, e.g., the naming scheme. However, once this initial learning phase is mastered, users are able to implement complex operations. Our students have confirmed this judgment when we have requested their feedback.

The various parts of C$\boxed{A}$RET can be discussed from different perspectives. In the following, we will go through several categories that we found important for judging the value of the platform.

Bindings Through using SSTs in our experiments, we have made the experience that the entrance barrier to using SSTs is very low. After using the language models of ECLIPSE JDT and RESHARPER, the API of SSTs is very similar to use and can also be traversed conveniently with a visitor. While our own perception may be biased, also our students have built sophisticated tools on top of SSTs, e.g., a novel tool for source-code differencing (Section 9.1.2). This proves that out-standers can understand the SST interface and it makes us confident that it is usable.

Our own experience has shown us that the close connection between C$\boxed{A}$RET and our datasets makes it easy to get started and no further preparation is necessary. It is particularly useful for source-code that the data can directly be processed. While source-code datasets usually need to be made compilable first, our representation has fully-qualified type information built-in. The ease of use has proven especially valuable in our collaborations with students. They were quickly able to work with the datasets and for example traverse the syntax tree of the SSTs.

The datasets are made available in a JSON serialization format. This format is both an advantage and a drawback at the same time. While being easy to support and read in new languages, the format is very clumsy. Stored files become large and (de-) serialization is very slow. We think that, being standardized and easy to support, the chosen format exactly fills the required role. We acknowledge though that it has the characteristics of a transport format and might represent a bottle-neck in high-performance calculations. Future work should provide migration tools that enable a migration to binary formats to improve the performance, both to increase the responsiveness of FEEDBAG++ and for offline processing tasks in the C$\boxed{A}$RET platform.

Integration The second point of discussion covers the possibility to integrate C$\boxed{A}$RET. Other researchers will have build their own environment built already, in which they run their experiments. We acknowledge that, to be useful in practice, a platform must not only be accessible and understandable, it must be easy to integrate.

We designed C$\boxed{A}$RET's as a library. It does not require the user to comply with specific framework requirements and it can be used within an existing code base. While this gives the developer the freedom to use the components in arbitrary ways, it makes it possible to break the system or its benefits through inappropriate extensions. An infrastructure cannot prevent all possible problems by design. For example, one lesson that we have learned in our PBN$_{\text{BMF}}$ extension (Section 10.1.3) is that it is important to think early about parallelization of any component that is involved. While the original PBN pipeline is trivial to parallelize (see Section 7.3), we have used Matlab for the matrix factorization in the extension, which turned out to be the bottleneck of the implementation. The Matlab implementation is highly optimized, but we only had a single license and could not distribute the calculation over several machines, which has strongly limited the performance compared to the other parallelized algorithms.

The C$\boxed{\text{A}}$RET platform provides the means to build an infrastructure for building RSSE. However, the platform concept only works through synergy effects: users are attracted through reusable components that provide value to them. Once they finish their research, they contribute the resulting components themselves, which may attract further users. To make this possible, it is necessary to keep the initial effort low that is necessary to get things started and to mobilize new users. To achieve this, we provide an extensive set of examples that explain how to use the platform. However, the current C$\boxed{\text{A}}$RET implementation is not available through common dependency mechanisms, e.g., Maven. It is required to fork our repository and to work within this checked out clone. Future work should spend the effort and create a proper release, e.g., as a Maven package, to make it easier to integrate.

Writing Static Analyses SSTs provide a very lightweight approach for creating static analyses on source code, e.g., collecting invoked method calls from the source code. We experienced this as a big advantage in our own research (Section 10.1.1), because we did not need to configure any analysis toolkits or compile any source code. However, it might become necessary to implement more sophisticated analyses like complex data-flow analyses, and that it is preferable to use specialized static analysis tools for this task. Future work should explore the possibilities to export SSTs into other representations, e.g., back to source code, to allow the application of existing analysis frameworks. In fact, we make it easy to transition from one to the other and already provide components that can generate source code in Java or in C# syntax from SSTs. We are not aware of any limitations that would prevent an export to Jimple, the intermediate representation used by the static analysis framework Soot[5]. In this way, the analysis could leverage many of the existing components of the specialized tool. We advocate that the results of such an external analysis should be processed into a form that can be integrated back into C$\boxed{\text{A}}$RET though to make them available without the necessity of running an external tool. For example, while Soot could be used to run a points-to analysis, the results should be mapped back to SSTs to make it available as a service to others that want to reuse this result. It is important to allow the use of specialized tools without loosing the desirable properties introduced by the C$\boxed{\text{A}}$RET platform, namely comparability, replicability, and standardization.

Standardization One of the main goals and drivers behind the creation of C$\boxed{\text{A}}$RET was the desire to standardize research methodology, tools, and datasets to improve comparability of research results. The C$\boxed{\text{A}}$RET platform solves this on two levels. In the small, we provide the extensible PBN pipeline that provides a standardized evaluation pipeline for object-usage based RSSE. We have successfully used it to compare prediction quality, memory consumption, and inference speed of different recommenders (Section 11.1). At large, we have used the platform for evaluation (Section 11.2) and sketched a generic benchmark for method-call recommenders (Section 11.3). While this benchmark is still a theoretical construct, it makes it obvious that the standardized data structures of the C$\boxed{A}$RET platform are not tied to a specific application, but that they are applicable to many problems.

Toolsmiths will compose the various components of C$\boxed{A}$RET to create novel tools. The platform encourages working with interfaces and standardized data representations, which standardizes the data exchange and makes it easy to build reusable components for others. It also facilitates the composability of the various components and makes it unnecessary to use converters every time data is exchanged between existing tools. The tooling provided in C$\boxed{A}$RET is language and IDE independent, offering reuse opportunities for studies across language and IDE boundaries.

In summary, the C$\boxed{A}$RET platform represents a solid foundation for research. It enabled all experiments that we have presented in this thesis and provide the means for a wide range of research questions. We found it easy to integrate in our research and have experienced various advantages, especially when writing static analyses. The standardized components and data structures that have been introduced by C$\boxed{A}$RET made it easy to write components that are reusable by others. Its value does not come at the cost of usability. We could confirm the usability through our students that have also worked with the platform and that have provided numerous contributions.

13.5 Datasets

The C$\boxed{A}$RET platform provides the concepts and tools that facilitate writing static analyses for RSSE and to conduct empirical studies on developers. To instantiate the platform, data is necessary that can be analyzed in the experiments. We have compiled two kinds of datasets for this thesis that are available for research with our platform. First, we have collected an interaction dataset in a field study (Section 8.2) and in a industrial collaboration (Section 9.2) that has to stay non-disclosed. Second, we have compiled a dataset of released source-code from GitHub (Section 8.1).

The datasets make C$\boxed{A}$RET useable and they were involved in all our applications that have been presented throughout the thesis. We have used them to study source-code evolution (Section 9.1), developer interactions (Section 9.2), to build RSSE (Chapter 10), and both datasets were important in our evaluations (Chapter 11).

Both datasets might be unrepresentative samples of the general population. In case of the interaction dataset, we request demographic information from our participants. However, these only cover basic properties and do not allow to partition our participants into categories that go beyond self-estimated experience. Future work should extend the questionnaire and introduce a concept to capture the role of the developer in the project for which the activities were tracked. The situation is similar for the dataset of released source code. We sampled a large set of C# solutions from Github, but the main selection criterion was to complement the interaction dataset. Future work should establish additional measurements to quantify the representativeness of the dataset and extend the sample size. In addition, we have only included C# repositories, which makes it hard to compare to existing techniques which are mostly evaluated on Java code. We don't think that this leads to an unfair comparison, because it equally affects all systems. However, the languages differ in significant details and we expect that different recommenders might be useful for developers.

One of the ugly details of the interaction dataset is its potential to be misused by third parties. Our dataset license strictly forbids misusing its contents, but we cannot keep criminal-minded individuals from matching the data with other public information. For example, by matching file names and time stamps to files from public repositories to get user names or email addresses. We did not find any way to avoid this limitation other than the hashing or anonymization of sensitive information like file names. However, this would also strongly affect the usefulness of the dataset.

Over the last years, we have been spending a lot of effort on continuously advertising our research and the field study. This lead to an increasing number of study participants and allowed us to build a dataset of recorded interaction data with a respectable size. This dataset enables investigations of research questions at a much larger scale than typical experiments that are conducted with a low number of students or research assistants. However, the dataset is not yet *big* as in big data. While the number of projects included in the dataset of released code is easy to increase, we still need to find better ways to engage more developers in our field study.

The interaction dataset is currently organized by participant, which makes it necessary to identify subsequent uploads and merge them. The current strategy to merge uploads leaves room for improvement. We catch many cases, but some cannot be captured conceptually. This situation could be improved in the future with a registration system for user accounts. A clear tradeoff exists between consistency guarantees and the ease-of-use for developers though. Future work should analyze whether a lazy approach with optional registrations is feasible. While optional registrations would not solve the problem entirely, because many participants will not register, registered users would provide the ground truth to evaluate novel merging strategies that introduce heuristics to improve the current state, e.g., by considering a list of edited files. A registration would also enable the creation of additional services, e.g., a web-dashboard similar to CODE-A-LIKE[22], as an incentive for registration.

In summary, we provide two datasets for the C$\boxed{A}$RET platform that contain captured interaction data and released source code. The datasets complement each other and have enabled research in several directions, including answering research questions that could not be addressed before. The datasets remove the burden from the researcher to setup a proper environment first and makes it easy to get started. The size and the representativeness of the compiled data should be further improved in future work. However, our statistics show that their size is larger and more representative than many empirical studies we found in related work. In addition, our experiments have proven that it is possible to use them in experiments.

Chapter Summary

This thesis has discussed the value and the shortcomings of the C$\boxed{A}$RET platform, which consists of several connected pieces that work smoothly together. In this chapter, we have identified and discussed minor limitations in in all pieces that can be improved in the future. The discussion has shown, however, that the orchestration of all parts enables novel research on developers and RSSE that was not possible before. This thesis provides proof that the platform is usable and the cooperation with our students that successfully used C$\boxed{A}$RET in their own research projects has shown that it is usable by developers other than the creators.

14 Outlook

This thesis has presented exciting results of technical research in the previous part. Instead of closing a line of research, many opportunities for future work have popped up, some easier to achieve than others. In this chapter, we will present an outlook into future work that we anticipate and we we will divide it into implementation and research. The implementation part covers future work on the infrastructure and the technical foundation. The research part covers research that is either enabled by the infrastructure or that presents interesting new directions into which the whole platform could evolve.

14.1 Implementation

This thesis had an extensive implementation part through the creation of the interaction tracker FeedBaG++, the C$\boxed{A}$RET platform, and their corresponding data structures. While their current implementation is very stable, we see potential for several extensions that could be interesting for future works.

Improve Source-Code Representation The SST representation has proven its usefulness for the use case that has been discussed in this thesis. The big idea that added the most value was to embed fully-qualified type information in the source-code representation. While the decision to normalize source code helped us as well to write static analyses, it makes other tasks harder like studying source-code evolution.

One of the lessons learned of our work was that these two ideas should be separated in the future. Future work should find a source-code representation that is closer to the original source code, which also contains invalid parts of an AST, while still including fully-qualified references to types and type elements. The normalization is advantageous too and we want to keep it. However, we imagine that splitting out the SST creation into a separate transformation that is available on top of the future representation might be more beneficial. Another option would be to keep the current transformation and provide a de-normalization component that converts SSTs back into the imaginary future representation.

More Instrumentations So far, the development of FeedBaG++ and C$\boxed{A}$RET was driven by the desire to replicate existing works, but the results of this thesis now provide an excellent opportunity to further extend the implementation with instrumentations that cover other specialized context-information types. Examples would be capturing more details about in-IDE events (e.g., more debugging details) or additional events that happen outside of the IDE (e.g., an instrumentation of the browser). This opens up the excellent chance to pursue future research questions as the driver for future extensions to our extensible platform. However, every extension always comes at the cost of privacy and might also repel participants. Future work should study the effect on privacy and propose a more fine-grained way to control the captured data.

Provide Additional Default Components The C$\boxed{\text{A}}$RET platform caters for the needs of RSSE and supports basic analyses through reusable analysis components. So far, only modules are available that were needed in our own experiments. The platform facilitates the creation of additional components that can be reused by other researchers and we see a big chance to make it valuable for research in this area. Examples of such components are additional static analyses (e.g., escape analysis for objects), transformations of source code (e.g., slicing transformation or alpha conversion of variable names), or recurring infrastructural tasks (e.g., more data cleanup and noise reduction). The great benefit of the platform is that it creates the potential to implement these components once, whenever they are needed for the first time, and then integrate them into the platform for others to reuse.

Tool Platform We provide several in-IDE services that simplify the access to activities and the edited source code of the developer. In contrast to hooking into the corresponding parts of the IDE themselves, researchers can reduce their implementation effort by reusing our platform instead. We already provide tooling to clean the captured interaction data, e.g., by removing duplicate events, and to simplify the access to the edited source code to simplify static analyses on top of it, e.g., through granting access to the SST representation of the source code under edit.

These services provide a great chance for reuse when implementing novel software development tools and having such a platform is key to enhance the current state of the art. However, FeedBaG++ is the only consumer of many of these services right now. Future work could improve our service API and establish it as a standardized platform. This could lower the barrier to bring research tools into the IDE.

Continuity The biggest challenge for future work on FeedBaG++ is to keep up with the development of the IDEs. The interaction tracker has a stable implementation, but the experience of the KaVE project has shown that it is necessary to constantly update the tooling and to spend time with the maintenance of the instrumentation. Over the course of its lifetime, we already had to compensate several significant updates to the environment, namely of the ReSharper framework, that introduced breaking changes, which broke our instrumentation. The bane of a holistic interaction tracker is that it covers a great part of the framework API. Regardless of the breaking change that is introduced in an update, the tracker will likely be affected. Once active development is ceased, the interaction tracker will soon be outdated. Worse than that, it will loose most of its user base as developers move on to newer versions of ReSharper.

It would be a pity to loose an excellent opportunity to collect such realistic data in the field. However, the only encouraging detail is that even if FeedBaG++ will be outdated, the collected datasets and the C$\boxed{\text{A}}$RET platform that allows working with them will last and can still build the foundation for future research in this area.

Overall, the nature of this thesis made it necessary to spend significant implementation efforts while building the infrastructure required by our research to create the scientific contributions. The current state of the platform provides ample opportunity to conduct more future research, but it also represents a solid platform for potential future extensions. To stay relevant and to continue the ongoing data collection in the field study, it is also necessary to spend continuous maintenance efforts to keep the implementation up-to-date and compatible to ReSharper.

14.2 Research

This thesis has shown that it is possible to conduct exciting research based on the C🄰RET platform and the datasets. The presented work does not exhaust the available possibilities though. In this section we envision future research questions that are either immediate next steps that are made possible through the results of this thesis or novel research opportunities that could be seen as a continuation of this work. We will discuss both in the following.

Immediate Next Steps

The following research questions are enabled through the created data and infrastructure of this thesis and could be answered in immediate next steps.

Replications One of the core ideas of this thesis is to provide a foundation that improves comparability of research results. Therefore, we have created a benchmark for method-call recommendation systems that can be used for a fair comparison of different approaches. Most RSSE works don't share their implementations though and we only provide the four recommendation engines that were built by us. So far, the high effort required for understanding and rebuilding kept us from re-implementing other engines. An immediate next step of future work could change this and spend the engineering effort to re-build more method-call recommenders. This would serve two purposes: first, it would replicate existing works and enable a fair comparison on common ground. Second, a large-scale comparison of such systems would emphasize the importance of a common benchmark for research. The existence of such a comparison would raise the bar for the expectations towards future evaluations of method call recommenders, which ultimately leads to a higher quality of research in this area.

Method-call recommenders only represent a single use case though and future work could use our datasets to design other benchmark suites. We envision that a similar strategy can be followed in many other research areas, like different examples of support through source-based systems (e.g., code search or snippet completion) or also in the area of activity-based support (e.g., in navigation tasks).

Personas So far, all RSSE that we are aware of pursue a *one-size-fits-all* approach. They build one system that is to be used by all developers. Intuitively, we assume that different people have different information needs and require different kinds of support. This intuition is usually formalized through the identification of relevant *persona*. For example, in UI design, it is a common technique to fix target persona, e.g., *elderly person*, *power-user*, or *color-blind person*, and to reflect all design decisions with respect to each persona. Future work should study persona in the context of RSSE. Do different personas need to use completely different systems that address the differing information needs or do all users rely on the same systems that are able to distinguish usage profiles and can cater for special needs of their users?

For illustration, let's compare the information needs of a novice developer and of an expert, when using a specific API. We imagine that a novice programmer would benefit from optimistic recommendation of API elements that help him to discover yet unknown parts of an API. An optimization towards a high recall makes sure that a wanted element is always recommended. A more experienced developer, on the other hand, might be annoyed by unnecessary proposals. We assume that recall can be sacrificed for a high precision, because the developer already knows the API.

Personas have already been studied in the context of navigation models of software developers [65, 104], but we assume that they are widely applicable in the area of RSSE. Future work can build upon our platform to analyze this assumption.

Intermediate Representation We introduced SSTs as a novel intermediate representation for source code in this thesis. While the representation was a means for us to enable our experiments, we envision future work enabled through the representation.

One opportunity we see is using it in datasets that contain source code as embedded element. For example, a dataset of StackOverflow posts also contains source code that is part of a post. Previous works have introduced ways to parse these posts into heterogeneous ASTs [182] or to resolve typing information in the posts [236]. However, it is still hard to write static analysis on these snippets. A simple solution would be to combine both previous works that have focused on parsing and type resolution and store the resulting snippets in SSTs such that the same analyses and processing tools can be applied on them as on transformed source code from other sources.

Another line of future research we foresee is improving the intermediate representation itself. Very similar to the FAMIX meta model [241], it could serve as a means to design cross-language analyses or to investigate the differences between APIs and their usages in different programming languages. Having an IR as common ground opens the possibility to have a single toolchain to process and evaluate programs originally written in different programming languages. Right now, the intermediate representation can represent C# and we are currently working on an IR transformation for Java. Future work could investigate possibilities of language-independent transformations and refactorings that go beyond simple structural changes. Different semantics of the original programming languages make it necessary to define semantics for the intermediate representation and to create mappings for all supported language constructs to guarantee soundness of the transformation.

SSTs are currently only semi-self-contained. While the intermediate representation contains fully-qualified references to all types and type elements, the declarations of these referenced types are not contained in the SST snapshot. While types that are defined in a public dependency can be reconstructed later, e.g., by establishing a type table that allows lookups (see Section 4.3), local references cannot be reconstructed. One possible extension to SSTs is to extend the representation for the snapshot to include *TypeShapes* of all (transitively) referenced types, similar to what we already capture for the type hierarchy of the current type. This technique would further increase the redundant information that is stored in the snapshots, because the information would be included in multiple snapshots, even though might not have changed. However, capturing all information would create a consistent slice of the type system that contains all relevant information for the current snapshot.

Improving the Datasets We have spent major effort for this thesis to compile two datasets of developer interactions and of released source code. Now that the data representations and the required tooling is available, we envision that these datasets can be extended even further in future work. We see three opportunities to do so.

Vertical Extension Future work can extend the sizes of the datasets by analyzing additional repositories and by including more users in the field study. Adding repositories is straightforward, as we only need to identify more repositories and run our transformation on them. To acquire more users, it is necessary to increase our advertisement efforts or to provide value to the users. We plan to do both steps ourselves, but also encourage others to create their own datasets of source code and interactions using FeedBaG++ and C$\boxed{A}$RET and to share them.

Horizontal Extension A second option is to extend the kind of data that is captured to get a more precise picture of the development process, e.g., by including additional source-code elements or more context information for interactions.

For this work, we focused on representing source code and development activities to provide fundamental data for research on RSSE. However, it could also be possible to complement the captured source code with other datasets like commit messages or bug reports. We believe that we can design representations for other inputs in the same way we designed SSTs for source code: Survey RSSE that use the respective type of input, identify specific input requirements, design an abstract representation for the input, and provide a converter to that representation. Afterwards, reusable transformations and analyses can be built on top of the abstract representation, like we built C$\boxed{A}$RET's modules for method-call completion systems.

Representativity Empirical studies require on large input datasets, but the significance of the results often depends on the quality of the datasets. We envision an extension to the datasets that focuses on the improvement of their representativeness. Nagappan et al. [160] described an algorithm to determine the representativeness of experiments and we have already sketched an algorithm to achieve more representative experiments [187]. We envision that additional meta-data about the projects or further demographic information about the FeedBaG++ users can be used to cluster participants or projects into *families*. The goal is to cover as much variety of the input as possible and to avoid a bias towards a single family.

Overall, we are convinced that providing good datasets that can be mined by the research community is a key aspect of research on software engineering. We strongly believe that future work can build upon the current state of the datasets and that various opportunities exist to make them even more valuable by either increasing their size, their precision, or the quality of the datasets.

Novel Research Opportunities

In addition to the immediate steps, we have identified several novel research opportunities as a direct continuation of the work presented in this thesis.

Cross-cutting Research The biggest opportunity that is opened by this thesis is that it became easy to answer research questions that cross the boundaries of a single type of input. Our datasets provide the unique opportunity to connect source changes with developer activities, because are perfectly aligned in enriched event streams. One example of such an alignment is the combination of captured edits of testing related source code changes with test run information. This allows us to study several interesting test related activities like test refactoring, test selection, and also test writing activities of developers. Another example of interesting research is to revisit studies on navigation models for developers. With our dataset, we can differentiate read-only navigation from navigation that resulted in edit operations, we capture how long developers read a specific file or if files are opened through shortcut or by clicking at them manually. This allows studying navigation events while connecting them to activities like testing, debugging, and editing. The dataset is not tailored to this kind of research and contains very general information. Many more opportunities for research might exist that we have not thought about so far.

Re-Enable Incentives in FeedBaG The FEEDBAG++ interaction tracker contains several incentives that we planned to include in the release as a means to increase our user base through provided value and advertisement. However, we decided to disable all these incentives in our field study that created the interaction dataset. This avoids uncontrolled bias in the data and allows using it as ground truth in evaluations.

We envision that future work gets back to this question and reconsiders the decision, as enabling these incentives would attract a large number of participants for the field study. A significantly increased number of participants goes together with a much higher representativeness of the collected data and would certainly justify a tradeoff to some degree. A follow-up experiment could re-enable the incentives to get more users and test how different the biased dataset really is, compared to the unbiased one. Related work has reported that novelty effects wear off fast [30, 230] and we are expecting that the effect is negligible for long time participants. It could be studied, if the effect can be isolated and affected parts removed, maybe by ignoring data collected within the first week of participation of a developer.

Use of (Intelligent) Code Completion Our study on the usage of VISUAL STUDIO by professional software developers has confirmed the results of previous studies that found that code completion is one of the most important in-IDE tools. While we know now *that* it is used, we still don't know *how* developers are using it.

Previous work has already tried to answer this question from two different perspectives. For example, by conducting qualitative surveys of existing recommender systems [252] or through empirical work [129] with focus on understanding the usage intentions and reflecting on the actual behavior of developers. While in the former the conclusions are only based on the reviewed tools and not on actual developers, the latter presents results from a study of 6 developers and is based on a very small dataset of 192 code completion usages. Our dataset of developer interactions is several orders of magnitude large and contains almost 200,000 interactions with the code completion. This allows to re-visit this research question to replicate these results.

An additional line of research that opens up through the results of this thesis is to study the effect of intelligent code completion on developers. The news regularly contain stories about people who drove their car into lakes, because their navigation system told them so. Future work could analyze whether this blind trust towards recommender systems also exists among software developers through enabling our intelligent code completion system. This is possible by adding artificial proposals to the code completion that are completely bogus and by analyzing whether these proposals are actually being selected.

Other research questions that could be answered with respect to the code completion usage is whether an intelligent system changes the way code completion is used, empirically analyze the best number of proposals shown to developers, or comparing different code completion strategies in the field instead of in a benchmark.

Crowd-Sourcing Empirical Studies This thesis has introduced FEEDBAG++, a powerful interaction tracker that can capture activity information from developers. We have used the tracker ourselves in a large field study and FEEDBAG++ can also be used in controlled experiments to capture interactions in a fine granularity. In addition, we envision several scenarios in which the tool might be useful for future work.

Future work could analyze how conducting empirical studies that involve an IDE could be better supported through more appropriate infrastructure. Especially the setup of the environment represents a large burden for the experimenter right now. Following previous work of Van Gorp et al. [246], we have conducted preliminary experiments[1] to study whether it is possible to provide such a general infrastructure for empirical study that can be reused to reduce the overhead when conducting experiments. Our results show that it is possible to provide a centralized and scalable infrastructure that significantly reduces the setup overhead for empirical studies. We have proven that we could replicate the data collection of our field study in a controlled experimental setting.

We see two steps in which future work can extend these preliminary results. First, future work could use the infrastructure to run a controlled experiment to get a controlled dataset. We'd like to compare this dataset to the data collected in the uncontrolled field study and analyze their commonalities and differences.

Second, if this idea is further extended, it might be possible to run such an infrastructure in the cloud and conduct future empirical studies in less controlled environments like on-demand workforce services like AMAZON MECHANICAL TURK.[44] Future work should study the tradeoff between sacrificing the total control over study participants and the representativeness gain enabled by much higher numbers of study participants. Previous work of Keimel et al. [101] has shown that such a setting is possible when testing video quality, but it is still unclear, if the conclusion also holds in different domains or experimental settings.

Versioning 2.0 An established line of research studies the history of source code. Many works rely on extracting changes from revisions contained in large repositories of open-source projects (e.g., [66]). Others collect a fine-grained history during development. Both approaches have disadvantages. While the history captured in repositories is very coarse grained, the history captured in the IDE is much more fine-grainer and stored in the granularity of saves. However, it is typically studied on a textual level and does not consider typing information.

The C$\boxed{A}$RET interaction dataset also contains a fine-grained change history of source code. We capture changes on the finest granularity of edits, as well as interaction events that mark saves and commits. We envision several strategies for grouping these fine-grained changes into coherent change sets. For example, time-based (e.g., per day, per hour, fixed-size window, timeout-based) or activity-based (e.g., on edit, on save, on commit). An intriguing idea for future work is to learn a model that can identify abstract task boundaries in enriched event stream. These boundaries would represent a very interesting scheme to group coherent changes and would provide excellent meta-data for other approaches that analyze the events.

We envision that future work analyzes the effect of different granularities on clients that are based on historic information (e.g., change patterns [167], commit summarization [207], or untangling of changes [45]). Systems like the EVOLIZER [66] tool suite infer semantic changes from the program history. Our datasets provides an excellent

[1] These experiments have been conducted by Can Pekesen [176] as part of his Bachelor thesis.

opportunity to extend its representation to make it easy to access the captured program history on various stability levels. Untangling changes and providing coherent changesets seems to be a crucial preprocessing step for such an application.

Framework Evolution and Migration Developers use APIs of frameworks and libraries on a daily basis. New versions of APIs often extend their functionality or fix bugs. However, updates often introduce incompatible, breaking changes to the APIs that break their clients. If developers update a dependency to use a new functionality or change it to an alternative implementation, they have to adapt the API usage.

The naming scheme for types that is used in Simplified Syntax Trees preserves information about the originating assembly of a type. This makes it very easy to extract information about changes from the program history that are specific to a given framework. We envision future work that uses this information to build tools that support developers with adapting source code to breaking changes in APIs or with migrating existing source code to an alternative API implementation (e.g., changing a specific library to a more performant alternative).

Chapter Summary

In summary, this outlook has shown that this thesis opens several new interesting questions for future work. There is ample opportunity for work both on the technical side to further extend and improve the infrastructure, but also on the scientific side to answer several immediate research questions or use the platform for a continuation of the work in related areas.

15 Summary

This thesis has the goal to pave the road towards a general platform for in-IDE experiments that facilitates studies on the development process. Previous work on developer behavior either studies the artifacts of the development process after the fact or observes developers while using an IDE. In contrast to that, we propose to better align process information with fine-grained context information, like source code snapshots, to combine the advantages of both and to facilitate future research.

To achieve this, we have created the meta models *enriched event stream* and *simplified syntax trees* after surveying the requirements from related work. We build infrastructure around these models that consists of several strongly connected components: bindings for C# and JAVA, the interaction tracker FEEDBAG++ that collects interaction data in VISUAL STUDIO, and the C$\boxed{A}$RET platform that provides composable analyses and reusable transformations around these models in JAVA. It is infeasible to evaluate such a platform as a whole, so instead, we have reported on our own experience using it. To this end, we have conducted a large field-study to collect a dataset of developer activities from a significant number of participants and have complemented it with a dataset of released source code. These datasets were used in our own technical research to prove the usefulness of C$\boxed{A}$RET. Some of the addressed research questions have been enabled through the platform, other experiments replicated existing works as a proof of concept. We have already extensively discussed our experience and while we found opportunities for improvement in future work, we found the platform to be a solid foundation for research.

To assess our contributions on a more qualitative level, we will reflect on the initially formulated problem statements of this thesis. It becomes immediately obvious that our contributions make it possible to revise all of them into "solution statements". Our two meta models build the foundation of this thesis. They are not specialized on a certain use case and capture a wide range of information. Their *General Data Schema* enables various studies on the development process. We have built the C$\boxed{A}$RET platform around these models, which makes it *Easy to Get Started.* Researchers can use it to conduct experiments on our infrastructure, which significantly *Reduces Cost* and allows to focus on the actual experimental design. At the same time, a *Very Low Risk* is involved, because little time has to be invested to decide on the applicability in a specific use case. In addition to the infrastructure, we also provide datasets that further add to these benefits. They are not only conveniently usable in C$\boxed{A}$RET, due to their significant sizes, it is also safe to assume that experiments will have *Representative Results.* Additionally, the public availability and extensibility of our complete tooling make experiments on our infrastructure *Easy to Reproduce* and allow to collect additional datasets, which enables an *Easy Replication* of our experiments in a different environment or on different data.

The second focus of this work has been static analyses of source code. Our meta model for source code to enable the creation of shared tools and to facilitate the creation of static analysis. Its source-code representation is self-contained so *No Project Setup Is Necessary* to work with the source code, which makes it *Easy to Get Started* in a completely different use case. The meta model is designed to be used for both released source code and for source code snapshots taken in-IDE. Our implemented transformation *Avoids Invalid Source Code by Design* by ignoring unparseable or ill-typed parts and all shared tools build on it therefore automatically support working with *Incomplete Source Code.* We could also prove the applicability of the meta model in various experiments, showing once again the *Generality of the Data Schema.*

Overall, we think that the work presented in this thesis paves the road for novel studies. Using our datasets, we see a unique opportunity to study the development process. Our platform has the potential to be of great value to the whole community, because it has not been possible before to study the combination of both data sources with such a good alignment. Creating the platform is only the first step though, but maturing it is an endeavor that can only be achieved through joined forces. Everybody is encouraged to use and extend our tools and datasets. We strongly believe that we took the first step towards more standardized, comparable, and reproducible experiments on software engineering. While we have used C$\boxed{\text{A}}$RET to create exciting results, we strongly believe that our experiments have only scratched the surface of the potential hidden in the datasets. We expect that future work will use them to answer questions that go far beyond the experiments presented in this thesis.

Links

All links verified on May 31, 2017.

[1] Creative Commons License, `https://creativecommons.org/licenses/by-sa/4.0/`

[2] Opal, `http://www.opal-project.de/`

[3] ReSharper, `https://www.jetbrains.com/resharper/`

[4] ReSharper SDK, `https://www.jetbrains.com/help/resharper/sdk/`

[5] Soot, `https://sable.github.io/soot/`

[6] Wala, `http://wala.sourceforge.net/`

[7] FeedBaG++, `https://resharper-plugins.jetbrains.com/packages/KaVE.Project/`

[8] ReSharper Gallery, Download Statistics over the past 6 weeks, `https://resharper-plugins.jetbrains.com/stats/packages`

[9] KaVE Project Website, `http://www.kave.cc/`

[10] KaVE Datasets, `http://www.kave.cc/datasets/`

[11] KaVE List of Supported Event Types, `http://www.kave.cc/feedbag/event-generation`

[12] Tutorials: How to Use C$\boxed{\text{A}}$RET, `https://github.com/stg-tud/kave-java/tree/master/exec/exec.examples/src/main/java/exec/examples`

[13] Eclipse Usage Data Collector, `http://www.eclipse.org/epp/usagedata/`

[14] Eclipse Usage Data Collector - Aggregated Archive, `http://archive.eclipse.org/projects/usagedata/`

[15] Blaze Data, `https://abb-iss.github.io/DeveloperInteractionLogs/`

[16] FAMIX Meta model of source code, `http://www.themoosebook.org/book/internals/famix`

[17] srcML, `https://github.com/srcML`

[18] srcML Stereocode, `https://github.com/srcML/stereocode#stereotypes`

[19] OpenHub, `https://www.openhub.net/`

[20] Wikipedia: "Privacy by Design" (does not fully match "Datensparsamkeit"[21]), `https://en.wikipedia.org/wiki/Privacy_by_design`

[21] Wikipedia: "Datensparsamkeit" (German), `https://de.wikipedia.org/wiki/Datenvermeidung_und_Datensparsamkeit`

[22] CodeAlike, `https://codealike.com/`

[23] JUnit, `http://junit.org/`

[24] StackOverflow data dumps, https://archive.org/details/stackexchange

[25] ANTLR - Parser Generator Framework, http://www.antlr.org/

[26] NCrunch, http://www.ncrunch.net/

[27] Understand the Pitfalls of Benchmarking Java Code,
http://www.ibm.com/developerworks/java/library/j-benchmark1/

[28] TIOBE Index, http://www.tiobe.com/tiobe-index/

[29] Open Hub, https://www.openhub.net/

[30] TensorFlow, https://www.tensorflow.org/

[31] Apache Hadoop, http://hadoop.apache.org/

[32] Apache Mahout, http://mahout.apache.org/

[33] Json.NET, http://www.json.net

[34] GSON, https://github.com/google/gson

[35] Google Protocol Buffers, https://developers.google.com/protocol-buffers/

[36] Power BI, https://powerbi.microsoft.com/

[37] R Project, https://www.r-project.org/

[38] Software Engineering Open Science Portal, http://openscience.us/

[39] Grimoire Lab, http://grimoirelab.github.io/

[40] SWT: The Standard Widget Toolkit, http://www.eclipse.org/swt/

[41] Eclipse Update Site, Kepler Release,
http://download.eclipse.org/releases/kepler/

[42] MSDN Security Design Guidelines for Web Services,
https://msdn.microsoft.com/en-us/library/ff649737.aspx

[43] "Issue 32", http://gousios.gr/blog/Issue-thirty-two/

[44] Amazon Mechanical Turk, https://www.mturk.com/

[45] Goggle Trend Analysis of Top IDEs, http://pypl.github.io/IDE.html

[46] Simian: Similarity Analyser, http://www.harukizaemon.com/simian/

Bibliography

[1] M. Allamanis and C. Sutton. Mining Idioms from Source Code. In *International Symposium on Foundations of Software Engineering*. ACM, 2014.

[2] S. Amann, S. Proksch, and M. Mezini. Method-call Recommendations from Implicit Developer Feedback. In *International Workshop on CrowdSourcing in Software Engineering*. ACM, 2014.

[3] S. Amann, S. Proksch, and S. Nadi. FeedBaG: An Interaction Tracker for Visual Studio. In *International Conference on Program Comprehension*. IEEE, 2016.

[4] S. Amann, S. Proksch, S. Nadi, and M. Mezini. A Study of Visual Studio Usage in Practice. In *International Conference on Software Analysis, Evolution, and Reengineering*. IEEE, 2016.

[5] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and D. Hou. CSCC: Simple, Efficient, Context Sensitive Code Completion. In *ICSME*, 2014.

[6] N. Ayewah, D. Hovemeyer, J. Morgenthaler, J. Penix, and W. Pugh. Using Static Analysis to Find Bugs. *IEEE Software*, 25(5):22–29, 2008.

[7] G. J. Badros. JavaML: A Markup Language for Java Source Code. In *International World Wide Web Conference on Computer Networks : The International Journal of Computer and Telecommunications Networking*, 2000.

[8] S. Bajracharya, J. Ossher, and C. Lopes. Sourcerer: An Internet-scale Software Repository. In *International Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, 2009.

[9] S. Bajracharya, J. Ossher, and C. Lopes. Searching API Usage Examples in Code Repositories with Sourcerer API Search. In *Workshop on Search-driven Development: Users, Infrastructure, Tools and Evaluation*, 2010.

[10] B. Basten, M. Hills, P. Klint, D. Landman, A. Shahi, M. J. Steindorfer, and J. J. Vinju. M3: A general model for code analytics in rascal. In *International Workshop on Software Analytics*. IEEE, 2015.

[11] J. Beel, M. Genzmehr, S. Langer, A. Nürnberger, and B. Gipp. A Comparative Analysis of Offline and Online Evaluations and Discussion of Research Paper Recommender System Evaluation. In *International Workshop on Reproducibility and Replication in Recommender Systems Evaluation*, 2013.

[12] J. Beel, B. Gipp, S. Langer, and C. Breitinger. Research-paper recommender systems: A literature survey. *International Journal on Digital Libraries*, 17(4):305–338, 2016.

[13] M. Beller, G. Gousios, A. Panichella, and A. Zaidman. When, How, and Why Developers (Do Not) Test in Their IDEs. In *Joint Meeting on Foundations of Software Engineering*, 2015.

[14] M. Beller, N. Spruit, and A. Zaidman. How Developers Debug. *PeerJ Preprints*, 5:e2743v1, 2017.

[15] M. Beller, A. Zaidman, and A. Karpov. The Last Line Effect. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, pages 240–243. IEEE Press, 2015.

[16] J. Bevan, E. J. Whitehead, Jr., S. Kim, and M. Godfrey. Facilitating Software Evolution Research with Kenyon. In *International Symposium on Foundations of Software Engineering*, pages 177–186. ACM, 2005.

[17] H. Beyer and K. Holtzblatt. *Contextual Design: A Customer-Centered Approach to Systems Designs*. Morgan Kaufmann, 1997.

[18] W. C. Booth, G. G. Colomb, and J. M. Williams. *The craft of research*. University of Chicago Press, 2003.

[19] A. Bragdon, R. Zeleznik, S. P. Reiss, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola Jr. Code bubbles: a working set-based interface for code understanding and maintenance. In *Conference on Human Factors in Computing Systems*, pages 2503–2512. ACM, 2010.

[20] F. P. Brooks, Jr. *The Mythical Man-month (Anniversary Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[21] P. F. Brown, P. V. Desouza, R. L. Mercer, V. J. D. Pietra, and J. C. Lai. Class-based N-gram Models of Natural Language. *Computational Linguistics*, 1992.

[22] M. Bruch, M. Monperrus, and M. Mezini. Learning from Examples to Improve Code Completion Systems. In *International Symposium on the Foundations of Software Engineering*. ACM, 2009.

[23] M. Böhme, E. O. Soremekun, S. Chattopadhyay, E. J. Ugherughe, and A. Zeller. Where is the Bug and How is it Fixed? An Experiment with Practitioners. In *International Symposium on Foundations of Software Engineering*, 2017.

[24] J. M. Carroll. The Adventure of Getting to Know a Computer. *Computer*, 15(11):49–58, Nov. 1982.

[25] J. M. Carroll. *Scenario-based Design: Envisioning Work and Technology in System Development*. John Wiley and Sons, 1995.

[26] E. Cergani, S. Proksch, S. Nadi, and M. Mezini. Addressing Scalability in API Method Call Analytics. In *International Workshop on Software Analytics*. ACM, 2016.

[27] O. Chapelle and Y. Zhang. A Dynamic Bayesian Network Click Model for Web Search Ranking. In *International Conference on World Wide Web*, 2009.

[28] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change Detection in Hierarchically Structured Information. In *ACM SIGMOD Record*, 1996.

[29] S. F. Chen and J. Goodman. An empirical study of smoothing techniques for language modeling. In *Annual Meeting on Association for Computational Linguistics*, 1996.

[30] R. Clark and B. Sugrue. *Research on Instructional Media, 1978-1988*, pages 327–343. Instructional Technology: Past, Present, and Future. Libraries Unlimited, 1991.

[31] M. L. Collard, M. J. Decker, and J. I. Maletic. srcML: An Infrastructure for the Exploration, Analysis, and Manipulation of Source Code: A Tool Demonstration. In *International Conference on Software Maintenance*. IEEE, 2013.

[32] M. Conklin, J. Howison, and K. Crowston. Collaboration Using OSSmole: A Repository of FLOSS Data and Analyses. In *International Workshop on Mining Software Repositories*. ACM, 2005.

[33] J. Coplien and K. Beck. The Column Without a Name: After All, We Can't Ignore Efficiency - part 2. C++ Report, July 1996.

[34] C. S. Corley, F. Lois, and S. Quezada. Web Usage Patterns of Developers. In *International Conference on Software Maintenance and Evolution*. IEEE, 2015.

[35] T. Cover and P. Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1):21–27, 2006.

[36] D. Cubranic and G. C. Murphy. Hipikat: Recommending Pertinent Software Development Artifacts. In *International Conference on Software Engineering.* IEEE, 2003.

[37] B. Dagenais and L. Hendren. Enabling Static Analysis for Partial Java Programs. In *Conference on Object-oriented Programming Systems Languages and Applications.* ACM, 2008.

[38] V. Dallmeier and T. Zimmermann. Extraction of Bug Localization Benchmarks from History. In *International Conference on Automated Software Engineering*, 2007.

[39] K. Damevski, D. Shepherd, J. Schneider, and L. Pollock. Mining Sequences of Developer Interactions in Visual Studio for Usage Smells. *IEEE Transactions on Software Engineering*, 2016.

[40] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Symposium on Operating System Design and Implementation*, 2004.

[41] R. DeLine, M. Czerwinski, and G. Robertson. Easing program comprehension by sharing navigation data. In *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*, pages 241–248. IEEE, 2005.

[42] Blizzard. Diablo 3, 2012.

[43] S. Deterding, D. Dixon, R. Khaled, and L. Nacke. From game design elements to gamefulness: Defining "gamification". In *Proceedings of the 15th International Academic MindTrek Conference: Envisioning Future Media Environments*, MindTrek '11, pages 9–15, New York, NY, USA, 2011. ACM.

[44] M. Dias. *Supporting Software Integration Activities with First-Class Code Changes.* PhD thesis, Laboratoire d'Informatique Fondamentale de Lille, 2015.

[45] M. Dias, A. Bacchelli, G. Gousios, D. Cassou, and S. Ducasse. Untangling Fine-Grained Code Changes. In *International Conference on Software Analysis, Evolution, and Reengineering.* IEEE, 2015.

[46] M. Dias, D. Cassou, and S. Ducasse. Representing Code History with Development Environment Events. In *International Workshop on Smalltalk Technologies*, 2013.

[47] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automated detection of refactorings in evolving components. In *European Conference on Object-Oriented Programming*, pages 404–428. Springer, 2006.

[48] D. Dig, K. Manzoor, R. E. Johnson, and T. N. Nguyen. Effective software merging in the presence of object-oriented refactorings. *IEEE Transactions on Software Engineering*, 34(3):321–335, 2008.

[49] J. P. Djajadiningrat, W. W. Gaver, and J. Fres. Interaction Relabelling and Extreme Characters: Methods for Exploring Aesthetic Interactions. In *Conference on Designing Interactive Systems: Processes, Practices, Methods, and Techniques*, 2000.

[50] D. Draheim and L. Pekacki. Process-Centric Analytical Processing of Version Control Data. In *International Workshop on Principles of Software Evolution*, 2003.

[51] S. Ducasse, M. Lanza, and S. Tichelaar. Moose: an extensible language-independent environment for reengineering object-oriented systems. In *International Symposium on Constructing Software Engineering Tools*, volume 4, 2000.

[52] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen. Boa: A Language and Infrastructure for Analyzing Ultra-large-scale Software Repositories. In *International Conference on Software Engineering.* IEEE, 2013.

[53] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen. Boa: Ultra-Large-Scale Software Repository and Source-Code Mining. *Transactions on Software Engineering and Methodology*, 2015.

[54] R. Dyer, H. Rajan, H. A. Nguyen, and T. N. Nguyen. Mining Billions of AST Nodes to Study Actual and Potential Usage of Java Language Features. In *International Conference on Software Engineering*. ACM, 2014.

[55] P. Ebraert, J. Vallejos, P. Costanza, E. Van Paesschen, and T. D'Hondt. Change-oriented software engineering. In *International Conference on Dynamic Languages*, 2007.

[56] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus. Fine-Grained and Accurate Source Code Differencing. In *International Conference on Automated Software Engineering*, 2014.

[57] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *International Conference on Software Maintenance*, 2003.

[58] Z. Fitz-Walter, D. Tjondronegoro, and P. Wyeth. Orientation Passport: Using Gamification to Engage University Students. In *Proceedings of the 23rd Australian computer-human interaction conference*, pages 122–125. ACM, 2011.

[59] B. Fluri, M. Wuersch, M. Pinzger, and H. Gall. Change Distilling: Tree Differencing for Fine-grained Source Code Change Extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, 2007.

[60] B. Fluri, M. Wursch, and H. C. Gall. Do code and comments co-evolve? On the relation between source code and comment changes. In *Working Conference on Reverse Engineering*. IEEE, 2007.

[61] B. Fluri, J. Zuberbühler, and H. C. Gall. Recommending Method Invocation Context Changes. In *International Workshop on Recommendation Systems for Software Engineering*. ACM, 2008.

[62] M. E. Fonteyn, B. Kuipers, and S. J. Grobe. A description of think aloud method and protocol analysis. *Qualitative Health Research*, 3(4):430–441, 1993.

[63] S. R. Foster, W. G. Griswold, and S. Lerner. WitchDoctor: IDE support for real-time auto-completion of refactorings. In *International Conference on Software Engineering*, pages 222–232. IEEE Press, 2012.

[64] C. Franks, Z. Tu, P. Devanbu, and V. Hellendoorn. CACHECA: A Cache Language Model Based Code Suggestion Tool. In *International Conference on Software Engineering*. IEEE, 2015.

[65] T. Fritz, D. C. Sheperd, K. Kevic, W. Snipes, and C. Braeunlich. Developers' code context models for change tasks. In *International Symposium on the Foundations of Software Engineering*, 2014.

[66] H. C. Gall, B. Fluri, and M. Pinzger. Change Analysis with Evolizer and ChangeDistiller. *IEEE Software*, 2009.

[67] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley, 1995.

[68] X. Ge, Q. L. DuBose, and E. Murphy-Hill. Reconciling manual and automatic refactoring. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 211–221. IEEE, 2012.

[69] D. M. German. Mining CVS Repositories, the SoftChange Experience. *Evolution*, 245(5,402):92–688, 2004.

[70] G. Ghezzi and H. C. Gall. *Distributed and Collaborative Software Analysis*, pages 241–263. Springer, Heidelberg, Germany, 2010.

[71] T. Gîrba and S. Ducasse. Modeling History to Analyze Software Evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(3):207–236, 2006.

[72] M. W. Godfrey and L. Zou. Using Origin Analysis to Detect Merging and Splitting of Source Code Entities. *Transactions on Software Engineering*, IEEE, 2005.

[73] V. U. Gómez, S. Ducasse, and T. D'Hondt. Ring: a unifying meta-model and infrastructure for Smalltalk source code analysis tools. *Computer Languages, Systems & Structures*, 38(1):44–60, 2012.

[74] V. M. González and G. Mark. "Constant, Constant, Multi-tasking Craziness": Managing Multiple Working Spheres. In *Proceedings of the Conference on Human Factors in Computing Systems*, CHI '04, pages 113–120. ACM, 2004.

[75] J. M. González-Barahona and G. Robles. On the Reproducibility of Empirical Software Engineering Studies Based on Data Retrieved from Development Repositories. *Empirical Software Engineering*, 2012.

[76] G. Gousios. The ghtorrent dataset and tool suite. In *Working Conference on Mining Software Repositories*, MSR '13, pages 233–236, Piscataway, NJ, USA, 2013. IEEE Press.

[77] G. Gousios and D. Spinellis. A Platform for Software Engineering Research. In *International Working Conference on Mining Software Repositories*, 2009.

[78] G. Gousios and D. Spinellis. Alitheia Core: An Extensible Software Quality Monitoring Platform. In *International Conference on Software Engineering*, 2009.

[79] T. Gvero, V. Kuncak, I. Kuraj, and R. Piskac. Complete Completion Using Types and Weights. In *Proceedings of Conference on Programming Language Design and Implementation (PLDI)*, 2013.

[80] A. E. Hassan. The road ahead for mining software repositories. In *Frontiers of Software Maintenance (FoSM '08)*, pages 48–57. IEEE, 2008.

[81] A. E. Hassan and R. C. Holt. Replaying Development History to Assess the Effectiveness of Change Propagation Tools. *Empirical Software Engineering*, 11(3):335–367, 2006.

[82] H. Hata, O. Mizuno, and T. Kikuno. Historage: Fine-grained Version Control System for Java. In *International Workshop on Principles of Software Evolution and Annual Workshop on Software Evolution*. ACM, 2011.

[83] L. Heinemann, V. Bauer, M. Herrmannsdoerfer, and B. Hummel. Identifier-based Context-dependent API Method Recommendation. In *European Conference on Software Maintenance and Reengineering*. IEEE, 2012.

[84] H. Hemmati, S. Nadi, O. Baysal, O. Kononenko, W. Wang, R. Holmes, and M. W. Godfrey. The MSR Cookbook: Mining a Decade of Research. In *Working Conference on Mining Software Repositories*. IEEE Press, 2013.

[85] I. Herraiz, D. Izquierdo-Cortazar, F. Rivas-Hernandez, J. Gonzalez-Barahona, G. Robles, S. Duenas-Dominguez, C. Garcia-Campos, J. Gato, and L. Tovar. FLOSSMetrics: Free/Libre/Open Source Software Metrics. In *European Conference on Software Maintenance and Reengineering*, March 2009.

[86] K. Herzig, S. Just, and A. Zeller. The Impact of Tangled Code Changes on Defect Prediction Models. *Empirical Software Engineering*, 21(2):303–336, 2016.

[87] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the Naturalness of Software. In *International Conference on Software Engineering*, 2012.

[88] R. Holmes and G. C. Murphy. Using Structural Context to Recommend Source Code Examples. In *Proceedings of International Conference on Software Engineering (ICSE)*, 2005.

[89] R. Holmes and R. J. Walker. Customized Awareness: Recommending Relevant External Change Events. In *International Conference on Software Engineering*, pages 465–474. ACM, 2010.

[90] R. Holmes, R. J. Walker, and G. C. Murphy. Approximate Structural Context Matching: An Approach to Recommend Relevant Examples. *Transactions on Software Engineering*, 2006.

[91] A. Hora, A. Etien, N. Anquetil, S. Ducasse, and M. T. Valente. APIEvolutionMiner: Keeping API Evolution Under Control. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*, pages 420–424. IEEE, 2014.

[92] O. Hummel, W. Janjic, and C. Atkinson. Code Conjurer: Pulling Reusable Software out of Thin Air. *Software, IEEE*, 25(5):45–52, 2008.

[93] W. S. Humphrey. *A Discipline for Software Engineering.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[94] A. Izmaylova, P. Klint, A. Shahi, and J. J. Vinju. M3: An Open Model for Measuring Code Artifacts. *CoRR*, abs/1312.1188, 2013.

[95] C. Jaspan and J. Aldrich. *Checking Framework Interactions with Relationships.* Springer, 2009.

[96] A. Jbara and D. G. Feitelson. How programmers read regular code: A controlled experiment using eye tracking. In *International Conference on Program Comprehension*, 2015.

[97] D. Jin and J. Cordy. Ontology-Based Software Analysis and Reengineering Tool Integration: The OASIS Service-Sharing Methodology. In *International Conference on Software Maintenance*, 2005.

[98] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 672–681. IEEE Press, 2013.

[99] P. M. Johnson, H. Kou, J. Agustin, C. Chan, C. Moore, J. Miglani, S. Zhen, and W. E. J. Doane. Beyond the Personal Software Process: Metrics Collection and Analysis for the Differently Disciplined. In *International Conference on Software Engineering*. IEEE Computer Society, 2003.

[100] P. M. Johnson, H. Kou, M. Paulding, Q. Zhang, A. Kagawa, and T. Yamashita. Improving Software Development Management Through Software Project Telemetry. *IEEE software*, 22(4):76–85, 2005.

[101] C. Keimel, J. Habigt, C. Horch, and K. Diepold. QualityCrowd: A Framework for Crowd-Based Quality Evaluation. In *Picture Coding Symposium*, 2012.

[102] M. Kersten and G. C. Murphy. Mylar: A Degree-of-interest Model for IDEs. In *International Conference on Aspect-oriented Software Development*, 2005.

[103] M. Kersten and G. C. Murphy. Using Task Context to Improve Programmer Productivity. In *International Symposium on Foundations of Software Engineering.* ACM, 2006.

[104] K. Kevic and T. Fritz. Towards Developer- and Task-Tailored Navigation Models. In *International Workshop on Context in Software Development*, 2014.

[105] K. Kevic, B. M. Walters, T. R. Shaffer, B. Sharif, D. C. Shepherd, and T. Fritz. Tracing Software Developers' Eyes and Interactions for Change Tasks. In *Joint Meeting on Foundations of Software Engineering*, 2015.

[106] P. Klint, T. v. d. Storm, and J. Vinju. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *International Working Conference on Source Code Analysis and Manipulation*, SCAM'09, 2009.

[107] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on software engineering*, 32(12):971–987, 2006.

[108] A. Kołakowska, A. Landowska, M. Szwoch, W. Szwoch, and M. R. Wróbel. Emotion Recognition and its Application in Software Engineering. In *International Conference on Human System Interactions.* IEEE, 2013.

[109] C. Kolassa, D. Riehle, and M. A. Salim. The empirical commit frequency distribution of open source projects. In *International Symposium on Open Collaboration*, 2013.

[110] Y. Koren, R. Bell, and C. Volinsky. Matrix Factorization Techniques for Recommender Systems. *Computer*, 2009.

[111] P. Lam, E. Bodden, O. Lhotak, and L. Hendren. The Soot Framework for Java Program Analysis: A Retrospective. In *Cetus Users and Compiler Infastructure Workshop*, CETUS'11, 2011.

[112] H. A. Landsberger. *Hawthorne Revisited: Management and the Worker, Its Critics, and Developments in Human Relations in Industry.* ERIC, 1958.

[113] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining Mental Models: A Study of Developer Work Habits. In *Proceedings of the International Conference on Software Engineering*, ICSE '06, pages 492–501. ACM, 2006.

[114] J. Laval, S. Denier, S. Ducasse, and J.-R. Falleri. Supporting simultaneous versions for software evolution assessment. *Science of Computer Programming*, 76(12):1177–1193, 2011.

[115] S. Lee and S. Kang. Clustering navigation sequences to create contexts for guiding code navigation. *Journal of Systems and Software*, 86(8):2154–2165, 2013.

[116] T. Lee, J. Nam, D. Han, S. Kim, and H. P. In. Micro Interaction Metrics for Defect Prediction. In *Symposium on Foundations of Software Engineering.* ACM, 2011.

[117] Z. Li and Y. Zhou. PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code. In *International Symposium on Foundations of Software Engineering.* ACM, 2005.

[118] E. Lippe and N. van Oosterom. Operation-based Merging. In *Symposium on Software Development Environments*, 1992.

[119] B. Liskov. Keynote address - data abstraction and hierarchy. In *Conference on Object-oriented Programming Systems, Languages and Applications.* ACM, 1987.

[120] B. Livshits and T. Zimmermann. Dynamine: finding common error patterns by mining software revision histories. In *International Symposium on Foundations of Software Engineering.* ACM, 2005.

[121] W. Maalej. Intention-Based Integration of Software Engineering Tools. *PhD Thesis, Technische Universität München, Germany*, 2010.

[122] W. Maalej and M. Ellmann. On the Similarity of Task Contexts. In *International Workshop on Context for Software Development*, 2015.

[123] W. Maalej, M. Ellmann, and R. Robbes. Using Contexts Similarity to Predict Relationships Between Tasks. *Journal of Systems and Software*, 2017.

[124] W. Maalej, T. Fritz, and R. Robbes. Collecting and Processing Interaction Data for Recommendation Systems. In *Recommendation Systems in Software Engineering*, pages 173–197. Springer, 2014.

[125] W. Maalej and A. Sahm. Assisting engineers in switching artifacts by using task semantic and interaction history. In *International Workshop on Recommendation Systems for Software Engineering.* ACM, 2010.

[126] T. W. Malone. Toward a Theory of Intrinsically Motivating Instruction. *Cognitive science*, 5(4):333–369, 1981.

[127] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid Mining: Helping to Navigate the API Jungle. In *Conference on Programming Language Design and Implementation.* ACM, 2005.

[128] D. Mandrioli and B. Meyer. Design by Contract. *Advances in Object-Oriented Software Engineering*, page 1, 1991.

[129] M. Mărăşoiu, L. Church, and A. Blackwell. An Empirical Investigation of Code Completion Usage by Professional Software Developers. In *Psychology of Programming Interest Group*, 2015.

[130] B. Martin, B. Hanington, and B. M. Hanington. *Universal Methods of Design: 100 Ways to Research Complex Problems, Develop Innovative Ideas, and Design Effective Solutions.* Rockport Publishers, 2012.

[131] R. C. Martin. *Agile Software Development: Principles, Patterns, and Practices.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.

[132] R. C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship.* Pearson Education, 2009.

[133] D. Maulsby, S. Greenberg, and R. Mander. Prototyping an intelligent agent through wizard of oz. In *Conference on Human Factors in Computing Systems.* ACM, 1993.

[134] E. Mayo. *The Social Problems of an Industrial Civilisation.* Routledge, 1949.

[135] P. W. McBurney and C. McMillan. Automatic Documentation Generation via Source Code Summarization of Method Context. In *International Conference on Program Comprehension.* ACM, 2014.

[136] A. McCallum, K. Nigam, and L. H. Ungar. Efficient Clustering of High-Dimensional Data Sets with Application to Reference Matching. In *Proceedings of the 6th International Conference on Knowledge Discovery and Data Mining*, pages 169–178, New York, NY, USA, 2000. ACM.

[137] F. Mccarey, M. Ó. Cinnéide, and N. Kushmerick. Rascal: A Recommender Agent for Agile Reuse. *Artificial Intelligence Review*, 24(3-4):253–276, 2005.

[138] R. McCarney, J. Warner, S. Iliffe, R. van Haselen, M. Griffin, and P. Fisher. The Hawthorne Effect: A Aandomised, Controlled Trial. *BMC Medical Research Methodology*, 7(1):30, 2007.

[139] K. Mens and A. Lozano. Source Code-Based Recommendation Systems. In *Recommendation Systems in Software Engineering*, pages 93–130. Springer, 2014.

[140] A. N. Meyer, T. Fritz, G. C. Murphy, and T. Zimmermann. Software Developers' Perceptions of Productivity. In *International Symposium on Foundations of Software Engineering*, pages 19–29. ACM, 2014.

[141] A. N. Meyer, T. Fritz, G. C. Murphy, and T. Zimmermann. Software Developers' Perceptions of Productivity. In *Proceedings of the International Symposium on Foundations of Software Engineering*, FSE '14, pages 19–29. ACM, 2014.

[142] B. Meyer. *Object-oriented Software Construction.* Prentice Hall, 1988.

[143] A. Michail. Data Mining Library Reuse Patterns in User-selected Applications. In *International Conference on Automated Software Engineering.* IEEE, 1999.

[144] A. Michail. Data Mining Library Reuse Patterns Using Generalized Association Rules. In *International Conference on Software Engineering.* ACM, 2000.

[145] P. Miettinen, T. Mielikäinen, A. Gionis, G. Das, and H. Mannila. The Discrete Basis Problem. *Transactions on Knowledge and Data Engineering*, 2008.

[146] Y. M. Mileva, A. Wasylkowski, and A. Zeller. Mining Evolution of Object Usage. In *European Conference on Object-Oriented Programming.* Springer, 2011.

[147] R. Minelli and M. Lanza. Visualizing the Workflow of Developers. In *Working Conference on Software Visualization*, 2013.

[148] R. Minelli, A. Mocci, and M. Lanza. I Know What You Did Last Summer: An Investigation of How Developers Spend Their Time. In *International Conference on Program Comprehension*, 2015.

[149] R. Minelli, A. Mocci, and M. Lanza. The Plague Doctor: A Promising Cure for the Windw Plague. In *International Conference on Program Comprehension*, 2015.

[150] R. Minelli, A. Mocci, R. Robbes, and M. Lanza. Taming the ide with fine-grained interaction data. In *International Conference on Program Comprehension*, 2016.

[151] H. Mintzberg. The Nature of Managerial Work. *Theory of Management Policy*, 1980.

[152] M. Monperrus, M. Bruch, and M. Mezini. Detecting Missing Method Calls in Object-Oriented Software. In *European Conference on Object-oriented Programming*, 2010.

[153] L. Moonen. Generating Robust Parsers Using Island Grammars. In *Working Conference on Reverse Engineering*. IEEE, 2001.

[154] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, and A. Marcus. How Can I Use This Method? In *International Conference on Software Engineering*. IEEE, 2015.

[155] B. Morschheuser, C. Henzi, and R. Alt. Increasing Intranet Usage through Gamification. In *Hawaii International Conference on System Sciences*, 2015.

[156] G. Murphy, M. Kersten, and L. Findlater. How Are Java Software Developers Using the Elipse IDE? *IEEE Software*, 2006.

[157] E. Murphy-Hill and G. C. Murphy. Recommendation Delivery. In *Recommendation Systems in Software Engineering*. Springer, 2014.

[158] E. W. Myers. An o(ND) Difference Algorithm and its Variations. *Algorithmica*, 1986.

[159] S. Nadi, S. Krüger, M. Mezini, and E. Bodden. "Jumping Through Hoops": Why do Developers Struggle with Cryptography APIs? In *International Conference on Software Engineering*, 2016.

[160] M. Nagappan, T. Zimmermann, and C. Bird. Representativeness in software engineering research. Technical report, Microsoft Research, 2012.

[161] M. Nagappan, T. Zimmermann, and C. Bird. Diversity in Software Engineering Research. In *International Symposium on Foundations of Software Engineering*. ACM, 2013.

[162] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *International Conference on Software Engineering*. IEEE, 2005.

[163] I. Neamtiu, J. S. Foster, and M. Hicks. Understanding Source Code Evolution Using Abstract Syntax Tree Matching. In *International Workshop on Mining Software Repositories*, 2005.

[164] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. *Lecture Notes in Computer Science*, 2304:213–228, 2002.

[165] S. Negara, M. Codoban, D. Dig, and R. E. Johnson. Mining fine-grained code changes to detect unknown change patterns. In *International Conference on Software Engineering*, pages 803–813. ACM, 2014.

[166] S. Negara, M. Vakilian, N. Chen, R. E. Johnson, and D. Dig. Is it dangerous to use version control histories to study source code evolution? In *European Conference on Object-Oriented Programming*, pages 79–103. Springer, 2012.

[167] A. T. Nguyen, M. Hilton, M. Codoban, H. A. Nguyen, L. Mast, E. Rademacher, T. N. Nguyen, and D. Dig. Api code recommendation using statistical learning from fine-grained changes. In *International Symposium on Foundations of Software Engineering*. ACM, 2016.

[168] A. T. Nguyen and T. N. Nguyen. Graph-based Statistical Language Model for Code. In *International Conference on Software Engineering*, 2015.

[169] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. V. Nguyen, J. Al-Kofahi, and T. N. Nguyen. Graph-based Pattern-oriented, Context-sensitive Source Code Completion. In *International Conference on Software Engineering*, 2012.

[170] H. A. Nguyen, A. T. Nguyen, and T. N. Nguyen. Using Topic Model to Suggest Fine-Grained Source Code Changes. In *International Conference on Software Maintenance and Evolution*. IEEE, 2016.

[171] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Graph-based Mining of Multiple Object Usage Patterns. In *International Symposium on Foundations of Software Engineering*. ACM, 2009.

[172] J. Nielsen. *Usability engineering*. Elsevier, Amsterdam, Netherlands, 1994.

[173] J. D. Novak and A. J. Cañas. The Theory Underlying Concept Maps and How to Construct and Use Them. Technical Report HMC CmapTools 2006-01, Institute for Human and Machine Computation, 2008.

[174] C. Omar, I. Voysey, M. Hilton, J. Aldrich, and M. Hammer. Hazelnut: A Bidirectionally Typed Structure Editor Calculus. In *Symposium on Principles of Programming Languages*, 2017.

[175] H. M. Parsons. What happened at Hawthorne? *Science*, 183(4128):922–932, 1974.

[176] C. Pekesen. A Cloud-based Universal Platform for Crowdsourcing Empirical Case Studies. *Bachelor Thesis, Technische Universität Darmstadt, Germany*, 2016.

[177] D. Perry, N. Staudenmayer, and L. Votta. People, Organizations, and Process Improvement. *IEEE Software*, 11(4):36–45, 1994.

[178] D. E. Perry, N. A. Staudenmayer, and L. G. Votta. People, Organizations, and Process Improvement. *IEEE Software*, 1994.

[179] L. Pickard, B. Kitchenham, and S. Linkman. An investigation of analysis techniques for software datasets. In *International Software Metrics Symposium*, 1999.

[180] M. Pinzger, E. Giger, and H. Gall. Handling Unresolved Method Bindings in Eclipse. Technical report, Department of Informatics, University of Zurich, Switzerland, 2007.

[181] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza. Mining StackOverflow to Turn the IDE Into a Self-Confident Programming Prompter. In *International Conference on Mining Software Repositories*. ACM, 2014.

[182] L. Ponzanelli, A. Mocci, and M. Lanza. StORMeD: Stack Overflow Ready Made Data. In *International Conference on Mining Software Repositories*, 2015.

[183] M. Pradel, P. Bichsel, and T. Gross. A Framework for the Evaluation of Specification Miners Based on Finite State Machines. In *International Conference on Software Maintenance*, 2010.

[184] M. Pradel, C. Jaspan, J. Aldrich, and T. R. Gross. Statically Checking API Protocol Conformance with Mined Multi-object Specifications. In *International Conference on Software Engineering*. IEEE, 2012.

[185] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim. Template-based Reconstruction of Complex Refactorings. In *International Conference on Software Maintenance*. IEEE, 2010.

[186] S. Proksch. Empirical Study of Impact Factors on the Quality of Code Recommender Models. *Master Thesis, Technische Universität Darmstadt, Germany*, 2011.

[187] S. Proksch, S. Amann, and M. Mezini. Towards Standardized Evaluation of Developer-assistance Tools. In *International Workshop on Recommendation Systems for Software Engineering*. ACM, 2014.

[188] S. Proksch, S. Amann, and S. Nadi. Enriched Event Streams: A General Dataset For Empirical Studies On In-IDE Activities Of Software Developers. In *International Conference on Mining Software Repositories – Accepted Mining Challenge*, 2017.

[189] S. Proksch, S. Amann, S. Nadi, and M. Mezini. A Dataset of Simplified Syntax Trees for C#. In *International Conference on Mining Software Repositories*. ACM, 2016.

[190] S. Proksch, S. Amann, S. Nadi, and M. Mezini. Evaluating the Evaluations of Code Recommender Systems: A Reality Check. In *International Conference on Automated Software Engineering*. ACM, 2016.

[191] S. Proksch, V. Bauer, and G. C. Murphy. How to Build a Recommendation System for Software Engineering. In *Software Engineering*. Springer, 2015.

[192] S. Proksch, J. Lerch, and M. Mezini. Intelligent Code Completion with Bayesian Networks. *Transactions of Software Engineering and Methodology. ACM*, 2015.

[193] S. Proksch, S. Nadi, S. Amann, and M. Mezini. Enriching In-IDE Process Information with Fine-grained Source Code History. In *International Conference on Software Analysis, Evolution, and Reengineering*, 2017.

[194] R. Purushothaman and D. E. Perry. Toward Understanding the Rhetoric of Small Source Code Changes. *Transactions on Software Engineering*, IEEE, 2005.

[195] V. Raychev, M. Vechev, and E. Yahav. Code Completion with Statistical Language Models. In *Conference on Programming Language Design and Implementation*. ACM, 2014.

[196] S. Reuß. On the Effects of Pointer Analysis on Intelligent Code Completion. *Master Thesis, Technische Universität Darmstadt, Germany*, 2016.

[197] T. J. Reynolds and J. Gutman. Laddering theory, method, analysis, and interpretation. *Journal of Advertising Research*, 28(1):11–31, 1988.

[198] I. Rish. An Empirical Study of the Naive Bayes Classifier. In *Proceedings of the Workshop on Empirical Methods in Artificial Intelligence*, 2001.

[199] R. Robbes and M. Lanza. A Change-Based Approach to Software Evolution. *Electronic Notes in Theoretical Computer Science*, 2007.

[200] R. Robbes and M. Lanza. Characterizing and Understanding Development Sessions. In *International Conference on Program Comprehension*. IEEE, 2007.

[201] R. Robbes and M. Lanza. How Program History Can Improve Code Completion. In *International Conference on Automated Software Engineering*, ASE'08, 2008.

[202] R. Robbes and M. Lanza. SpyWare: A Change-aware Development Toolset. In *International Conference on Software Engineering*, ICSE '08, pages 847–850, New York, NY, USA, 2008. ACM.

[203] R. Robbes and M. Lanza. Improving Code Completion with Program History. *Automated Software Engineering*, 17(2):181–212, 2010.

[204] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford. Automated API Property Inference Techniques. *Software Engineering, IEEE Transactions on*, 39(5):613–637, 2013.

[205] M. P. Robillard, W. Maalej, R. J. Walker, and T. Zimmermann. *Recommendation Systems in Software Engineering*. Springer, 2014.

[206] G. Robles. Replicating MSR: A Study of the Potential Replicability of Papers Published in the Mining Software Repositories Proceedings. In *Working Conference on Mining Software Repositories*, 2010.

[207] P. Rodeghero, C. McMillan, P. W. McBurney, N. Bosch, and S. D'Mello. Improving Automated Source Code Summarization via an Eye-tracking Study of Programmers. In *International Conference on Software Engineering*, 2014.

[208] T. Roehm, N. Gurbanova, B. Bruegge, C. Joubert, and W. Maalej. Monitoring User Interactions for Supporting Failure Reproduction. In *International Conference on Program Comprehension*, 2013.

[209] A. Rountev, A. Milanova, and B. G. Ryder. Points-to Analysis for Java Using Annotated Constraints. In *Conference on Object-oriented Programming, Systems, Languages, and Applications*, 2001.

[210] C. Sadowski, K. T. Stolee, and S. Elbaum. How Developers Search For Code: A Case Study. In *International Symposium on Foundations of Software Engineering*, 2015.

[211] G. Salvaneschi, S. Amann, S. Proksch, and M. Mezini. An empirical study on program comprehension with reactive programming. In *International Symposium on Foundations of Software Engineering*, 2014.

[212] G. Salvaneschi, S. Proksch, S. Amann, S. Nadi, and M. Mezini. On the Positive Effect of Reactive Programming on Software Comprehension: An Empirical Study. *Transactions on Software Engineering*, 2017.

[213] A. L. Santos, G. Prendi, H. Sousa, and R. Ribeiro. Stepwise API Usage Assistance Using N-Gram Language Models. *Journal of Systems and Software*, 2016.

[214] A. A. Sawant and A. Bacchelli. A dataset for api usage. In *International Conference on Mining Software Repositories*, 2015.

[215] J. Sayyad Shirabad and T. Menzies. The PROMISE Repository of Software Engineering Databases. School of Information Technology and Engineering, University of Ottawa, Canada, 2005.

[216] J. Schlitzer. Improving Object-Usage Detection with Method-Call Inlining. *Bachelor Thesis, Technische Universität Darmstadt, Germany*, 2015.

[217] K. A. Schneider, C. Gutwin, R. Penner, and D. Paquette. Mining a Software Developer's Local Interaction History. In *International Workshop on Mining Software Repositories*, MSR'04, 2004.

[218] J. M. Schultz and M. Liberman. Topic Detection and Tracking using idf-Weighted Cosine Coefficient. In *Proceedings of the DARPA Broadcast News Workshop*, pages 189–192, Athens, Greece, 1999. Morgan Kaufmann Publishers, Inc.

[219] M. Shepperd and G. Kadoda. Using Simulation to Evaluate Prediction Techniques [for Software]. In *International Software Metrics Symposium*. IEEE, 2001.

[220] S. Sheth, G. Kaiser, and W. Maalej. Us and them: A study of privacy requirements across north america, asia, and europe. In *International Conference on Software Engineering*, 2014.

[221] O. Shivers. Control flow analysis in scheme. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 164–174, New York, NY, USA, 1988. ACM.

[222] O. Shivers. Data-flow analysis and type recovery in scheme. In *Topics in Advanced Language Implementation*. The MIT Press, Cambridge, MASS, USA, 1991.

[223] O. Shivers. The semantics of scheme control-flow analysis. In *Proceedings of the Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 190–198, New York, NY, USA, 1991. ACM.

[224] A. Sillitti, A. Janes, G. Succi, and T. Vernazza. Collecting, Integrating and Analyzing Software Metrics and Personal Software Process Data. In *Euromicro Conference*. IEEE, 2003.

[225] J. Singer, R. Elves, and M.-A. Storey. Navtracks: Supporting navigation in software. In *13th International Workshop on Program Comprehension (IWPC'05)*, pages 173–175. IEEE, 2005.

[226] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil. An Examination of Software Engineering Work Practices. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '97, pages 21–35. IBM Press, 1997.

[227] V. Singh, L. L. Pollock, W. Snipes, and N. A. Kraft. A case study of program comprehension effort and technical debt estimations. In *International Conference on Program Comprehension*, 2016.

[228] W. Snipes, V. Augustine, A. R. Nair, and E. Murphy-Hill. Towards Recognizing and Rewarding Efficient Developer Work Patterns. In *International Conference on Software Engineering*. IEEE Press, 2013.

[229] W. Snipes, E. Murphy-Hill, T. Fritz, M. Vakilian, K. Damevski, A. Nair, and D. Shepherd. A Practical Guide to Analyzing IDE Usage Data. In *The Art and Science of Analyzing Software Data*. Morgan Kaufmann, 2015.

[230] W. Snipes, A. R. Nair, and E. Murphy-Hill. Experiences Gamifying Developer Adoption of Practices and Tools. In *International Conference on Software Engineering*, 2014.

[231] J. Spacco, J. Strecker, D. Hovemeyer, and W. Pugh. Software Repository Mining with Marmoset: An Automated Programming Project Snapshot and Testing System. In *International Workshop on Mining Software Repositories*, 2005.

[232] D. Steele-Johnson, R. S. Beauregard, P. B. Hoover, and A. M. Schmidt. Goal Orientation and Task Demand Effects on Motivation, Affect, and Performance. *Journal of Applied psychology*, 85(5):724, 2000.

[233] B. Steensgaard. Points-to Analysis in Almost Linear Time. In *Symposium on Principles of Programming Languages*, 1996.

[234] B. Steinert, D. Cassou, and R. Hirschfeld. Coexist: Overcoming aversion to change. In *Symposium on Dynamic Languages*, 2012.

[235] A. Strehl, J. Ghosh, and R. Mooney. Impact of similarity measures on web-page clustering. In *Workshop on Artificial Intelligence for Web Search*, pages 58–64, Palo Alto, CA, USA, 2000. AAAI.

[236] S. Subramanian, L. Inozemtseva, and R. Holmes. Live API Documentation. In *International Conference on Software Engineering*, pages 643–652. ACM, 2014.

[237] T. Teitelbaum and T. Reps. The Cornell Program Synthesizer: A Syntax-directed Programming Environment. *Communications of the ACM*, 1981.

[238] E. Tempero. Towards a curated collection of code clones. In *International Workshop on Software Clones*, 2013.

[239] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. Qualitas corpus: A curated collection of java code for empirical studies. In *2010 Asia Pacific Software Engineering Conference (APSEC2010)*, 2010.

[240] S. Thummalapenta and T. Xie. Parseweb: A Programmer Assistant for Reusing Open Source Code on the Web. In *International Conference on Automated Software Engineering*. ACM, 2007.

[241] S. Tichelaar, S. Ducasse, S. Demeyer, and O. Nierstrasz. A Meta-Model for Language-Independent Refactoring. In *International Symposium on Principles of Software Evolution*, 2000.

[242] Z. Tu, Z. Su, and P. Devanbu. On the Localness of Software. In *International Symposium on Foundations of Software Engineering*. ACM, 2014.

[243] V. Uquillas-Gomez. *Supporting Integration Activities in Object-Oriented Applications*. PhD thesis, Université des Sciences et Technologie de Lille-Lille I, 2012.

[244] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson. Use, Disuse, and Misuse of Automated Refactorings. In *International Conference on Software Engineering*. IEEE, 2012.

[245] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, R. Zilouchian Moghaddam, and R. E. Johnson. The Need for Richer Refactoring Usage Data. In *Workshop on Evaluation and Usability of Programming Languages and Tools*. ACM, 2011.

[246] P. Van Gorp and P. Grefen. Supporting the Internet-Based Evaluation of Research Software with Cloud Infrastructure. *Software and Systems Modeling*, 2012.

[247] D. Waddington and B. Yao. High-fidelity C/C++ Code Transformation. *Science of Computer Programming*, 2007.

[248] C. Wasson. Ethnography in the field of design. *Human organization*, 59(4):377–388, 2000.

[249] P. Weissgerber and S. Diehl. Identifying Refactorings from Source-Code Changes. In *International Conference on Automated Software Engineering.* IEEE, 2006.

[250] A. Wexelblat and P. Maes. Footprints: History-rich tools for information foraging. In *Conference on Human Factors in Computing Systems*, CHI '99, pages 270–277, New York, NY, USA, 1999. ACM.

[251] D. Wightman, Z. Ye, J. Brandt, and R. Vertegaal. Snipmatch: Using source code context to enhance snippet retrieval and parameterization. In *Annual Symposium on User Interface Software and Technology.* ACM, 2012.

[252] J. Wu, L. Shen, W. Guo, and W. Zhao. How Is Code Recommendation Applied in Android Development: A Qualitative Review. In *International Conference on Software Analysis, Testing and Evolution*, pages 30–35. IEEE, 2016.

[253] W. Wu, Y.-G. Guéhéneuc, G. Antoniol, and M. Kim. AURA: A Hybrid Approach to Identify Framework Evolution. In *International Conference on Software Engineering.* ACM, 2010.

[254] Z. Xing and E. Stroulia. UMLDiff: An Algorithm for Object-oriented Design Differencing. In *International Conference on Automated Software Engineering.* ACM, 2005.

[255] Z. Xing and E. Stroulia. API-Evolution Support with Diff-CatchUp. *Transactions on Software Engineering*, 2007.

[256] F. Xu, J. Weber, and D. Buhalis. Gamification in Tourism. In *Information and Communication Technologies in Tourism*, 2014.

[257] X. Yang, Y. Guo, and Y. Liu. Bayesian-inference based recommendation in online social networks. In *INFOCOM, 2011 Proceedings IEEE*, pages 551–555, Piscataway, NJ, USA, 2011. IEEE.

[258] A. Ying, G. Murphy, R. Ng, and M. Chu-Carroll. Predicting Source Code Changes by Mining Change History. *Transactions on Software Engineering*, IEEE, 2004.

[259] Y. Yoon and B. A. Myers. Capturing and analyzing low-level events from the code editor. In *Workshop on Evaluation and Usability of Programming Languages and Tools.* ACM, 2011.

[260] Y. Yoon, B. A. Myers, and S. Koo. Visualization of Fine-Grained Code Change History. In *Symposium on Visual Languages and Human-Centric Computing.* IEEE, 2013.

[261] C. Zhang, J. Yang, Y. Zhang, J. Fan, X. Zhang, J. Zhao, and P. Ou. Automatic Parameter Recommendation for Practical API Usage. In *International Conference on Software Engineering.* IEEE, 2012.

[262] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. MAPO: Mining and Recommending API Usage Patterns. In *European Conference on Object-oriented Programming.* Springer, 2009.

[263] H. Zhong, L. Zhang, and H. Mei. Inferring Specifications of Object-oriented APIs from API Source Code. In *Asia-Pacific Software Engineering Conference*, 2008.

[264] M. Zimmermann. Using Domain Knowledge to Improve Source Code Differencing. *Bachelor Thesis, Technische Universität Darmstadt, Germany*, 2016.

[265] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl. Mining Version Histories to Guide Software Changes. *Transactions on Software Engineering*, IEEE, 2005.

[266] M. Züger, C. Corley, A. Meyer, B. Li, T. Fritz, D. Shepherd, V. Augustine, P. Francis, N. Kraft, and W. Snipes. Reducing Interruptions at Work: A Large-Scale Field Study of FlowLight. In *Conference on Human Factors in Computing Systems*, 2017.