IDE 2.0: Leveraging the Wisdom of the Software Engineering Crowds

Vom Fachbereich Informatik der Technischen Universität Darmstadt genehmigte

Dissertation

zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

vorgelegt von

Diplom-Informatiker Marcel Bruch

geboren in Limburg an der Lahn.

Referent:	Prof. Dr. Mira Mezini
Korreferent:	Prof. Dr. Andreas Zeller
Datum der Einreichung:	02.09.2011
Datum der mündlichen Prüfung:	20.10.2011

Erscheinungsjahr 2012

Darmstädter Dissertationen
D17

Acknowledgements

Though only my name appears on the cover of this dissertation, a great many of people have contributed to its production. I owe my gratitude to all those people.

First, my advisor Prof. Dr. Mira Mezini, who gave me the opportunity to conduct my research in the area of intelligent IDEs and who supported me with countless discussions and helpful comments. Also, I owe her many thanks for enabling me to present my ideas to a much larger audience than I ever expected to reach, and for providing kind words at times when things did work as well as they should.

I would also like to thank my co-advisor Prof. Dr. Andreas Zeller for his time spent on critically reading the work and all other committee members who let me successfully finish and conclude this chapter of my life.

Many thanks also go to all my co-authors for the amazing and hard work they put into the project, and of course to all of my 58 students that supported this research work – it was my pleasure to learn from and work with you!

When working such a long time at the same place, there is naturally much you take away that is not only related to research but to social experiences. Thanks go out to my colleagues for lots of hilarious moments but also for several grounding experiences that have left their imprint on me and will hopefully never fall into oblivion.

Furthermore, I would like to thank my parents for their valuable support during my studies (and hopefully thereafter).

And of course there is the one and only who continuously pushed me to the limits, who at the same time protected me from getting lost in my work, who supported me in my many difficult moments and who will always motivate me to continue. My love Kristin.

Academic Résumé

November 2006 – October 2011

Doctoral studies at the chair of Prof. Dr. Mira Mezini, Software Technology Group, Fachbereich Informatik, Technische Universität Darmstadt.

October 1999 – August 2006

Studies of Computer Science with the subsidiary subject of Business Administration at the Technischen Universität Darmstadt.

Abstract

During the past decades, software systems have grown significantly in size and complexity, making software development and maintenance an extremely challenging endeavor. Integrated Development Environments (IDEs) greatly facilitate this endeavor by providing a convenient means to browse and manipulate a system's source code and to obtain helpful documentation on Application Programming Interfaces (APIs).

But according to Ko et al. [KMCA06], developers spent up to 40 % of their time with searching example usages and analyzing how to use a given API correctly—quite a lot of time that is not used for its actual objective, the development of new features, but for understanding existing code and learning how to use existing APIs.

We argue that there is great space for improvement by exploiting collective intelligence, the knowledge of the masses. The leveraging of user data to build intelligent and user-centric web-based systems is the source of our inspiration. A Web 2.0 site allows its users to interact with each other as contributors to the website's content, in contrast to websites where users are limited to the passive viewing of information that is provided to them.

This mindshift in enableing users to contribute to a site has significant impact on the quality of services of existing websites. Amazon, the leading internet sales platform for books and electronic devices, for instance, creates recommendations based on purchase behaviors of its customers or finds interesting similar products based on how customers interact with search results. Netflix, a video-on-demand service, features a web application that leverages user ratings on movies to recommend likely interesting movies to other users. These systems have in common that they leverage *crowds* to continuously improve the quality of their services, either through implicit feedback (e.g., user click-through behaviors), explicit feedback (e.g., ratings for movies) or user-generated content (e.g., product reviews and movie critiques).

We argue that today's IDEs behave more like traditional "Web 1.0" applications in the way that they do not enable their users to contribute and share their knowledge with others, neither explicitly nor implicitly, and thus hinder themselves to effectively exchange knowledge among developers. This thesis investigates how successful concepts of the Web 2.0 can be brought to today's IDEs and shows how existing IDE tools like code completion, documentation viewers, code-search, and bug detectors can be improved by leveraging the knowledge that is available in the developer's IDE.

Zusammenfassung

Innerhalb der letzten Jahrzehnte sind Softwaresysteme erheblich in Größe und Komplexität gewachsen. Dieser enorme Komplexitätsanstieg macht die Entwicklung und Wartung solcher Systeme zu einem extrem schwierigen und kostenintensiven Unterfangen. Integrierte Entwicklungsumgebungen unterstützen Softwareentwickler bei dieser Arbeit - beispielsweise indem sie den Entwickler beim Suchen und Manipulieren von Quellcode unterstützen oder Dokumentation kontext-sensitiv innerhalb der Entwicklungsumgebung zur Verfügung stellen.

Trotz dieser Unterstützung verbringen Softwareentwickler laut einer Studie von Ko et al. [KMCA06] bis zu 40% ihrer Zeit mit der Suche nach und der Analyse von Beispielcode, um zu verstehen, wie existierende Softwarekomponenten korrekt eingesetzt werden können. Zeit die somit nicht dem eigentlichen Ziel - der Entwicklung neuer Software - zur Verfügung steht, sondern in das Erlernen bestehender Systeme fließt.

Diese Studie zeigt eine Ineffizienz im Entwicklungsprozess auf, die nur unzureichend durch bestehende Ansätze abgedeckt wird. Wir behaupten das aktuelle Entwicklungsumgebungen (neben dem Compiler eines der wichtigsten Werkzeuge für Softwareentwickler) ihren Nutzern bei diesen Problemen erheblich besser unterstützen könnten und noch viel Raum für Verbesserungen bieten – zum Beispiel durch die Nutzung von kollektiver Intelligenz - dem Wissen der Massen.

Die Nutzung von Benutzerdaten, um intelligente und benutzerzentrische web-basierende System zu bauen (bekannt unter dem Begriff Web 2.0), ist die Quelle unserer Inspiration. Eine Web 2.0 Anwendung erlaubt ihren *Benutzern* neue Inhalte zu erstellen - im Gegensatz zu klassischen Web 1.0 Anwendungen in denen der Benutzer auf das passive Lesen der Informationen beschränkt ist, die ihm durch die Webseitenanbieter dargeboten werden.

Der Sinneswandel, Webseitennutzern zu erlauben zu den Inhalten einer Website beizutragen, hat enorme Auswirkungen auf die Qualität der angebotenen Dienste. Amazon, die führende Internet Verkaufsplattform für Bücher und elektronische Geräte, erzeugt beispielsweise Empfehlungen basierend auf dem Kaufverhalten seiner Kunden oder identifiziert relevante, ähnliche Produkte basierend auf dem Navigationsverhalten, wie Benutzer mit den Ergebnissen einer Suchanfrage umgehen. Netflix, ein Video-on-Demand Anbieter, betreibt eine Webseite die Benutzerbewertungen für Filme verwendet um anderen Kunden Filme vorzuschlagen, die zu den persönlichen Vorlieben (Action, Romantik, Trick etc.) passen.

Diese Systeme haben gemeinsam, dass sie Benutzerdaten verwenden, um ihre Dienste kontinuierlich zu verbessern - sei es durch implizites Feedback (zum Beispiel das Klick-Verhalten),

durch explizites Feedback (z.B. durch die Bewertungen von Filmen) oder durch benutzergenerierten Inhalte (z.B. durch das Schreiben von Filmkritiken).

Wir sind der Meinung, dass sich heutige Entwicklungsumgebungen wie traditionelle Web 1.0 Anwendungen verhalten. Sie ermöglichen es ihren Nutzern nicht, Wissen und Erfahrungen mit anderen Nutzern zu teilen (weder implizit noch explizit) und verhindern dadurch den effektiven Austausch von Wissen unter Softwareentwicklern. Diese Doktorarbeit untersucht, wie erfolgreiche Konzepte des Web 2.0 in heutige Entwicklungsumgebungen integriert werden können und zeigt, wie existierende Werkzeuge wie beispielsweise Code Completion, Dokumentation, Beispielcode-Suche und Bug-Detektoren durch das Nutzen von kollektiven Wissens verbessert werden können.

Contents

1 Introduction **1**
- 1.1 The Vision of *"IDE 2.0"* . . . 1
- 1.2 Contributions of the Thesis to the Vision . . . 8
- 1.3 Structure of the Thesis . . . 10

2 Recommending Likely Method Calls for Code Completion **13**
- 2.1 Introduction . . . 13
- 2.2 Example-Based Code Completion Systems . . . 15
- 2.3 Gathering Example Data . . . 23
- 2.4 Evaluation . . . 35
  - 2.4.1 Data Set . . . 35
  - 2.4.2 Scenarios . . . 35
  - 2.4.3 Metrics . . . 36
- 2.5 Results . . . 38
- 2.6 Eclipse Integration . . . 40
- 2.7 User Study . . . 43
  - 2.7.1 Setup . . . 43
  - 2.7.2 Results . . . 43
- 2.8 Summary . . . 44

3 Mining Subclassing Directives from Example Code **47**
- 3.1 Introduction . . . 47
- 3.2 Four Kinds of Subclassing Directives . . . 49
  - 3.2.1 Method Overriding Directives . . . 49
  - 3.2.2 Method Extension Directives . . . 49
  - 3.2.3 Method Call Directives . . . 50
  - 3.2.4 Class Extension Scenarios . . . 50
- 3.3 Techniques to Mine Subclassing Directives . . . 51
  - 3.3.1 Mining Overriding Directives . . . 51
  - 3.3.2 Mining Extension Directives . . . 53
  - 3.3.3 Mining Call Directives . . . 53
  - 3.3.4 Mining Class Extension Scenarios . . . 53
  - 3.3.5 Defining Importance Levels . . . 54

3.3.6 Implementation . . . 54
3.4 Case Study . . . 55
3.4.1 Set Up and Overview of the Results . . . 55
3.4.2 Disagreeing Documented and Mined Directives . . . 56
3.4.3 Documented and Not Mined Directives . . . 58
3.4.4 Not Documented Important Directives . . . 58
3.4.5 Relevance of Class Extension Scenarios . . . 59
3.4.6 The Expert Viewpoint . . . 61
3.5 Integrating Subclassing Directives in IDEs . . . 62
3.6 Summary . . . 64

4 Detecting Missing Method Calls in Object-Oriented Software 67
4.1 Introduction . . . 67
4.2 The Importance of Detecting Missing Method Calls . . . 69
4.2.1 Problems Related to Missing Calls are Real and Hard to Understand . . . 69
4.2.2 Missing Method Calls Survive Development Time . . . 70
4.2.3 Recapitulation . . . 71
4.3 The DMMC System . . . 71
4.3.1 Overview . . . 72
4.3.2 Extracting Type-Usages . . . 73
4.3.3 Exactly and Almost Similar Type-usages . . . 74
4.3.4 S-score: A Measure of Strangeness for Object-Oriented Software . . . 75
4.3.5 Predicting Missing Method Calls . . . 75
4.4 Evaluation . . . 76
4.4.1 The Correctness of the Distribution of the S-Score . . . 77
4.4.2 The Ability of S-score to Catch Degraded Code . . . 78
4.4.3 The Performance of the Missing Method Calls Prediction . . . 79
4.4.4 Finding New Missing Method Calls in Eclipse . . . 81
4.4.5 Summary of the Evaluation . . . 89
4.5 Missing Method Call Detection in the Development Process . . . 89
4.6 Summary . . . 90

5 Learning Rankings for Code Search 93
5.1 Introduction . . . 93
5.2 State of the Art in Code Search . . . 95
5.3 Learning Ranking Functions . . . 97
5.3.1 Scoring Documents . . . 97
5.3.2 Exploiting Relevance Feedback . . . 98
5.3.3 Exploiting Explicit Feedback . . . 98
5.3.4 Exploiting Implicit Feedback . . . 100

5.4 Approach . . . 101
5.4.1 Workflow Overview . . . 101
5.4.2 Search Index and Query Creation . . . 102
5.4.3 Ranking Function . . . 104
5.4.4 Code Summarization . . . 105
5.5 Evaluation Setup . . . 105
5.5.1 Benchmark . . . 106
5.5.2 Evaluated Systems . . . 111
5.5.3 Evaluation Metrics . . . 111
5.6 Results . . . 113
5.6.1 Comparing System Performances . . . 113
5.6.2 Learning from User Click-through Data . . . 115
5.6.3 Threats to Validity . . . 116
5.7 Summary . . . 117

6 Related Work 119
6.1 Documentation Mining . . . 119
6.2 Code-search . . . 122
6.3 Code Completion . . . 125
6.4 Bug Detection . . . 127
6.5 Others . . . 129

7 Conclusions and Future Work 133
7.1 Conclusion . . . 133
7.2 Future Work . . . 133

Bibliography 137

List of Figures

1.1 Thesis Vision: Linked global knowledge bases 2
1.2 Eclipse Code Recommenders Project Logo 10

2.1 Encoding Framework Usages as Boolean Vectors 18
2.2 From Observations to Recommendations 19
2.3 Object Usage Data Structure 24
2.4 Example Class Hierarchy 26
2.5 Example of a callgraph created with WALA's default settings. 28
2.6 Code Snippet Before & After Code Completion 37
2.7 Performance of *EcCCS*, *FreqCCS*, *ArCCS* and *BMN* 38
2.8 F1 per Expected Method Call Recommendations 39
2.9 Integration of our Example-based Code Completion into Eclipse 41

3.1 API Documentation containing a method overriding directive 49
3.2 An Example to Assess the Correctness of the Metric *ovLikelihood* 52
3.3 Mapping Code to a Binary Space to Mine Extension Scenarios with LCA . . . 54
3.4 Mined Class-level Subclassing Directives 63
3.5 Mined Method-level Subclassing Directives 64

4.1 Exactly-Similar and Almost-Similar Type-Usages 72
4.2 Extraction Process of Type-Usages in Object-Oriented Software. 73
4.3 An example computation of the likelihoods of missing method calls 76
4.4 Distribution of the S-Score based on the type-usages of SWT 77
4.5 Scatter Plot of the Type-Usages of SWT 79
4.6 Font inconsistencies in Eclipse caused by a missing method call to `setFont` . . 86
4.7 Excerpt of revision 1.5 of BrowserDescriptorDialog.java 86
4.8 The DMMC system in batch mode. The output could be text or XML. 90

5.1 Improving rankings leveraging *explicit* feedback 99
5.2 The workflow of $\text{CoRe}_{\text{LUCID}}$ 101
5.3 $\text{CoRe}_{\text{LUCID}}$ Integration into Eclipse 103
5.4 Average Precision and Kendall's τ performances 114
5.5 Performance-to-Feedback Ratio 115

1 Introduction

1.1 The Vision of *"IDE 2.0"*

> Under the right circumstances, groups are remarkably intelligent and are often better than the smartest person in them.
>
> – James Surowiecki: *Wisdom of the Crowds*

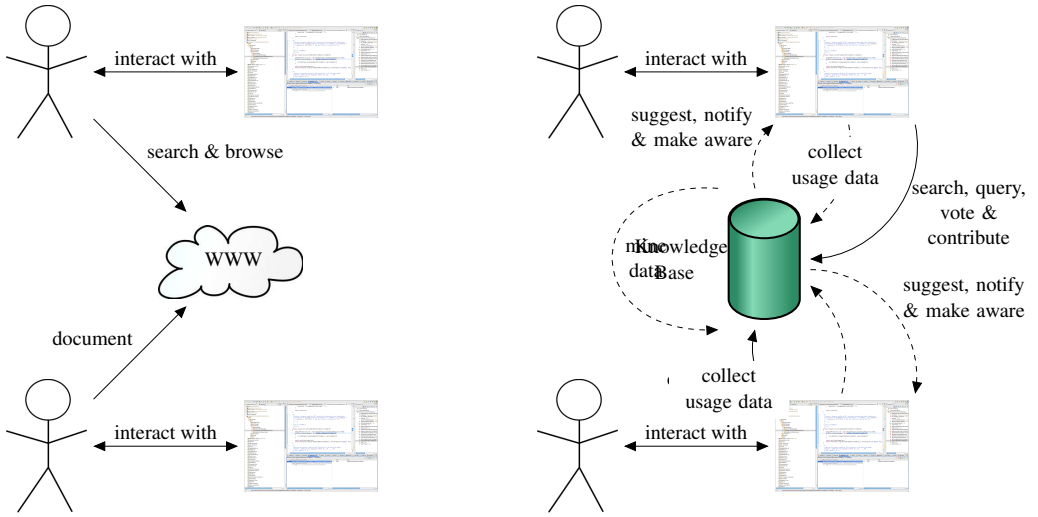
During the past decades, software systems have grown significantly in size and complexity, making software development and maintenance an extremely challenging endeavor. Integrated Development Environments (IDEs) greatly facilitate this endeavor by providing a convenient means to browse and manipulate a system's source code and to obtain helpful documentation on Application Programming Interfaces (APIs).

Yet, we argue that IDEs could support their users much better by harnessing collective intelligence, the knowledge of the masses. The *Web 2.0* is the source of our inspiration. In Web 2.0, applications allow its users to directly and indirectly contribute to the website's content—in contrast to Web 1.0 sites where users are limited to the passive viewing of information that is provided to them. Amazon, the leading internet sales platform for books and electronic devices, for instance, creates recommendations based on purchase behaviors of its customers or finds interesting similar products based on how customers interact with search results. Netflix, a video-on-demand service, features a web application that leverages user ratings on movies to recommend likely interesting movies to other users. Last.FM, a personalized internet radio station, recommends its users new songs similar to their personal preferences and enable users to form spontaneous communities around their favorite artists and music.

All these systems have in common that they leverage *crowds* to continuously improve the quality of their services, either through implicit feedback (e.g., user click-through behaviors), explicit feedback (e.g., ratings for movies and songs) or user-generated content (e.g., product reviews and movie critiques).

We argue that today's IDEs behave more like traditional Web 1.0 applications in the way that they do not enable their users to contribute and share their knowledge with others, neither explicitly nor implicitly, and thus hinder themselves to effectively exchange knowledge among developers.

But what would it mean to bring collective intelligence into software development and hence into today's IDEs?



(a) Presence: Isolated IDEs, manual search

(b) Future: An integrated, pro-active knowledge base

Figure 1.1: Our vision: in the future, IDEs will be linked through global knowledge bases

Figure 1.1a shows the current state of the practice: software developers use IDEs that are "integrated" only in the sense that they integrate all tools necessary to browse, manipulate and build software on a single machine. If a programmer has a question about a particular piece of code, for instance an API, she has to browse the web for solutions—by hand. After she has found the solution and solved her problem, the newly gained knowledge usually stays on the developer's mind only and is not passed to other developers.

Figure 1.1b shows our vision of how IDEs will work in the near future: IDEs will support developers through integration with a global knowledge base. This knowledge base will receive information from implicit and explicit user feedback. By implicit feedback we mean anonymized usage data that the cross-linked IDEs will send to the knowledge base automatically and spontaneously (in the figure, we represent such spontaneous activity through dashed arrows). The knowledge base will also comprise explicit user feedback in the form of user-written documentation, error reports, manuals, etc.

Crucially, the knowledge base itself is intelligent: it will use novel data-mining techniques to integrate the different sources of information to produce new information that has added value. For instance, if the knowledge base discovers that Java developers who write an `equals` method often write a `hashCode` method on the same type at the same time, or do so after an extended debugging session, then the knowledge base may be able to discover the important rule that, in Java, every type that implements `equals` should also implement `hashCode`, and that missing this rule likely causes bugs.

But if today's IDEs are 1.0, then what is the current state-of-the-art in research and how will future IDEs leveraging the wisdom of the masses look like?

From IDE 1.0 towards IDE 2.0

In the following we give three examples of how research in collective intelligence can improve existing IDE services. We split the discussion of each example into three sections: **IDE 1.0** sections describe the state-of-the-art in today's IDEs. Under **IDE 1.5**, we briefly summarize current research to improve IDE 1.0 services. And finally, the **IDE 2.0** sections discuss how collective intelligence could solve some of the issues of current research approaches, and thus, further advance IDEs towards the vision of IDE 2.0.

Intelligent Code Completion

IDE 1.0: Code completion is a very popular feature of modern IDEs, a life without which many developers find hard to imagine. One major reason for its popularity is that developers are frequently unaware of the methods they can invoke on a given variable. Here, code completion systems (CCSs) serve as an API browser, allowing developers to browse methods and select the appropriate one from the list of proposals. However, current completions are either computed by rather simplistic reasoning systems or are simply hard-coded. For instance, for method completion, CCSs base their proposals only on the receiver's declared type. This often leads to an overwhelming number of proposals. For instance, triggering code completion on a variable of `javax.swing.JButton` results in 381 method proposals. Clearly, developers only need a fraction of the proposed methods to make their code work. Code templates are an example for hard-coded proposals. Templates (like the Eclipse SWT Code Templates) serve as shortcuts and documentation for developers. Unfortunately, manual proposal definitions are labor intensive and error prone, and thus, only a very limited number of such templates exist. SWT in its latest version, for instance, comes with 35 templates dealing with object creation patterns while Eclipse UI or Eclipse JFace offer no templates at all.

IDE 1.5: Researchers have recognized these issues. For instance, approaches exist that analyze client code to learn which methods the clients frequently use in certain contexts, and rearrange method proposals according to this notion of relevance [BMM09, RL08, KM06]. Tools like XSnippet, Prospector and Parseweb [MXBK05, SC06, TX07] attempt to solve the issue of hard-coded code templates by also analyzing source code and identifying common patterns in code. Although obviously useful, these systems didn't make it into current IDEs. We argue that the primary reason for this is the lack of a continuously growing knowledge base: To build reliable models, source-code based approaches require example applications and full knowledge about the execution environment (i.e., class-path, library versions, etc.). However, finding a sufficiently large set of example projects is difficult and tedious, and creating models for new

frameworks is too time-consuming yet. While such approaches can sufficiently support a few selected APIs, we argue that they do not scale when tens of thousands of APIs should be supported.

IDE 2.0: So, how can we build continuously improving code completion systems then? To solve the scalability problem, code completion systems must allow users to share usage information among each other in an anonymized and automated way—from within the developer's IDE. This continuous data sharing allows recommender systems to learn models for every API that developers actually use. IDEs are very powerful when it comes to extracting information: they actually have access to information like the execution environment or the class-path of a program and even about user interactions.

But the new, massive data sets derived from this information pose a challenge. We will likely require new algorithms to find reliable and valuable patterns in this data.

Whatever means future code completion systems will use to build better recommendation models, the systems will be based on shared data. It will be the users who provide this data and it is important to realize that, as the user base grows, the recommendation systems will be able to continuously improve over time. Very soon such systems will make intelligent completions that are useful for novice developers and experts alike.

Example & Code-Snippet Recommendations

IDE 1.0: Source-code examples appear to be highly useful to developers, whenever the documentation of the API at hand is insufficient [Rob09]. This is evident by the rise of code search engines (CSEs) over the last few years, e.g., Google Codesearch, Krugle, and Koders, just to name a few. However, current CSEs almost exclusively use standard information-retrieval techniques that were developed for text documents. While source code is text, it also bears important inherent structure. Disregarding this structure causes less effective rankings and misleading code summaries.

IDE 1.5: Researchers have presented a number of approaches [HM05a, LBN+09a, ZXZ+09, Sin06, HFW07, LLBO07, BOL10] that improve certain aspects of CSEs. All these approaches exploit structure, like inheritance relations, method calls, type usages, control flow, and more. However, they face two severe problems. First, source code provides much more structure than text. Thus, ranking systems have to take into account many more features when ranking the results for a search query. Consequently, it is hard to derive optimal weights for these features, so that the resulting scoring function will perform as well as possible. Often, a fixed scoring system will perform "well enough" but not be optimal, and thus, won't create the best possible results. Another issue with current CSEs is that they ignore the personal prior knowledge of the user who issued the query. Many current web search engines now support "personalized

search", which leverages the personal background and interests of a user to find documents that are likely to be interesting for this user, but not necessarily for others. Current CSEs lack such functionality, and thus, lavish their users with lots of examples that do not contain any new information to them.

IDE 2.0: How can one improve rankings and realize personalized search in CSEs? The key to solving both problems is to *leverage implicit user feedback*. To solve the manual-weight-tweaking problem of search engines, recent work [Joa02] has shown that leveraging observations of how users interact with the search results can significantly improve the precision of existing search engines. The authors used the information whether or not the user inspects a search result to automatically adjust feature weights. From these observations they produce an optimized ranking where all inspected results are listed above those that the user did not investigate and used these rankings as input for an machine learning algorithm that uses these rankings as optimization criterion to determine the best feature weights. To implement personalized code search engines, one can infer the personal background (or experience) of a developer by the code she has already written. Then, CSEs could first display code examples that are similar to examples previously explored or, on demand, code examples that allow the developer to learn new information. We are certain that IDE services in general, not only those that we discussed, can greatly benefit from leveraging implicit user feedback.

Extended Documentation

IDE 1.0: Software engineers widely accept that documenting software is a tedious job. Especially open-source projects frequently lack sufficient resources to produce comprehensive documentation. Both Sun/Oracle and the Eclipse Foundation recently started to address this problem by opening their documentation platforms to their users. Eclipse asks its users to provide and update tutorials at the central Eclipse Wiki. Sun/Oracle's "Docweb" allows users to edit JavaDoc API documentation, and to provide code examples or cross references to other interesting articles in the web. These tools aim to leverage a Wikipedia-style approach tailored to software documentation. Past experience has shown, however, that such systems often suffer from a lack of user participation. We believe that the primary cause for this lack of participation is the fact that people may not be willing to document APIs which they have no control over, because these APIs may change rapidly at any time: they may be completely outdated in just a few months.

IDE 1.5: Recent research therefore addresses the problem from another angle, enriching existing documentation with automatically mined documentation [BMM10, LWC09, KLwHK10]. Such approaches identify frequent patterns or interesting relations in code, and generate helpful guidelines from these relations. However, generated documentation may not always be helpful. Like text mining, documentation mining uncovers *any* relation between code elements, no mat-

ter whether or not this relation is useful to consider. The problem is aggravated by the fact that it is sometimes the non-obvious and hidden relations that are the most useful. Another drawback of mining approaches is that they cannot provide rationales for their observations, leaving it up to the developer to make sense of the data.

IDE 2.0: How could collective intelligence address the issues mentioned above? The key to a solution is a mixture of *explicit user feedback* and *user-provided content*. In the future, we expect generated documentation to be judged by thousands of users, enabling people to evaluate the quality of their services immediately—tool developers and documentation providers alike. Furthermore, we expect collective intelligence to enable us to migrate documentation from older to newer versions more easily. For example, when a new version of an API becomes available, *explicit user feedback* will make apparent which parts of the documentation remain valid for the newer version and which parts require updating. Explicit user feedback will also allow users to attach rationale to mined documentation, allowing the documentation to not only state *that* users must follow a certain principle but *why*.

All these examples are, however, just the tip of the iceberg and the past has proven that research community always comes up with new ideas what else could be improved to support developers on their work. *But why do we call it IDE 2.0?*

From Web 2.0 to IDE 2.0

We have used the analogy to "Web 2.0" to indicate that this new generation of web applications and our view of future IDEs have something in common. In the following, we discuss the similarities between Web 2.0 and IDE 2.0 to make this analogy more concrete.

We define a set of principles that we expect successful IDE 2.0 services to follow. Some of the concepts are paraphrased from Tim O'Reilly's principles for successful Web 2.0, described in his article "What is Web 2.0?".

1. The Web as Platform. The web as platform is the core concept of Web 2.0. In various ways, clients and servers share data over the web. We expect the same to hold for future collaborative IDE 2.0 services. These services rely on client-side usage data and thus, the web is also fundamental to them. A notable difference between IDE 2.0 and Web 2.0 is that IDEs offer a much broader spectrum of data and also allow for client-side pre-processing of data like static code analysis. Such pre-processing may even be crucial to allow for proper privacy. Furthermore, one needs to distribute to clients recommendation models that are built on the server-side. Local databases or caches can increase the scalability of these systems; this is crucial when dealing with millions of request per day. Whatever the particular technology may be, the web will be the platform for IDE 2.0.

2. Data is key. Data is key to any IDE 2.0 service. However, here we fundamentally differ from Tim O'Reilly's understanding of who owns this data. In Web 2.0, data is the key factor for the success of an application over its competitors. In contrast, we strongly believe in *Open Data*: all collected data is publicly available. This fosters a vital ecosystem around the concepts of IDE 2.0 and enables sustainable research. Successfully IDE 2.0 services will use both raw data and derived knowledge and will facilitate innovation instead of locking in data or users.

3. Harnessing Collective Intelligence. Leveraging the wisdom of the crowds is the third fundamental concept of successful Web 2.0 applications—and the same holds for IDE 2.0. The examples introduced in the previous section used either user-provided content (like source code, updated documentation or code snippets), implicit feedback (like user click-through data used to improve rankings), or explicit feedback (like ratings for judging the quality of relevance of generated documentation) to build new kind of services. It is important to recognize that, while individuals may be able to build these services, these services cannot unleash their potential without the crowds sharing their knowledge. Only with collective intelligence, IDE services like intelligent code completion, example recommenders or even smart documentation systems become possible. Yochai Benkler's work about commons-based peer production [**?**] gives interesting insights into what motivates individuals to contribute to projects like IDE 2.0.

4. Rich User Experiences. The appearance of AJAX gave web applications a new look and feel, bringing web applications closer to desktop applications than ever before. In the context of IDE 2.0, intelligent, context-sensitive recommender systems will evolve that recommend relevant APIs or documentation where appropriate and help to reduce the clutter in IDEs at the same time. However, providing rich user experiences is fundamental for users to accept such services. Similar to Google Search, simple and intuitive interfaces seamlessly integrated into existing IDE concepts like code completion, quick fixes etc. are the major key to success.

5. Lightweight Programming Models. In web 2.0, mashups (applications that combine several other (web) applications to build new services on top of existing ones) evolved, building new services the initial application developers never considered. Excellent IDE 2.0 services will encourage others to build their services on top of existing ones by providing public and easy-to-use APIs. Clearly, in the early days we expect such services to be data-driven, i.e., they will leverage the same data for enhancing several aspects of current IDEs or to port existing services to other IDEs. Note that Open Data is necessary to enable such services. However, over time, services will use other services to build what we call IDE mashups.

Summary

The concepts behind Web 2.0 are a great fit for future IDE services and we expect future services to meet many if not all of the above properties.

However, this vision is neither self-fulfilling nor trivial to achieve. The Software Engineering research community has to play a key role in unleashing the full power of the crowds. First and foremost, it has to provide an appropriate environment for building and evaluating IDE 2.0 services. Strong partners like the Eclipse Foundation or Sun/Oracle already support and promote such new IDE concepts today, and their help will be crucial to providing access to large user communities in the future. But there is an incentive for these partners: they will profit from new *exciter* features, making the IDE itself appear very innovative.

Second, the Software Engineering research community is the link between practitioners and researchers in machine learning. Most IDEs only contain instances of rather primitive machine-learning algorithms. It will be our job to identify the problems that developers face in their day-to-day work, to provide appropriate data as input for machine learners, and to evaluate and reintegrate these results into IDEs. Thus, IDE 2.0 research will create new fascinating and challenging applications of machine learning beyond the current markets.

To sum up, we believe IDE 2.0 services have much potential to improve developer productivity and provide a fantastic playground for new algorithms. They bring together several research communities at the same time, to solve a new generation of challenges in Software Engineering. When tackling the problem now and in a farsighted, IDE 2.0 will be one of the major research areas of the near future.

But if IDE 2.0 is still a vision, what are the contributions of this thesis?

1.2 Contributions of the Thesis to the Vision

The idea of mining source code to extract information valuable to developers new to a framework or application has been around for almost a decade now. The advances in static analysis, in machine learning, as well as in the large-scale availability of open source code repositories gave rise to this promising research area.

Roughly at the same time, the notion of “Web 2.0” appeared. Although the term suggests a new version of the World Wide Web, it did not refer to an update to any technical specification, but rather to cumulative changes in the ways vendors, software developers and end-users use the Web. The exact meaning of Web 2.0 has been discussed many times since, but was finally coined to a large extent by Tim O’Reilly’s talk at the O’Reilly Media Web 2.0 conference in late 2004 and his later article “What is Web 2.0?” in 2005.

When this research started in November 2006, the Web 2.0 was omnipresent and the success of web applications like Amazon demonstrated clearly how the principles of Web 2.0 led to very

successful applications. More important, it showed how services greatly improved by leveraging the information how users interact with the application and by allowing users to contribute to these services by the means of explicit feedback or user-provided content.

Inspired by this development, we started out to see whether the concepts of Web 2.0 can be successfully transferred to improve today's IDEs. At the beginning of this thesis, the working vision was to create a code completion feature which would be way smarter than the one in today's IDEs by leveraging the knowledge of how other developers used or extended a given API before. It should do so by analyzing the widely available, large-scale code repositories which we considered as some kind of implicit user feedback on how to use or extend a given API. At that time it was not clear that such an approach would work, and we had to evaluate numerous of different mining techniques, static analyses, and evaluation scenarios until the intelligent completion system worked in the way as it is today and presented in this thesis.

Code completion is not the only service that may benefit from Web 2.0 concepts. IDEs do a great job when it comes to searching and presenting documentation on how to use an API. However, one recurring problem with open source software (OSS) is that its documentation is frequently outdated, misleading or even missing. In these cases, example code that used an API successfully before becomes an interesting source of information again. However, manually extracting patterns from source code is a tedious and time-consuming process and the question arises whether we couldn't support developers better by automatically extracting frequent patterns from example code and provide them as additional documentation. In this thesis we present an approach to mining so called subclassing directives from example code.

Bugs or wrong API usages are another severe problem for developers. Tools like FindBugs have been around for years now and have become an inherent part of today's build systems. However, one unpleasant weakness of these tools is that they identify incorrect API usage based on predefined, manually crafted rule-sets. Creating such rule-sets, however, is a tedious and time-consuming job, and thus, only few rule-sets exist. Consequently, such tools cannot prevent developers from misusing APIs they don't have rule-sets for. Given our experience in recommending likely method calls for code completion, we evaluated whether deviations from some commonly observed usage patterns may be good indicators for potential bugs in code – and came up with a system that was able to find a dozen bugs in Eclipse 3.5, a major code base.

Another challenging research area is the development of code-search engines. The challenge with code-search is: given some kind of a query; how to identify those code snippets deemed relevant by a developer and rank them according to her notion of relevance. But ranking such code snippets is a non-trivial task. Typically, dozens of (potentially interdependent) features exist that may be taken into account for judging the relevance of a snippet; manually tweaking such systems to produce optimal results becomes a very challenging task. To overcome this we worked on an approach that learns ranking functions for code-search engines by leveraging user relevance feedbacks (especially user click-through data). This approach is able to perform equally well as state-of-the-art code-search engines just after a few training queries. Because

Figure 1.2: Eclipse Code Recommenders Project Logo

of it's self-adapting capabilities this approach may pave the way for personalized code-search engines in the near future.

Four tools build around user feedback and example code. But how do they contribute to IDE 2.0?

What constitutes IDE 2.0 is a continuous round-trip of knowledge sharing and self-improving services which are tightly integrated into your IDE. Given this definition of IDE 2.0, the tools presented in this thesis are more IDE 2.0-*ready* than 2.0 yet. Further research is needed to see whether the concepts of the tools we have worked on in the past four years can actually keep what they promise. Therefore, we started to spread the vision of IDE 2.0 in 2009 and have been speaking on ten Eclipse DemoCamps, three industry conferences, published several interviews and articles in press since, and had the chance to learn a lot about what developers would like to see in their IDE.

Based on the gained experiences and the gathered feedback, we decided to propose the Code Recommenders research project as an Eclipse project, which finally has been accepted as such in January 2011. The main motivation behind this step is to build a community around research tools like those presented in this thesis, and thus enable researchers to quickly evaluate their proposed ideas, learn about new obstacles that arise from these ideas, and to bring these tools out to the developers.

1.3 Structure of the Thesis

This thesis is structured as follows.

Chapter 2 - Example-Based Code Completion The suggestions made by current IDE's code completion features are based exclusively on the static type system of the programming language. As a result, often proposals are made which are irrelevant for a particular working

context. Also, these suggestions are ordered alphabetically rather than by their relevance in a particular context. In chapter 2, we present intelligent code completion systems that learn from existing code repositories. We have implemented three such systems, each using the information contained in repositories in a different way. We perform a large-scale quantitative evaluation of these systems, integrate the best performing one into Eclipse, and evaluate the latter also by a user study. Our experiments give evidence that intelligent code completion systems which learn from examples significantly outperform mainstream code completion systems in terms of the relevance of their suggestions and thus have the potential to enhance developers' productivity. This work is based on preliminary work published at Eclipse Technology eXchange Workshop (ETX) 2006 [BSM06], the International Workshop on Recommendation Systems in Software Engineering (RSSE) 2008 [BSM08], ACM Recommender Systems (RecSys) 2009 [WKB09], and has been finally published at Foundations of Software Engineering (FSE) 2009 [BMM09]

Chapter 3 - Mining Subclassing Directives To help developers in using frameworks, good documentation is crucial. However, it is a challenge to create high quality documentation especially of hotspots in white-box frameworks. This chapter presents an approach to documentation of object-oriented white-box frameworks which mines from client code four different kinds of documentation items, which we call subclassing directives. A case study on the Eclipse JFace user-interface framework shows that the approach can improve the state of API documentation w.r.t. subclassing directives. The results of mining sublassing directives from code have been published at Working Conference for Mining Software Repositories (MSR) 2010 [BMM10].

Chapter 4 - Detecting Missing Method Calls When using object-oriented frameworks it is easy to overlook certain important method calls that are required at particular places in code. In this chapter, we provide a comprehensive set of empirical facts that show that missing method calls may cause subtle problems that are hard to understand and localize. We propose a new system which automatically detects them at both software development and quality assurance phases. The evaluation shows that it has a low false positive rate (<5%) and that it is able to find missing method calls in the source code of the Eclipse IDE. This system has been published at the European Conference of Object-oriented Programming (ECOOP) 2010 [MBM10].

Chapter 5 - Learning Rankings for Code-Search Code-search engines leverage various search features including well known information retrieval ones such as tf-idf, cosine similarity, subword matching, etc. as well as structural properties of code such as used types or called methods, etc. But when leveraging dozens of such (potentially interdependent) features, manually tweaking these systems to produce optimal results becomes a very challenging task. We present an approach that learns ranking functions for code-search engines by leveraging user relevance feedbacks, especially user click-through data. We implemented a code-search engine

based on the features of Strathcona, which we consider to represent the state of the art of code search engines that employ manual tweaking of feature weights. We experimentally show that user click-through data can actually improve on manually tweaked systems even with a small number of collected user feedbacks. This approach to building personalized, self-updating code-search engines has not been published yet.

Chapter 6 - Related Work Recommender systems and mining approaches that leverage information hidden in large-scale code repositories exist in many flavors. This chapter presents tools and approaches related to our work grouped by the four tools presented in previous chapters: documentation mining and presentation, code-search engines, code completion systems, and bug detection tools.

Chapter 7 - Conclusions and Future Work The tools presented in chapters 2 to 5 have improved the state-of-the-art in practice and research alike. This chapter, however, critically reviews what has been achieved so far and what actions have to be taken to see the IDE 2.0 vision come to fruition. Furthermore, it outlines future steps for improving existing approaches and outlines further research opportunities.

2 Recommending Likely Method Calls for Code Completion

2.1 Introduction

The code completion feature of modern integrated development environments (IDEs) is extensively used by developers, up to several times per minute [MKF06]. The reasons for their popularity are manifold. First, usually only a limited number of actions are applicable in a given context. For instance, given a variable of type `java.lang.String`, the code completion system would only propose members of this class but none of, say, `java.util.List`. This way, the code completion prevents developers from writing incompilable code by proposing only those actions that are syntactically correct. Second, developers frequently do not know exactly which method to invoke in their current context. Code completion systems like that of Eclipse use pop-up windows to present a list of all possible completions, allowing a developer to browse the proposals and to select the appropriate one from the list. In this case, code completion serves both as a convenient documentation and as an input method for the developer. Another beneficial feature is that code completion encourages developers to use longer, more descriptive method names resulting in more readable and understandable code. Typing long names might be difficult, but code completion speeds up the typing by automating the typing after the developer has typed only a fraction of the name.

However, current mainstream code completion systems are fairly limited. Often, unnecessary and rarely used methods (including those inherited from superclasses high up in the inheritance hierarchy) are proposed. Current code completion systems are of little use especially when suggestions are needed for big (incoherent) classes with a lot of functionality that can be used in many different ways.

For illustration, consider the public interface of the class SWT[1] `Text`, which consists of more than 160 callable methods. Whenever querying the system for instances of type `Text` an overwhelming number of proposals are made—including all methods of `java.lang.Object` (e.g., `equals()`, `notify()`, or `wait()`)! But in fact methods like `wait()` are never invoked on an instance of `Text` from within the whole Eclipse codebase—after all, a software with several millions of lines of code. Recommending this method every time is rarely helpful to a developer and unnecessarily bloats the code completions, thereby counteracting the (last two) advantages

[1] SWT is a graphical user interface library.

of code completion systems.

The method `wait()` is not an isolated phenomenon. By analyzing the Eclipse codebase, we found that typically no more than five methods are invoked on SWT `Text` instances. Thus, a developer needs to pick the right 5 method calls from the list of 160 proposals. Even with Eclipse's capability to narrow down the list of proposals with each new character typed by the developer, the list still remains unnecessarily large.

Callable methods differ not only with respect to the frequency of their use; they also often differ with respect to the specific contexts in which they are typically used. Several factors such as the current location in code, e.g., whether the developer is currently working in the control-flow of a framework method [BSM08], or the availability of certain other variables in the current scope affect the relevance of a method.

For illustration, consider the situation of a developer creating a dialog window to gather user input using a text widget. Typically, text widgets have a two-phase life-cycle: (a) they are configured and placed in a visual container during dialog creation and (b) they are queried for user input after the dialog window is closed. These two phases are typically encoded in different methods of the dialog. While widget configuration takes place within `Dialog.create()`, reading the input occurs within `Dialog.close()`. Depending on the dialog method within which the developer needs suggestions for method calls on a text widget, different methods are relevant. Within `Dialog.create()`, relevant methods are text widget creation methods and setter methods for its visual properties; within `Dialog.close()` the `getText()` method is relevant.

The suggestions made by mainstream code completion systems are based exclusively on the information given by the static type system of the programming language. This seems too primitive given that the examples discussed above suggest that taking into account factors like the frequency of certain method calls in certain contexts might help to improve the accuracy of the proposals made by code completion systems.

Our hypothesis we evaluate in this chapter is as follows: The quality of suggestions and hence the productivity of software development can be improved by employing *intelligent code completion systems* capable of

1. filtering those elements from the list of proposals which are irrelevant for the current context, thus, disburdening a developer from knowing all (unnecessary) details of the API used, instead allowing him to focus only on API elements that are actually relevant.
2. assessing the relevance of every proposal (e.g., by using a relevance ranking), thus allowing a developer to quickly decide which recommendations are relevant for the task at hand

We propose *intelligent code completion systems* that learn from existing code repositories by searching for code snippets where a variable of the same type as the variable for which the developer seeks advice is used in a similar context. We have built three prototype code com-

pletion engines of this kind, each using the information contained in repositories in different ways. The first prototype uses the frequency of method calls as a metric to decide about their relevance. The second code completion uses association rule mining to search the code repository for frequently occurring method pairs. Finally, the last and most advanced code completion system recommends method calls as a synthesis of the method calls of the closest source snippets found. To build this system, we have modified the *k nearest neighbors* algorithm [CH67], a classical and efficient machine learning algorithm, to fit the needs of code completion. We call the resulting algorithm *best matching neighbors (BMN)* algorithm.

To evaluate our hypothesis we use a large scale evaluation process for recommender systems, sketched in [BSM08], along with standard information retrieval performance measures. By large scale, we mean that the system is evaluated with a test bed of more than 27,000 test cases. The evaluation results prove the efficiency of the learning code completion engines in general and of the *best matching neighbors (BMN)* algorithm in particular. In order to demonstrate that the *best matching neighbors (BMN)* code completion algorithm, which gets the best quantitative evaluation, has the potential to increase developer productivity, we show how it can be seamlessly integrated into the default Eclipse development widgets and evaluate its usefulness by means of a user study.

The remainder of this chapter is organized as follows. In Section 2.2 we present learning algorithms for code completion systems under investigation and especially our main contribution: the *best matching neighbors (BMN)* algorithm. Section 2.3 details on the data collection and static analysis applied to collect the evaluation data set. Section 2.4 presents the setup of the quantitative evaluation of the learning code completion systems which are compared against each other and against the default Eclipse code completion system in Section 2.5. Section 2.6 presents the prototype implementation of the BMN algorithm and its integration into the Eclipse IDE. Section 2.7 shows the results of a user study that used our tool in real-life situations. Section 2.8 concludes with a summary of the results.

2.2 Example-Based Code Completion Systems

The Eclipse code completion system (EcCCS for short) uses the type of a variable and suggests all callable method names based on this information. For this purpose, it only needs the information about the type hierarchy. This system serves as baseline for more intelligent code completion systems (CCS for short). Clearly, since it proposes all possible method names, the "correct" methods are always among its suggestions. The question is how many irrelevant recommendations are also made.

Our hypothesis, which we will test in the evaluation section, is that EcCCS makes too many irrelevant recommendations, since it does not take into consideration the context in which a particular object is used. This motivates our work on more intelligent code completion systems that learn how to use objects of a particular type in a particular context from code other developers

have written in similar situations.

Three New Code Completion Systems

We have implemented three code completion systems that learn from existing example code. These systems use different kind of information as the basis for completing calls.

A frequency based code completion system A plausible approach for intelligent CCS is to determine the relevance of each method based on the frequency of its use in the example code. The rationale for this strategy is: The more frequently a method has been used the more likely it is that other developers will use the same method. By implementing and evaluating such a frequency based code completion system (FreqCCS for short), we want to find out to which extent such a rather simple frequency based relevance ranking system can help developers to find the right completions.

An association rule based code completion Association rule mining is a machine learning technique for finding interesting associations among items in data [AIS93]. The problem of mining association rules is to find all rules $\texttt{A} \rightarrow \texttt{B}$ that associate one set of items with another set. Association rules have already been used in the context of recommendation systems for software engineering. Codeweb [Mic00] generates usage pattern documentation. Fruit [BSM06] makes interactive usage recommendations when developing software using frameworks. Both approaches do not handle recommendations on variable level. To the best of our knowledge, there is no publication that proposes to apply association rules in the context of code completion systems.

However, it is possible to use an example codebase to mine variable-scoped association rules and to use the context of a variable for determining the rules to select. In a nutshell, association rules can be used for code completion systems as well.

Consider the introductory example of using a `Text` widget again. The data mining process of association rules may identify two different usages: 1) Object creation which involves a constructor call and calls to several setter methods like `setText()`, and 2) object interrogation, when the object is queried for its state by calling getters like `getText()`. An association rule would be then "If a new instance of `Text` is created, recommend `setText()`". Another rule would be "If in `Dialog.close()`, recommend `getText()`".

We have implemented such an association rules based code completion system (ArCCS for short).

The Best Matching Neighbors code completion This new code completion system is based on a modification of the k-nearest-neighbor (kNN) machine learning algorithm [CH67].

To the best of our knowledge, kNN has not yet been used for code completion system. For this reason, we present this code completion engine in detail in the following section.

The Best Matching Neighbors Completion System

In this section we present the *best matching neighbors algorithm* (BMN for short). BMN adapts the k-nearest-neighbor (KNN) algorithm [CH67] to the problem of finding method calls to recommend for particular objects.

The KNN algorithm comes from the pattern recognition research. It is a classification algorithm. For instance, in the context of image recognition, given the image of a letter, the algorithm predicts the letter. The intuition of the algorithm is based on a common sense rule which can be pronounced as follows: to predict something according to some observation, let's find in one's experience a similar situation, and predict what actually happened at the end.

The KNN algorithm (applied as multi-label classifier) fits remarkably well to the problem of code completion: when a developer wants to complete code at the usage pattern level, she might start to search for code fragments very similar to the already written ones (e.g., using Google Code) and takes inspiration from the rest of the code. In a nutshell, our algorithm works as follows:

1. Extract the context of the variable;
2. Search for variables used in similar situations in an example codebase;
3. Synthesize method recommendations out of these nearest snippets.

More specifically, given a local variable `t` in the code under development, the system extracts and encodes the context as a feature vector. Based on this information, the algorithm searches the example base for object usages that are *close* to the usage being codified. From the close examples, the algorithm recommends the methods that are most likely to be used. The BMN code completion system (BMNCCS for short) is a tailoring of the KNN algorithm to the context of code completion. Our tailorings are as follows:

1. The way of extracting the context of the variable and encoding it as a feature vector;
2. The design of a meaningful and efficient distance measure between the code being written and the snippets of the example code base;
3. The selection mechanism of nearest neighbors;
4. The synthesis of method recommendations out of these nearest snippets.

Codebase Variables as Boolean Vectors

Given a local variable `t` in the code the developer is working on, BMN extracts three pieces of information: (i) the type of the variable, (ii) the method context the variable is used in, and (iii) a

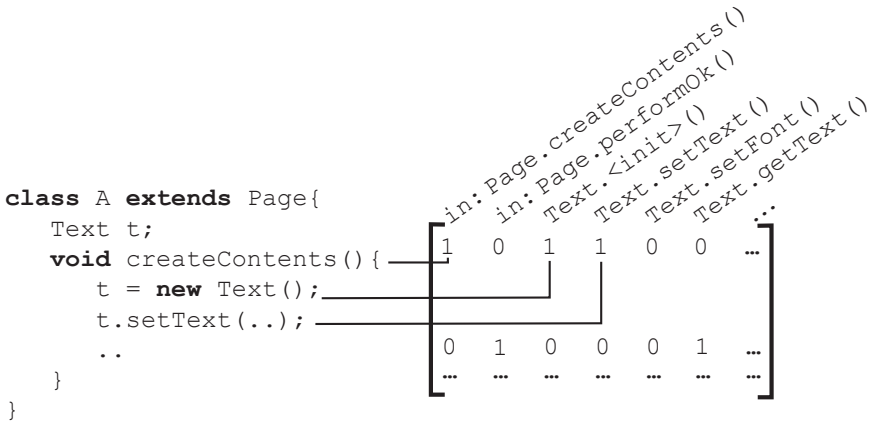


Figure 2.1: Encoding Framework Usages as Boolean Vectors

list of methods that have been invoked on `t` so far. This information is then encoded as Boolean vector.

Figure 2.1 illustrates this extraction and encoding process: Let's assume that there is a class `A` that extends `Page` and overrides `Page.createContents()`. Assume further that `A.createContents()` declares a variable of type `Text` and invokes the methods `new Text()` and `setText()` on it.

The right-hand side of Figure 2.1 shows the Boolean encoding of `t`. For each method call the column index is set to 1 if the corresponding method calls has been observed and to zero otherwise. In addition to the method calls some fraction of the Boolean vector is reserved to encode the method context, i.e., the framework method an object usage has been observed in. In the case of `t` for example, the column indicies of *Text.<init>()*, *Text.setText()*, and *in:Page.createContents()* are set to 1, whereas the remaining indicies become 0. This process is repeated for every variable that is observed in the control-flow of `A.createContents()` and every public method declared in `A`. This process ends up with an example code base represented by a set of feature vectors, i.e., a Boolean matrix for each variable type.

The above code snippet, however, shows a rather simple scenario. It does not contain any branches or calls to private methods nor does it pass objects to other objects or static methods. Real code is slightly more complex than this and several decisions have to be made that govern how object usages are extracted from code.

The most relevant one is how we deal with subsequent calls to private methods, static methods, and calls from nested to enclosing classes (as frequently done by anonymous listeners). For each class we run a static analysis for all its each public methods and built an interprocedural, context-sensitive callgraph that includes all method calls on `this` and declared in the same class

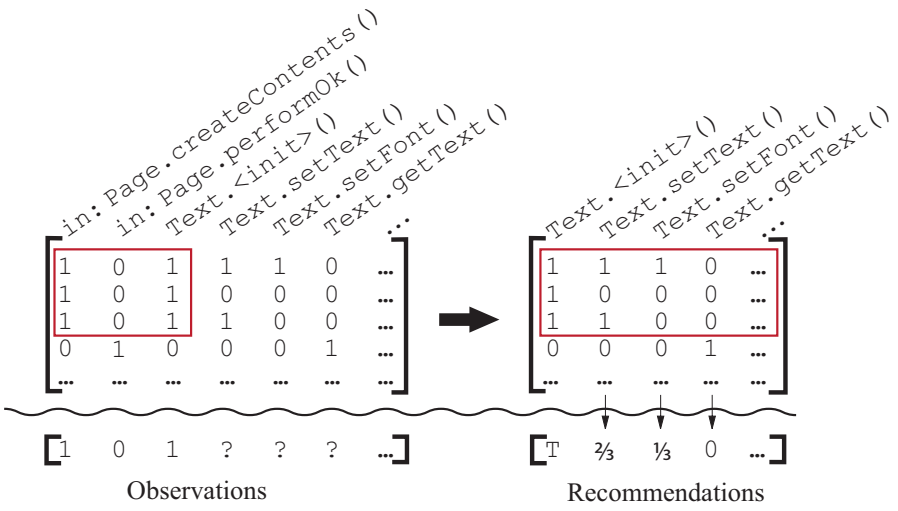


Figure 2.2: From Observations to Recommendations

or any of its superclasses as well as static methods declared elsewhere. In the case of calls from inner classes the call graph also included the calls to the methods of its enclosing class. Pointers to local variables and fields were kept intact over these method calls so that calls to same objects but in different methods could been tracked. In other words, we performed an inlining of all methods called from the entry point of our static analysis to get a rather complete object usage protocol.

The static analysis used for extraction was build on the WALA bytecode toolkit[2].

Let k be the number of variables extracted, i be the number of method contexts found in the training code base, and j be the number of all callable methods on all framework types. Then the usage Boolean matrix U has k rows and $i + j$ columns.

Since a variable can be in one method context only, the matrix U has the following property: for each observed usage v, there is a single context feature $f_t, 0 < t \leq i$ such that $v_f = 1$, i.e., each usage happens in the context of exactly one method; Such a usage Boolean matrix is shown at the left hand side of figure 2.2.

A Distance Measure for Code Completion

The KNN algorithm default distance measure is the Euclidean distance. First, we observe that since we are in a Boolean feature space, the Euclidean distance is exactly the square root of the

[2] Seehttp://wala.sf.net.

Hamming distance [3], as proved in the following equation.

$$
\begin{aligned}
\text{euclidean}(u, v) &= \sqrt{\sum_{i=1}^{n} (u_i - v_i)^2} \\
&\text{since } u_i, v_i \text{ are 0 or 1} \\
&= \sqrt{|u_1 - v_1| + \cdots + |u_n - v_n|} \\
&= \sqrt{\sum_{i=1}^{n} \text{diff}(u_i, v_i)} \\
&= \sqrt{\text{hamming}(u, v)}
\end{aligned}
$$

where $\text{diff}(u_i, v_i) = 1 \iff u_i \neq v_i$

This observation leads to a performant implementation of the code completion system since the calculation of Hamming distance is efficient (in comparison to the calculation of Euclidean distance).

Second, the KNN algorithm default distance measure is based on the full feature space. We show in the following that it does not fit to code completion.

The top left matrix in figure 2.2 is an encoding of an example code base as feature vectors. The bottom left corner of the figure shows a context (observation) given to the code completion system. The observation vector at the bottom left corner of figure 2.2 encodes the situation when the code completion is triggered in the context of overriding the method `Page.createContents()` and after an instance of `Text` is created. The ones and zeros in the context vector denote facts we are sure about. For instance, for the observed vector of figure 2.2, we are sure that we are within a method that overrides `createContents()` and that we are not within a method that overrides `performOK()`. However, we cannot say for sure whether the developer does not want to use `setText()`, or whether she simply has not yet used it, or whether she does not even know about its existence. Question marks in the vector encoding the context denote such uncertainties.

Using a distance on the whole feature space requires treating uncertainty as zeros. It turns out that in a real world feature space there are much more uncertain feature values (i.e., question marks in figure 2.2) than certain feature values. In such a case, the Hamming distance captures only the noise due to uncertainty. We encountered this problem empirically. This can be motivated by a probabilistic model.

Let us model the terms of the Hamming distance, $diff(u_i, v_i)$, as a random variable following the Bernoulli distribution, which has the following characteristics:

[3] The Hamming distance between two feature vectors is the number of positions for which the corresponding features are different.

$$\begin{aligned} E(\text{diff}(u_i, v_i)) =& p \\ \text{var}(\text{diff}(u_i, v_i)) =& p \cdot (1-p) \end{aligned}$$

Let us now consider the variance of the Hamming distance by splitting its terms in two groups: related to certain and uncertain information. Let also assume that these two groups are uncorrelated, then thanks to the Bienaymé formula, we can write:

$$\begin{aligned} \text{var}(\text{hamming}(u_i, v_i)) &= \text{var}(\sum_i \text{diff}(u_i, v_i)) \\ &= \text{var}(\sum_{i \in certain} \text{diff}(u_i, v_i)) \\ &+ \text{var}(\sum_{i \in uncertain} \text{diff}(u_i, v_i)) \end{aligned}$$

which, thanks to the central limit theorem, can be transformed to:

$$\begin{aligned} \text{var}(\text{hamming}(u_i, v_i)) = & \sqrt{n_{certain} \cdot p \cdot (1-p)} \\ & + \sqrt{n_{uncertain} \cdot p \cdot (1-p)} \end{aligned}$$

Since $p(1-p)$ is a constant and $n_{uncertain} \gg n_{certain}$ in our feature space, the Hamming distance is driven in this context by $n_{uncertain}$, i.e., it captures only the noise of uncertainty. Practically, this means that using the KNN algorithm on the whole feature space gives poor results for code completion. Our solution to this problem is to compute the distance on a partial feature space, based only on certain information of the observed context.

The first modification made to the initial KNN algorithm is to compute the distance based on *certain information* only. Given an incomplete vector (`icompVec`) encoding the context, the algorithm iterates through all rows of the Boolean usage matrix and determines the distance between each row and `icompVec`, based on the columns of `icompVec` that contain certain information. This is illustrated in figure 2.2, where only the three first columns of the matrix are used to compute the distance between the context and the example codebase. Eventually, there are three snippets close to the observation at an equal Hamming distance of 0: the three first rows of the example code base (marked by a red rectangle).

To sum up, the BMN system computes the distances between the current programming context and the example codebase based on the Hamming distance on a partial feature space.

The Selection Mechanism of Nearest Neighbors

In a standard pattern recognition context, when using the KNN algorithm, features are real numbers. Hence the distance between the observation and the database is also a real number that allows a complete ordering of neighbors.

In the context of code completion and with the distance measure described in the previous section, it turns out that a lot of neighbors are at the same distance of the input vector. It means that there is no complete ordering of the neighbors and it does not make sense to select the K nearest neighbors, whatever K is. It is very likely to have other neighbors that are exactly equally distant to the input observation. For instance, in the example of figure 2.2, there are three equally close neighbors.

To address this problem, we build equivalence classes based on the calculated distance and then take the set with the smallest distance. This set contains best matching neighbors; hence the name of the algorithm. It is used to compute the method call recommendations.

Synthesizing Recommendations

Since the KNN algorithm is a classification algorithm, it retrieves a class (for instance 'A', 'B', etc. in the context of letter recognition).

The BMN algorithm acts differently. Once the nearest snippets are selected, it computes the likelihood of the missing method calls based on their frequency in the nearest snippets, i.e., by counting the occurrence of each method call in the selected snippets and dividing it by the total number of selected snippets.

In the example in figure 2.2, the nearest snippets of the context are the three first rows. The BMN algorithm then looks at the methods called in each snippet and counts their occurrence. For instance, the method `Text.setText()` occurs in two out of the three records and `Text.setFont()` occurs only once. Method `Text.getText()`, however, is never observed, whenever the constructor call has been observed. When dividing these numbers by the total number of selected rows, we get the following recommendations: $\frac{2}{3}$ of the closest examples called `Text.setText()`, $\frac{1}{3}$ called `Text.setFont()`, and none of the examples matching the current observation called `Text.getText()`.

The BMN algorithm is parameterized with a threshold, called a filtering threshold. If the likelihood of a method call is higher than the filtering threshold, the methods are recommended and ordered by their likelihood. In the example of figure 2.2, considering a threshold of 50%, the BMN code completion system recommends only one method call to `setText()`; the method call has a likelihood of 66%.

To sum up, the BMN algorithm identifies method calls to be recommended to the user based on their frequencies in the *selected* nearest neighbors.

2.3 Gathering Example Data

To work properly, the algorithm needs static analyses that extract relevant API usage information from example code. This section describes how we obtained the codebase and discusses several key decisions we made regarding static analysis and their impact on the analysis results.

Example Data Collection

Example applications are fundamental to all mining approaches we present in this thesis. Alas, obtaining a sufficient amount of data is sometimes a very labor-intensive task. Consequently, approaches are needed that can collect as much data as possible with minimal effort.

Several approaches have been deployed in related work: Gruska et al. [GWZ10], for instance, analyzed 6,000 open source C projects taken from the Gentoo Linux distribution. Bajracharya et al. [BOL10] developed a crawler that steadily scans project hosting sites like Sourceforge and Google Code for changes. Whenever a project changes, the crawler downloads and rebuilds the projects and updates the analysis codebase. Other approaches, as for instance applied by Thummalapenta et al. [TX07], leverage code-search engines like Google Codesearch, Krugle, or Koders to obtain code examples for their analyses.

For the evaluation of our tools we followed a different approach. We applied them to several popular APIs of the Eclipse Rich Client Platform (Eclipse RCP), namely, Eclipse SWT, Eclipse JFace, and Eclipse UI. We chose these frameworks because of our own familiarity with these APIs and the availability of a huge example codebase: for Eclipse RCP thousands of extensions and standalone applications exist; many of which can be installed easily through the *Eclipse Marketplace*[4]. For our studies we developed a crawler that leverages the Eclipse Marketplace to automatically download and analyse available extensions. These extensions declare their dependencies similar to Maven inside so called manifest files which specify both the names of the dependencies and acceptable version ranges. Based on this information all required dependencies can be determined transitively, downloaded and analyzed in a fully automated process.

At the time of writing, the Eclipse Marketplace consists of more than 1100 extensions with in total more than 11 gigabyte of zipped example code. For evaluation of the three completion systems only a subset of the available data, Eclipse 3.4.2 Classic Edition, has been used.

Static Analysis for Call Recommendations

This section elaborates on how we perform our static analysis to collect object usage information from code.

We use the T. J. Watson Libraries for Analysis toolkit, WALA in short, developed at IBM Re-

[4] See `http://marketplace.eclipse.org`.

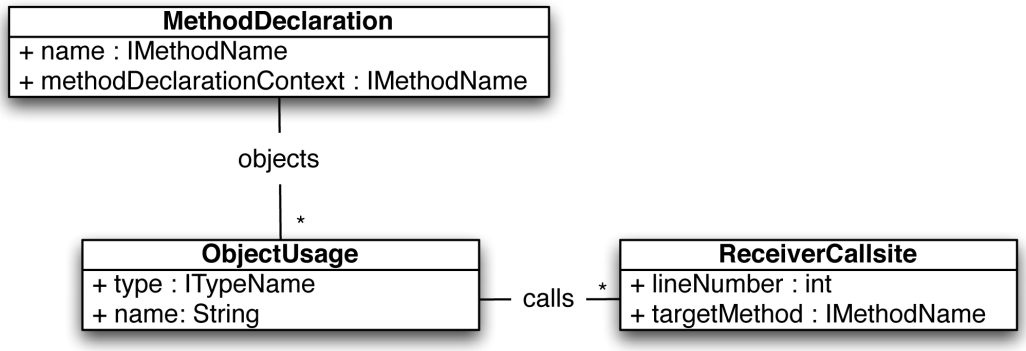


Figure 2.3: Object Usage Data Structure

search. WALA is a fully-featured bytecode analysis toolkit with built-in support for creating context-sensitive interprocedural callgraphs including points-to analysis, code slicing, and data flow analysis. It also provides various customization opportunities and supports partial program analysis.

Excellent explanations of the concepts of static analysis have been given elsewhere such as in the *Purple Dragon Book* [ALSU06]. The terminology we use in the following is borrowed from there. In this section, we will elaborate on key decisions we made for our static analysis.

Information extracted from source code

The completion engine presented in this chapter computes its recommendations based on three facts: (i) the type of the receiver (e.g. `org.eclipse.swt.widget.Text`), (ii) the methods that already have been invoked on the receiver, and (iii) the method declaration context in which the code completion was triggered.

Figure 2.3 shows an excerpt of the data structure used to store object usage data. For each method we analyze, a `MethodDeclaration` object md is created that stores the name of the method as well as the name of the method declaration context (explained below). There is one object usage per object obj_i referred to within md. The `ObjectUsage` of an object obj_i stores a list of `ReceiverCallsite`s, one for each callsite location at which obj_i is used as receiver for a method call. The `ReceiverCallsite` stores the location of this callsite (line number) as well as the name of target method as given in the bytecode.

While the notions of callsite and object usage are self-explaining, the term *method declaration context* warrants further explanation. As illustrated in Section 2.1, it makes a fundamental difference whether a `Text` widget is used inside a method that overrides `Window.create()` or `Window.close()`. For instance, in `Window.create()` a text widget is typically created and configured by calling several setter methods whereas in `Window.close()` its state is queried by

Defined Methods in MyDialog	Method Declaration Context	Entrypoint
`+MyDialog.<init>()`	`+Dialog.<init>()`	Yes
`+MyDialog.create()`	`+Window.create()`	Yes
`-MyDialog.createHeaderArea()`	`null`	No
`-MyDialog.createContentArea()`	`null`	No
`+MyDialog.performOK()`	`null`	No

Table 2.1: Example Method Declaration Contexts

calling methods like `getText()`. A method declaration context encodes this information, and thus, helps us later to recommend likely methods based on the information *where* an instance is used.

The declaration context of a method m is defined as follows:

Case 1: m overrides another method higher in the inheritance hierarchy. The declaring context of m is the topmost occurrence of m in the inheritance hierarchy. We consider interface methods as being overridable too. In the case that several adjacent topmost declarations exist, we select the topmost method declared in the superclass. If several adjacent interface methods are topmost, the one first observed is selected.

Case 2: m is a constructor. For constructors the notion of *overriding* does not hold since every constructor must call a constructor of its superclass. Following such a constructor call chain to its topmost constructor always leads to the no-arg constructor of `java.lang.Object`, which would be of limited use for the recommendation process. Thus, the declaration context of a constructor m is defined as the first super-constructor called from within its body.

Case 3: m does not override another method. In this case the method declaration context of m is `unknown`; it cannot be used to infer meaningful information for the recommendation process.

Figure 2.4 illustrates this concept by an example. The class `MyDialog` extends `Dialog` which in turn extends `Window`. `MyDialog` defines several methods: A public constructor[5], two public methods (`create()`, `performOK()`), and two private methods (`createHeaderArea()`, `createContentArea()`). Table 2.1 gives the method declaration contexts of each method of `MyDialog`: the declaration context of `MyDialog.<init>()` is `Dialog.<init>()`, the declaration context of `MyDialog.create()` is `Window.create()`, whereas the remaining methods have no declaration contexts since they do not override any other method declaration.

[5]Note, we consider constructors as methods with special semantics in our static analysis. When speaking of methods this typically includes constructors unless explicitly excluded or listed separately.

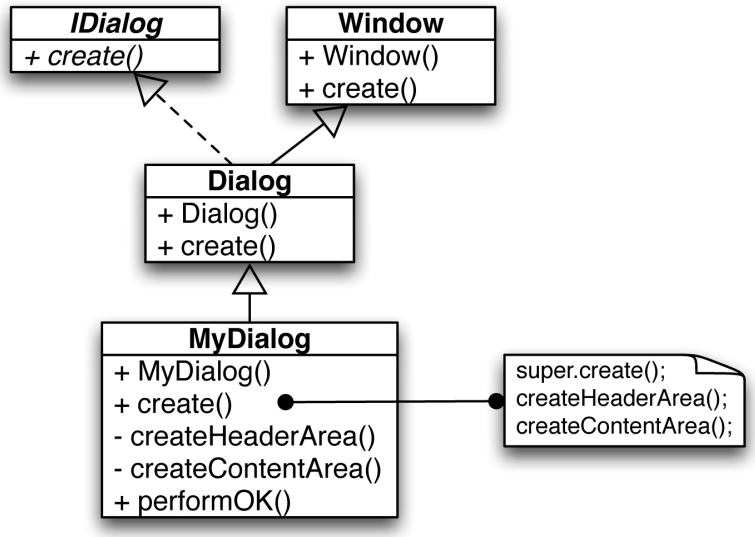


Figure 2.4: Example Class Hierarchy

Key Design Decisions of the Static Analysis

Doing whole-program analyses is infeasible when analyzing large and modular applications such as Eclipse. Consequently, strategies are needed that allow us to analyze only parts of the application while maintaining acceptable precision and performance. In the following, we elaborate on various adaptions we made to (i) the entrypoint selection for static analysis, (ii) callgraph creation, (iii) dealing with imprecise type information, and (iv) our handling of pointer aliases, pointer ambiguities, and control-flow decisions.

Entrypoint Selection

One decision we made was to analyze non-abstract classes only. This was motivated by the assumption that abstract classes leave important details of the implementation to the extending subclass. The risk of learning incomplete and thus misleading patterns from such code is high. But ignoring the code contained in abstract classes completely also seems wrong since subclasses may reuse some logic contained in the abstract base class. The interprocedural callgraph we build during analysis, is configured to follow calls into the base class to cover such cases as we shall see later.

For all concrete classes in the corpus we visit each method and look up its method declaration context. If a context is found, the method is marked as entrypoint. By construction, this includes all constructors and all methods that override at least one method declaration but excludes all

other methods such as static and private methods. The motivation not to use static and private methods as entrypoints is basically the same as for methods in abstract classes. We assume that static methods are rather utility methods that perform a certain small task, and thus, do not contain complete object usages. Thus, when using static methods as entrypoints for the interprocedural callgraph (and thus ignoring the callers of this static method) it's likely that the analysis finds a lot of incomplete usage patterns. For private methods we assume that they contain small, well modularized code fragments that (similar to utility methods) encode a specific subtask. We assumed that it's very likely that those object usage patterns we are interested in are often scattered over several methods. Consequently, using private methods as entrypoints (without considering their callers) is likely to result in incomplete, and thus, misleading usage patterns. The next sections provide several examples that further illustrate this issue in more detail.

Summarizing, as entrypoint for our static analyses it only makes sense to use methods that are part of an API defined by some superclass.

Callgraph Construction

After the entrypoints are determined, the callgraph is constructed. Given a method with a method declaration context, WALA creates an interprocedural callgraph using this method as entrypoint. For each callgraph a heap model is created to track all objects observed during the abstract interpretation of the code (referred to as *instance keys* in WALA) and their aliases (referred to as *pointer keys*). This information is used later to construct the list of all objects that may potentially appear as receiver at a given callsite.

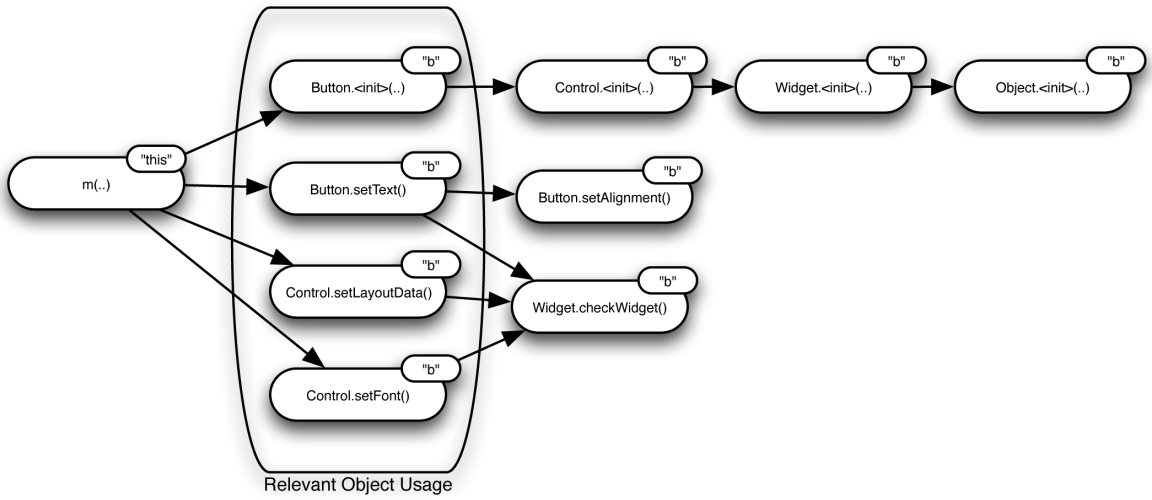
Building callgraphs in this way, however, would waste a lot of resources. To learn how developers use APIs we are only interested in a subset of method calls—those a developer actually has to take care of. Figure 2.5 illustrates this by an example: 2.5a shows a code snippet that illustrates how developers typically instantiate and configure an instance of `Button`. The callgraph this sequence would generate, is depicted in 2.5b[6]. It contains many more nodes than actually relevant to learning how to use the API: relevant are only the calls that happen in `m()`,i.e., the calls the developer has to write. The remaining calls that happen internally, i.e., behind the public interface of `b` are irrelevant for the developer. Following these irrelevant calls is resource intensive and time consuming, and slows down the whole analysis process. Consequently, we skip those calls in the callgraph construction phase to safe memory and computation time.

However, as we shall see later in this section, object usages are frequently scattered over several methods. This requires us to create a minimal interprocedural callgraph starting from m in order to obtain as complete as possible usage patterns. In the following, we will explain a set of tweaks we did to make the analysis suitable to learn object usage patterns from code with reasonable performance.

[6]Note, for brevity we already skipped method calls to other objects such as `java.lang.String`.

```
1  public void m(..) {
2    Button b = new Button(..);
3    b.setText(..);
4    b.setLayoutData(..);
5    b.setFont(..);
6  }
```

(a) Example of an desireable object usage to learn from.



(b) Effective callgraph following all calls inside `b`. Calls to other objects such as strings etc. in `b` omitted.

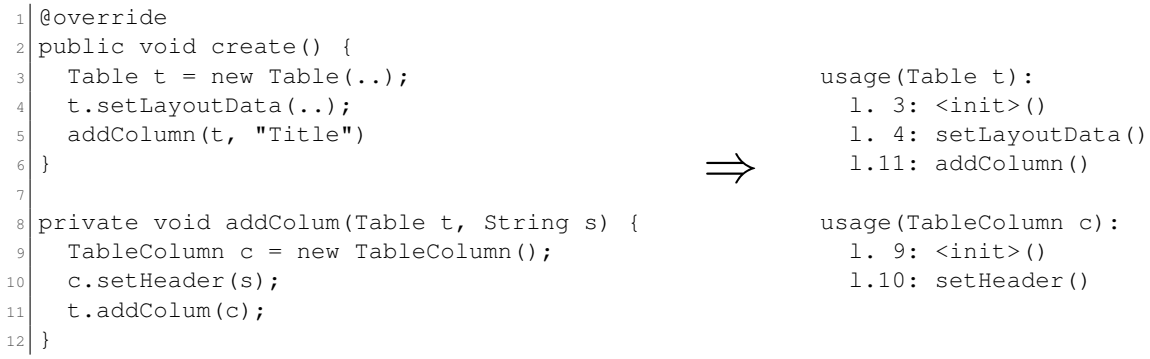
Figure 2.5: Example of a callgraph created with WALA's default settings.

Follow calls to static methods. In the previous section we discussed why static methods shouldn't be used as entrypoints for static analysis. However, not considering these methods for callgraph construction also seems inappropriate as the code example below illustrates. In this example, ignoring the call to `Dialog.applyDialogFont()` would result in a missed call to `Composite.setFont()`, and thus, leads to an incomplete object usage observation.

```
1  public class MyDialog {
2    @Override
3    public void create() {
4      container = new Composite(..);
5      container.setLayoutData(..);
6      Dialog.applyDialogFont(container);
7    }                                                    usage(Composite container):
8  }                                                  ⇒       l. 4: <init>()
9                                                             l. 5: setLayoutData()
10 public class Dialog {                                      l.14: setFont()
11   public static void applyDialogFont
12                       (Control c) {
13     Font f = Resources.getDialogFont();
14     c.setFont(f);
15   }
16 }
```

However, including static methods into the callgraph is not a good decision at all times. Frequently, utility methods perform some additional calls to query the state of an object before taking any actions. For instance, let's assume that `applyDialogFont()` may have checked that the composite has no font configured by verifying that `c.getFont() == null`. In that case, the observed usage pattern would include an additional (safety-check) call to `getFont()` which is typically not necessary since a developer knows exactly the state of `c`. As a result the system learns an unusual usage pattern for `Composite`s. From manual inspection we concluded that following calls to static methods more often improves the observed usage patterns than it harms. However, in the long run improved recommender systems may take into account parameter callsites and refine their proposals based on this information. This may alleviate this problem.

Follow self-calls declared in same class. For callgraph creation we follow all self-calls (i.e., calls to `this`) if the target method is defined in the same class as the entrypoint (e.g., private methods). This allows us to observe object usages scattered over several methods in the same class. The code snippet below gives an example of such a scenario along with the object usage information extracted from this snippet. It shows that the object usage of `t` is scattered over two different methods, namely `create()` and `addColum()`. WALA's built-in heap model and points-to analysis allows us to keep track of this scattered usage and associates all calls to the appropriate objects.



Follow self-calls declared in superclasses. Following calls into the superclass is similar to following calls to static methods, and thus, following them has the same benefits and drawbacks. One difference to static methods is that calls to superclasses are potentially polymorphic and thus requires proper method dispatching. The code snippet below gives an example of such a polymorphic call. `MyDialog.create()` calls the super-implementation `Dialog.create()` which calls the polymorphic `adapt()` method. At runtime, the virtual machine would dispatch this call to `MyDialog.adapt()`, and thus, the static analysis should do so too.

Our analysis correctly dispatches these calls to the appropriate method in the hierarchy, and thus, is able to find object usages scattered across the inheritance hierarchy too.

```
1  public class MyDialog extends Dialog{
2    @override
3    public Control create() {
4      Control c = super.create()
5    }
6
7    protected adapt(Control c){
8      c.setFontSize(..);
9      c.setForeground(..);
10     c.setBackground(..);
11   }
12 }
13
14 public class Dialog {
15   @override
16   public Control create() {
17     Composite c = new Composite()
18     c.setLayoutData(..);
19     adapt(c)
20     return c;
21   }
22
23   protected adapt(Control c){
24     // do nothing
25   }
26 }
```

⇒

```
usage(Composite c):
  l.17: <init>()
  l.18: setLayoutData()
  l. 8: setFontSize()
  l. 9: setForeground()
  l.10: setBackground()
```

Follow calls to enclosing classes. Java allows developers to specify *nested* classes, i.e., classes defined inside another class or method declaration. Each instance of these classes has a reference to an enclosing instance (i.e. an instance of the enclosing class), except for local and anonymous classes declared in static context. Hence, a nested class can implicitly refer to instance variables and methods of the enclosing class.

This feature of the Java language is quite often used by developers. One common example of such a usage are listener classes. Here, developers register an anonymous listener class on, for instance, an UI widget. The UI framework calls this listener if an appropriate event occurs. Often the listener does not contain the code to handle this event but delegates to a method of the enclosing class.

In this case we make an exception to the *Follow no objects other than* `this` rule. Otherwise we would loose interesting and relevant information. Thus we configured the static analysis to follow calls from inner to their enclosing classes and the other way around.

The code snippet below illustrates a situation where following calls to the enclosing class makes sense. The method `SelectionAdapter.widgetSelected()` serves as entrypoint for the static analysis. Internally, the method delegates its work to `handleClick()` which invokes `getSelection()` on an instance of `Button`. Not following the call to the enclosing class

would lead to an empty, and thus, incomplete usage pattern for `Button`s.

```
1  public void create() {
2    // ..
3    b.addSelectionListener(
4      new SelectionAdapter(){
5
6        @Override
7        public void widgetSelected(Event e){
8          handleClick();                                    ⇒    usage(Button b):
9        }                                                          1.14: getSelection()
10     });
11 }
12
13 private void handleClick() {
14   boolean v = b.getSelection();
15   // ..
16 }
```

Miscellaneous Adaptions

Dealing with conditional branches. Sometimes the actual object usage depends on several conditions at runtime. For instance, based on some object state a method may or may not be called. We ignore these potentially different execution paths and merge all conditional object usages into a single object usage:

```
1    Widget w = ..
2    if(!w.isDisposed()) {                                    usage(Widget w):
3      Control[] children = w.getChildren();          ⇒         1. 2: isDisposed()
4      // ..                                                    1. 3: getChildren()
5    }                                                         1. 6: dispose()
6    w.dispose();
```

Merging different execution paths into one object usage may seem odd at first since we then recommend methods that may only be executed in specific situations. For instance, given the example above, the information is lost that `w.getChildren()` is only called if `w` was not disposed before.

We think that it is essential for a developer to know that these calls occur typically together in the same method, and that a developer can easily make sense out of the information even without the exact control-flow or sequence information. Thus, it seems reasonable to ignore control-flow and sequence information for code completion (but not for checking!).

However, making control-flow information more explicit may also help developers to understand relations between method calls more quickly. Later versions of our analysis will collect this

information and consider it in the recommendation process.

Dealing with pointer ambiguity. If a receiver callsite can be executed with several different objects, the call is added to the object usage of each of them. For instance, given the conditional statement below, this would result in a call to `Button.setText()` for the instances `c` and `d`. Obviously, adding this call to both object usages is wrong since at runtime only one object receives the call to `setText()`. At analysis time, however, it is not decidable and thus both objects get this call.

Please note that aliases (i.e., several pointer keys point to the same instance key) are resolved by WALAs points-to analysis.

```
1  Button b = null;
2  if(cond) {
3      b =c;
4  } else {
5      b = d;
6  }
7  b.setText();
```

⇒

```
usage(Button c):
  l. 2: setText()

usage(Button d):
  l. 2: setText()
```

Dealing with method returns. We assume that repeated calls to a method return different objects. For instance, we assume that repeated calls to `c.getFont()` as shown in the code snippet return different instances of `Font`.

```
1  Composite c = ..
2  Font f = c.getFont();
3  f.getFontData();
4  f = c.getFont();
5  f.getGC();
```

⇒

```
usage(Composite c):
  l. 2: getFont()
  l. 4: getFont()

usage(Font f):
  l. 3: getFontData()

usage(Font f):
  l. 5: getGC()
```

This is motivated by the fact that we cannot know a priori which object a method is going to return; partial program analysis makes this impossible. Thus, some heuristic is needed.

A more sophisticated strategy could have been chosen such as treating methods with no arguments to always return the same instance and methods with arguments to return always different instances. However, as we shall see below, there is no clear *best* solution. We identified four different cases: two cases that support the assumptions above and two counterexamples which violate them.

Case 1: Methods that take one or more arguments and return different objects. An example of such a method is `List.get(int)`. Such methods are likely to return different objects each time they are invoked.

Case 2: Methods with no arguments that return the same instance. *Getter* methods belong into this group. For them, it seems reasonable to assume that repeated calls always return the same instance.

Case 3: Methods that take arguments but always return the same instance. Methods of *Builder* classes[7] are a counterexample to case 1. These methods always take an argument but always return `this` to allow developers to save a few keystrokes.

Case 4: Methods with no arguments that return different objects. The well known `Iterator` class with its `Iterator.next()` method belongs into this group. Here, each repeated call to `next()` always returns a different object. This example acts as a counterexample to case 2.

These examples show that without further analysis of the called method it is undecidable whether a method returns the same or different objects. Thus, we implemented the simplest solution to return a new instance key for each invocation. However, this decision may have some impact on the quality of the patterns which we describe below using a `StringBuilder` code example:

```
1  new StringBuilder()
2      .append(s1)
3      .toString();
```

$\Rightarrow$

```
usage(StringBuilder ?):
  1. 1: <init>()
  1. 2: append()

usage(StringBuilder ?):
  1. 3: toString()
```

In this example, our analysis finds two objects with different method calls. As a result, we would fail to find the pattern "after calling `StringBuild.append()` one has to call `StringBuild.toString()` to obtain the final string". Further experiments will be conducted to determine the impact of this design decision.

Exception Handling. Throw statements as well as try/catch/finally blocks are considered as normal instructions and code blocks respectively. We deem potential exceptional edges in the control-flow graph as irrelevant for learning API usage patterns, and thus ignore them completely.

[7] We refer to the Builder pattern as described by Joshua Bloch in Effective Java 2, but not the GoF design pattern.

2.4 Evaluation

In the following, we evaluate the Eclipse code completion and the three code completion systems (CCS) introduced in Section 2.2.

2.4.1 Evaluation Data Set

We measure the ability of the code completion systems under investigation to predict method calls on objects of the Standard Widget Toolkit (SWT) [Ecl06], a graphical user interface library. The SWT library has been chosen for several reasons:

- There are an important number of open source programs that uses SWT. These programs constitute the "knowledge" base from which BMN, FreqCSS, and ArCSS systems can learn.
- Also many frameworks rely on the SWT framework. Developers must know the concepts of both frameworks, e.g., where to place the method calls to SWT instances in the context of the other framework etc. As discussed in the introductory example, the knowledge about the currently overridden framework method can be leveraged by a code completion system to improve the proposals.
- SWT is used in many projects, hence a code completion system for it could be immediately beneficial to a great audience of developers.

We collected the whole Eclipse 3.4.2 codebase for conducting the experiment. This codebase grounds both the training data for code completion system and the test data for creating the evaluation scenario.

2.4.2 Evaluation Scenario

The code completion systems under investigation are evaluated with several thousands of queries following the automated evaluation process presented in [BSM08]. In the following, the main steps of the evaluation process and their rationale are briefly summarized.

1. The initial data set is randomly split in two parts: 90% grounds the initial knowledge of the code completion system, the remaining 10% are called test data and are used to create evaluation scenario. This step ensures that the system is not evaluated with queries for which it has the *exact* information to answer them. This is a crucial requirement for evaluating machine learning systems [Bis06].
2. For each Boolean feature vector of the test data, some method calls (i.e., some 1s) are removed. The resulting degraded vector will be used as a query to the system. The removed method calls constitute an expectation, similar to the expected value of a unit test case. This step simulates a real programming situation, where the developer needs assistance

after she has already written some code. We remove half of the method calls of the initial feature vector thus simulating the situation where the developer has already performed half of the job. This choice is a pragmatic trade-off between: (a) providing information that enable to make intelligent context-dependent predictions, and (b) withholding information from the system to complicate the task of prediction.

The dataset we used for this evaluation contained more than 27,000 examples usages, and the number of method calls in these examples varied from 2 up to 46 methods calls.

The randomization in step #1 ensures that, on an average, the query distribution between small and hard tasks reflects the real distributions of such situations in the codebase.

3. For each query, each code completion system is asked to return a prediction of the missing method calls.
4. For each prediction and the corresponding expectation, evaluation metrics defined in Sec. 2.4.3 are calculated.

To achieve a standard machine learning evaluation, we repeat the evaluation process 10 times; this is known as 10-fold cross validation [Koh95]. Since the initial dataset consists of 27,000 records, the code completion systems are actually evaluated against $(27,000 * 0.1) * 10 = 27,000$ real world programming queries.

2.4.3 Evaluation Metrics

The objective of this evaluation is to figure out to which extent the four code completion systems under investigation, EcCCS, FreqCCS, ArCCS, and BMNCCS, can: (i) identify relevant methods, i.e., methods that are actually used at the end by the developer to complete her code, and (ii) weed out irrelevant proposals, i.e., those that did not make it into the final code, without sifting out too many of the relevant methods.

The measures we use for assessing the performance are precision, recall, and the F1-measure. These measures are commonly used for evaluating the performance of information retrieval systems like (web) search engines or generally any kind of recommender systems. The intelligent code completion systems that are subject of this evaluation are basically recommender systems in a particular domain.

To explain the meaning of the metrics consider the code snippet depicted in Listing 2.6. This listing shows an incomplete usage of a `Text` widget on the left hand side and the final code after as it should be. Assume that a developer who knows exactly the methods to call on `t` but nevertheless pressed *Ctrl-Space* in line 2 on the left code snippet to query the code completion system. Assume further that the system returned three recommendations: `setText()`, `setLayoutData()` and `addListener()`. The developer investigates these recommendations and decides to apply the first two proposals but to ignore the third one. Furthermore,she decides to add a new, not recommended call to `setFont()`.

```
1 Text t = new Text(..);
2 t.|
```

(before)

```
1 Text t = new Text(..);
2 t.setText(..);
3 t.setLayoutData(..);
4 t.setFont(..);
```

(after)

Figure 2.6: Code Snippet Before & After Code Completion

Let us measure the performance of the CCS for this query. The developer added three method calls to the code. Two calls were predicted correctly by the system (`setText()`, `setLayoutData()`); the third call (`setFont()`) was not proposed. Thus, the system proposed 2/3rd of the calls the developer actually needed. The recall of the CCS in this case is thus 2/3. More formally, recall is defined as the ratio between the relevant (correct) recommendations made by the system for the given query and the total number of recommendations that it *should* have made:

$$Recall = \frac{Recommendations_{\ made \cap relevant}}{Recommendations_{\ relevant}}$$

Clearly, a system that recommends all possible methods would always achieve great recall. But if only one out of a hundred recommendations is actually needed, the system is not really intelligent. This is where the second interesting question, namely, how many false recommendations the system made, comes into place. In our example, 2/3rd of the recommendations actually made it into the final code. This is the precision that the CCS achieved for the query. Formally, precision is defined as the ratio between relevant recommendations made and the *total* number of recommendations made by the system for a particular query:

$$Precision = \frac{Recommendations_{\ made \cap relevant}}{Recommendations_{\ made}}$$

Both values together allow to assess the quality of a CCS for a given query. In order to summarize how the systems perform on several queries the precision and recall values are averaged over all queries using *micro-averaging* as described in [tre90]. After averaging, the performance of each completion system is boiled down to a pair of numbers. However, with two numbers characterizing the "goodness" of systems, the question arises when a system is “better” than another. It might be the case that one system has a very high recall but a very low precision. Another system might have only a medium recall but also a medium precision. The F-Measure[vR79] has been widely accepted by the research community as a means to correlate precision and recall by computing their harmonic mean:

$$F = \frac{(1 + \beta^2) \cdot precision \cdot recall}{\beta^2 \cdot precision + recall}$$

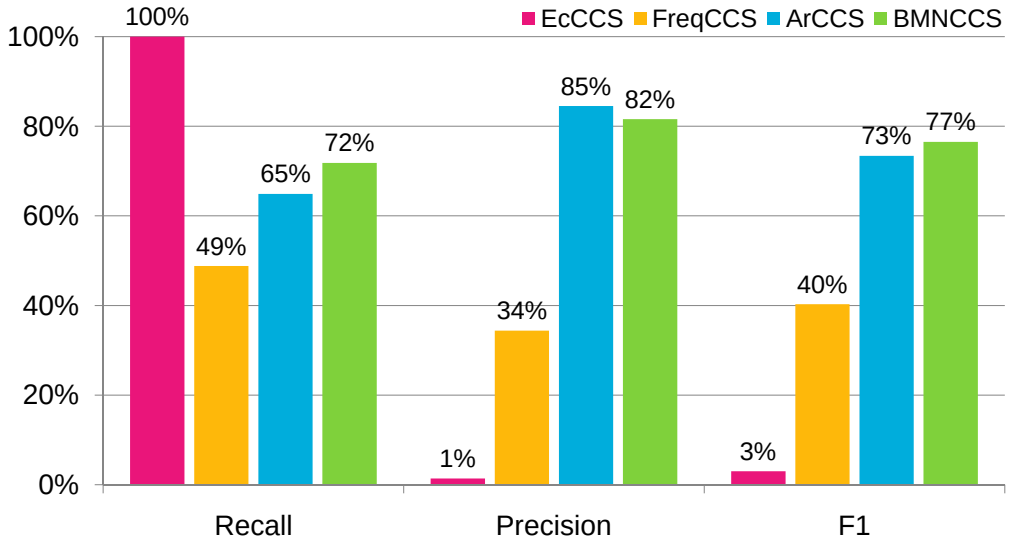


Figure 2.7: Performance of *EcCCS*, *FreqCCS*, *ArCCS* and *BMN*

The F-Measure allows to emphasize either recall *or* precision by accordingly assigning the β parameter. For our evaluation, we equally weight precision and recall ($\beta = 1$). The resulting formula is called the F1 measure [vR79].

In addition to summarizing the performance in a single number, the F1-Measure has another important role in our evaluation. Each CCS under evaluation has a set of specific parameters (e.g., minimum likelihood thresholds etc.) that affect its performance. To produce comparable results we need to minimize the effects of accidental parameter guessing. For this purpose, we used the F-Measure as an optimization criterion for each CCS: We run several dozen experiments for each CCS to figure out the best parameter settings that maximized the F-Measure.

2.5 Results

Figure 2.7 summarizes the results of the evaluation. It shows the recall, precision and F1 levels reached by each code completion system. As already mentioned, all systems were tuned to maximize F1 with the corresponding algorithm parameters (e.g., the filtering threshold of FreqCSS and BMN), except EcCCS which has no parameter to be tuned.

The EcCCS performs worst. As expected, it proposes all relevant methods (i.e., it has a recall of 100 %), but its low precision shows that 99 % of its recommendations were not what the developer actually needed, thus false proposals in our setup. This result clearly suggests that current code completion systems provide a large room for improvements.

The frequency-based CCS achieves significantly higher precision than the EcCCS; yet, its over-

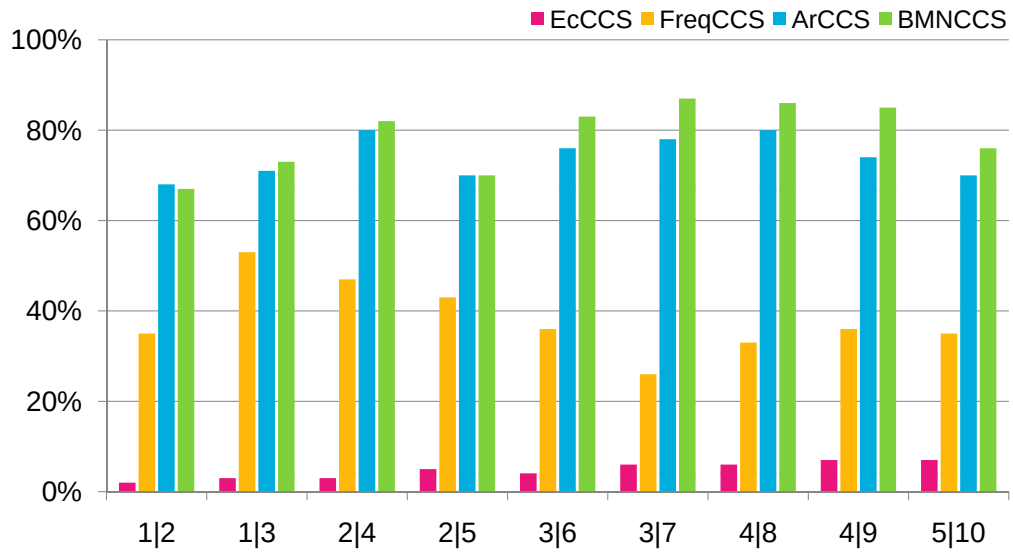


Figure 2.8: F1 per Expected Method Call Recommendations

all performance is disappointing. Only one out of three proposals was actually correct. Furthermore, the system only finds half of all relevant method calls. Even if the underlying idea would seem intuitive, we doubt that code completion systems based purely on the frequency of method calls would meet the developer expectations.

The rule-based approach significantly improves both values: It finds 65% of all relevant methods and, what is really interesting, 85% proposals were correct. This shows that the proposals made by the ArCCS are highly reliable and developers can trust recommendations made by this system.

Finally, the BMNCCS achieves a 10 % higher recall than ArCCS and a slightly lower precision. In total, from all evaluated code completion systems the BMN system achieves the highest F1 value, thus performs best with respect to this evaluation setup.

Figure 2.8 enables us to refine these conclusions by decomposing the F1 performance along the different query settings. The results are grouped by the total size of method calls contained in the example snippets. For illustration of this figure, consider the second group of bars labeled with “1|3”. The bars summarize the performance of the code completion systems for the following queries: Given one randomly selected method call, the system was asked to return the two missing calls; in total the example snippet contained 3 calls. This figure lets us draw the following conclusions.

First, the overall performance of the evaluated systems stays in the same order of magnitude. Since the F1 measure stays around 80%, we can say that the BMN code completion system is able to predict method calls based on large usage patterns involving dozens of method calls.

Second, the gap between ArCCS and BMN widens when the completion task size increases. While ArCCS and BMN are roughly equivalent for predicting method calls based on short usage patterns, BMN is significantly better for large usage patterns.

To sum up, the best code completion according to this evaluation is the BMN system. It finds 72% of all relevant method calls and 82% of the recommended method calls are actually correct. The performance results of this new system give encouraging evidence that intelligent code completion systems are not a Utopia.

Discussion

In the following, we discuss the the generalizability of the evaluation results.

- Our evaluation deals with Eclipse code, i.e., our context (encoded as feature vectors) contains information about the object-oriented context of a variable. For instance, the context might say that the current variable is in a class that extends `DialogPage`, and in a method that overrides a specific method of `DialogPage`, say `setControl`. This additional information may improve the performance of the system, and weaken the generalizability.
- Our evaluation deals with SWT classes. It seems that SWT classes contain more logic and more complex usage patterns than most of the classes of the Java Runtime Environment (e.g., `java.net.URI`). Again, the generalizability of the very high precision and recall might be discussable for low-level APIs, such as default Java API.
- Evaluation queries are built by randomly selecting method calls from real variables of the codebase, without taking into account the order. This may lead to unrealistic queries. For instance, if one considers the ordered calls on a variable, we may randomly drop out the middle third of the calls. In this case, it is unlikely that a developer would write code this way and that the code completion system would be queried with this context. We are in the process of obtaining empirical results to figure out: 1) whether our approximation increases or decreases the overall precision or recall and 2) to what extent the order of method calls matters (for instance, methods that configure an object may be called in different orders).

2.6 Eclipse Integration

In this section, we present our prototype integration of the BMN engine into the Eclipse IDE. We decided to integrate BMN since it was the engine with the best performance according to the evaluation. The goal for building the prototype was to enable the user study presented in Section 2.7 to give us insight on the practical relevance of code completion systems capable of learning from example code. We present the prototype by elaborating on three main differences between of our prototype to the default Eclipse code completion system. Our statements are illustrated

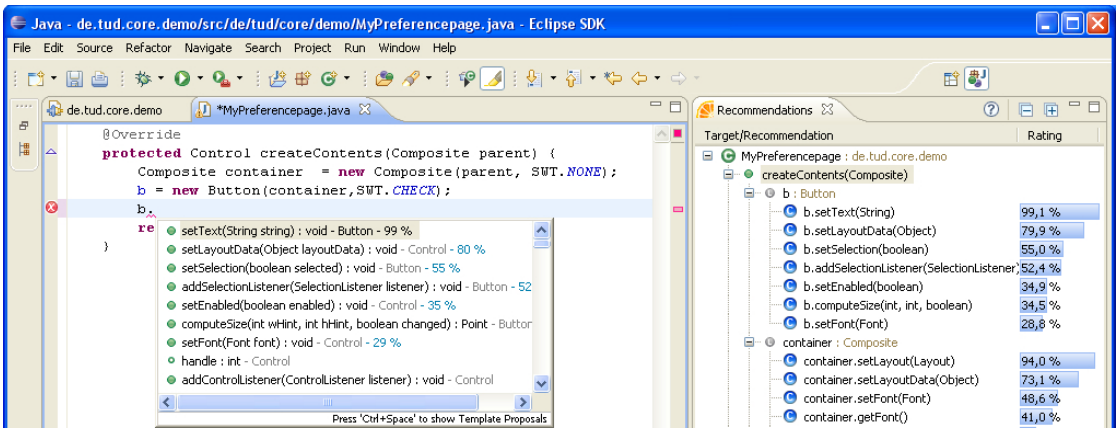


Figure 2.9: Integration of our Example-based Code Completion into Eclipse

by figure 2.9, which shows a screenshot of our prototype.

Size of the method name list First of all, an advanced code completion system substantially reduces the number of methods suggested to those with high probability of being relevant in a particular context. We agree with Robbes and Lanza[RL08] that we cannot expect the programmer to scroll down a huge list of method names in order to find the good one. This would introduce a cognitive context switch in the programmer activity.

Our system filters recommendations based on a threshold on the confidence value of the recommendations. The higher the threshold, the lower the number of recommendations in the code completion widget. Based on our experience in using earlier versions of the prototype in day-to-day development, the default filtering threshold is set to 25%.

Confidence value The default code completion widget is enhanced with the confidence values of the recommendations (for instance, consider in the screenshot the 99% confidence value of method `setText`). This value is an indicator for the developer of what to do next. We use the following guidelines to interpret the confidence values.

A very high confidence value ($\sim$ >90%) indicates that unless the developer explicitly knows that she is implementing a boundary case, this method has to be called.

A high confidence value ($\sim$ >50%) means that this method can probably be called. From the programmer view point, the interpretations can be:

1. “I know the usage pattern I am implementing, I know this method and I am happy the tool agrees with me”. *The tool may strengthen the confidence the programmer has in her knowledge and in her code.*

2. "I did not plan to use this method. Maybe I am wrong in using this class. Let's read the corresponding method documentation". *Our code completion system can be used as a knowledge watchdog. The difference between the developer's plan and the confidence value is a kind of warning.*
3. "Hey, what is this method? I don't know it. Let's read the corresponding method documentation; it may be important." *Our code completion system helps the programmer to discover new features. It assists her in improving her knowledge.*

A low confidence value ($\sim$ <50%) is a reminder of potential methods to call in special usage patterns. Since special usage patterns are rarely used, the programmer forgets them easily. In this case, our code completion system can act as a reminder.

Note that the thresholds for interpreting the confidence values are fuzzy. When one gets used to work with a code completion system with confidence values, each programmer tends to tune these values w.r.t. the framework used and to her own experience. Note also that the validity of these guidelines depends highly on the application of the single responsibility principle [Mar03] in the class being used. If a class can be used in several different ways, i.e., it violates the single responsibility principle, the confidence values computed by our code completion system automatically decrease and the thresholds loose their relevance.

Dedicated view We propose to add a new view to Eclipse, dedicated to code completion, shown on the right hand side of figure 2.9. In contrast to the code completion window (which shows recommendations for a single variable only) this view provides a summary of all recommendations (including their relevance) available for the given class and its variables. This view serves as a browser, allowing a developer to search through all recommendations and supports her to understand the instance usage concepts.

Conclusion Our new code completion system is tightly and seamlessly integrated into the Eclipse IDE. It does not break the usual programmer way of working: Advanced code completion is still available with `Ctrl+Space` (the default keyboard shortcut that every Eclipse programmer knows). Code completion still works without noticeable delays. Computing the distances between the current context and the codebase is not an issue in terms of performance. The improvements compared to the existing Eclipse code completion widgets are threefold:

1. irrelevant methods are removed,
2. method recommendations are always given together with a confidence value, and
3. we introduce a new view dedicated to code completion.

The latest prototype can be installed from `http://eclipse.org/recommenders/download/`.

2.7 User Study

In Section 2.4 we demonstrated the effectiveness of *intelligent code completion systems* in identifying relevant methods for a given context using a large-scale automated evaluation approach. In this section, we report on the user feedback of 10 experienced Java programmers who tested the BMN system for its usefulness. The user study consists of three phases: (i) completing a predefined task, (ii) filling a questionnaire after the programming task, and (iii) giving an interview one or two days after returning the questionnaire.

2.7.1 Setup

For the user study each subject had to develop a small graphical user interface using the Eclipse SWT UI Framework. This user interface required several graphical SWT widgets like `Button`s, `Text` fields, `Combo` boxes etc., as well as some interactions between these widgets in response to some user interactions like enabling and disabling of widgets etc. The appearance of the interface and the dynamic behavior was specified with an annotated screencast.[8] Furthermore, we provided the basic application code as a downloadable Eclipse project to free developers from Eclipse-specific tasks unrelated to the user study as such.

Overall, ten subjects participated in the user study. All subjects had several years of experience with the Java programming language and were familiar with Eclipse and its code completion system. Half of the subjects did not have any experience with the SWT framework; the other half had at least several months of experience in developing SWT applications and, thus, were acquainted to the concepts of the SWT framework.

2.7.2 Results

For the questionnaire and interviews we asked the subjects to answer several questions concerning the usefulness of the proposed system and to give their personal rating to these questions. For the personal ratings the subjects had to choose one of *Strong Agree (++)*, *Weak Agree (+)*, *Weak Disagree (–)*, *Strong Disagree (––)*, or *No Answer (∘)* if none of the previous was deemed appropriate. The significant questions from the questionnaire are given below (Questions 2 and 5 are summaries from interviews as these questions were not based on ratings).

1. Did the system propose relevant method calls? (5++ / 4+ / 1 ∘) This question aims to identify whether subjects' perception was in alignment with the numerical results of the automated evaluation process. Nine out of ten subjects were pleasantly surprised about the quality of the recommendations made by the system but some criticized that in a few cases the system also recommended unrelated methods, which lead to deductions in the rating.

[8] See `http://www.stg.tu-darmstadt.de/research/core/`.

2. Did the system correctly rank the proposals by relevance? (Interview) This question aims to identify how valuable the subjects found the two ways for presenting relevance of a recommendation to the users supported by the system, namely, (a) by presenting the likelihood along with the proposal and (b) by ranking the recommendations by the identified relevance. Most developers stated that they largely ignored the probability of a proposal and just followed the order of the proposals on the code completion window. The interviews suggest that displaying the exact relevance (percentage) is not a necessary feature as long as the most relevant recommendations are found on top of the list.

3. Did the tool speed up your development compared to the default Eclipse code completion? (4++ / 5+ / 1 –) One reason for the success of code completion is that it speeds up coding. This question aims to catch the subjective perception whether intelligent code completion systems can *further* improve development speed compared to the default Eclipse code completion. We take the obtained feedback as a positive indicator of the usability of the tool that shows its potential for future research.

4. Is the tool well integrated into Eclipse? (7++ / 1+ / 2 – –) The aim of this question is (i) to identify how pleased the subjects were with the implementation, and (ii) whether some issues with the interface existed that might affect other aspects of the tool. The user feedback suggests that the integration into the Eclipse's code completion is well accepted by the users. Two subjects complained that the JavaDoc for the SWT framework was missing, thereby, reducing the usability of the tool. Since this is probably due to a misconfiguration of the Eclipse system itself, these complains do not diminish the general positive feedback about the integration. Based on this feedback, we conclude that the survey results were not distorted by implementation issues.

5. Did the tool help you to understand the concepts of the framework? (Interview) Since the intelligent CCS recommends likely method calls, one hypothesis was that the tool also might support developers in understanding the framework concepts faster. The interview showed that this hypothesis is currently unfounded. Usually, framework concepts have an abstraction level much higher than method calls, thus, presenting likely methods did not help the novice subjects of this user study to grasp the underlying framework concepts. Furthermore, the subjects identified several other issues with learning a framework which we will elaborate on in Section 2.8.

Summarizing, the promising results of the user study show that it is reasonable to assume that developers would accept intelligent code completion systems as a valuable extension to the toolbox available for modern software engineering.

2.8 Summary

In this chapter, we introduced the concept of *example-based code completion system*. These systems are *intelligent*, their knowledge is based on information mined in an example codebase.

They improve the state-of-the-art of code completion systems in today's IDEs by producing context-sensitive and relevant method call recommendations, while remaining seamlessly integrated into the IDE.

We presented three example based code completion systems. We conducted a large scale evaluation of these three systems with 27,000 real world code completion queries extracted automatically from the example code base. We showed that these systems dramatically outperform the type-based Eclipse code completion system. The best system built is able to predict 82% of the method calls that are actually needed by the programmer (recall) and 72% of the recommended method calls are relevant (precision). It is based on a variant algorithm of the machine learning algorithm K-nearest neighbors, which we called Best Matching Neighbors (BMN).

We performed a user study involving 10 subjects to figure out whether real world developers could benefit from such a new code completion system. The results are promising: 9 out 10 subjects think that an example-based code completion system speeds up the development.

3 Mining Subclassing Directives from Example Code

3.1 Introduction

Object-oriented frameworks are a great vehicle in supporting code reuse [Lew95, FSJ99]. White-box frameworks are those frameworks that use inheritance, i.e., application-specific code consists of subclasses of framework classes [Sch97]. White-box frameworks, while very flexible, are difficult to learn and use [Joh92, Sca06, Rob09]. For example, instantiating the JFace white-box framework[1] to program a user interface requires that the developer identifies the right classes to extend among 203 available public classes and that she correctly overrides methods among 20 overridable methods per class in average[2].

To help developers in using frameworks good documentation is crucial. However, it is a challenge to create high quality documentation for white-box frameworks [Blo08], especially given the complexity of today's frameworks [KRW07]. As a result, framework users often miss the correct piece of documentation as recently shown by Robillard [Rob09].

The documentation of object-oriented frameworks may contain different kinds of information [FSJ99]: high-level description of the architecture (e.g., used design patterns and class diagrams), information about what the code does, code snippets, directives stating how to use framework classes or methods, etc. This chaper focuses on directives stating how to use the framework. More specifically, we use the term ***"subclassing directive"*** to designate pieces of documentation related to how to subclass a framework class or how to override a framework method.

In this chapter, we present an approach to improve the quality of *"subclassing directives"* of white-box frameworks. The core idea is that subclassing directives can be reverse-engineered from application-specific code (called *client code* hereafter), meaning that *how-to-use* documentation of a particular software artifact can be inferred from how it is actually used. Bloch [Blo08] (Item 17, pp. 87-92) urges developers of extensible classes to test them by writing subclasses before delivering them, arguing that it is the actual extensions of a base class that reveal the relevant hotspots and not only those that are exposed and documented. This fits with our intuition that actual usages found in existing clients are a good source for mining subclassing directives.

[1] JFace grounds the Eclipse IDE.

[2] For the source of these numbers see section 3.4.

These mined directives can be used by developers of new clients as a complementary source of information in addition to the API documentation delivered with the framework. They can also be used by framework developers who can peruse the list of inferred subclassing directives to eventually identify incorrect or incomplete documentation, to update the existing documentation, and improve the quality of framework subclassing directives.

An directive may be incorrect if its formulation clashes with actual usages. For instance, a method that is documented as *Subclasses may override* whereas 100% of client code do override it, is probably incorrect. Documentation is incomplete when some directives are not documented at all: for instance our approach finds directives of the form *Subclasses may override, but must call the super implementation* (100% of client code does so) which are not documented in the API documentation.

More specifically, our contributions are as follows:

1. We propose four different kinds of subclassing directives and present arguments for them. We show that they are complementary and that having just one or the other is not sufficient. We argue that when developers do not have this information they lose time in understanding how to use a framework or in solving a bug related to a violation of an undocumented directive.
2. We present an approach to mine framework subclassing directives from client code. We present one mining technique per proposed subclassing directives. Three of these techniques are based on metrics gathered from client code, the fourth one is based on a machine learning clustering algorithm.
3. We present a case-study to validate the proposed approach. The subject of the case study is the Eclipse JFace framework, a powerful, open-source, and industry-proven framework developed by IBM. This case study shows that our approach improves both the correctness and the completeness of subclassing directives present in API documentation.
4. We present a tool, called $\text{CoRe}_{\text{ExtDocs}}$, that implements the proposed approach. $\text{CoRe}_{\text{ExtDocs}}$ presents the mined subclassing directives as an extension to the API documentation in the Eclipse IDE for Java. It is public available at `http://www.eclipse.org/recommenders/`.

The reminder of this chapter is structured as follows. Section 3.2 presents four kinds of subclassing directives. Section 3.3 presents techniques to mine each of them from client code. The case study evaluating the approach is presented in section 3.4. Then, section 3.5 presents the integration of the approach into the Eclipse IDE. Section 3.6 summarizes the approach. Related work is discussed separately in Chapter 6.

3.2 Four Kinds of Subclassing Directives

A subclassing directive is a piece of documentation stating how to subclass a framework class. They can be of different kinds. We claim that the documentation of white-box frameworks requires four types of subclassing directives. In the following, we define them and give rationales for having each of them by discussing possible consequences if they are missing or if developers overlook them.

3.2.1 Method Overriding Directives

To instantiate a white-box framework, the developers need to know which framework classes are designed to be subclassed and which methods therein are designed to be overridden in an application-specific manner. A method overriding directive is a piece of documentation stating whether a framework method is designed to be overridden by client code. A class is documented as designed for subclassing if it contains at least one method overriding directive.

Method overriding directives are part of Johnson's patterns to document frameworks [Joh92], as shown by the following excerpt:

> *Each drawing element in a HotDraw application is a subclass of Figure, and must implement displayOn, origin, extent, and translateBy.*

In certain programming languages, some method overriding directives are enforced by the language itself. For instance, in Java, the `abstract` modifier for methods forces subclasses to override it, and the keyword `final` forces subclasses to use the framework implementation of the method.

Method overriding directives can be found in the API documentation of framework (an example is given in figure 3.1).

```
/**
 * Creates the control for the tool bar manager. Subclasses
 * may override this method to customize the tool bar manager.
 */
```

Figure 3.1: API Documentation containing a method overriding directive (`ApplicationWindow.createToolBarControl()` of Eclipse JFace)

3.2.2 Method Extension Directives

A method extension directive is a piece of documentation stating whether a method overriding a framework method should call the super-implementation. If the client-specific implementation

of a framework method does not call the super-implementation when required, it may violate internal framework protocols resulting in runtime problems.

For instance, if a programmer does not call the super implementation of the method `Dialog.close()` of JFace when overriding it, she gets a unclosable window which totally hangs the application. Also, she does not get a stack trace to localize the error. This shows the importance of having documented method extension directives.

To homogenize these directives, the programmers of Eclipse published a guideline to explain how to document them [dR01]. Programmers should use one of the following expressions: *subclasses may extend this method* or *subclasses may re-implement this method*. Extending means that subclasses have to call the super-implementation. Re-implementing means that subclasses must not call the super-implementation. Note that both directives are not supported by Java modifiers hence they have to be in the API documentation. Also, by default, a directive of the form *"Subclasses may override this method"* means that subclasses may or may not call the super-implementation.

3.2.3 Method Call Directives

Although the hotspot overriding is fully application-specific, frameworks may have expectations in terms of framework methods that should be called by the overriding code. A method call directive is associated to an overridable framework method and states which methods should be called inside client implementations of this framework method.

For illustration, consider an Eclipse JFace wizard page that creates some visual components to display information. The `createContents()` method is the place where to create application-specific components. The framework expects the developer to call `WizardPage.setControl()` somewhere in the body of the overriding `createContents()` in order to register the application-specific control. Omitting this call leads to a cryptic runtime error (*"Assertion failed"*) and no stack trace to localize the error.

Even if it's not explicit in Johnson's description of documentation patterns [Joh92], there are such directives in the real patterns he presents:

> *However, methods that change some attribute of a figure must notify the objects that depend on it. This is done by sending the* `willChange` *message to itself before changing the attribute, and sending the* `changed` *message to itself afterwards.*

3.2.4 Class Extension Scenarios

The directives discussed so far concern individual methods. However, just giving a novice user of a white-box framework a "flat" set of methods that could be overridden leaves her with many open questions. Which methods should she actually override? The methods with a computed

likelihood higher than a threshold? Or, are there other togetherness criteria determining "typical units of co-overridden methods"? The criteria to choose subsets of methods to override together may depend on the framework and on the class.

This is the rationale for defining a *class extension scenario* as a set of typically co-overridden methods. Class extension scenarios are ready-to-use. Developers who have never extended a particular class, can rely on them to choose the set of methods to override together. Without them, developers lose time in answering the boundary problem aforementioned.

The Eclipse website has a section containing tutorial articles. Some of them are about how to typically subclass framework classes (e.g., the tutorial about `PreferencePage`[3]).

Note on Importance Level

TODO Move to evaluation? There is a concern which crosscuts the four aforementioned directives: whether the directive is a strong requirement (e.g., *Subclass must call the super implementation*) or a weak one (e.g., *This method may be overridden*). We can identify two ways of indicating importance levels.

First, they can be expressed with modal verbs (e.g., may, should, must, etc.). This approach fits well with natural language and fuzzy expert knowledge. Johnson [Joh92] uses them, and the Eclipse guidelines for subclassing directives as well [dR01]. Second, previous work about the mining of subclassing directives [Mic00, Vil03a, TX08] generally uses importance values (e.g., probability).

3.3 Techniques to Mine Subclassing Directives

We now define techniques to mine subclassing directives from client code of white-box frameworks. Mined directives can be used by framework developers to improve the quality of the documentation and by framework users to find the pieces of information they need to correctly use the framework, thus complementing the documentation delivered with the framework.

3.3.1 Mining Overriding Directives

To determine the likelihood that a method is designed for being overridden, we define a metric called *ovLikelihood*, which represents the importance of overriding a framework method. A method overriding directive is created for each method whose value of *ovLikelihood* is not null. In the following, we give its definition and illustrate that it is meaningful by considering cases

[3] See `http://www.eclipse.org/articles/Article-Preferences/preferences.htm`.

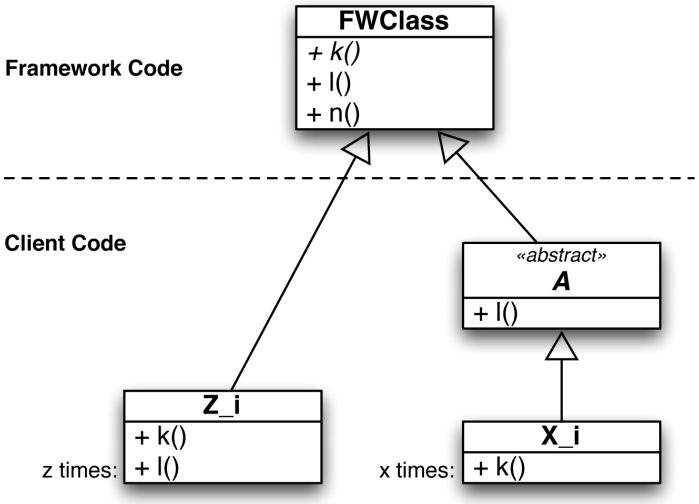


Figure 3.2: An Example to Assess the Correctness of the Metric *ovLikelihood*

where we know the likelihood value. The definition of *ovLikelihood* is as follows:

$$ovLikelihood(fwMeth) = \frac{\sum_{clCl} ov_{clCl,fwMeth}}{\sum_{clCl} ov_{clCl,fwMeth} + \sum_{clCl} not_{clCl,fwMeth}}$$

Thereby, for any framework class `fwCl`, framework method `fwMeth`, and non-abstract client subclass `clCl`: $ov_{clCl,fwMeth} = 1$, if `clCl` overrides `fwMeth` directly or inherits an overriding implementation from an intermediate client superclass; $not_{clCl,fwMeth} = 1$, if `clCl` does not override `fwMethod` at all.

To understand the properties of this metric, let us now consider the class diagram depicted in figure 3.2. `FWClass` is a framework class, `A` is a class in client code that overrides `FWClass` and that it abstract. There are also z concrete classes ($Z_1 \ldots Z_n$) that override `FWClass` and x concrete classes ($X_1 \ldots X_n$) that override `A`.

Method `k` is an abstract framework method that *must* be overridden. Hence, its value has to be 100% ($ovLikelihood_k = \frac{x+z}{x+z} = 100\%$) Method k also illustrates the rationale of discarding abstract client classes from the counting: if we do not discard them, then $ovLikelihood_k = \frac{x+z}{x+z+1} < 100\%$ which is incorrect.

Even if l is overridden in an intermediate abstract client class A, it counts as actually overridden in all concrete subclasses of A; hence the value of $ovLikelihood_l$ must be 100% and this is indeed what our metric calculates ($ovLikelihood_l = \frac{x+z}{x+z} = 100\%$).

Method n is never overridden, hence the value of $ovLikelihood_n$ must be 0% ($ovLikelihood_n = \frac{0}{0+x+z} = 0\%$).

3.3.2 Mining Extension Directives

To mine extension directives, we propose a metric that counts the number of methods that override a framework method and call the super-implementation (i.e., extend the framework method):

$$exLikelihood(fwMeth) = \frac{\sum_{clCl} super_{clMeth,fwMeth}}{\sum_{clCl} ov_{clCl,fwMeth}}$$

where $ov_{clCl,fwMeth} = 1$, if `clCl` contains a method `clMeth` which overrides `fwMeth`; $super_{clCl,fwMeth} = 1$, if `clCl` contains a method `fwMeth` which overrides `fwMeth` and call the super implementation.

A method extension directive is created for each method whose value of *exLikelihood* is not null.

3.3.3 Mining Call Directives

To mine a call directive for a framework method `fwMeth`, we propose to collect all self-calls executed within the control flow starting from each method overriding `fwMeth`. The following defines a metric which represents the importance of calling a framework method `fwMeth2` in the control flow of another framework method `fwMeth1`

$$clLikelihood(fwMeth1, fwMeth2) = \frac{\sum_{clCl} call_{clCl,fwMeth1,fwMeth2}}{\sum_{clCl} ov_{clCl,fwMeth}}$$

where $ov_{clCl,fwMeth} = 1$, if `clCl` contains a method which overrides `fwMeth`; $call_{clCl,fwMeth1,fwMeth2} = 1$, if `clCl` contains a method which overrides `fwMeth1` and calls `fwMeth2` in its control flow.

A call directive is created for each pair of framework methods, whose value of *clLikelihood* is not null.

3.3.4 Mining Class Extension Scenarios

We propose to use a clustering algorithm on client code to mine the set of methods commonly overridden together. For each framework class `fwCl`, one selects the client classes that subclass `fwCl` and clusters the methods that are often overridden together. As with other subclassing directives, mining existing clients enables to discover extension scenarios that have not been covered by tutorials.

For each framework class `fwCl` a binary matrix is build. Each row of the matrix represents a subclass of `fwCl`; each column represents an overridable method of `fwCl`. Whenever a subclass `i` overrides a framework method `j` the position `(i,j)` of the binary matrix is 1, it is 0 otherwise.

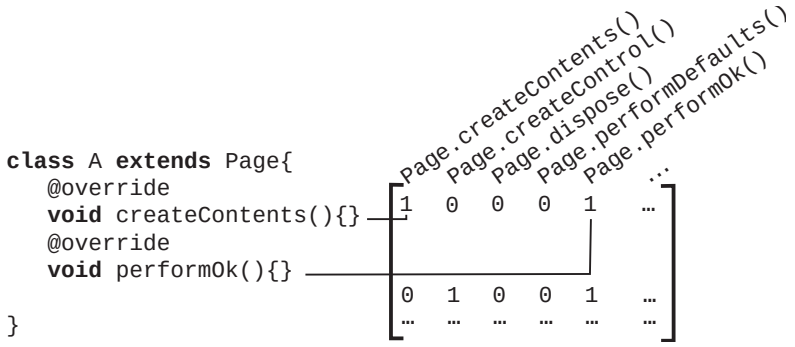


Figure 3.3: Mapping Code to a Binary Space to Mine Extension Scenarios with LCA

For illustration, figure 3.3 shows the binary matrix of a class `Page` and elaborates on the row for a subclass `A`, whose code is shown on the left-hand side.

Since the data grounding the clustering algorithm is binary, we use a data mining algorithm called *Latent Class Analysis (LCA)* appropriate to such binary data [LH68]. For each framework class, the algorithm outputs zero or more class extension scenarios. We define a default extension scenario as a scenario which covers at least 5 percent of the data, a heuristic which gives satisfying results according to our experience. If several scenarios satisfy these constraints, the default scenario is simply the one that has the greatest probability (as given by LCA).

3.3.5 Defining Importance Levels

As discussed in 3.2.4, the importance level of each directive can be represented by modal verbs or importance values. We propose to give the programmers both, for instance, *Subclass may override this method (32%)*. Our rationales are 1) modal verbs are intuitive and accessible to novice users and 2) importance values are useful for users who know how to interpret them.

We propose the following heuristics to map importance values to modal: 100%→MUST; 80%-100%→ SHOULD, 20%-80%→ MAY; 1%-20%→ RARELY. It seems satisfactory according to our own experience and according to the users of the tool. Validating them by a controlled user study is left out of the scope of this paper and one of the areas for future work.

3.3.6 Implementation

We have implemented a static analysis for each technique described above. The analyses target Java bytecode and use the IBM WALA bytecode toolkit. The implementations of *ovLikelihood* and *exLikelihood* are simple countings on a direct representation of the code. The implementation of *clLikelihood* is based on interprocedural call graphs that included all calls methods defined in the extending type or any class in between the extending class and the framework

baseclass. To cluster the co-overridden methods (the class extension scenarios), we have reused an implementation of LCA provided by NASA: Autoclass[4].

3.4 Case Study: Improving the Documentation Quality of a Mature Framework

In this section, we evaluate whether our system is able to improve the quality of the API documentation of a real-world framework with respect to subclassing directives.

3.4.1 Set Up and Overview of the Results

The subject of the study is JFace, a white-box framework dedicated to user interfaces. It is a representative of heavily used frameworks: it grounds the Eclipse IDE, it is more than 6 years old, and it is used daily by hundreds of developers. So, one can expect its documentation to be already improved several times and hence of good quality. We postulate that if our approach is able to produce directives that complement JFace's documentation or to produce directives that are more precise, it can do so for documentation of less quality as well.

For the study, we compared the subclassing directives in the API documentation of JFace with directives that were automatically extracted by applying our system to client code of JFace. The client code used for the study consisted of 600 MB of mature available Eclipse plugins[5]. Since our codebase is composed of mature code only, we can consider that the extracted directives are correct by construction.

To obtain the list of subclassing directives that are already present in the documentation of JFace, we analyzed the documentation of JFace by performing the following process.

1. We wrote a trivial static analysis which counts the number of public classes and their respective overridable methods (i.e., public/protected and not final methods). We found a total of 203 public classes which have in average 20 overridable methods.
2. We analyzed the collected client code and listed those framework classes that are extended at least ten times. Altogether there were 45 such classes (i.e., approx. one fourth of public classes).
3. We read the JavaDoc documentation of each framework method of the collected classes as well as the JavaDoc documentation of the containing class and reported the overriding directives, whenever available. Altogether 632 methods were analyzed. We found a total of 237 documented directives.

Since step 3 is manual and error-prone, we performed them twice and consolidated the results.

[4] See `http://ti.arc.nasa.gov/project/autoclass/`.

[5] For sake of replicability, the complete list of plugin ids and versions is available upon request.

Table 3.1 presents an overview of the results of comparing subclassing directives of the JFace API documentation with subclassing directives automatically mined by our system. The first part gives an overview of the documentation (of the 45 classes manually analyzed). The second part reports on agreeing documented and mined directives. The third part concerns disagreeing documented and mined directives. The fourth part is dedicated to documented directives that have no correspondence in the mined directives. Finally, the last row in the table concerns mined directives for which we could not find corresponding directives in the documentation.

One finding that is not reported in Table 3.1 but which we find interesting to emphasize is that only 30% of overridable methods are actually overridden. This is an empirical proof that the visibility modifiers alone are not sufficient as subclassing directives.

In the following subsections, we first elaborate on the findings reported in parts three, four, and five of Table 3.1. Subsequently, we evaluate the mined extension scenarios. We conclude this section by presenting and discussing the viewpoint of Boris Bokowski, leader of the JFace development team at IBM Canada, further called *the expert*, who kindly agreed to comment a sample of directives mined by our system that we sent to him.

3.4.2 Disagreeing Documented and Mined Directives

In this section, we elaborate on the set of documented directives with corresponding mined directives, but with mismatching importance levels. This set breaks down as follows:

- There are 3 methods that are documented as *Subclasses must override* or *Subclasses should override* whereas the overriding frequency in client code is less than 40%. Along the same line, we found 2 *Subclasses must override*, while client code does not always follow them (importance values: 93% and 94%), this indicates a possible change from *must override* to *should override*.
- We found 3 methods that are documented as *Subclasses may override*, while their overriding importance in client code is greater than 80%. Those methods should be documented as *Subclasses should override*. Furthermore, two of them have an importance value of 100% (all client classes overridden them) which would literally mean a *Subclasses must override*. Since we could not state whether it is really a "must" contract or a particularity of our code base, we prefer generating a *Subclasses should override*.
- For extension directives, there are 9 directives of the form *Subclasses must extend* or *Subclasses should extend* in the documentation, while the actual extension frequency in the codebase is less than 80%.
- Further, there are 28 directives of the form *Subclasses may extend* in the documentation, while their mined importance is higher than 80%.

Directive	#
Documented	**237**
Overriding Directive	153
Extension Directive	69
Call Directive	15
Agreeing Documented and Mined	**181**
Overriding Directive	138
Extension Directive	32
Call Directive	11
Disagreeing Documented and Mined	**45**
Overriding Directive	8
Extension Directive	37
Call Directive	0
Documented and Not Mined	**11**
Overriding Directive	7
Extension Directive	0
Call Directive	4
Not documented and Important in Actual Usages (importance $> 80\%$)	**129**
Overriding Directive	4
Extension Directive	67
Call Directive	58

Table 3.1: Improving the Subclassing Directives of the JFace Framework

The discussion above reveals non-negligible discrepancy between the importance level of documented directives and importance levels deduced from existing clients. Given that the codebase that we use consists of production-level client code, the mined directives reflect information that was actually needed at a point in time by the developers of the codebase. Hence, we consider the disagreeing documented directives misleading: Developers probably lose some of their valuable time in investigating and finding out that the misleading directives in the documentation are not useful for them. Hence, we conclude that the results reported in this sub-section show that our system is able to improve the correctness of the API documentation w.r.t. subclassing directives.

3.4.3 Documented and Not Mined Directives

There are documented subclassing directives of JFace which are never followed in client code. In the following, we present them and some possible explanations.

- There are 7 methods documented as *Subclasses may override*, which are never overridden in client code. It seems that JFace designers thought that it could be useful to make these methods overridable, but the reality somehow invalidated their assumptions. As emphasized by Bloch [Blo08], it is really difficult to know a priori which hotspots to provide in a base class.
- There are no unused extension directives.
- There are 4 call directives which are never followed in client code.
  - One directive is expressed as *"Use removeAll for clean up references"*. We assume that the scope of this directive is not subclasses but external users of this class.
  - Two directives are expressed as *"Subclasses should call this method at appropriate times"*. The vagueness of the directive indicates that its author did not have a precise contract in mind.
  - One directive is expressed as *"This method is really only useful for subclasses to call in their constructor"*. As for overriding directives, this may be an incorrect guess of the method usages at design time of the framework.

3.4.4 Not Documented Important Directives

In this subsection, we discuss in more detail the mined directives with a high importance level that are missing in the API documentation. For each of them, we looked in the corresponding API documentation whether it is documented or not (in both the method-level and the class-level documentation). The results of this study are as follows:

- There are 4 overriding directives extracted from client code with an importance of greater than 80% which are missing in the documentation.

- The system extracts 50 *must* extension directives that are not documented (and 17 undocumented *Subclasses should call the super-implementation*). We assume that this kind of contracts is widely yet implicitly used by framework designers while not being integrated as a documentation best practice. Developers probably lose some of their valuable time in finding out that they are expected to call the super-implementation for these particular methods.
- There are 58 mined call directives with *clLikelihood* $> 80\%$ which are not documented. This clearly indicates that framework designers often use implicit call contracts while they (or framework documenters) often do not document them, or are not aware of their importance for the client code.

The observations just reported show that our system is able to improve the completeness of the API documentation w.r.t. subclassing directives.

3.4.5 Relevance of Class Extension Scenarios

We used internet tutorials about JFace to evaluate the relevance of class extension scenarios mined by our system. We carefully read the available tutorials (both text and code snippets) to extract what are the recommended methods to override, which together form a reference extension scenario. We then compared the reference scenarios with the mined ones. For instance, the official Eclipse tutorial *"Preferences in the Eclipse Workbench UI"*[6] explains how to subclass `PreferencePage` by overriding three methods (`createContents, performDefaults, performOk`). This reference scenario perfectly matches the mined one for `PreferencePage`.

In all we analyzed 31 different tutorials from the IBM developer website[7], the Eclipse website [8] and Javaworld [9]. We found 14 tutorials that contain 25 different reference extension scenarios for JFace.

Table 3.2 presents the results of this evaluation. Each row in the table is about a different JFace class. The first column shows the name of the class and the tutorials that were used for finding reference scenarios for that class. The second column shows the mined extension scenarios in italics (there could be several mined and reference scenarios per framework classes) followed by different reference scenarios. The third column indicates whether the reference scenarios match the mined scenarios.

A quantitative summary of the data in the table 3.2 is as follows: a) 18 mined scenarios perfectly match the reference. a) 5 mined scenarios partly match the reference counterparts (they are either superset or subset of overridden methods); b) 2 reference scenarios do not have a mined

[6] See `http://www.eclipse.org/articles/Article-Preferences/preferences.htm`.
[7] See `http://www.ibm.com/developerworks`.
[8] See `http://www.eclipse.org/articles/`
[9] provides "Solutions for Java developers", see `http://www.javaworld.com`.

Context, Tutorial Title and Source	*Mined (in italic)* & Reference Scenario	Match
Class `Wizard`	*addPages, performFinish*	
Extending the Generic Workbench (IBM)	addPages,performFinish	OK
Creating JFace Wizards (Eclipse)	addPages, performFinish, canFinish (opt),	OK
Class `WizardPage`	*createControl*	
Extending the Generic Workbench (IBM)	createControl	OK
Customizing Eclipse RCP applications (IBM)	createControl	OK
Creating JFace Wizards (Eclipse)	createControl, canFlipToNextPage, getNextPage (opt)	x
Class `LabelProvider`	*getImage, getText*	
Using Images in the Eclipse UI (Eclipse)	getText, dispose	x
Using the Jface Image Registry (IBM)	getImage, getText	OK
Using the Eclipse GUI outside the Eclipse Workbench, Part 1 (IBM)	getText	x
Class `ViewerSorter`	*pattern1: category; pattern2: compare*	
Using the Jface Image Registry (IBM)	category	OK
How to use the JFace Tree Viewer (Eclipse)	category	OK
How to use the JFace Tree Viewer (Eclipse)	compare	OK
Building and delivering a table editor with SWT/JFace (Eclipse)	compare	OK
Class `ViewerFilter`	*select*	
Using the Jface Image Registry (IBM)	select	OK
Customizing Eclipse RCP applications (IBM)	select	OK
How to use the JFace Tree Viewer (Eclipse)	select	OK
Class `Action`	*run*	
Rich clients with the SWT and JFace (JavaWorld)	run	OK
Creating an Eclipse View (Eclipse)	run	OK
Class `DialogCellEditor`	-	t.f.d
Take Control of Your Properties (Eclipse)	openDialogBox	-
Extending The Visual Editor: Enabling support for a custom widget (Eclipse)	openDialogBox, doSetValue, updateContents	-
Class `PreferencePage`	*createContents, performOk, performDefaults*	
Preferences in the Eclipse Workbench UI	createContents, performDefaults, performOk	OK
Simplifying Preference Pages with Field Editors (Eclipse)	createContents, performOk, performDefaults	OK
Class `FieldEditorPreferencePage`	*createFieldEditors*	
Simplifying Preference Pages with Field Editors (Eclipse)	createFieldEditors	OK
Mutatis mutandis - Using Preference Pages as Property Pages (Eclipse)	addField, performOK, createContent	x
Mutatis mutandis - Using Preference Pages as Property Pages (Eclipse)	createFieldEditors	OK
Class `TitleAreaDialog`	*createDialogArea, okPressed, configureShell, createButtonsForButton, createContents*	
Extending The Visual Editor: Enabling support for a custom widget (Eclipse)	createContents, createDialogArea	x

Table 3.2: Reference Extension Scenarios to Evaluate the Validity of Mined Ones (in italic)

counterpart;

We further made the following qualitative observations:

- We expected to find more informal tutorials from technology guru's blogs or community-based sites. We were surprised to find so many reference scenarios in the Eclipse and IBM websites. This indicates that authoritative sources (Eclipse and IBM) consider extension scenarios as an important documentation artifact.
- However, these authoritative tutorials cover only 10 classes of JFace. There are many more of importance which are not documented by an extension scenario. This is exactly where our approach makes its contribution, by providing default extension scenarios when no others are available.

3.4.6 The Expert Viewpoint

To get an unbiased view on the observed differences between mined and documented directives, we asked Boris Bokowski, leader of the JFace development team at IBM Canada, to comment on a sample of mined directives, which we formulated as suggestions of change to the API documentation. The size of the sample (9 items) was chosen so that the questionnaire can be answered in less than 20 minutes. The directives we included in the sample were selected according to the following characteristics: First, they are related to an important and known class of JFace so as to be sure that the expert is fluent with this class; second, they have a very high importance value so as to reflect the added value of using our approach to find important incorrect or missing directives. In the following, we summarize the feedback that we got from the expert.

The expert agreed on two suggestions to change two *"Subclasses may override"* directives in the API documentation to *"Subclasses should override"* directives, as mined by our system. He asked us to fill respective entries in the bug repository of Eclipse, which we did[10]. For illustration, we elaborate on one of these changes. The API documentation states for `PreferencePage.performOK` that *Subclasses may override*. However, since our algorithm found out that the method was actually overridden in 84% of the client subclasses, we suggested to change the directive to *Subclasses should override*.

We sent the expert three suggestions for changes concerning three different *"Subclasses may extend"* directives found in the documentation. According to the Eclipse guidelines this actually means *"may override, should extend"*. Our system mined three different directives: (i) *"may override, must extend"*, (ii) *"should override, may extend"*, and (iii) *"may override, should extend"* for the respective three methods. The expert basically agreed that the first two suggestions were meaningful. Yet, he argued for not performing the first change we suggested as "it would render existing code incompatible with the specification". He disagreed on the third suggestion

[10] cf. bugs # 288461 and # 288462, `https://bugs.eclipse.org/bugs/show_bug.cgi?id=288461`.

arguing that our suggestion was merely making explicit the actual meaning of *"Subclasses may extend"* according to the Eclipse guidelines. While this is somehow a matter of taste, we still find that dissociating overriding and extension directives makes the directives more clear. The fact that we mined three different interpretations of the *"Subclasses may extend"* directive, which the expert principally agreed on, indicates that simply stating *"Subclasses may extend"* is very misleading.

The expert reviewed an extension directive with a very high importance (i.e., *Subclasses should call super*), for which he answers *"I don't see why we would require subclasses to call the super implementation"*. We analyzed the client classes that support this mined directive. It turned out that (i) the support of this directive is low (11), and (ii) the clients override this method just to add some application-specific logic, which makes sense to be called at this point in the framework control-flow, but which is not related to the initial intent of this method. This motivated us to set up a higher filter on the support of each directive in newer versions of the tool.

The expert reviewed two mined method call directives. Unfortunately, due to the specific directives selected for review and some unclarity in the expert's comments, we can not derive any generalizable conclusions.

The expert reviewed one mined default extension scenario. It is about extending the class `TrayDialog` which extends `Dialog`. While the documentation of `Dialog` related to subclassing is quite comprehensive, the subclassing documentation of `TrayDialog` is really scarce. The mined default extension scenario for `TrayDialog` consists of 11 methods to override. The expert answered that *""documentation items like this are useful*. In addition, he noted that they are *"probably better suited for a tutorial rather than the API specification."*.

The second part of the expert comment raises questions about where to put subclassing directives. In API documentation? In tutorials? Should we duplicate documentation at the class level documentation and at the method level? How to handle documentation that targets different kinds of users (novice versus experts), etc.? A thorough investigation of these questions including large-scale user studies has been out of scope of this work. For now, our answer to these questions is that subclassing directives should be integral part of the API documentation in a way that we elaborate on in the next section.

3.5 Integrating Subclassing Directives in IDEs

Subclassing directives are of primary importance for framework users and hence should be tightly integrated into the development environments so that developers find and use them easily. In the following, we present an integration proposal of our mined subclassing directives into the Eclipse IDE.

First, the initial documentation written by the framework developers is the primary source of information. It contains a lot of valuable information, e.g., the functional goal of a method,

its design rationales, etc. The mined directives complement the original API documentation. Second, since there is already a JavaDoc viewer in every default Eclipse installation and that developers may already use it, we propose to extend this viewer by enriching the initial API documentation with the mined subclassing directives (and not to create a new view). Finally, since mined directives are relevant at both the class level and the method level API documentation, the extended viewer supports both.

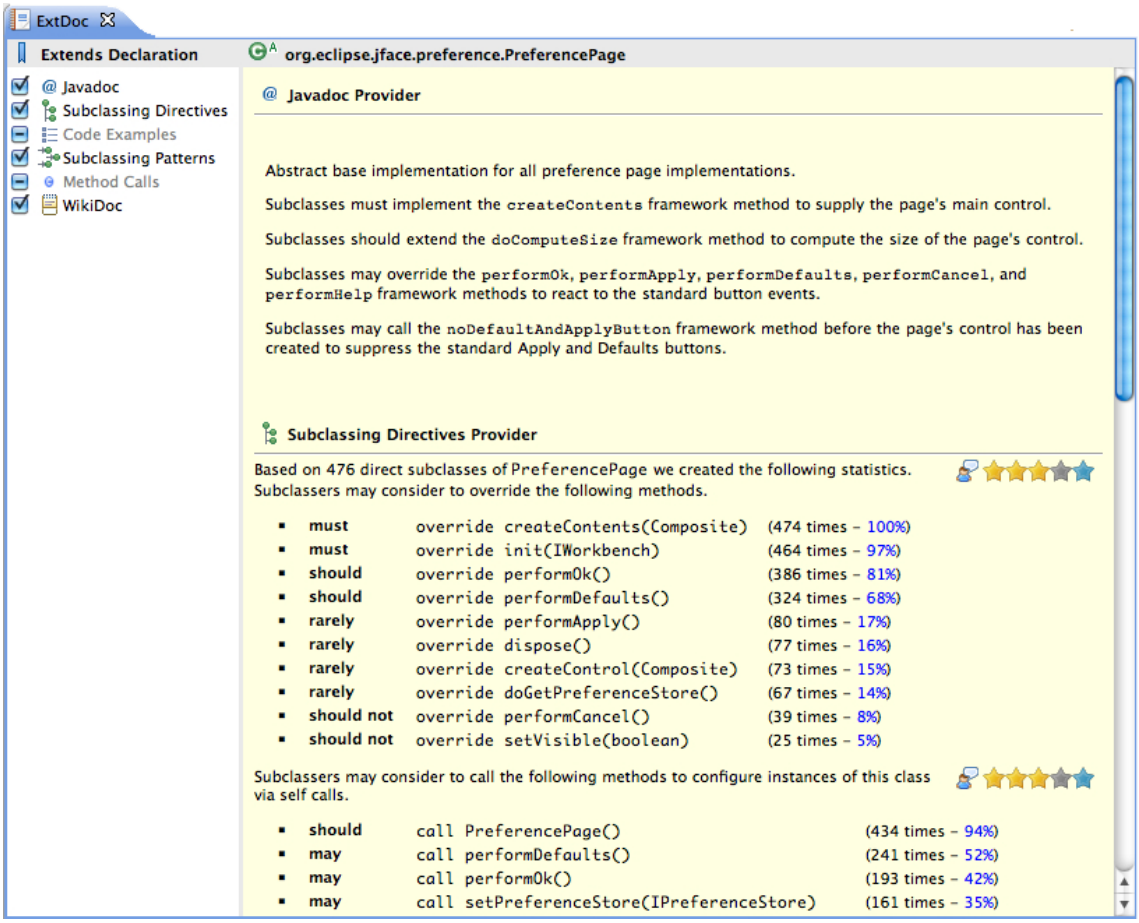


Figure 3.4: Class-level API Documentation Extended with Automatically Mined Subclassing Directives.

When developers browse the class level documentation, they are shown the list of overriding directives, the list of method call directives, and the extension scenarios that have a high support. Directives are shown in natural language (e.g., *Subclasses may override*) together with the underlying importance value. Figure 3.4 shows a screenshot of our viewer displaying the API documentation (both hand-written and generated directives) for the class `PreferencePage` of

the JFace framework.

When developers browse the method level documentation, the documentation view acts differently. It gives the initial documentation, and in addition adds the mined overriding directive, the extension directive, and the method call directives, if any. Figure 3.5 shows an example of the method level documentation. Note that both screenshots show directives for real classes of JFace and that real data underlies them.

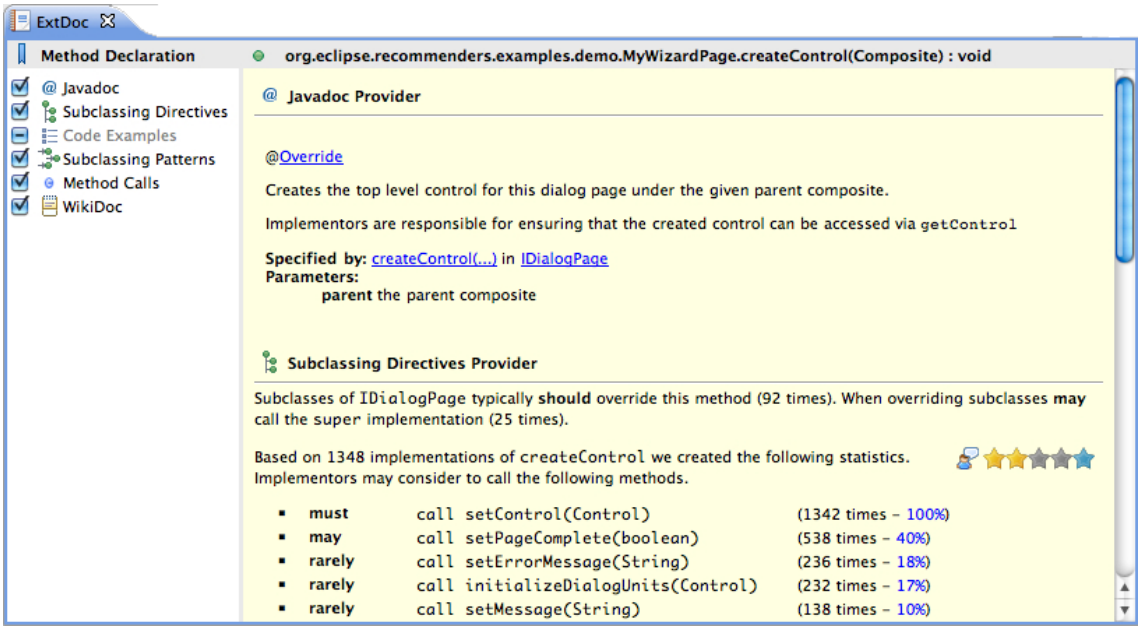


Figure 3.5: Method-level API Documentation Extended with Automatically Mined Subclassing Directives.

This integration is seamless with respect to the IDE and the usual way of programming with Eclipse. Developers have new information at the place they naturally would look at: the JavaDoc viewer. If no information is available for certain classes or certain frameworks, the viewer is simply the default one and shows only the initial API documentation.

3.6 Summary

In this chapter we presented a new approach to improve the quality of white-box framework documentation. For each framework class, our system mines from client code a set of subclassing directives that are all required to quickly and correctly extend the framework: (ii) what methods to override, (ii) should the overriding method call the super implementation, (iii) is the overriding method expected to call certain framework methods, and (iv) what are the methods usually

overridden together. This approach is complementary to manually written API documentation. Framework developers can update the documentation accordingly and framework users can access to the mined directives as an add-on to the original documentation in the IDE.

A case study evaluates the approach. We mined subclassing directives for the Eclipse's JFace framework for user-interfaces. We found that the existing API documentation is both incorrect (45 incorrect pieces of documentation) and incomplete (129 missing high-importance directives) compared to current usages of JFace.

4 Detecting Missing Method Calls in Object-Oriented Software

4.1 Introduction

"Thanks for letting me know about [...] the missing method call". This was written by a programmer on an Internet forum[1]. This quote indicates that missing method calls may be the source of software defects that are not easy to detect without assistance. Actually, problems related to missing method calls pop up in forums[1], in newsgroups[2], in bug reports[3], in commit texts[4], and in source code[5]. For a more systematic analysis of the problem, we performed a comprehensive study in a well-delimited scope: the Eclipse bug repository contains at least 115 bug reports related to missing method calls (cf. section 4.2.2). The analysis shows that issues caused by missing method calls are manifold[6]: they can produce obscure runtime exceptions at development time, they can be responsible of defects in limit cases, and they generally reveal code smells. These observations have motivated the work presented in this chapter.

Our intuition is that missing method calls are a kind of *deviant code*. Previous research proposed different characterizations of *deviant code*. Engler et al. [ECH+01] and Li et al. [LZ05a] proposed two different characterizations for procedural system-level code. Livshits et al. [LZ05b] characterized deviant code as instance of error patterns highlighted by software revisions. Wasylkowski et al. [WZL07] described an approach based on mining usage patterns and their violations. However, the aforementioned proposals are either not dedicated to object-oriented code and subsequently to missing method calls, suffer from scalability issues, or have a high rate of false positives.

This chapter presents *a new characterization of deviant code* suitable to detect missing method calls. The pieces of code that we consider are *type-usages*. A type-usage is a list of method calls on a variable of a given type occurring somewhere within the context of a particular method

[1] See `http://www.velocityreviews.com/forums/t111943-customvalidator-for-checkboxes.html`.
[2] See `http://dev.eclipse.org/mhonarc/newsLists/news.eclipse.tools/msg46455.html`.
[3] See `https://bugs.eclipse.org/bugs/show_bug.cgi?id=222305`.
[4] See `http://mail.eclipse.org/viewcvs/index.cgi/org.eclipse.team.core/src/org/eclipse/team/core/mapping/provider/MergeContext.java?view=log`
[5] See `http://mail.eclipse.org/viewcvs/index.cgi/equinox-incubator/security/org.eclipse.equinox.security.junit/src/org/eclipse/equinox/security/junit/KeyStoreProxyTest.java?view=co`.
[6] These issues are further discussed in this chapter.

body. Our characterization of deviant code is done on top of two new definitions of similarity for type-usages, and we propose a new metric called *S-Score* to measure the degree of deviance w.r.t. missing method calls. Our tool produces warnings for type-usages whose *S-Score* is high. Hence, we classify our tool as a code warning tool according to the definition of Robillard [Rob08]: it *"helps identify elements that are likely to be more worthy of investigation than others"*. The tool is a **D**etector of **M**issing **M**ethod **C**all, which grounds its acronymic name: DMMC.

We use different techniques to evaluate the proposed approach. First, statistical methods are used to show that our characterization of deviant code makes sense for detecting missing method calls. Second, we propose and perform a quantitative evaluation based on the simulation of defects by degrading real software. The advantage of this evaluation technique is that it can be fully automated on a large scale while still involving likely defects. This evaluation technique shows that our approach produces less than 2% of false positives (out of +50000 simulated missing calls), a result that does not fit the findings by Kim et al. [KE07] that usually *"automatic bug-finding tools have a high false positive rate"*. One might suspect that the low false positive rate is due to our process of artificially creating missing method calls, which might not well simulate real missing method calls of real software. To ensure that this is not the case, our last evaluation technique was to apply the tool to reveal problems related to missing method calls in real software: user-interface widgets of the Eclipse IDE codebase (44435 type-usages of `org.eclipse.swt.*` out of 1847431 LOC). We analyzed 19 high-confidence warnings found by our tool and filed the corresponding bug reports if appropriate: 6 of them are already fixed in the latest version of the Eclipse codebase. The results of this last evaluation confirm the findings of the automatic quantitative evaluation.

To summarize, the contributions of this work are:

- A comprehensive set of empirical facts on the problems caused by missing method calls. We present +30 examples of real software artifacts affected by missing method calls, a comprehensive study of this problem in the Eclipse Bug Repository, and an extensive analysis of the missing calls our tool found in Eclipse (including an analysis of their causes and their solutions).
- A new characterization of *deviant code* dedicated to missing method calls. This new characterization advances the state-of-the-art especially in terms of rate of false positive.
- A new strategy to evaluate code warning tools, based on the simulation of defects by degrading real software.

The reminder of the chapter is structured as follows. In section 4.2, we elaborate on empirical facts about missing method calls. Section 4.3 presents our approach and the underlying algorithm. Section 4.4 presents the evaluation strategy and results. Section 4.5 discusses the integration of the approach in the software development process. Section 4.6 concludes the work. Related Work is discussed in chapter 6.

4.2 The Importance of Detecting Missing Method Calls

This section presents empirical facts supporting the following claims: (a) problems related to missing method calls do happen in practice and can be difficult to understand, and (b) they survive development time.

4.2.1 Problems Related to Missing Calls are Real and Hard to Understand

Let us tell a little story that shows that missing method calls are likely and can be the source of real problems. The story is inspired from several real world posts to Internet forums and mailing lists[7]. Sandra is a developer who wants to create a dialog page in Eclipse. She finds a class corresponding to her needs in the API named `DialogPage`. Using the new-class-wizard of Eclipse, she automatically gets a code snippet containing the methods to override, shown below:

```
1 public class MyPage extends DialogPage {
2  @Override
3  public void createControl(Composite parent) {
4   // TODO Auto-generated method stub
5  }
6 }
```

Since the API documentation of `DialogPage` does not mention special things to do, Sandra writes the code for creating a control, a `Composite`, containing all the widgets of her own page. Sandra knows that to register a new widget on the UI, one passes the parent as parameter to the Composite constructor.

```
1  public void createControl(Composite parent) {
2    Composite mycomp = new Composite(parent);
3    ....
4  }
```

Sandra get the following error message at the first run of her code (the error log is unfortunately empty)!

```
1 An error has occurred. See error log for more details.
2 org.eclipse.core.runtime.AssertionFailedException
3 null argument:
```

When extending a framework class, there are often some contracts of the form "*call method x when you override method y*", which need to be followed. The Eclipse JFace user-interface framework expects that an application class extending `DialogPage` calls the method

[7] E.g., `http://dev.eclipse.org/mhonarc/newsLists/news.eclipse.tools/msg46455.html` and `http://dev.eclipse.org/newslists/news.eclipse.platform.rcp/msg10075.html`

`setControl` within the method that overrides the framework method `createControl`. However, the documentation of `DialogPage` does not mention this implicit contract; Sandra thought that registering the new composite with the parent is sufficient.

The described scenario pops up regularly in the Eclipse newsgroup[8] and shows that one can easily fail to make important method calls. Furthermore, the resulting runtime error that Sandra got is really cryptic and it may take time to understand and solve it.

Sandra had to ask a question on a mailing list to discover that this problem comes from a missing call to `this.setControl`. After the addition of `this.setControl(mycomp)` at the end of her code, Sandra could finally run the code and commit it to the repository; yet, she lost 2 hours in solving this bug related to a missing method call.

4.2.2 Missing Method Calls Survive Development Time

Missing method calls are not all detected before committing code to the version repository. To support this claim, we have searched for bug descriptions related to missing method calls in the Eclipse Bug Repository[9].

Our search process went through the following steps: 1) establish a list of syntactic patterns which could indicate a missing method call, 2) for each pattern of the list created in the previous step, query the bug repository for bug descriptions matching the pattern 3) read the complete description of each resulting bug report to assess whether it is really related to missing method calls.

To know that a report is really due to a missing method call or not, we read the whole sentence or paragraph containing the occurrence of the syntactic pattern. This gives a clear hint to assess whether this is a true or a false positive. For instance, bug #186962 states that *"setFocus in ViewPart is not called systematically"*: it is validated as related to missing method call; bug #13478 mentions that *"CVS perspective should be called CVS Repository Exploring"*: it is a false positive.

Table 4.1 summarizes the results. For illustration consider the numbers in the first raw, which tell that 49 bug reports contain the syntactic pattern "should call", and 26 of them are actually related to missing method calls. In all 211 bug reports are found by the syntactic patterns we have used, and 117 of them are actually related to missing method calls. This number shows that missing method call survive development time, especially if we consider that the number is probably an underestimation, since we may have missed other syntactic patterns. Indeed, we will also show in the evaluation section that we are able to find other missing method calls in Eclipse.

[8] Cf. the Google results of "`setcontrol+site:http://dev.eclipse.org/mhonarc/newsLists/`")
[9] See `http://bugs.eclipse.org`

Pattern	Matched	Confirmed
“should call”	49	26
“does not call”	39	28
“is not called”	36	26
“should be called”	34	9
“doesn’t call”	16	13
“do not call”	10	6
“are not called”	7	0
“must call”	7	4
“don’t call”	6	2
“missing call”	6	2
“missing method call”	1	1
Total	211	117

Table 4.1: The number of bug reports in the Eclipse Bug Repository per syntactic pattern related to missing method calls. The second column shows the number of occurrences of the pattern, the third one is the number of bug reports that are actually related to missing method calls after manual inspection.

4.2.3 Recapitulation

These empirical facts show that a detector of missing method calls: (a) can help programmers like Sandra write better code in a shorter time, and (b) can help maintainers solve and fix bugs related to missing method calls. Also, from a quality assurance perspective, such a code warning tool lists places in code that are likely to contain missing method calls and that are worth being investigated before they produce a real bug or hinder maintenance.

4.3 The DMMC System

The DMMC system is a missing method call detection system. It operates statically by analyzing software source code and outputting a list of places in code where there may be problems due to missing method calls.

Our intuition is that a piece of code is likely to host defects if there are few similar pieces of code and a lot of slightly different pieces of code. Let us consider an analogy for illustrating the idea. In a restaurant, there is one place p with one fork and one spoon. In the whole restaurant, there is a single other place with one fork and one spoon, i.e., there is one similar but not identical other place (the color of the spoon may change). However, there are 99 other places with one fork, one spoon, and one knife. It is very likely that there is an issue with place p, which can be formulated as *"A knife is missing"*.

```
class A extends Page {
  @Override
  Button createButton() {
    Button b = new Button();
    ...(interlaced code)
    b.setText("hello");
    ...(interlaced code)
    b.setColor(GREEN);
    return b;
  }
}
```

```
class B extends Page {
  @Override
  Button void createButton() {
    ... (code before)
    Button aBut = new Button();
    ...
    aBut.setText("great");
    aBut.setColor(RED);
    return aBut;
  }
}
```

```
class C extends Page {
  Button myBut;
  @Override
  Button void createButton() {
    myBut = new Button();
    myBut.setColor(PINK)
    nyButton.setText("world");
    myBut.setLink("http://bit.ly");
    ... (code after)
    return myBut;
  }
}
```

Figure 4.1: Exactly-Similar and Almost-Similar Type-Usages

The rest of this section adopts and formalizes this intuition for object-oriented software. We explain the foundations of our system in natural language before formalizing them using set theory and first-order logic.

4.3.1 Overview

The pieces of code we consider in our system are *type-usages*. A type-usage is a list of method calls on a variable of a given type occurring in the body of a particular calling method. Figure 4.1 shows three different code excerpts that are used to illustrate this definition and the ones that follow. All excerpts are from extensions of class `Page`, more precisely from re-implementations of the inherited method `createButton`; there is one type-usage per code excerpt, all are usages of the class `Button`, i.e., type-usages of `Button`. For instance, the excerpt at the right-hand side contains a type-usage of the form `tu1 = {Button.<init>, Button.setText(), Button.setColor()}`.

We have clarified the meaning of type-usages, we can now informally define two measures of similarity between type-usages: *exact-similarity* and *almost-similarity*.

A type-usage is *exactly-similar* to another type-usage if it is used in the method body of a similar method and if it contains the same method calls. For instance, in figure 4.1 the type-usage in class B (middle excerpt) is exactly-similar to the type usage of class A (left-hand excerpt): (a) they both occur in the body of the method `Button createButton()`, i.e., they are used in the same context, and (b) they both have the same set of method calls. We use the term "similar" to highlight that at a certain level of detail the type-usages related by exact-similarity are different: variables names may be different, interlaced and surrounding code, as well.

A type-usage is *almost-similar* to another type-usage if it is used in a similar context and if it contains the same method calls plus another one. In figure 4.1 the type-usage in class C (right-hand excerpt) is almost-similar to the type usage of class A (left-hand excerpt): they are used in the same context, but the type-usage in class C contains all methods of A plus another one: `setLink`. We need the term *almost-similar* to denote that the relationship between two

```
class A extends Page {
  Button b;
  @Override                                          T(b) = 'Button'
  Button createButton() {                            C(b) = 'Page.createButton()'
    b = new Button();                                M(b) = {<init>, setText, setColor}
    b.setText("hello");
    b.setColor(GREEN);                               T(t) = 'Text'
    ...(other code)                                  C(t) = 'Page.createButton()'
    Text t = new Text();                             M(t) = {<init>}
    return b;
  }
}
```

Figure 4.2: Extraction Process of Type-Usages in Object-Oriented Software.

type-usages is more similar than different, i.e., there is some similarity, while being not *exactly-similar*.

Our system is built on the assumption that a type-usage is deviant if: 1) it has a small number of other type-usages that are *exactly-similar*. and 2) it has a large number of other type-usages that are *almost-similar*. Similarly to the restaurant problem described above, a deviant type-usage may reveal an issue in software. Given the intuitive definitions so far, we are now able to give an overview of the logic of our system:

1. Extract every type-usage x from software;
2. For every type-usage x:
   a) Search for type-usages that are exactly-similar according to our definition of *exact-similarity*;
   b) Search for type-usages that are almost-similar according to our definition of *almost-similarity*;
   c) Compute a measure of strangeness. We call this new measure *the S-score*;
   d) Synthesize a list of likely missing method calls;
3. Output an list of deviant type-usages ordered by S-score and their missing method calls.

In the following, we elaborate on the internals of each of these steps.

4.3.2 Extracting Type-Usages

The DMMC system uses the WALA byte-code analysis toolkit to extract type-usages from the source code. For each variable x in the code, we extract the declared type $T(x)$ of the variable containing the type-usage, the context $C(x)$ of `x`, that we define as the signature of the containing method, and the names of the methods invoked on `x` within $C(x)$, $M(x) = \{m_1, m_2, ..., m_n\}$). If there are two variables of the same type in the scope of a method, they are two type-usages extracted.

Figure 4.2 illustrates the encoding of the extracted type-usages by an example. A code snippet is shown on the left-hand side of the figure; the corresponding extracted type-usages are shown on the right-hand side of the figure. There are two extracted type-usages, for Button `b` and for Text `t`. The context is the overriden method `createButton` for both. The set of invoked methods on `b` is $M(b) = \{< init >, setText, setColor\}$, `t` is just instantiated ($M(t) = \{< init >\}$).

4.3.3 Exactly and Almost Similar Type-usages

We define a relationship E over type-usages of object-oriented source code that expresses that two type-usages x and y are exactly-similar if and only if:

$$\begin{aligned} xEy \iff & T(x) = T(y) \\ & \wedge\, C(x) = C(y) \\ & \wedge\, M(x) = M(y) \end{aligned}$$

We then define for each type-usage x the set of exactly-similar type-usages:

$$E(x) = \{y | xEy\}$$

Note that since the relationship holds for the identity, i.e., xEx is always-valid, $E(x)$ always contains x itself, and $|E(x)| \geq 1$.

We define a relationship A over type-usages of object-oriented source code that expresses that two type-usages are almost-similar. A type-usage x is almost-similar to a type-usage y if and only if:

$$\begin{aligned} xAy \iff & T(x) = T(y) \\ & \wedge\, C(x) = C(y) \\ & \wedge\, M(x) \subset M(y) \\ & \wedge\, |M(y)| = |M(x)| + 1 \end{aligned}$$

For each type-usage x of the codebase, the set of almost-similar type-usages is:

$$A(x) = \{y | xAy\}$$

Note that contrary to $E(x)$, $A(x)$ can be empty and $|A(x)| \geq 0$.

4.3.4 S-score: A Measure of Strangeness for Object-Oriented Software

Now we want to define a measure of strangeness for object-oriented type-usages. This measure will allow us to order all the type-usages of a codebase so as to identify the top-K strange type-usages[10] that are worth being manually analyzed by a software engineer.

We define the S-score as:

$$\text{S-score}(x) = 1 - \frac{|E(x)|}{|E(x)| + |A(x)|}$$

This definition is robust with regard to the limit cases: if there are no exactly-similar type-usages and no almost-similar type-usages for a type-usage a, i.e., $|E(a)| = 1$ and $|A(a)| = 0$, then $S - score(a)$ is zero, which means that a unique type-usage is not a strange type-usage at all. On the other extreme, consider a type-usage b with $|E(b)| = 1$ (no other similar type-usages) and $|A(b)| = 99$ (99 almost-similar type-usages). Intuitively, a developer expects that this type-usage is very strange, may contain a bug, and should be investigated. The corresponding S-score is 0.99 and supports the intuition.

4.3.5 Predicting Missing Method Calls

For the type-usages that are really strange, i.e., that have a very high S-score, the system recommends a list of method calls that are likely to be missing.

Core Algorithm: The recommended method calls $R(x)$ for a type-usage x are those calls present in almost-similar type-usages but missing in x. In other terms:

$$R(x) = \{m | m \notin M(x) \wedge m \in \bigcup_{z \in A(x)} M(z)\}$$

For each recommended method in $R(x)$, the system gives a likelihood value $\phi(m, x)$. The likelihood is the frequency of the missing method in the set of almost-similar type-usages:

$$\phi(m, x) = \frac{|\{z | z \in A(x) \wedge m \in M(z)\}|}{|A(x)|}$$

For illustration, consider the example in figure 4.3. The type-usage under study is x of type `Button`, it has a unique call to the constructor. There are 5 almost-similar type-usages in the

[10] K may depend on the size and the maturity of the analyzed software.

$$
\begin{aligned}
T(x) &= Button \\
M(x) &= \{< init >\} \\
A(x) &= \{a, b, c, d\} \\
M(a) &= \{< init >, setText\} \\
M(b) &= \{< init >, setText\} \\
M(c) &= \{< init >, setText\} \\
M(d) &= \{< init >, setText\} \\
M(e) &= \{< init >, setFont\}
\end{aligned}
$$

$$R(x) = setText, setFont$$

$$\phi(setText) = \frac{4}{5} = 0.80$$

$$\phi(setFont) = \frac{1}{5} = 0.20$$

Figure 4.3: An example computation of the likelihoods of missing method calls

source code (a, b, c, d, e). They contain method calls to `setText` and `setFont`. `setText` is present in 4 almost-similar type-usages out of a total of 5. Hence, it's likelihood is $4/5 = 80\%$. In this situation, the system recommends to the developer the following missing method calls: `setText` with a likelihood of 80% and `setFont` with a likelihood of 20%.

Variant with Filtering In the example of figure 4.3, it is much more likely that the missing method call we are searching is `setText` rather than `setFont`, and it seems interesting to set up a threshold t on the likelihood before recommending a missing method call to the user. This defined a filtered set of recommendations $R_f(x)$ which is:

$$R_f(x) = \{m | m \in R(x) \wedge \phi(m, x) > t\}$$

This variant of the system is called DMMC filter, as opposed to DMMC-core.

4.4 Evaluation

We propose and conduct an evaluation for the proposed DMMC system, which combines different techniques to validate the system from different perspectives:

- We validate the S-score measure by showing that (a) it is low for most type-usages of real software, i.e., that the majority of real type-usages is not strange (cf. 4.4.1), and (b) it is able to catch type-usages with a missing method call, i.e., that the S-score of such type-usages is in average higher than the S-Score of normal type-usages (cf. 4.4.2).
- We show that our algorithm produces good results, i.e., predicts missing method calls that are actually missing (cf. 4.4.3).
- We evaluate whether our system is able to find meaningful missing method calls in mature software (cf. 4.4.4).

4.4.1 The Correctness of the Distribution of the S-Score

We collected the whole Eclipse 3.4.2 codebase for conducting this experiment (564 plugins). To allow future comparisons with other approaches and replication studies, this dataset is publicly available upon request. From this codebase, our static analysis collected 44435 type-usages whose type belongs to the Standard Widget Toolkit (SWT). For each of them, we have computed the sets of exactly and almost-similar type-usages ($E(x)$ and $A(x)$) and their S-score.

Histogram of the S-score

Since Eclipse is a real-world and mature software, we assume that most of its type-usages have a low degree of strangeness, i.e., a low S-score. Figure 4.4 validates the assumption: 78% of the SWT type-usages have indeed a S-score less than 10%, 90% of the SWT type-usages have a S-Score less than 50%. Indeed, the distribution has an unsurprising exponential shape, which is regular in software [BFN+06].

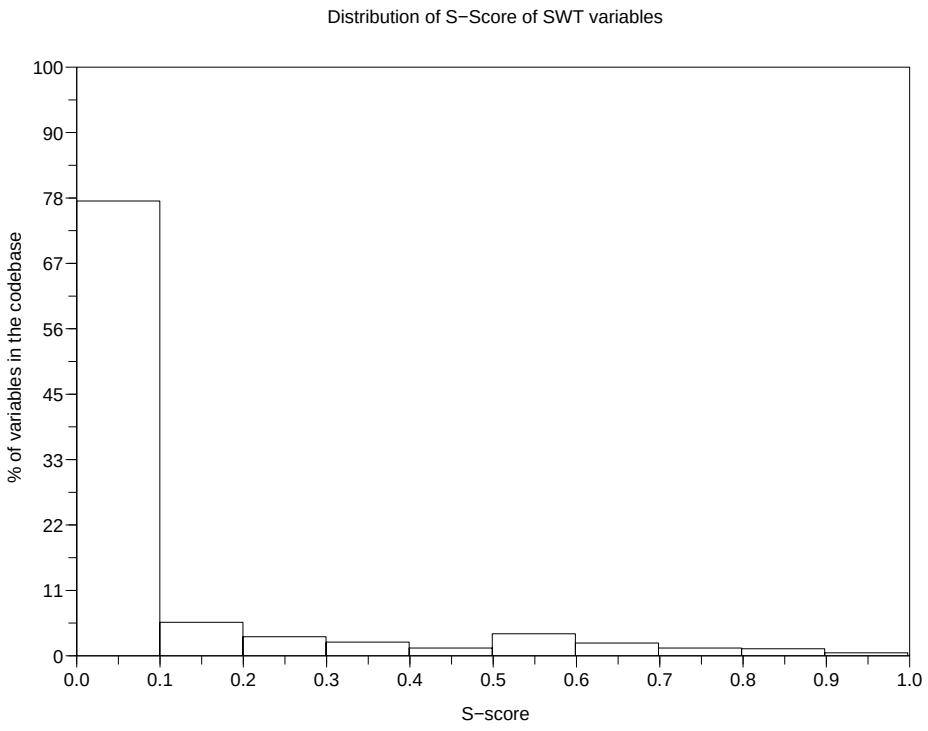


Figure 4.4: Distribution of the S-Score based on the type-usages of SWT in the Eclipse codebase. Most type-usages have a low S-Score, i.e., are not strange.

Representing the S-score in a 2D Space

Independently of the S-score, we also assume that most of the type-usages of Eclipse have a low number of almost-similar type-usages.

The number of exactly and almost-similar type-usages ($|E(x)|$ and $|A(x)|$) defines a two-dimensional space, in which we can plot the type-usages of a software package. A scatter plot in this space enables us to graphically validate our assumption, i.e., to see whether the majority of points are in the bottom of the figure. Figure 4.5 represents the SWT type-usages that we have extracted in this 2D space.

We make the following observations. First, the cloud of points is much more horizontal along the x-axis, which validates our assumption. Second, points depicted as diamonds (e.g., the point at coordinate $(189, 3)$) represent type-usages whose S-score is greater than 97%. The figure shows that strange type-usages are all located in the same zone: the top left-hand side part of the figure, which can somehow be called the "zone of strange" (parameterized by a threshold on the S-score). There are 25 type-usages in this zone, out of a total of 44435 type-usages. This can not be clearly seen in the figure, since the space is discrete and some points are exactly at the same place. All type-usages of this zone of strange will be analyzed in section 4.4.4.

4.4.2 The Ability of S-score to Catch Degraded Code

Even if the distribution of the S-score seems reasonable, we want to be sure that a faulty type-usage would be caught by the S-score. For this to be true, a type-usage with a missing method call should have a higher S-score than a normal type-usage.

To assess the validity of this assumption, our key insight is to simulate missing method calls. Given a type-usage from real software, the idea is to remove one by one each method call[11], and to check whether the S-score of the artificially created faulty type-usage has a higher S-score than the original one. This validation strategy has several advantages: (a) there is no need to manually assess whether a type-usage is faulty, we know it by construction, (b) it is based on real data (the type-usages come from real software), (c) it is on a large-scale (if the codebase contains N type-usages, with m_i method calls, the detection system is tested with $\sum m_i$ different queries.

We have conducted this evaluation on our SWT dataset. The evaluation strategy described in the previous paragraph creates 55623 different simulated missing method calls. We have checked whether the S-Score of these different simulated bugs is higher than the S-Score of the original (non-degraded) type-usage. The results of this evaluation is that 94% (52154/55623) of the degraded type-usages are more strange than the initial one i.e., they have a S-Score higher than the S-score of the initial one.

[11] If a type-usage contains at least 2 method calls.

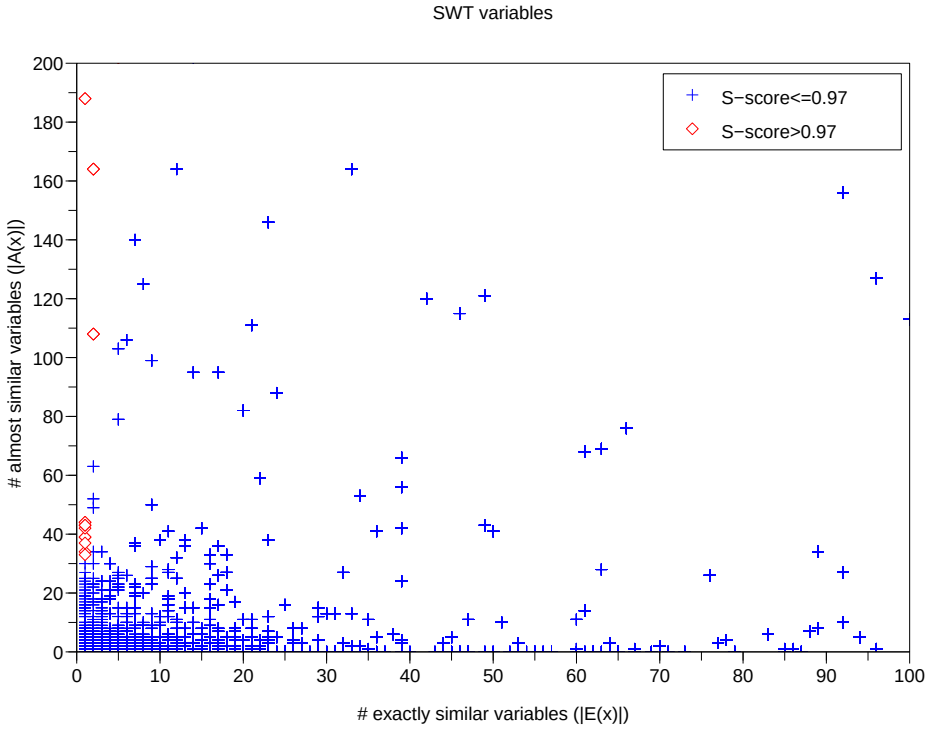


Figure 4.5: Scatter Plot of the Type-Usages of SWT in the Eclipse codebase. The red diamonds indicate very likely issues (i.e., their S-score in greater than 0.97).

Furthermore, the difference of the S-score is high as shown by table 4.2: the average S-score of normal data is 0.16 (low) and the average S-score of degraded data is 0.76 (high). The percentile shown on rows #2 and #3 of the table strengthen the confidence in this finding, while 62% of the normal type-usages have a S-score lower than 0.1, there is only 0.3% of degraded type-usages whose S-score is under 0.1. These numbers validate that the S-score correctly recognizes faulty type-usages.

4.4.3 The Performance of the Missing Method Calls Prediction

The third evaluation of our system measures its ability to guess missing method calls. The assumption underlying this evaluation is that our algorithm to predict missing method calls (cf. 4.3.5) should be able to predict calls that were artificially removed.

We have used the same setup as for evaluating the characteristics of the S-score (cf. 4.4.2), i.e., we have simulated missing method calls. However, instead of looking at the difference of S-

	Normal type-usages	Degraded type-usages
Mean S-score	0.16	0.76
S-score<0.1	62%	0.3%
S-score>0.9	0.6%	35%

Table 4.2: The S-Score for initial and degraded data. The S-Score is able to capture faulty type-usages.

score between real and degraded data, we have tried to guess the method call that was artificially removed with the technique described in 4.3.5.

For instance, given a real type-usage of the codebase representing a `Button` and containing `<init>` and `setText`, we test the system with two different queries: 1) `<init>` only and 2) `setText` only. The system may predict several missing method calls, but a prefect prediction would be `setText` as missing method call for the first query and `<init>` for the second query.

Hence, the system is evaluated with the same number of queries as in 4.4.2 i.e., 55623 artificial bugs. Then, we can measure the relevance of the missing method calls that are predicted. We measure the relevance of a single query using precision and recall:

- **PRECISION** is the ratio between the number of correct missing method calls (i.e., that were actually removed during the degradation of the real type-usage) and the number of guessed missing method calls. Note that the precision is not computable if the system outputs nothing, i.e., if the number of guessed missing method calls is null.
- **RECALL** is the number of correct missing method calls over the number of expected answers. In our evaluation setup, the number of correct missing method calls is either 0 or 1 and the number of expected answers is always one, hence the recall is a binary value, either 0 or 1;

We measure the overall performance of the system using the following metrics:

- **ANSWERED** is the percentage of answered queries. A query is considered as answered if the system outputs at least one missing method call.

$$ANSWERED = \frac{N_{answered}}{N_{query}}$$

- **MEANPRECISION** is the mean of the precision of answered queries. Since the precision is not computable for empty recommendations (i.e., unanswered queries),

$$MEANPRECISION = \frac{\sum_i PRECISION_i}{N_{answered}}$$

- **MEANRECALL** is the mean of the recall of all queries[12], i.e.,

$$MEANRECALL = \frac{\sum_i RECALL_i}{N_{query}}$$

MEANRECALL is directly related to ANSWERED, since an unanswered query has a null recall. Hence, the lower *ANSWERED*, the lower *MEANRECALL* and vice versa.

MEANPRECISION describes the rate of false positives: the lower *MEANPRECISION*, the greater the number of false positives. Hence, we would like to have a high *MEANPRECISION*. Also, even if the precision is high, the system might simply recommend nothing for most queries (cf. formula above). Hence, a good system must have a high *MEANPRECISION* and a high *ANSWERED*, which means it is right when it predicts a missing method calls **and** it does not miss too many missing method calls.

Results

We evaluated the DMMC system based on the evaluation process and performance metrics presented above. We have evaluated the two algorithms presented in 4.3.5 (DMMC-core and the variant with filtering DMMC-filter). The filtering version simply removes certain recommendations from the initial set of recommended method calls $R(x)$. Hence, the filtering version will mechanically have lower or equal ANSWERED and MEANRECALL. However, we hope that the filtering strategy would increase MEANPRECISION.

Table 4.3 presents the results. The three performance metrics of DMMC-core are high: the PRECISION of 84% shows that the core-system has a low false positive rate while still being able to answer 80% (ANSWERED) of the generated queries. Second, table 4.3 validates the filtering strategy defined in 4.3.5. As shown by the reported numbers, it significantly improves the precision. With a filter of 90%[13], the system is able to have a precision of 98% while still answering 67% of the queries. These numbers validate the ability of the system to correctly detect missing method calls.

4.4.4 Finding New Missing Method Calls in Eclipse

The evaluation results presented in 4.4.2 and 4.4.3 suggest that a software engineer should seriously consider to analyze and change a type-usage, if it has a high S-score and recommended methods with a high likelihood (say $\phi(m, x) > 90\%$). However, it may be the case that our

[12] Setting the denominator of MEANRECALL to $N_{answered}$ would be misleading because a system that predicts something only for 1% of the queries could still have a high MEANRECALL

[13] This threshold was chosen by intuition: it implies that a method call should be predicted if it is present in most of the almost-similar type-usages. Our various experiments showed that this threshold is not sensitive, all values from 80% to 95% produce results of the same order of magnitude.

System	N_{query}	ANSWERED	MEANPRECISION	MEANRECALL
DMMC-core	55623	80%	84%	78%
DMMC-filter (90%)	55623	67%	98%	66%

Table 4.3: Performance metrics of two variants of the DMMC system. Both have high precision and recall.

process of artificially creating missing method calls does not reflect real missing method calls that occur in real software.

As a counter-measure to this threat of validity, in this fourth evaluation, we applied the DMMC system to Eclipse v3.4.1. We searched for missing method calls in the Standard Widget Toolkit of Eclipse (SWT) type-usages of the Eclipse codebase. We chose to search missing method calls related to the SWT for the following reasons. First, we can reuse the extracted dataset used for the automatic evaluation. Second, finding method calls that remain in the user interface of Eclipse after several years of production is ambitious:

- since the community of users is large, the software is used daily in plenty of different manners, and missing method calls had a chance to produce a strange behavior. Furthermore, bugs in the user interface are mostly *visible* and easy to localize in code.
- since the community of developers is large and the codebase is several years old, most of the code has been read by several developers, which increases the probability of detecting suspicious code;

The following shows that we actually found some missing method calls related to the user interface of Eclipse.

We analyzed the strangest type-usages found by DMMC in Eclipse v3.4.1. They are in the zone of strange of figure 4.5 and have all a S-score greater than 0.97. For each strange type-usage, we tried to understand the problem so as classify it as a true or false positive and so as to file a bug in the Eclipse Bug Repository.

Table 4.4 gives the results of this evaluation. The first column gives the location of the strange type-usage. The second column gives the acronym of the problem underlying the strange type-usage (the following subsections elaborate on each such problem). The third column gives the S-score of the type-usage and the fourth column the bug id of our bug report or "NR" for non-reported. The last column indicates the feedback of the Eclipse developers based on the bug report. $\surd\surd$ means that the head version of Eclipse is already patched accordingly, $\surd$ means that the bug is very likely to be validated[14] but not yet fixed, x indicates that the bug has been marked as invalid, and finally a question mark "?" indicates that the comments and the status of the bug report do not yet allow us to conclude. We now elaborate on the kind of problems revealed by the detected missing method calls.

[14]E.g., similar to accepted reports and assigned to a particular developer.

Location	Problems	S-score	Bug Id	Val.
ExpressionInputDialog.okPressed	SA	0.99	296552	√√
ExpressionInputDialog.close	EB+VAPIBP	0.99	296494	?
RefactoringWizardDialog2.okPressed	SU	0.99	296585	x
NameValuePairDialog.createDialogArea	VAPIBP	0.98	296581	√√
AlreadyExistsDialog.createDialogArea	VAPIBP	0.98	296781	√
CreateProfileDialog.createDialogArea	VAPIBP	0.98	296782	√
AboutDialog;.createDialogArea	EB+VAPIBP	0.98	296578	?
TextDecoratorTab.<init>	WA	0.98	NR	
UpdateAndInstallDialog:createDialogArea	VAPIBP	0.98	296554	√√
RefactoringStatusDialog;.createDialogArea	VAPIBP	0.97	296784	√
AddSourceContainerDialog:createDialogArea	VAPIBP	0.97	296481	√√
GoToAddressDialog:createDialogArea	VAPIBP	0.97	296483	√√
TrustCertificateDialog.createDialogArea	EB	0.97	296568	√
TitleAreaDialog;.createDialogArea	FP	0.97	NR	
StorageLoginDialog.createContents	WA	0.97	NR	
BrowserDescriptorDialog.createDialogArea	WA	0.97	NR	
StandardSystemToolbar.<init>	FP	0.97	NR	
ChangeEncodingAction$1.createDialogArea	SA+VAPIBP	0.97	275891	√√
JarVerificationDialog.createDialogArea	EB+VAPIBP	0.97	296560	?

Table 4.4: Strange type-usages (S-Score > 0.97) in Eclipse, and the corresponding problems and bug reports.

SA: Software Aging

Missing method calls may reveal problems related to software aging. Let us elaborate on the two corresponding strange type-usages of table 4.4.

According to the system, `ExpressionInputDialog` contains strange code related to closing and disposing the widgets of the dialog. Our manual analysis confirms that there are plenty of strange things happening in the interplay of methods `okPressed`, `close` and `dispose`. We found out that these strange pieces of code date from a set of commits and a discussion around a bug report [15]. Even if the bug was closed, the code was never cleaned. In this case, software aging comes from measures and counter-measures made on the code in an inconsistent manner.

The other example is in `ChangeEncodingAction` which contains code related to a very old version of the API and completely unnecessary with the current version. Following our remark about this class to the Eclipse developers, the code has been actualized. In this case, software aging comes from changes of the API that were not reflected in client code.

SU: Software Understanding

Missing method calls may reveal problems related to software understanding. The third warning produced by the system concerns `RefactoringWizardDialog2.okPressed`. The system misses a call to `dispose` for certain widgets involved. The reason is that `okPressed` usually closes the dialog and dispose widgets. However, `RefactoringWizardDialog2.okPressed` uses the dialog itself to show an error, hence does not dispose anything, which is a very strange manner according to the Eclipse practices. Hence, this piece of code has a very high S-score. Interestingly, the code was so hard to understand that we created an incorrect bug report, which was invalidated by an Eclipse developer. Since it would require a significant refactoring to clean this strange code (and decrease its S-score), the developer did not modify this method. This case study validates the warning produced by a high S-score and also shows that it is not always possible to solve a warning by simply adding the missing method call.

VAPIBP: Violation of API Best Practices

Strange type-usages often reveal violations of API best practices. An API best practice is a programming rule which is not enforced by the framework code or the programming language. In the following, we discuss several API best practices of Eclipse whose violations can be detected by our system.

[15] See `https://bugs.eclipse.org/bugs/show_bug.cgi?id=80068`

Call super.createDialogArea: It is standard to create the new container widget of a dialog using the framework method `createDialogArea` of the super class `Dialog`. This best practice is documented in the API documentation of `Dialog`. However, certain type-usages do not follow this API best practice and create an incorrect clone of `super.createDialogArea`: there is an important method call present in `super.createDialogArea` and which is missing in the clone (e.g., setting the dialog margin using `convertVerticalDLUsToPixels`). For instance, `AddSourceContainerDialog` instantiates and initializes the new Composite by hand and `UpdateAndInstallDialog` uses an ad hoc method: both are not 100% compliant with `super.createDialogArea` and trigger a very high S-score. Both violations have been reported and are now fixed in the Eclipse codebase.

Setting fonts: A best practice of Eclipse consists of setting the font of new widgets based on the font of the parent widget and not on the system-wide font. Not following this best practice may produce an inconsistent UI, as shown figure 4.6. It is a screenshot of a dialog of Eclipse Ganymede resulting from a font change. It is inconsistent because it contains at the same time big and small fonts.

To our knowledge, this API best practice is not explicitly documented but pops up in diverse locations such as: newsgroups[16], bug reports[17], and commit texts[18]. The programming rule associated to this API best practice is to call `getFont` on the parent widget and to call `setFont` on the newly created widget. Figure 4.7 illustrates this point by showing the result of a commit which solves a violation of this best practice: the new code at the right hand side contains the previously missing method calls. Our system automatically detects the missing calls related to such violations.

Don't set the layout of the parent: Another API best practice of Eclipse consists of never setting the layout of the parent widget, i.e., not calling `setLayout` on the parent. Our system finds violations of the API best practice. At first sight, it seems contradictory since our system searches for missing rather than extraneous method calls. However, there is a logical explanation.

When one overrides `createDialogArea`, there are always two composites to work with: the `parent` and the newly created one for the dialog area, which we call `newcomp`. Their responsibilities are different, and so are the typical methods that developers call on them. Typically, one calls `setLayout` on the newly created `Composite`, but never on the parent.

[16] See `http://mail.eclipse.org/viewcvs/index.cgi/org.eclipse.ui.browser/src/org/eclipse/ui/internal/browser/BrowserDescriptorDialog.java`

[17] See `https://bugs.eclipse.org/bugs/show_bug.cgi?id=175069` and `https://bugs.eclipse.org/bugs/show_bug.cgi?id=268816`

[18] See `http://mail.eclipse.org/viewcvs/index.cgi/org.eclipse.ui.ide/src/org/eclipse/ui/internal/ide/dialogs/ResourceInfoPage.java?sortby=log&view=log` and `http://mail.eclipse.org/viewcvs/index.cgi/org.eclipse.ui.browser/src/org/eclipse/ui/internal/browser/BrowserDescriptorDialog.java`

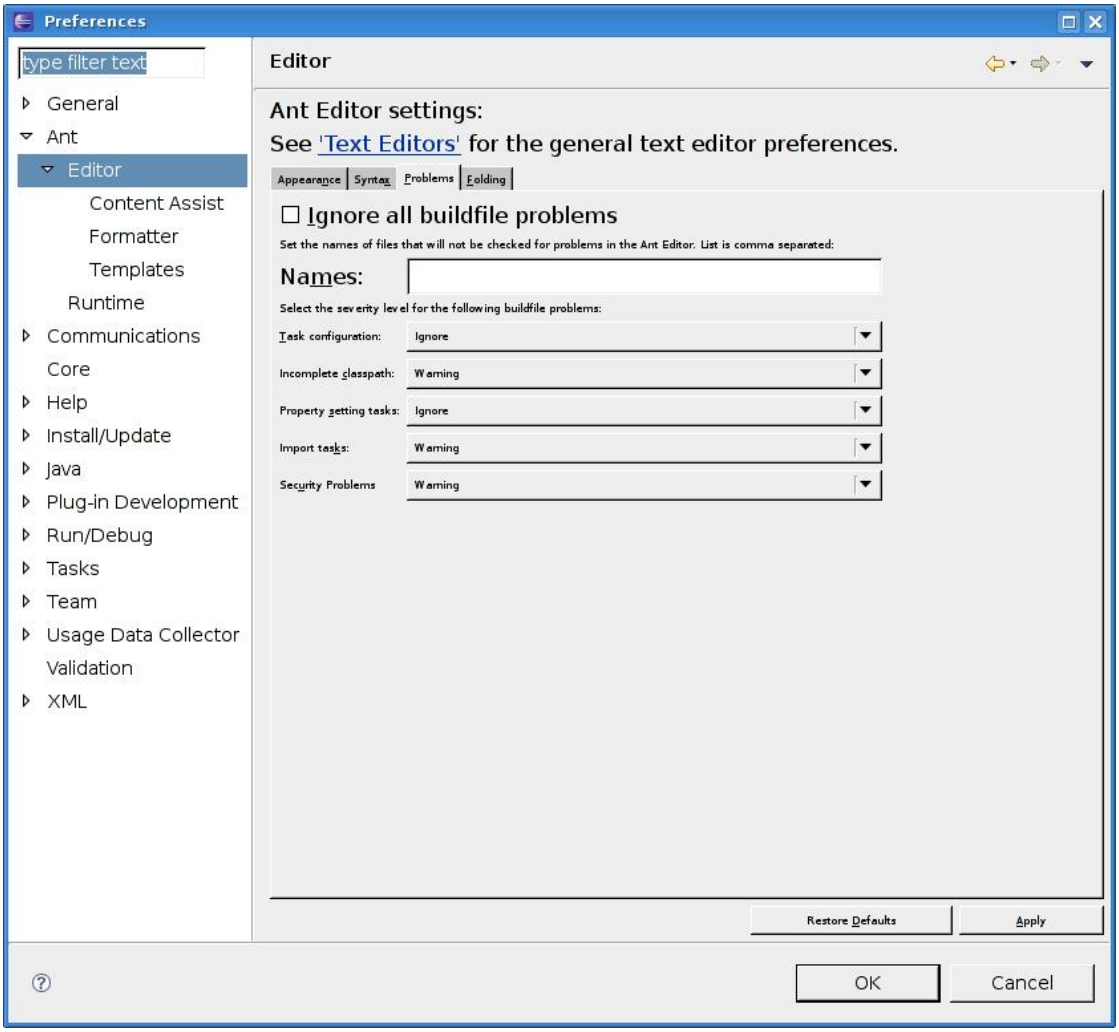


Figure 4.6: Font inconsistencies in Eclipse caused by a missing method call to `setFont`

```
protected Control createDialogArea(Composite parent) {
    Composite composite = (Composite) super.createDialogArea(parent);
    ((GridLayout)composite.getLayout()).numColumns = 3;
```

```
protected Control createDialogArea(Composite parent) {
    Font font = parent.getFont();
    Composite composite = (Composite) super.createDialogArea(parent);
    ((GridLayout)composite.getLayout()).numColumns = 3;
    composite.setFont(font);
```

Figure 4.7: Excerpt of revision 1.5 of Apr. 10 2006 of `BrowserDescriptorDialog.java`. Two missing method calls related to setting fonts are added.

When a developer accidentally calls `setLayout` on the parent widget, the set of almost exact type-usages consists mostly of `newcomp`-based type-usages and not of `parent`-based type-usages. In other terms, the system believes that this type-usage is a new created widget and misses the corresponding calls. In this case, the system is right in predicting a strange type-usage but wrong as far as the predicted missing method call is concerned. That is why software engineers always have to analyze and understand the causes of the strange type-usage before adding the predicted missing method call.

For illustration, let us consider such a violation in table 4.4: in the class `ChangeEncodingAction` of Eclipse Ganymede, there is a call to `setLayout` on the parent. To confirm the analysis we have just presented, we asked the Eclipse developers if this code is correct: they agreed on our diagnosis, filed a bug in the repository[19] and changed the code of `ChangeEncodingAction` accordingly[20].

Calling dispose: The SWT toolkit uses operating system resources to deliver native graphics and widget functionalities. While the Java garbage collector handles the memory management of Java objects, it can not handle the memory management of operating system resources. Not disposing graphical objects is a memory leak, which can be harmful for long-running applications. For instance, the following code of `ExpandableLayout` produces a high S-Score (0.96):

```
1 // Location: ExpandableLayout.layout
2 size= FormUtil.computeWrapSize(new GC(..),..)
```

The newly created graphical object (`new GC()`) is not assigned to a variable. However, the Java compiler inserts one in the Java bytecode. Since the method `computeWrapSize`, which receives the new object as a parameter, does not dispose the new object, it is never disposed. That's why our system predicts a missing call to `dispose`. This problem was filed and solved in the Bugzilla repository independently of our work[21].

FP: False Positive

Our system suffered from two false positives in this evaluation setup.

The first one is rather subjective (`TitleAreaDialog`), it is about the creation of a `Composite` instance, already discussed above (cf. *Call super.createDialogArea*) On the one hand, the problematic type-usage should use a `super.createDialogArea` for being initialized, and then it should be tailored by overriding certain default choices. In this perspective, it is an incorrect clone and a violation of the API best practice. On the other hand, the initialization code is quite

[19] See `https://bugs.eclipse.org/bugs/show_bug.cgi?id=275891`

[20] See `http://mail.eclipse.org/viewcvs/index.cgi/org.eclipse.ui.editors/src/org/eclipse/ui/texteditor/ChangeEncodingAction.java?r1=1.19&r2=1.20`

[21] See `https://bugs.eclipse.org/bugs/show_bug.cgi?id=257327`.

different compared to the body of the framework method, hence it is almost not anymore a clone! There is no objective and clear separation between a clone and a source of inspiration.

The second false positive reveals that our algorithm is sensitive to the tyranny of majority, when the three following conditions are met: 1) there is huge number of almost-similar type-usages, 2) there is a small number of exactly-similar type-usages and 3) the type-usage is correct, i.e., it is normal to have only this set of method calls. One type-usage in the manually analyzed warnings turned out to be such a false positive (`StandardSystemToolbar`). Let us explain it briefly. Most SWT widgets uses a layout manager based on grids (`GridLayout`), hence most SWT objects have the corresponding layout data (`GridData`) set using the method call `setLayoutData`. However, the caller of `StandardSystemToolbar.createDialogArea` uses a `CLayout` which does not require having the layout data set; and `StandardSystemToolbar.createDialogArea` logically does not call `setLayoutData`. However, the tyranny of majority makes of our system believes that a call to `setLayoutData` is missing in this context.

EB: Encapsulation Breaking

In three cases among the strangest type-usages, the system finds breakings of object encapsulation, a particular case of the law of Demeter [Lie89]. While our system is not designed to find such violations, it turns out that these violations are also caught by the S-score. For instance, let us consider the following excerpt of `TrustCertificateDialog`:

```
1 certificateChainViewer = new TreeViewer(composite, SWT.BORDER);
2 certificateChainViewer.getTree().setLayout(layout);
3 certificateChainViewer.getTree().setLayoutData(data);
```

This code contains two violations of the law of Demeter, which both break the encapsulation provided by a `TreeViewer`. Our system detects them because these two violations are reflected in bytecode with two type-usages containing a single method call each. However, `Tree` objects never have only one call to `setLayout` or to `setLayoutData`, and the S-score of the two type-usages are accordingly very high.

WA: Workaround

The analysis of 3 strange type-usages revealed a phenomenon that we call *workarounds*. Both use an inappropriate widget to create a filler (an empty `Label` and an empty `Composite`). While this works (the resulting UI is satisfying), fillers are normally set using margins and paddings of `GridLayout`. In some special cases, the standard way of creating fillers is not perfect-looking and such workarounds are useful. The system catches these workarounds since the corresponding type-usages do not contain the usual method calls (e.g., `setText` on a `Label`).

In such cases, the client code is neither faulty nor bad designed, and we did not file bug reports because they are due to the usage of a workaround that addresses a limitation of the API itself.

4.4.5 Summary of the Evaluation

To conclude this section, we sum up the main results of our evaluation of the DMMC system:

- The S-score captures faulty type-usages.
- The algorithm which predicts missing method calls achieves a precision of 84% in guessing simulated missing method calls.
- A pragmatic variant of the algorithm improves the precision up to 98%, i.e., less than 2% of false positives.
- Our analysis of the strangest type-usages of the Eclipse codebase showed that the S-Score produced 17/19 true positive warnings and 2/19 false positive warnings. Furthermore, the bug reports we wrote have already resulted in 6 patches to the Eclipse source code (7 are still pending). These results confirm the very high precision obtained with the simulation of missing method calls.

4.5 Missing Method Call Detection in the Development Process

We pointed out in 4.2 that there are several manners of using DMMC. At development time, DMMC outputs the very likely missing method calls in the problem view of Eclipse. The corresponding prototype is an incubator project of the Code Recommender System that we build at TU Darmstadt[22].

At maintenance and quality assurance time, our missing method call detector is used in batch mode. The software engineer has to give the tool a set of Java class files (e.g., in a JAR file). Figure 4.8 illustrates the command line usage of our prototype and its output. Since the output consists of a list of recommendation, it can be easily integrated into any development tool and process (e.g. into an XML file or a table in a user-interface).

We now present a checklist that helps engineers interpret the missing method calls predicted by the DMMC system. This checklist comes from our own experience on using the system for searching for missing method calls in Eclipse.

1. **What is the responsibility of this method call?** The first thing to do is to carefully read the documentation of the missing method call to understand what its function is. This

[22] See `http://www.stg.tu-darmstadt.de/research/core/`.

```
## eclipse.jar is the name of the software to be analyzed
$ java -jar de.tud.st.DMMC.jar eclipse.jar
1. Analyzing source code ...
2. Computing E(x) and A(x) for 40000 type-usages ...
3. Computing S-score ...
4. Ordering by S-score ...
5. Computing missing method calls ...
type-usage: Composite
  location: GotoAddressDialog:createDialogArea, line 300
  S-score: 0.97
  missing call: <init>
....
```

Figure 4.8: The DMMC system in batch mode. The output could be text or XML.

gives crucial insights on what type of problems we might encounter when this method call is missing.

2. **Is the surrounding code of the type-usage strange?** Interestingly, our evaluation on Eclipse showed that often, although a method call is really missing, the solution is not to insert the missing call. It's rather more meaningful to identify and fix the warning at a larger scope (e.g., fixing a violation of API best practices). Hence, it is very important to analyze the context of a strange type-usage before modifying the code.
3. **Can this missing method call produce a bug in special use cases?** Also, if the context is correct, one may imagine a special usage of the software to let the problem appear (e.g., the font inconsistencies of figure 4.6).
4. **Is it a false positive due to the tyranny of majority?** Section 4.4.4 highlighted that our system is sensitive to tyranny of majority: developers might assume this if the two previous analyses (2 and 3) were inconclusive.

To sum up, the interpretation of the *predicted* missing calls is not straightforward but the time spent in such an analysis is rewarded by an improvement of the software quality and the diminution of the number of latent defects (cf. the evaluation on Eclipse). Hence, we do believe that using our approach systematically in software development processes helps produce better software and is eventually economically valuable.

4.6 Summary

In this chapter, we have presented a system to detect missing method calls in object-oriented software. Providing automated support to find and solve missing method calls is useful at all

moments of the software lifetime, from development of new software, to maintenance of old and mature software.

The evaluation of the system showed that: 1) the system has a precision of 98% in the context of an automatic evaluation which simulates missing method calls (55623 defects simulated) and 2) the high confidence warnings produced by the system on the Eclipse codebase are relevant and 6 of them were convincing enough for the Eclipse developers to patch the codebase. This is promising, especially if one considers the usual high false positive rates discussed in the literature.

5 Learning Rankings for Code Search

5.1 Introduction

Application frameworks are an integral part of today's software development process—this is hardly surprising given their potential benefits such as reduced costs, higher code quality, and shorter time to market. But before a framework can be used efficiently, developers have to learn its correct usage which often results in high initial training costs.

To reduce these costs, framework developers provide diverse documentation addressing different information needs. Tutorials, e.g., describe typical usage scenarios, and thus provide an initial insight into the workings of the framework. Alas, their benefit quickly disappears when problems have to be solved that differ from standard usage scenarios. Now, API documentation is scanned for hints relevant to the problem at hand. If the needed information is not found there, the most costly part of the research begins: The source code of other programs is searched for examples of successful *similar* usage of the framework. Open source hosting sites such as Google Code, Sourceforge, etc., offer a tremendous amount of potentially helpful code snippets, which by far exceed the number any human user could possibly sift through manually.

To support developers on searching example code, dozens of so called *code-search engines* have been developed by both academia and industry. Some well known representatives from industry are Koders (2004), Krugle (2005), or Google Codesearch (2006); academic projects include (among others) Strathcona (2005), XSnippet (2006), Parseweb (2007), and Sourcerer/SAS (2010).

However, designing effective ranking functions for code examples is a challenging task. Standard retrieval functions designed for document search do not work well on code [HM05b, BOL10]; hence, code-search engines are increasingly leveraging structural information available in code, such as inheritance relations, types and methods used or called in code, name similarities, or code complexity, to name just a few.

While undoubtedly meaningful, indexing such new information comes with its own challenge: The definition of the "right" scoring function to produce meaningful rankings, i.e., the definition of *good features* and the *best weights* for these features. With current code search technology, the administrator has to manually tweak the search engine (features in use and their weights) with the help of test queries until the produced rankings conform to her expectation.

There are two problems with this approach. First, performing the task is tedious and for systems

using dozens of (potentially interdependent) features guessing the right weights bears a significant risk of using non-optimal weights. Second, different users may have different views on what makes a good example, which in turn may not match the engine administrator's view.

This is where the approach presented in this chapter makes its major contribution. It leverages implicit and explicit user feedback to continuously learn about examples that were considered helpful to *automatically* improve upon the ranking functions by striving to rank the snippets considered as useful by the user on top of the list with the snippets considered as less useful further down. We use Support Vector Machines optimized for such ranking problems based on the work by Joachims [Joa02] for text-based document search engines. To the best of our knowledge, no previous approach features self-improved ranking functions based on developer feedback, in particular on the usage of click-through data [Joa02].

We have implemented versions of our approach with explicit and implicit user feedback. While explicit feedback seems an obvious choice, it is difficult to obtain [Joa02]. Hence, other sources of information need to be leveraged. One such source is how a user interacts with the search results: A user click on a given search result is considered "equivalent" to explicit positive feedback.

There is one precondition to using click-through data as positive feedback: The ability of the search engine to automatically generate meaningful summarizations [SJ07] of the retrieved documents. Such a summarization typically provides some insight into the document's content enabling the user to judge whether the document is likely to contain the needed information. Unlike automated summaries for text documents that have a long history in information retrieval research, creating summaries for source code has not been a very active research area. We consider the automatic creation of meaningful summaries of code examples that match a user query to be a second contribution of the work presented this here.

The results of several experiments show that leveraging user feedback, especially click-through data, can indeed *automatically* produce valuable feature weights, which yield rankings that in average outperform state-of-the-art code search engines, as represented by Strathcona [HM05b]. They also show that rankings improve with increasing user feedback and provide some valuable insights for future work on ranking functions for code search engines.

The remainder of the chapter is structured as follows. Section 5.2 presents the state-of-the-art in (example) code search. Section 5.3 provides background information on learning ranking functions by leveraging implicit and explicit user feedback. Section 5.4 presents our approach to learn such optimized ranking functions for code-search engines. The set up for our experiments is described in Section 5.5; Section 5.6 presents the results of the experiments. Section 5.7 concludes the work presented in this chapter.

5.2 State of the Art in Code Search

Text-based code search engines such as Google Code Search [goob], Krugle [kru], and Koders [kod] try to bring the success of document retrieval techniques to code example retrieval. Developers formulate queries in terms of API-keywords like class or method names. Code-search engines then search for code examples with these keywords and rank them according to measurements like tf-idf [CDMS08], cosine-similarity, subwords and others.

These code search engines treat source code files mostly as text documents. They make—what Krugle Engineers Ken Krugler and Grant Glouser call—a very "fuzzy parse" of source code by extracting only some information from code like the class' name or the names of its declared methods but treat the remaining parts of the source code merely as plain text documents. For instance, they do not build symbol tables as that would require processing all includes/import statements, nor do they resolve method call targets to their actually declaring type since this would require a full build or compilation of the source code. The rationale for the "fuzzy parse" of Krugle is the decision to enable processing files individually, without knowledge of build settings, compiler versions, etc. To achieve this goal imprecise query matchings are accepted but their effect is softened by various other information retrieval features like tf-idf, subwords matching etc. (cf. Apache Lucene in Action, 2nd Ed., chapter 12.1).

However, by ignoring the inherent structure of source code, a lot of similarity information that might improve search results gets lost. Furthermore, Krugle's way of indexing code allows for *ambiguity of terms*. Consider for instance a method call to `setMessage`. A quick search for all methods with the name `setMessage` shows that just in the Eclipse 3.6 code base there are 176 methods that have exactly the same name.

Many academic tools exist that advance the state-of-the-art of code-search engines in various directions. We will briefly discuss them and sketch which features they use to rank their recommended code examples.

To eliminate the ambiguity of terms code search engines have been developed that leverage structural information contained in source code. SOURCERER [BOL10] for instance has shown that the build environment could be easily reconstructed in many cases using the project's build and dependency information stored in various configuration files. For instance tools like Apache Maven, Apache Ivy, or Eclipse P2 contain enough meta-information to rebuild these dependencies with small effort, and thus make fuzzy parses unnecessary in many cases. SOURCERER and their recently proposed example code search engine SOURCERER SAS [BOL10] allows developers to search for classes, methods, class and method uses but also for the presence of loops or branches, number of declarations or micro patterns like *Stateless* or *Pseudoclass*. It extracts relations between API-elements from source code and uses additionally common text retrieval techniques to find code examples. Furthermore, it exploits a dependency counting feature similar to Google's PageRank to rank code snippets.

Similar to SOURCERER is JSearch [Sin06] which allows the developer to search for code exam-

ples with queries like “Uses class X and calls method Y”. JSearch finds matching code examples and ranks them according to the frequency of matches with the query.

ParseWeb [TX07] searches code examples by leveraging existing code-search engines but puts additional ranking logic on top of these examples like sequence heuristics taking into account the observed frequencies or the length of a matching method invocation sequence. Mapo [ZXZ+09] extends existing code search approaches by mining frequent method invocation sequences and recommending examples that have a certain overlap with such a mined sequential pattern and the (incomplete) code at hand. The Strathcona Example Recommendation Tool [HM05b] allows the developer to invoke a search from the Eclipse editor; Strathcona extracts structural information from the developer code and uses this information in six heuristics for finding matching code examples. Each heuristic either evaluates positive or negative for a code example, e.g., *Calls Best Fit* selects code examples that call at least 40 percent of the methods called in the developer code. Strathcona ranks code examples according to the number of heuristics they "satisfy". CodeGenie [LLBO07] features test-driven code search. Given a JUnit test-case for a method, CodeGenie finds code examples that satisfy this test-case.

Assieme [HFW07] takes a developer query, adds code-specific terms like “public”, “static” and “import” and forwards it to web search engines like Google. It then extracts code examples from the returned web pages and presents them to the developer. CodeBroker [YFR00] searches code examples by using the comments in the developer’s code at hand to find code examples that have similar comments. Hill and Rideout used code search also for automatic method completion [HR04] in JEdit. Whenever a developer invokes the method completion inside his code, the plug-in then uses clone detection techniques to find and rank code examples similar to the developer code and synthesizes method completions. XFinder [DR08] for instance finds code examples for Mismar guides [DO07]—step-by-step descriptions of how to accomplish certain tasks in a framework. eXoaDocs [KLwHK10] augments API-documents with code examples.

The list of approaches presented so far shows that state-of-the-art tools for code search leverage a rich feature set to rank code examples, ranging from plain text retrieval features like tf-idf, PageRank, Subwords matching, or cosine similarity to code specific features such as information about inheritance relations, method calls observed, used types in code, various method complexity metrics, declared fields, method call sequences, pattern overlap metrics, comments in source code, and many more. But, none of these approaches ever discussed how these features can be "ideally" combined. We argue that most these systems give away a huge potential to improve their services by ignoring the power of implicit user feedback.

The system we propose in this chapter does not propose any new features for code search. What distinguishes it from existing approaches is its ability to learn ranking functions from user feedback–whatever the basic feature set is that is used for searching—and related to it the ability to produce meaningful code summaries. In this paper we show as proof-of-concept how such a self-learning system could be implemented for code-search engines. Yet, the proposal prepares the way for leveraging implicit user feedback not only for code-search engines but

virtually for any system that ranks and scores any kind of code documents—as long as users are allowed to interact with the results and the systems are capable to track these interactions for later analysis.

5.3 Learning Ranking Functions

This section provides background information on learning optimized ranking functions in information retrieval by using explicit or implicit user feedback. Readers familiar with the concepts of learning ranking functions from implicit and explicit user feedback may skip this section.

5.3.1 Scoring Documents

Given some query, the list of documents that match the query can by far exceed the number a human user could possibly sift through. Accordingly, it is essential for a search engine to rank the documents matching a query. To do this, the search engine computes, for each matching document, a *score* with respect to the given query. The most general scoring function frequently used in information retrieval is the weighted zone scoring function [CDMS08] given below:

$$score(d, q) = \sum_{i=0}^{n} w_i \cdot f_i(d, q)$$

Given a query q the search engine determines the score of each document d in its database as follows. For each feature i the engine computes the document's feature score $f_i(d, q)$, and multiplies it with a feature weight w_i. The resulting values are then accumulated to a single score and finally rank-sorted in ascending order.

With this formula in place, the challenge is to determine (a) the 'right' features, and (b) the 'best' weights for these features, so that the helpful documents are placed on top of the list. While an initial set of features is typically decided on rather quickly, determining their weights is much more difficult.

The weights are traditionally determined manually by the search engine administrator. She uses a few test queries and manually checks whether the code snippets listed on top are actually those she would have expected. If not, she tweaks the weights until the rankings produced conform to her expectation.

More recently, machine learning techniques are used to automatically compute optimal weights based on training examples that have been judged editorially by the administrator or by having users explicitly rate the relevance of the proposed documents [CS01]. In information retrieval, this methodology is known as *machine-learned relevance*.

Unfortunately, judging the relevance of documents is expensive especially for collections that

change frequently (like open source hosting sites). Furthermore, only few users are willing to give explicit feedback, making significant amounts of such data difficult to obtain. An alternative approach is to extract implicit relevance feedback from search engine log files (such as in [CSS99, Joa02]). We implemented and compared both approaches in the context of code-search engines, thus, will describe the concepts of both in the following subsections.

5.3.2 Exploiting Relevance Feedback

Let's first look on how rankings can be optimized generally in an automated way by using any kind of relevance feedback. Assume we have a set of training examples (d, q)—pairs of a query q and a document d—together with a relevance judgment $r(d, q)$ for d on q. In the simplest form, each relevance judgment is either *Relevant* or *Non-relevant* typically mapped to the numerical values '1' for Relevant and '0' for Non-Relevant. The weights w_i are then 'learned' from these examples, such that the learned scores approximate the relevance judgments in the training examples by minimizing the quadratic error ϵ overall training examples Φ:

$$\epsilon(w, \Phi(d, q)) = \sum_{j=0}^{|\Phi|} (r(d_j, q_j) - score(d_j, q_j))^2$$

This formula describes the basic concept to solve this optimization problem; more sophisticated approaches exist, such as the Support Vector Machine approach [Joa02] we use in our system. However, all these algorithms are based on a loss/error function similar to the one given above. Thus, this light-weight introduction to learning optimized ranking functions should be sufficient to understand the basic concepts without going into the details of the optimization using Support Vector Machines and all kinds of optimization problems.

5.3.3 Exploiting Explicit Feedback

Let's introduce the concept of improving rankings based on explicit user feedback in a more visual way now. Assume a developer issued a query that returns four code examples as depicted in Figure 5.1. Assume further that the developer judged the quality of these examples by rating example 1 and 4 as being *Relevant* but rated example 2 being *Non-relevant* (the explicit feedback is indicated using plus and minus signs behind the examples).

From the user expectation we can derive an "optimized" ranking (i.e., better than the initial one) such as the one given on the right of Figure 5.1. Here, examples 1 and 4 are ranked on top, followed by the neutral (unrated) example 3, and then followed by the negatively rated example 2.

Intuitively, an optimal search engine would place examples that are rated helpful by the user on top of list, followed by less useful examples, and not useful examples at the bottom. *But how*

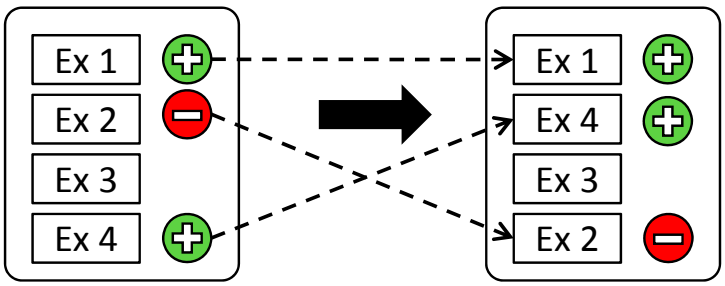


Figure 5.1: Improving rankings leveraging *explicit* feedback

can we leverage user feedback for particular examples to improve the search engine for future queries? The key here is a detailed analysis of the features that contributed to the selection of individual examples for which we have feedback. Roughly speaking, features that contributed to the selection of the examples for which we get positive feedback should be weighted higher than features that don't.

For illustration, assume that in our example we use the weighted zone scoring function with three features f_1, f_2, f_3. Table 5.2a gives a detailed view on the features, their weights and the extent in which they are present in the examples. The current weights imply that f_1, and f_3 seem to be equally important and f_2 only half as much as the other two features. The total of each feature score multiplied with its corresponding weight produces the final score, which is given in the bottom row of Table 5.2a.

Given the individual feature scores each example achieved without their weights, we can now start the optimization approach to learn new feature weights so that scoring these documents a second time with these new weights would result in our optimized ranking where example 1 and 4 are ranked on top and example 2 on the bottom. Table 5.2b shows *one* such optimized feature weighting that would have produced the optimized ranking. Assume that the algorithm has learned that f_1 seems to be very effective to find the relevant examples whereas f_3 seems to be overrated by mistake. Thus, the algorithm may correct the scoring function by using the new weight 2.0 for f_1, and a negative weight -1.0 for f_3 for subsequent requests.

Two important notes are due at this point. First, we might have produced a ranking with the fourth example on the first position in the ranking. Since there is no difference in the relevance of example 1 compared to example 4, this would be perfectly okay (in terms of the loss function used), as it would have produced an equivalent ranking. Second, tweaking the feature weights does not boost the positively rated examples only. It's rather a general way to boost *all* examples that possess the same features. For instance, the algorithm might learn that a high overlap between the called methods in the example code compared to the query is extremely important, and thus may assign larger weights to features that reflect this overlap.

Table 5.1: Example scoring with feature weights

feature	weight	Ex 1	Ex 2	Ex 3	Ex 4
f_1	1.0	1.0	0.2	0.5	0.8
f_2	0.5	0.4	0.0	0.8	0.0
f_3	1.0	0.2	1.0	0.1	0.1
score	—	1.4	1.2	1.0	0.9

(a) before optimization

feature	weight	Ex 1	Ex 2	Ex 3	Ex 4
f_1	2.0	1.0	0.2	0.5	0.8
f_2	0.5	0.4	0.0	0.8	0.0
f_3	-1.0	0.2	1.0	0.1	0.1
score	—	2.0	-0.6	1.3	1.5

(b) after optimization

5.3.4 Exploiting Implicit Feedback

The basic idea underlying implicit feedback is as follows. For each document returned, the search engine creates an *automated summarization* [SJ07] of the document. This summarization typically contains the document's title and sentences that contain some of the query terms, etc., and hence gives some insight into the document contents enabling a user to judge whether the document is likely to contain the needed information before clicking on the document's link to see the full content. Thus, it is reasonable to derive from a user click on a link that the document carries some valuable information, making it a weaker version of a positive feedback.

Such kind of implicit feedback is easy to collect from a search engine, available in masses, and always up to date. Furthermore, it has been shown that this approach works well on text search engines (in particular, Google) [Joa02]. However, for this approach to work the automatically created summaries must be meaningful, i.e., a user must be able to judge whether a document might be relevant by purely looking at them. If the summaries are meaningless, the user will click through all recommended documents, thus incorrectly marking them all as relevant, which makes the whole feedback useless.

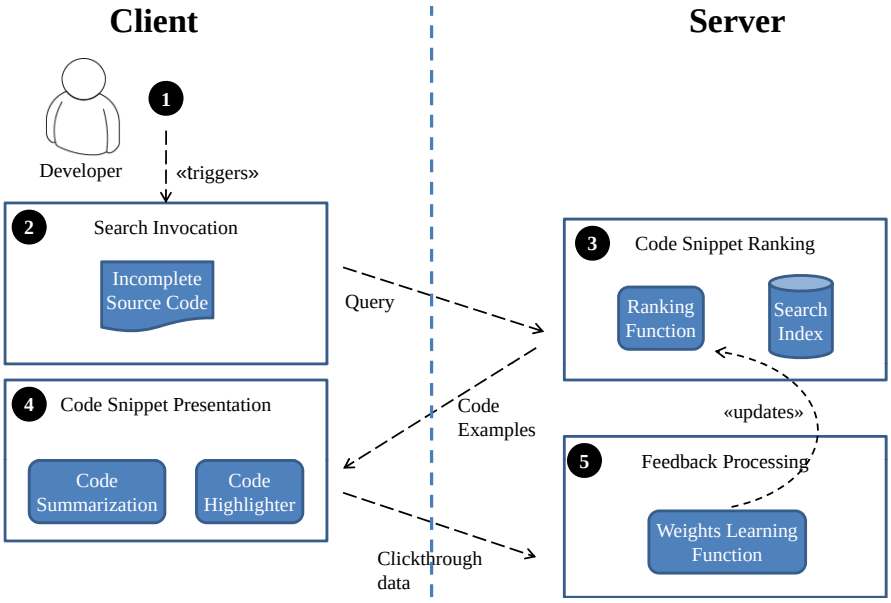


Figure 5.2: The workflow of CoRe$_{\text{LUCID}}$

5.4 Approach

In this section, we present CoRe$_{\text{LUCID}}$, a code recommender leveraging user click-through data, based on Strathcona's [HM05b] model. There are several reasons for using Strathcona as a baseline. First, it is publicly available, thus, enabling a reference comparison between a 'classic' approach working with manually-defined feature weighting and a pure feedback-based system. Second, the concepts behind Strathcona still represent the state-of-the-art, making this comparison valuable for the research community. Third, Strathcona's ranking system is a derivation of the weighted zone scoring function (each feature weight is set to 1.0) we introduced above. This makes it easy to interpret the findings we will present in section 5.6. Fourth, Strathcona is tightly integrated into the Eclipse IDE, thus enabling the collection of various kinds of (implicit) user feedback, which could not be tracked with pure web-based approaches, e.g., the detection of copy & paste events from a code example into the developer's code.

5.4.1 Workflow Overview

CoRe$_{\text{LUCID}}$ consists of two components: a client component living inside the developer's IDE and a server component responsible for searching the index, creating code summaries, and learning new ranking functions whenever new feedback data arrives. Figure 5.2 illustrates the basic workflow between client and server consisting of the following main steps:

1. The developer issues a code search request from inside her IDE via code completion or via the context menu.
2. The client component automatically extracts several code properties from the active editor, creates a **query** and sends it to the server.
3. The server queries its **search index** for code examples relevant to the given query, ranks these snippets according to its **ranking function**, and sends them back to the issuing client.
4. The results are displayed inside the IDE as **code summarizations**. Further, the client tracks how the developer interacts with the search results, i.e., which examples she looked at, which ones she judged as helpful, etc. This feedback is sent back to the server.
5. After a certain amount of so called **click-through data** has become available, the server evaluates the collected user feedbacks and learns a new ranking function according to the feedback so that it would have produced the best possible rankings for all queries.

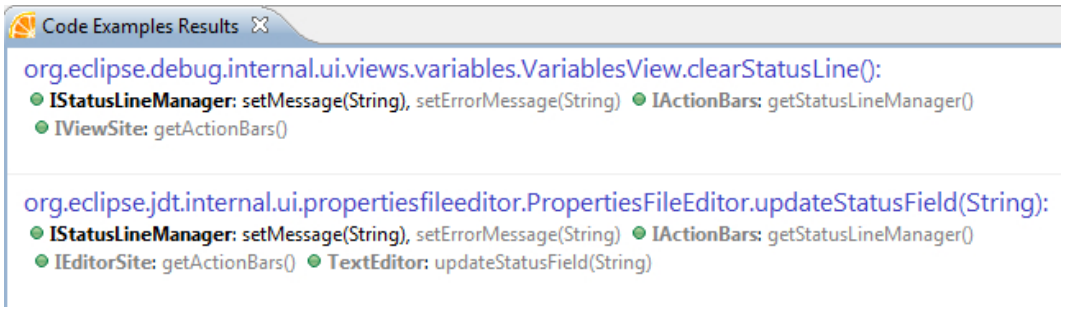
5.4.2 Search Index and Query Creation

The index is deliberately kept simple and is designed to be almost similar to the index structure used by Strathcona. It indexes two types of documents: methods and classes. All type and method names are stored with their full qualifying names.

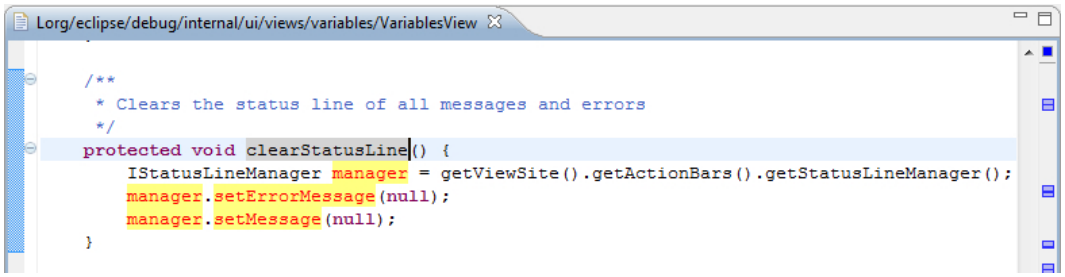
Method documents basically consist of an ID, the name of the overridden method (if any), a set of types used, and a set of methods used inside the body. Class documents subsume their corresponding method documents but add further information about superclasses, implemented interfaces, and declared fields. No further information is indexed. In total, our index stores the same six code properties for methods and classes as Strathcona (referred to as extends, implements, fields, overrides, calls, and uses code properties in the following).

```
Method: {
  id: <full qualified method name>
  source: <source code>
  overrides: <overridden method>
  used types: <set of types used in code>
  used methods: <set of methods used in code>
}

Class: {
  id: <full qualified class name>
  source: <source code>
  extends: <set of superclasses>
  implements: <set of implemented interfaces>
  fields: <set of declared field types>
  overrides: <set of overridden methods>
  used types: <set of types used in code>
  used methods: <set of methods used in code>
```



(a) Code Summarization View



(b) Source Code Highlighting in Example Code

Figure 5.3: CoRe$_{\text{LUCID}}$ Integration into Eclipse

```
18 }
```

We use Apache Lucene 3.0, but since Lucene is primarily designed to score text documents, we implemented our own scoring function that is suitable for scoring source code as described in the next section.

CoRe$_{\text{LUCID}}$ offers three types of search: *class*, *method*, or *selection*. Class queries include all types and methods used within the class' source code as well as fields, extended super-classes, and implemented interfaces. Method queries contain all types and methods used within the method's body and (if appropriate) the name of the overridden method. Selection queries create a subset of this data structure depending on the text selection in the active editor.

Depending on the query type, code-search engine-toolname automatically extracts the corresponding information from the active editor and fills a search request. The search request is basically a one-to-one match of index structure plus some additional information like the request ID, which is used later on for training the system:

```
1 Search Request: {
2   kind: <class | method | selection>
3   unique-user-id: <uuid>
4   unique-request-id: <random uid>
5   extends: <set of all superclasses>
```

```
  implements: <set of implemented interfaces>
  fields: <set of declared fields types>
  overrides: <set of overridden methods>
  used types: <set of types used in code>
  used methods: <set of methods used in code>
}
```

After the search request is created, it is sent to the server.

5.4.3 Ranking Function

Once a query arrives at the server, relevant documents (i.e., code snippets) must be identified and ranked. Strathcona uses six heuristics which leverage the indexed code properties. Some jointly aggregate several such properties. The 'calls with inheritance' heuristic, e.g., leverages the method calls information together with information about superclasses and implemented interfaces. We decomposed each of these heuristics into individual scoring features which allows tweaking and judging their relevance individually. For each of the six indexed code properties we defined two generic scoring features:

Query-to-Snippet Overlap Ratio: For each code property p, query q and document d, $q2s\text{-}score$ computes the overlap between the terms for p in q and d:

$$q2s\text{-}score(d, p, q) = \frac{|Q_p \cap D_p|}{|Q_p|}$$

Snippet-to-Query Overlap Ratio: For each p having at least one term in q, we also measure the overlap between the terms in d with the terms found in q using the following formula. This score becomes 0 if the query and snippet are identical, i.e., the snippet possesses the same items as the query, but gets higher the more terms are contained in the snippet but not in the query. The idea behind this score is to possibly penalize huge divergence between snippet and query using negative weights for this feature.

$$s2q\text{-}score(d, p, q) = 1 - \frac{|Q_p \cap D_p|}{|D_p|}$$

Calculating the $q2s$ and $s2q$ scores for all six indexed code properties results in 12 features for each document, which are aggregated into a final score using the weighted zone scoring function (cf. Section 5.3.1). Next, the list of relevant documents is sorted in ascending order by their score and is returned to the client along with the required details to create the code summarization in Eclipse.

Please note that we kept the scoring used for this evaluation deliberately simple to make the performance results comparable to Strathcona's. In our production system we have implemented

advanced features such as tf-idf, query enrichment, personalized feature weights, etc. However, these advanced features were turned off for the experiments presented in this chapter, as their use improves the overall performance of the system even in absence of user feedback. Hence, their use would distort the results of experiments aimed at backing the claim that learning from user feedback improves code search engines independent of the basic features in use.

5.4.4 Code Summarization

Figure 5.3a shows how summarizations for recommended example methods are displayed inside the IDE. The summarizations are created as follows:

Each code snippet is presented using a title and a *types and methods* usage block. The code snippet's declaration statement is used as title, i.e., for classes the full qualified class name followed by the extends and implements clauses with a list of the implemented interfaces is used; for methods we used the full qualified method signature. The usages block gives the methods used in the code snippet grouped by their declaring types and orders them by their frequency in the index, the assumption being that frequently used types and methods are more important than rarely used ones.[1] Furthermore, we put the blocks that contain methods and types used in the query at the beginning of the summarization and highlight terms occurring in both the query and the example document in bold. Finally, tool tips provide additional information, e.g., the individual feature scores of each example.

Based on this information the user chooses the examples to look at. After a click on a code example, the source code is displayed inside a new editor. To allow developer to quickly identify the relevant code fragments, statements that occurred in both query and code snippet are highlighted and their positions are marked (cf. Figure 5.3b).

5.5 Evaluation Setup

We evaluated the effectiveness of our click-through-based approach by answering the following four questions:

1. Can user click-through data actually be used to improve the performance of code-search engines, in particular when compared to explicit feedbacks?
2. Is such a system competitive to state-of-the-art code search engines (Strathcona)?
3. What is the optimal performance we can expected from such a user feedback driven approach (given the benchmark defined in the next section)?
4. How much click-through feedback is needed to improve code-search engines over time?

[1] We discard calls to synthetic (i.e., compiler-generated) and to private methods since they are not callable by clients and rarely contain valuable knowledge.

5.5.1 Benchmark

Since there is no widely accepted benchmark for comparing code-search engines, we had to generate the baseline rankings for the evaluation ourselves. The approach for doing so is described in this subsection.

We selected six example usage scenarios from the Eclipse FAQs.[2] Each scenario has different demands on what the search engine should return; some scenarios only apply in a certain class context, others make extensive use of several different Eclipse APIs, etc. Readers familiar with related work may be familiar with some of the scenarios, since they have been used by others [HM05b, TX07]:

Create AST. In the first scenario we simulate a developer who wants to parse some source code given as a string using the class `ASTParser`. Therefore she must obtain an instance of AST Parser via the method `AST Parser.newParser(AST.JLS3)`, call some additional methods to configure the parser and finally get an instance of `ASTNode` via the method `ASTParser.createAST(IProgressMonitor)`. Typically the obtained ast node is then casted into a `CompilationUnit`. The snippet below shows the code fragments we would like our search engine to return:

```
String source = ...
ASTParser parser = ASTParser.newParser(AST.JLS3);
parser.setSource(source.toCharArray[]);
parser.setResolveBindings(true);
parser.setKind(ASTParser.K_COMPILATION_UNIT);
CompilationUnit cu = (CompilationUnit) parser.createAST(null);
```

Listing 5.1: Expected code for creating an AST from source

For evaluation of the search engine, we assume that the developer already knows `ASTParser` and is aware of `ASTParser.setSource(char[])`. We assume further that she does not know how to create the parser and how to obtain an instance of `CompilationUnit`. The code used to create a query for this scenario is as follows:

```
private void createASTFromSource(String source) {
  ASTParser parser = null;
  parser.setSource(source.toCharArray());
  CompilationUnit cu;
}
```

Listing 5.2: Code fragment used to create the search query for creating an AST

Get Active Page. Let's assume a developer needs to obtain the currently active page of the Eclipse workbench. The common solution to obtain this instance is the call sequence

[2] See `http://wiki.eclipse.org/index.php/Eclipse_FAQs`.

`PlatformUI.getWorkbench().getActiveWorkbenchWindow().getActivePage()`. Often developers know that they can access the active page via the interface `IWorkbench` that is returned by `PlatformUI.getWorkbench()` but do not recall the exact call sequence or that they have to use `PlatformUI` to obtain the workbench. Additionally, using exactly this call sequence is discouraged. Developers should check the return values of the latter two methods for `null` since there may be no active window or active page. A helpful code example should look similar to the one given below:

```
1 IWorkbenchWindow window = PlatformUI.getWorkbench().
      getActiveWorkbenchWindow();
2 if(window == null) {
3   // handle
4 } else {
5   IWorkbenchPage page = window.getActivePage();
6   if(page == null) {
7     // handle
8   } else {
9     // use page
10  }
11 }
```

Listing 5.3: Expected code for obtaining the active page

Here we assume that the programmer already knows that she has to use `IWorkbench` and that she needs to obtain an instance of `IWorkbenchPage` but has no clue how to obtain any of these instances. The code used to create a query is as follows:

```
1 private IWorkbenchPage getPage() {
2   IWorkbench workbench;
3   return null;
4 }
```

Listing 5.4: Code fragment used to create the search query for obtaining the active page

Get Resource. In this scenario the developer needs to identify the resource underlying the currently active editor. For this, an instance of IEditorInput is given and an instance of IResource is sought. Most of the time the sought resource will actually be an instance of IFile, a child of IResource, thus, there are three ways to solve this scenario:

Cast the editor input to `IFileEditorInput` and use `IFileEditorInput.getFile()`. This only suffices if the editor is actually an editor with a file resource. Alternatively, one may use the utility method `ResourceUtil.getFile(IEditorInput)`. As for the previous solution this only suffices for editors with a file resource. Last, one may use the adapter interface, e.g. call `IEditorInput.getAdaptable(Class)`. This is the preferred solution. A excerpt of helpful code examples is given below:

```
1 IEditorInput input = ...
```

```
if(input instanceof IFileEditorInput) {
  IResource resource = ((IFileEditorInput)input).getFile();
  // use resource
} else {
  // handle
}

IEditorInput input = ...
IResource resource = ResourceUtil.getFile(input)
if(file == null) {
  // handle
} else {
  // use resource
}

IEditorInput input = ...
IResource resource = (IResource) input.getAdapter(IFile.class);
if (resource == null) {
  resource = (IResource) input.getAdapter(IResource.class);
}
if(resource == null) {
  // handle
} else {
  // use resource
}
```

Listing 5.5: Expected code snippets for getting the resource from editor input

As input to the query generator we assume that the developer has an instance of IEditorInput and wants to extract an instance of IResource as indicated by the paramter and method return value respectively. The code used to create a query for this scenario is as follows:

```
private IResource extractResource(IEditorInput input) {
  return null;
}
```

Listing 5.6: Code fragment used to create the search query for getting the resource from editor input

Create Marker. Assume a developer has to create a marker for a file. The marker itself is instantiated via the method `IFile.createMarker(String)`. Several attributes must be set for the marker so that Eclipse will display it. A generic example for this scenario may look as follows:

```
IMarker marker = file.createMarker(MARKER_ID);
marker.setAttribute(IMarker.LINE_NUMBER , ...);
marker.setAttribute(IMarker.CHAR_START , ...);
marker.setAttribute(IMarker.CHAR_END , ...);
marker.setAttribute(IMarker.MESSAGE , ...);
```

```
6  if (...) {
7    marker.setAttribute(IMarker.PRIORITY , IMarker.PRIORITY_HIGH);
8    marker.setAttribute(IMarker.SEVERITY , IMarker.SEVERITY_ERROR);
9  } else {
10   marker.setAttribute(IMarker.PRIORITY , IMarker.PRIORITY_NORMAL);
11   marker.setAttribute(IMarker.SEVERITY , IMarker.SEVERITY_WARNING);
12 }
```

Listing 5.7: Expected code for creating markers

As input to the query builder we assume that the developer did not know how to create the marker but does know that she will have to set attributes via `IMarker.setAttribute(String, ...)`. The code used to create a query for this scenario:

```
1 public IMarker createMarker(IFile file) {
2   IMarker m = null;
3   m.setAttribute("attribute", "value");
4   return null;
5 }
```

Listing 5.8: Code fragment used to create the search query for creating markers

Use Progress Monitor. In this scenario the developer uses a progress monitor to give feedback about a task. The progress monitor is accessed via an instance of `IProgressMonitor` and altered with various methods to reflect progress of the task. Once the task is finished one must call `IProgressMonitor.done()` inside a try-block and call the method in the `finally`-block. The snippet below shows the characteristics a code example should posess:

```
1 IProgressMonitor monitor = ...
2 try {
3   // use monitor
4 } finally {
5   monitor.done();
6 }
```

Listing 5.9: Expected code showing how to use a progress monitor

For this scenario we assume that the developer already uses a progress monitor, but did not know that he must call `IProgressMonitor.done()`. The code used to create a query for this scenario:

```
1 public void useProgressMonitor(IProgressMonitor monitor) {
2   monitor.beginTask("Performing task: ", 10);
3   monitor.subTask("step");
4   monitor.worked(1);
5 }
```

Listing 5.10: Code fragment used to create the search query for using progress monitors

Update Status Line. For the final scenario let's assume a programmer implements a view and aims to inform it's users about the progress of several long running operations. A common way to do so is to show short messages on the status line of the Eclipse workbench. To get access to the status line, she needs to get a handle on an `IStatusLineManager` which can be obtained by the call sequence `getViewSite().getActionBars().getStatusLineManager()` where `getViewSite()` is a method of the view base class `ViewPart`. The following code snippet shows the relevant cod fragment a search engine should return for this scenario:

```
1 IStatusLineManger statusLine = getViewSite().getActionBars().
      getStatusLineManager();
2 statusLine.setMessage(...);
```

Listing 5.11: Expected code for updating the status line

For evaluation we assume that the developer wants to obtain the status line manager inside an extension of `ViewPart`. The code used to create a query for this scenario is depicted below:

```
1 public class ExampleView extends ViewPart {
2   public void reportStatusMessage(String message) {
3     IStatusLineManager statusLine = null;
4     statusLine.setMessage(message);
5   }
6 }
```

Listing 5.12: Code fragment used to create the search query for updating status line

For each scenario, we queried Strathcona for its recommendations and investigated the usefulness of the returned ten recommendations manually by classifying them into one of five relevance groups:

1. Examples that present the helpful code clearly.
2. Examples that contain the helpful code, but do not present it well.
3. Examples that present some of the helpful code clearly.
4. Examples that contain some of the helpful code, but do not present it well.
5. Examples that contain no helpful code.

Based on this classification, we created a *perfect ranking* r^* by sorting the examples by relevance groups. If a group contained more than one example, Strathcona's ranking was used to sort the snippets within this group. The resulting rankings serve as our evaluation baseline.

There are two potential biases of our perfect rankings. First, the classification into relevance groups made by the authors may potentially impact the performance evaluation. However, since our classification is based on the recommended code snippets given in the Eclipse FAQ, we believe that this does not favor one engine over the other. Second, the use of Strathcona's ordering for examples belonging to the same relevance group gives Strathcona some advantage

when comparing the overall ranking quality. This bias is intended to ensure the most objective evaluation with Strathcona.

5.5.2 Evaluated Systems

We defined four systems, each leveraging different kind of information to rank code examples, as subjects for our evaluation:

The reference system (Strathcona) We evaluate the performance of Strathcona on the given scenarios to establish a baseline to compare the other approaches with.

The explicit feedback-based system (Explicit) As described in Section 5.3.3, systems can learn feature weights based on explicit relevance feedback. We allowed our users to judge the quality of the examples by marking them as relevant or non-relevant, and learned the feature weights based on these judgments. This system establishes another baseline to compare the click-through-based approach against.

The click-through-based system (Implicit) This system learns its weights based on the implicit user-feedback as described in Section 5.3.4. Our hypothesis is that a pure click-through-based system achieves a competitive performance compared to both Strathcona and an explicit feedback-leveraging system. The system is trained with user click-through data extracted from 55 queries.

The *optimal* system (Oracle) The *oracle* learns its weights directly from the optimal rankings r^* for the six evaluation scenarios as outlined in section 5.3.2. Obviously such a system is overfitting the data and may not generalize well for other scenarios. However, as it learns its weights from perfect rankings, the oracle should outperform all existing approaches; thus, it can serve as an upper bound for the performance a system may achieve over all evaluation scenarios.

5.5.3 Evaluation Metrics

While Section 5.5.2 defines the subjects of evaluation, this subsection defines the metrics used for the evaluation. The problem of information retrieval can be formalized as follows. For a query q and a document collection $D = d_1, ..., d_n$, the optimal retrieval system should return a ranking r^* that optimally orders the documents in D according to their relevance to the query. Typically, retrieval systems do not achieve this optimal ordering r^*. Therefore, an operational retrieval function f is evaluated by how closely its ordering r_f approximates the optimum for the given query.

But what is an appropriate measure of closeness or similarity between a ranking r_f and the optimal target ranking r^*? Two commonly used metrics in information retrieval are *Average Precision* [BYRN99] and *Kendall's τ* [Joa02]. Each focuses on slightly different aspects of the

ranking, which we will describe in the following.

Average Precision

Let's consider the ranking $r_f = \{ex_1, ex_2, ex_3, ex_4\}$ with ex_2 and ex_3 being relevant. To measure how well the system ranked the relevant documents on top, Average precision can be used as follows. Starting from the first rank, we count for each rank containing a relevant document how many relevant documents have been found so far and divide this number by the current rank. These values are then summed up and divided by the total number of relevant examples:

$$avg\text{-}prec(r_f) = \frac{\sum_{s=1}^{|S|}(\text{rel}(s) \times \text{prec(s)})}{|\{\text{relevant examples}\}|} \in [0, 1],$$

where s ranges over the ranks, S is the number retrieved examples, rel a binary function on the relevance of a given example at position s, and $prec(s)$ the precision at a given cut-off rank, defined as:

$$prec(s) = \frac{|\{\text{relevant retrieved ex. of rank s or less}\}|}{s} \in [0, 1]$$

For the example ranking given above Average Precision is

$$avg\text{-}prec(r_f) = \frac{0 + \frac{1}{2} + \frac{2}{3} + 0}{2} = \frac{7}{12} \approx 0.58.$$

Values close to 1 indicate that the relevant documents are ranked on top, whereas values close to 0 indicate that relevant examples are far down the ranking. Clearly, a code search engine should focus on achieving a very high Average Precision: a developer typically seeks good examples and may abort the search after the first helpful code example has been found.

Kendall's τ

For a binary relevance scale, Average Precision is frequently used. However, most information retrieval researchers agree that binary relevance (Relevant/Non-Relevant) is somewhat coarse since it does not allow judging the quality of complete ranking. To compare our feedback-based system with Strathcona we will thus also measure the performance of all systems by comparing them to the optimal ranking r^* using Kendall's τ, a frequently used measure to assess the overall quality of rankings in statistics.

Let's assume we got rankings r_a and r_b given below, each ranking the examples ex_1 to ex_5. Ranking r_b is almost similar to r_a but has flipped ex_1 with ex_3. Thus, both rankings disagree

in the way how they order three elements, namely, $\{ex_1, ex_3\}$, $\{ex_2, ex_3\}$, and $\{ex_1, ex_2\}$, and agree on the remaining 7 pairs.

$$ex_1 <_{r_a} ex_2 <_{r_a} ex_3 <_{r_a} ex_4 <_{r_a} ex_5$$
$$ex_3 <_{r_b} ex_2 <_{r_b} ex_1 <_{r_b} ex_4 <_{r_b} ex_5$$

To measure the similarity of both rankings we use Kendall's τ which is defined as follows:

$$\tau(r_a, r_b) = \frac{p - q}{p + q} \in [-1, 1]$$

whereby p denotes the number of concordant pairs and q denotes the number of discordant pairs between the rankings r_a and r_b . More formally, a pair (ex_1, ex_2) is said to be concordant, if the rankings for both elements agree, i.e., r_a and r_b either both rank ex_1 *over* ex_2 or both rank ex_1 *under* ex_2. Otherwise the pair is discordant. Now, given that our example rankings agreed on seven pairs and disagreed on three, $\tau(r_a, r_b) = 0.4$.

Generally speaking, the higher the value of $\tau(r_a, r_b)$ for two rankings r_a and r_b, the higher is the similarity of the rankings. In case $\tau(r_a, r_b) = 1$ both rankings are exactly the same; if $\tau(r_a, r_b) = -1$ then both rankings are reverse to each other. For our evaluation we will compare each ranking produced by one of our systems with the manually created optimal ranking r^* to measure how close the rankings get to the optimal results.

5.6 Results

We split the discussion of the evaluation results into two subsections: The first subsection answers the first three evaluation questions by presenting the performance of the four subject systems in terms of Average Precision and Kendall's τ. The second subsection shows how user feedback affects the performance of the system, and thus answers the fourth question stated in Section 5.5.

5.6.1 Comparing System Performances

Figure 5.4 presents the performance measures for each system and evaluation scenario as scatterplots. Each data point in the plot represents the system's performance; the x-coordinate giving the Average Precision the system achieved and the y-coordinate giving its Kendall's τ score, respectively. To determine whether a system s_1 is superior to another system s_2 one simply compares the systems' x and y coordinates. If the x coordinate of s_1 is greater it achieved a higher Average Precision than s_2; if the y coordinate of s_1 is greater than it archived a higher Kendall's τ score as s_2. A system clearly outperforms another one if it wins in both dimensions, i.e., it has both a higher precision and a higher Kendall score.

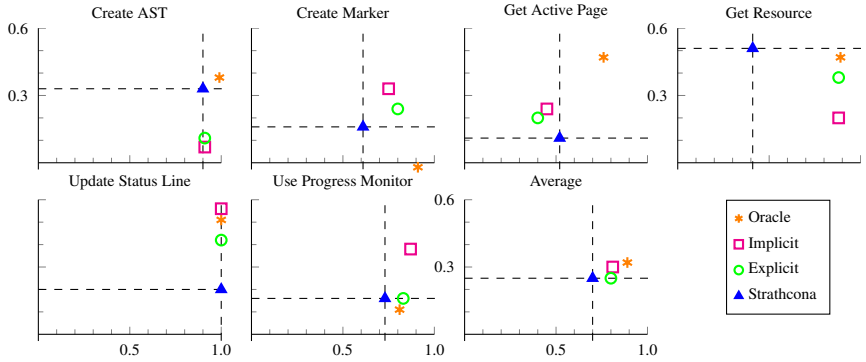


Figure 5.4: Average Precision and Kendall's τ performances

Given this simple metric, the click-through-based approach clearly wins 3 of 6 scenarios compared to Strathcona (Create Marker, Get Status Line, Use Progress Monitor). In the remaining three scenarios the results are divergent; Strathcona is superior in one dimension but has a weaker performance in the other. However, when looking at the average performance over all scenarios the click-through based approach wins over Strathcona w.r.t. either metric—despite Strathcona's advantage when measuring with Kendall's τ.

Let's discuss the systems' performances by looking on Average Precison first (by considering the x coordinates only). Given the experiment results, both feedback-based systems identified and ranked the actually relevant examples better than Strathcona in all but one scenario (*Get Active Page*). In average, Strathcona achieved an Average Precision of 0.7, Explicit and Implicit reached approx. 0.8 (+14 % rel./+10 % abs.); Oracle reached an Average Precision of 0.89 (+29 % rel./+14 % abs.). From this result we conclude that both feedback-based systems are actually able identify the relevant examples and present those high up in the ranking–with a slight advantage for the implicit system. Given that developers start traversing the results list from top down until the first helpful example has been found, this result is very promising and shows a clear advantage of our system.

However, when looking at Kendall's τ (by considering the y coordinates only) a more distinguished picture is drawn. In general, no system was able to create rankings very close to the perfect rankings. The achieved scores varied from -0.02 to 0.58 at most. Both Strathcona and Explicit reached a score of 0.25, Implicit reached 0.30 (+20 % rel./+5 % abs.), and Oracle achieved a score of 0.32 (+28 % rel./+6 % abs.). Interestingly, the explicit system did not achieve a better performance than Strathcona. Thus, our Support Vector Machine approach could not derive better feature weights from the rare explicit feedbacks. In contrast, the same algorithm improved the performance by 20 % by leveraging the implicit feedbacks extracted from the user interactions only. This observation gives evidence that the click-through-based approach performs better than systems that require explicit feedback to learn ranking functions.

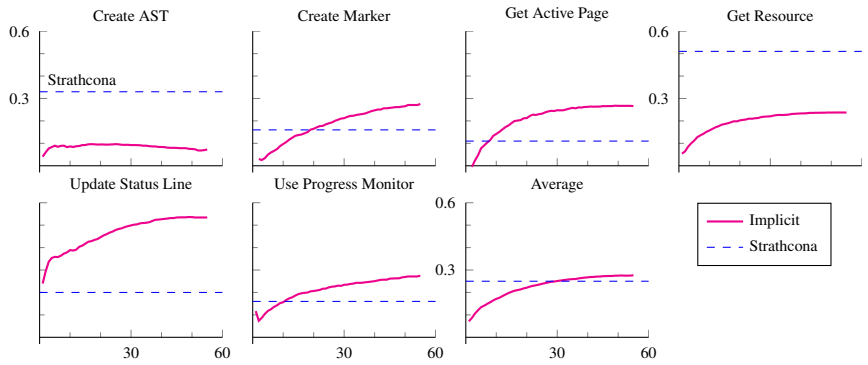


Figure 5.5: Performance-to-Feedback Ratio

It is worth highlighting that the system that creates the best rankings varies from scenario to scenario. For instance, Strathcona, Oracle, and Implicit each win two scenarios whereas Explicit does not win any scenario. Although we expected Oracle to define the upper performance bound for our systems, it failed drastically in two scenarios: *Create Marker* and *Use Progress Monitor*. We investigated the reason for these failures and found out that for the scenarios in our experiments conflicting feature weights would have been required: one that favors code examples using only few types and methods other than those given in the query, and another one that boosts examples that use more than the given types. For example, in the scenarios *Create Marker* and *Use Progress Monitor* the developer searched for code examples that showcase how to use methods she already defined in her code. A good code example will use these methods, but as few other methods as possible – additional methods would obscure the code example and make it less helpful. On the contrary, for the scenarios *Create AST*, *Get Active Page* and *Get Resource* the developer searched for code examples that contain a certain call sequence. Here code examples are favorable that call additional methods.

Such contradictions make it impossible to define a weight set that will perform perfectly well on all scenarios. Instead the learning algorithm favors weights that produced good rankings for *most* scenarios; thus, it did perform very well on four scenarios but not on the remaining two. This finding provides an interesting insight for the design of future example code search engines: What the developer needs cannot be expressed by just one feature set—or at least not with the existing feature set used. Further research is needed to learn new kind of ranking functions that leverage other features and/or variable feature weight sets to produce much better rankings.

5.6.2 Learning from User Click-through Data

After discussing the overall performance of the four subject systems we now answer the forth question of how implicit feedback affects the ranking capabilities of the system and how much

feedback is needed to optimize our code-search engine. We evaluated, therefore, how each new feedback obtained from our users improved the performance of our click-through-based system in each scenario. Figure 5.5 depicts these improvement curves and additionally plots Kendall's τ for Strathcona as a horizontal baseline for comparison. The x-axis gives the number of feedbacks we used to learn new feature weights starting from 1 up to 55 feedbacks. At each position we plotted Kendall's τ the Implicit system achieved for each scenario. Which feedback we used for training was chosen randomly. This random selection, however, lead to large peaks; thus, we repeated this process thirty times for each scenario and plotted the average values.

Figure 5.5 shows that with each feedback the system continuously improves over time for almost all scenarios. A big exception is the *Create AST* scenario where the quality of the system actually slightly decreases over time for reasons we already discussed above. Furthermore, Figure 5.5 shows that the implicit system stays below Strathcona's performance in the *Create AST* and *Get Resource* scenarios. However, our system achieved the same Average Precision as Strathcona in *Create AST* but a much higher Average Precision in *Get Resource* scenario; thus, our system shows a much better performance for the relevant examples than Strathcona. Here, the selection bias towards Strathcona for r^* shows its effect in reducing the overall performance of our system more than necessary. But despite this bias, over all scenarios our system still achieves a higher score than Strathcona. Furthermore, for the remaining four scenarios our system needed less than 30 feedbacks to exceed the performance of Strathcona.

Another observation that can be made is that some kind of saturation takes place letting the ascent of the curves getting lower and lower the more feedbacks are leveraged. Although the curves indicate that at least some improvement can still be expected, it seems that there is some upper bound to these curves asymptotically approximate to. However, from the diagram it also seems clear that this upper bound is not the one defined by the Oracle system—the system we learned from the optimal rankings r^* and expected to perform as an upper bound for all systems. However, from the current data it cannot be concluded yet where this upper bound lies.

5.6.3 Threats to Validity

There are some caveats with the current experiments, which we discuss in the following.

First, the evaluation used a relatively small set of user feedbacks (264 explicit and 506 implicit example ratings); further investigations are needed for larger, maybe conflicting, feedback sets.

Second, the systems we evaluated used a relatively small number of code features. Further experiments are needed for find out how the approach will perform for systems that leverage significantly more code features and how much user feedback will be required to train such systems. Yet, we are confident that these experiments will deliver similar results, given that others [Joa02] have successfully leveraged user feedback for "traditional" document search with more than 20.000 features and thousands of queries.

A further threat may be the scenarios used for the evaluation. However, the feedback-based systems were not trained on these scenarios, but on independent search queries issued by our anonymous test users. Thus, the improvements we presented are not specific to the scenarios used and we assume that they will generalize to other scenarios. Yet, further studies will be conducted to examine the generalizability of this approach—also to other domains than example recommendations.

5.7 Summary

In this chapter, we presented a code-search engine, which is based on the concepts of the Strathcona example recommender, but unlike Strathcona learns from user click-through data to continuously improve its ranking function. We showed that leveraging user click-through data is effective to improve the performance of today's code-search engines. In particular, we showed that even with a small number of user feedbacks the performance of existing, manually tweaked code-search engines could be reached and improved in most cases.

More importantly, the presented approach opens the door for several interesting research directions in the future.

By having ranking functions be automatically derived from user click-through data, it becomes easy to experiment with new features and/or combinations of features from different existing code-search engines.

Our experiments revealed the important insight that the needs of the developer in different API usage scenarios may be best served by conflicting feature weights. Further research is required to identify which feature sets and weights are appropriate to find those examples the developer actually needs in the context of a particular development task at hand.

Furthermore, we plan to exploit the capability of learning from user's behavior to build personalized code-search engines that are individually trained based on a user's individual click-through behaviors. Such personalized code search engines may also take into account a developer's prior knowledge by tracking the APIs she uses inside the IDE to build personalized degree-of-interest models for code search. These models could be used to automatically judge the novelty and interestingness of a recommended code snippet, and thus may work as another dimension to improve code-search engines.

Last but not least, we would like to highlight that the approach presented here, prepares the way for leveraging implicit user feedback not only for code-search engines but virtually for every system that ranks and scores any kind of code documents, as long as the systems allow users to interact with the results and are capable to track these interactions for later analysis.

6 Related Work

This chapter summarizes previous work closely related to the tools presented above: intelligent code completion systems, mined documentation, bug detection, and code search.

However, when discussing related work there is always a need to draw a boundary of what has to be discussion and what may be skipped. The approaches we present in this thesis leverage usage data collected from code repositories as well as from the observations how users interact with systems. This relates them with almost any approach that performs mining on software repositories or user interaction mining. Also the way how developers share their knowledge with others is changing – which brings social collaboration, security, and data privacy into focus of related work. To keep this section focused, we thus had to excluded several related but more distant research areas from the discussion such as specification mining, automated test generation, or any other social aspects.

In the following we briefly summarize approximately 30 research works grouped by their relation to our work. For each work we described briefly what problem it is solving, outline the approach applied to solve the problem, and summarize how the solution is evaluated. While reading one may find that some tools may be related to more than one group. In these cases, we associated it with the group we think it fits best.

6.1 Documentation Mining

CodeWeb [Mic00] and FrUiT [BSM06] use association rule mining to discover library usage rules in C++ and Java source code, respectively. For example, CodeWeb may find rules like *"If programmers call file::open then 95% of them also called file::close"*. CodeWeb presents the rules found as kind of browsable, extended manual which links the relevant code examples. FrUiT extends this approach by an interactive viewer tightly integrated into Eclipse which recommends relevant items based on the current editor's content. Both systems use inheritance taxonomies and statistical significance tests for pruning the number of association rules found. Both systems have been evaluated by a small qualitative study; no quantitative evaluation was performed nor a user study conducted.

Suade [Rob05] generates suggestions for software investigation. Developers who become stuck while exploring code to complete a change task can use Suade to trigger recommendations

about where to look next among all the related elements. The developer explicitly specifies a set of relevant fields and methods (the context elements), and Suade uses method-call and field-access relations to automatically retrieve related elements. It ranks the retrieved elements by extracting a dependency graph of all their static dependencies from the project's source code to the context elements, and then by applying heuristics to the graph's topology. For example, if a method calls only those methods that a developer specified as relevant, it's ranked higher than methods that call the context methods in addition to many others. In Suade, users create a context by dragging and dropping elements of interest into a view. Once they've specified a context, they can trigger a recommendation cycle. Suade displays recommendations as a list in a dedicated view. Users can drag recommended elements back into the context view to iteratively update recommendations. The system have been evaluated by a quantitative study on JHotdraw and Azureus to assess its general properties like the number of proposed related elements, stability and effect of changing parameters. In addition to this evaluation two manual case studies have been performed to assess the usefulness of the proposals made.

Altair [LWC09] automatically generates API function cross-references between related methods based on structural measures. Altair ranks related API functions for a given query according to pair-wise overlap, i.e., accessing the same data, call the same public functions, or use similar composite types. In contrast to other approaches Altair does not depend on specific client code to build such cross-references. In addition to presenting related functions, Altair also clusters related functions into "modules" using a spectral clustering technique, so that functions can be grouped by the general modules they belong to. The authors evaluate Altair's ranking algorithm by manually judging the its results for six scenarios (taken from the Apache Portability Runtime) and comparing them with Suade [Rob05], FRAN [SFDB07], and FRIAR [SFDB07]. The clustering approach is evaluated in terms of an ad-hoc precision and recall measurement against a manually defined gold-standard.

eXoaDocs [KLwHK10] belongs to the group of code-search engines such as StrathCona, XSnippet, and MAPO. In contrast to these tools, eXoaDocs does not target to support developers on their current working context in the IDE but to provide code examples for API methods without further context. These code examples are then added to existing API documentation. The process of finding relevant code snippets is as follows. For each API method, eXoaDocs searches the web (using code-search engines like Google Code, Krugle, Koders etc.) for example methods that use the API method in question. In a next step they create code summarizations from these candidate methods which are then clustered into groups of distinct usage examples. Finally, for each group the most representative example is selected and merged into the API documentation. Recently, eXoaDocs website started to capture explicit user feedback about the quality of a mined code example. Whether this information is used to continuously improve the system is yet unclear. The system itself has been evaluated in a user study with 24 subjects and four different tasks these subjects had to accomplish. Subjects were divided into two groups:

one using eXoaDocs, the other using JavaDocs.

SpotWeb [TX08] aims to identify the hotspots and coldspots of a framework. The term hotspot refers to classes and methods that are either extended or (re-)implemented or just used often by clients. Given a framework class, SpotWeb searches the web (using code-search engines like Google Code, Koders, or Krugle, etc.) for example classes that use the given framework class. For each framework class various usage metrics are computed based on the examples found, which serve as input to create rankings for the hotness of each framework class. Finally, all classes and methods that pass a given hotness threshold are marked as hotspots and presented to the user (for instance inside the Eclipse UI). Elements whose usage metrics are below a low threshold are marked as being coldspots. In addition, SpotWeb classifies hotspot classes into `hooks`, if the class is an interface or abstract, and `templates` otherwise. The effectiveness of the approach is evaluated by comparing the number of detected hotspots with (i) a manually create gold standard created from existing documentation and (ii) by comparing the results with the approach of Viljamaa [Vil03b] using precision and recall.

MAPO [ZXZ+09] is a tool that extracts API usage information from code snippets and mines patterns from them to leverage code searching. MAPO takes as input a set of open-source repositories and a coding context (e.g. a few API calls in a method body), and outputs code snippets related to the code context. MAPO is composed of three phases: API usage fact extraction, pattern mining, and a recommendation engine. The API usage fact extraction phase models how a client method uses an API by creating an ordered sequence of API method calls invoked by the client method. When control flow is present, and multiple possible sequences may be legal, MAPO finds all possible sequences and take a subset of sequences that cover all API calls in the client method. During the pattern mining phase, client methods are clustered according to the API methods they invoke and by the natural language terms found in their method name and enclosing class name. Each cluster is fed into a sequential pattern miner that looks for frequent subsequences of API method calls. The patterns are used to create a representative index of the cluster. Finally, MAPO uses the index to match user-entered code snippets to relevant clusters. The effectiveness of MAPO was shown by an experimental study of 13 scenarios taken from the Eclipse Graphical Editing Framework. For these scenarios the results returned by MAPO were compared it with those of Strathcona [HM05a] and Google code-search [gooa] and and its quality judged by the authors.

PopCon [HW08] provides a bridge between high-level design documentation and low-level API documentation by statically analyzing a framework and several of its clients and providing a ranked list of the relative popularity of its APIs. For example, if a programmer selects a package like `org.eclipse.jdt` she gets a list of the most-often used classes, methods, and fields of this package. PopCon has been designed to help developers to gain insight into which parts of

the framework are most often used and thus are likely to be good starting points to familiarize oneself with the API. From a technical perspective straight-forward to implement but in terms of usefulness potentially well suited for novices to get in touch with an API. PopCon has been evaluated by a small qualitative study; no quantitative evaluation was performed nor a user study conducted.

SemDiff [DR08]. Upgrading to the latest version of a library may break ones code; for instance, because a method that existed in the previous version of the library does not exist anymore. While the Java compiler helps to quickly spot this kind of problems, it cannot help programmers to solve it. SemDiff aims to support developers on this task by recommending replacement methods for adapting code to a newer library version. SemDiff mines the revisions of the framework's version control system to study the evolution of method calls. It is based on the assumption that calls to deleted methods will be replaced in the same change set by one or more calls to methods that provide a similar functionality. For each removed method it computes a list of potential replacements based on several statistics and finally presents these lists to the user inside Eclipse. The paper doesn't provide any details on how the tool was evaluated.

Jadeite [SMY09] is a JavaDoc-like documentation system that takes advantage of multiple users' aggregated experience to reduce difficulties that programmers have learning new APIs. It extends JavaDoc in three ways. First, it allows developers to specify placeholders for methods that actually do not exist in an API but users have expected them to be there. For instance, users of `javax.mail.MimeMessage` might expect this type to have a method called `send()`. Actually, there is no such method because messages are send using the class `javax.mail.Transport`. With Jadeite developers now can add a placeholder for `send` in `MimeMessage` which may refer a reader to `Transport.send()`. Second, Jadeite uses different font sizes in outline to indicate popular APIs, and third enriches existing method documentation with code snippets showing common uses of these APIs. These snippets and popularity indicators are taken from Google search results and number of hits for a given search term respectively. An evaluation of the approach is pending.

6.2 Code-search

CodeBroker [YF02] supports developers to locate reusable components in a reuse repository. Therefore, whenever a developer places the cursor inside a JavaDoc comment, the system automatically searches the repository for similar components, i.e., components with similar documentation. To determine the similarity between two JavaDoc comments it is using Latent Semantic Analysis. After the developer completed the method signature (or places the cursor after the opening curly brace) CodeBroker refines the query with the method signature information now available to further reduce the number of recommend components to only those that match

the requested signature. In addition it manages user-specific lists of "known components", i.e., classes a developer has already used or traversed before, which is then used to further prune the list of proposed components to those the developer is not familiar with. The authors conducted two evaluations. First, the authors defined 19 queries and evaluated the performance of CodeBroker against a manually defined gold-standard using averaged precision/recall curves. Second, the authors conducted a user study with five subjects and twelve experiments. The subjects interactions with the system were recorded and analyzed and the subjects interviewed after the experiments to gain insights into the usefulness of CodeBroker.

StrathCona [HM05a] is a code-search engine that retrieves relevant source code examples to help developers use frameworks effectively. For example, a developer who's trying to figure out how to change the status bar in the Eclipse IDE can highlight the partially complete code (the context) and ask Strathcona for similar examples. Strathcona extracts a set of structural facts from the code fragment, such as what types are referenced and what methods are called etc. and queries an example repository to search for occurrences of each fact in a code repository. Next, it uses a set of heuristics to decide on the best examples, which it orders according to how many heuristics select them, and returns the top 10 examples. Strahtcona was evaluated a user study with two subjects. Each subject had to solve four tasks. The subjects confirmed that Strathcona was able to recommend one useful example for the three (respectively two) of the four tasks.

ParseWeb [TX07] Sometimes programmers might want to call methods on an object of a particular type but don't know how to obtain objects of that type from objects available in their current programming context (for example, method parameters). ParseWeb recommends sequences of method calls starting from an available object type and producing a desired object type. ParseWeb analyzes example code found on the Web to identify frequently occurring call patterns that link available object types with desired object types. Developers use the tool by specifying available and desired object types and requesting recommendations. The authors conducted three experiments. First, they walked through two real-life problems and demonstrate how ParseWeb could have helped to solve these problems. Second, they compare ParseWeb with Strathcona [HM05a], Prospector [MXBK05] and Google code search [gooa] by exercising all tools on a set of ten manually defined queries. Third, ParseWeb was compared with the best performing tool, Prospector, on further 12 example queries. For judging the quality of each system, binary decisions whether or not the search call sequence was contained in the recommended example was sufficient.

SNIFF [CJS09] is a search engine for Java using free-form queries. The key idea of Sniff is to combine API documentation with publicly available Java code. Specifically, Sniff takes Java source code already available on the web and annotates it by appending each statement containing a method call with the method's JavaDoc description (if available). Furthermore,

after retrieving code fragments, SNIFF then performs an intersection of types in these code chunks to retain the most relevant and common part of the code chunks and ranks these pruned chunks using the frequency of their occurrence in the indexed code base. SNIFF was evaluated with three different experiments. First, SNIFF was compared against Prospector [MXBK05] and Google code search [gooa] in a controlled user experiment. For evaluation eight participants had to solve four programming problems; two with the aid of SNIFF, one with Google and one with Prospector. As a result, subjects with SNIFF were 40% faster than users of Prospector and Gooogle code search in all four tasks. In a second experiment, a set of manually defined natural queries where issued to SNIFF, Google code search [gooa], Koders [kod], and Krugle [kru]. For evaluation the quality of the first result was judged for relevance and counted. In this experiment, SNIFF was able to return a relevant hit on the first position in 87.5% of the queries followed by Koders, Google and Krugle with 62.5%, 25%, and 12.5% respectively.

Mica [SM06] is a web-search engine specially tweaked to support programmers better than plain Google web-search. Mica uses Google's web-search API to generate its search results but enriches the results provided by Google with some additional meta-data like a keywords sidebar containing frequently observed code terms like method or class names, and other visual improvements like indicating the origins of a web page by classifying it as official documentation or as containing source code, keyword highlighting etc. Interesting about Mica is in particular its keyword sidebar. *Keywords* according to Mica are all class and method names that occurred in the Java SDK library. If a web page contains one such keyword it is added to the keywords sidebar and is henceforth available for highlighting, search refinement operations etc. An evaluation of Mica's interface enrichments is missing.

Sourcerer [BOL10] is a code search platform that comes in many different flavors. For instance, CodeGenie [LLBO07] performs Test-driven-code-search, i.e., it searches code examples based on the identifiers used inside test cases. Sourcerer API Search [BOL10] searches example code snippets by enabling users to refine their queries for instance via tag clouds, and uses enriched source code processing by identifying similar other methods and many things more. It integrates some of the concepts of introduced by Mica like tag clouds to enable query refinement. Looking at the technologies leveraged inside Sourcerer is seems to be the most advanced code-search engine available. To relate this work with our work described in chapter 5: their scoring mechanism is hard coded using fixed weights for scoring code examples. Sourcerer could clearly benefit from the approach we presented in this thesis. Various aspects of the Sourcerer infrastructure are evaluated in different ways. One approach uses 25 control queries and measures the performance of various configurations using standard IR method such as precision and recall, ROC curves, and AUC. For details we refer to [LBN+09b];

6.3 Code Completion

Mylyn [KM06] introduced the concept of a task-focused interface. Software developers have to work on many different tasks (such as bugs, problem reports or new features) but each of them requires the developer to identify and remember which resources in a project are related to which task. Mylyn supports developer in that it makes the notion of a task explicit in the UI. Developers can now inform the IDE on which task they are going to work now and Mylyn tracks which elements a user is working with and starts filtering elements irrelevant to the active task from the UI. It internally maintains a*degree of interest model* that stores relevant elements but also drops elements which haven't been visited for a longer period. When a user switches to another task only those elements relevant to this activated task are shown. In addition, Mylyn's task focus also contributes to the code completion of the IDE by proposing the most relevant APIs higher on top. It learns what to put at the top based on the developer's personal usage history for the currently active context. Mylyn was evaluated by a four month field study with 16 subjects. To measure the effects of task-focused IDEs, the subjects' interaction history was tracked before Mylyn was used and after Mylyn was installed. Based on these two periods the *edit ratio*, i.e., the ratio between editing parts of the code and navigating to other locations in a project was computed and compared. The evaluation showed that task-focused UIs could significantly improve this ratio. In the meanwhile, Mylyn is one of the most frequently installed Eclipse extensions with hundreds of thousands downloads per month.

Prospector [MXBK05] supports developers on solving questions like how to get from one type in the API to another desired type. It is tightly integrated into the Eclipse code completion system to enable programmers to specify such queries directly from the editor. For example, given an incomplete assignment like `IWorkbenchHelpSystem hs = |`, a programmer may trigger Prospector immediately after the equals sign to find all potential paths through a so-called jungloid graph that may lead to an instance of the requested type. A jungloid graph is build from a code corpus. Nodes in a jungloid graph are types; edges in the graph represent ways of getting from one type to another, e.g., via field accesses, method returns, or downcasts. Edges like field accesses, or method returns are created from the static type system information available in the corpus whereas downcast information are taken from example code that actually applied a downcast on a given type. The resulting graph is used as input to the code completion system. Prospector has been evaluated by a set of 20 sample queries and a user study with eight subjects and four tasks.

xSnippet [SC06] is context-sensitive code assistant framework (similar to Prospector) that allows developers to query a sample repository for code snippets to the object instantiation task at hand. The relevance of a snippet is defined by the context of the code, both in terms of the parents of the class under development as well as lexically visible types. Queries, invoked from the Java editor, can range from the generalized object instantiation query that returns all possible

code snippets for the instantiation of a type, to the more specialized object instantiation queries that return either parent-relevant or type-relevant results. Queries are passed on to a graph-based snippet mining module that mines for paths that meet the requirement of the specified query. Paths here can be either within the method scope or outside of the method boundaries, ensuring that relevant code snippets that are spread across methods are discovered. All selected snippets are passed on to the Ranking module that supports four types of ranking heuristics: context-sensitive ranking, frequency-based ranking, length-based ranking, and a ranking heuristic that combines the three. Various aspects such as the impact of different ranking strategies have been evaluated by 17 object-instantiation specific programming tasks. All tasks were based on the Eclipse plug-in examples from *The Java(TM) Developer's Guide to Eclipse (The 2nd Edition)*. Furthermore, xSnippet was compared with Prospector on the same tasks and outperformed Prospector given these scenarios and "must-compile" evaluation strategy.

OCompletion [RL10] by Robbes and Lanza leverages the program's history to improve the quality of code completion. It maintains class-based working sets of methods that have been recently modified along with all the methods which are called in it. All the entries in these working sets are sorted by date, favoring the most recent entries. Whenever a user triggers code completion in a class, the corresponding class working set is loaded and the list of most recently used methods in this context is recommended to the user. For typed languages - or in situations where the type could be inferred from the IDE - it also leverages the type of the method receiver to further filter the list of proposals to those that can actually be applied. OCompletion was evaluated with a series of completion algorithms and in various completion scenarios. To obtain completion scenarios, a monitor logged which prefixes a user entered and which proposal was actually selected from the code completion pop-up. Based on this data the authors evaluated for different prefixes (varying from two to eight characters) which on which position each ranking strategy proposed the correct proposal (position 1,2,3,4-10, or *fail*). Furthermore, the tool was evaluated by a user survey with 29 participants.

Hill and Rideout [HR04] use code duplication mining techniques to extract likely completions of a method body based on what a developer has typed before. Their approach defines each method as a 154-dimensional vector: the number of lines of code, the number of arguments, a hash of the return type, the cyclomatic complexity, and the frequency count of each of the 150 Java Language token types. The vector v_q of a method being queried is compared to a set of pre-computed vectors v_t of methods in the example repository, and the best method match is returned to the developer, such that the difference between v_q and v_t is small. This enables the developer to see similar methods. The developer can than choose to complete the current method with this example snippet. For evaluation, the authors wrote 30 partial methods. The authors chose methods they consider typical for atomic clones such as listeners and interface implementations. For each method, they came up with three possible completions: a correct one, good one, and a mediocre one. The fourth completion was generated by the tool. The authors then

ranked the four methods relatively against each other and determined the overall best performing one based on these rankings.

Abbreviation completion [HWM11] is a multi keyword completion engine that allows developers to write statements using non-predefined abbreviated input. Let's assume that a developer enters `ch.opn(n|)` into the editor and triggers code completion. Abbreviation completion now automatically expands this input string into all possible completions that match the entered characters, for instance, **ch**ooser.show**Op**e**n**Dialog(**n**ull). It is based on Hidden Markov Models that and utilizes frequent keyword patterns learned from a corpus of existing code. The system has proven to speed-up typing up to 40%.

6.4 Bug Detection

PR-Miner [LZ05a] is a tool that uses association rule mining to extract implicit programming rules about a program (e.g., locking a shared resource before accessing it). Once identified, these rules can be used to find violations, which could possibly indicate bug locations. For example, an implicit programming rule could be to always follow a call to `lock()` with an eventual call to `unlock()` and a violation of this rule would be the omission of `unlock()`. PR-Miner represents implicit programming rules as association rules between program elements that frequently appear in function definitions. To find these rules, PR-Miner represents program elements items and functions as itemsets, where each itemset is considered as a transaction in an itemset database. Then PR-Miner mines the itemset database for frequent closed itemsets and generates association rules. Association rules are considered as implicit programming rules if they have a confidence value above 90%. An association rule with a confidence value lower than 100% indicates that it has been violated. For example (taken from the paper), if a rule $a, b \Rightarrow d$ has a support of 100 and a, b has a support of 101, there is only one out of 101 cases that has a, b but not d, and thus is likely to be a bug. PR-Miner was applied on Linux, PostgreSQL, and Apache HTTP Server. For each system the top 60 violations have been verified manually to differentiate bugs from false positives. From this set 16 bugs in Linux, 6 bugs in PostgreSQL, and 1 in the Apache HTTP Server could be identified and have been confirmed by the community.

DynaMine [LZ05b] analyzes source code check-ins to find highly correlated method calls as well as common bug fixes in order to automatically discover application-specific coding patterns. Therefore, it mines the revision history for pairs of method calls newly added in a revision. From these pairs DynaMine generates association rules of the kind "95% of all programmers that added a call `it.next()` also added a call to `it.hasNext()`". In addition to discovering coding patterns, DynaMine also enables programmers to check whether the own code satisfies these patterns. Therefore, a programmer selects several of these rules, let DynaMine instrument

the programmer's byte-code, and then executes the code. In the background, DynaMine collects the execution traces related to the selected rules and verifies after execution that all rules have been satisfied - or not. The intuition behind using dynamic traces instead of static analysis for verification is that it is sometimes hard to detect statically whether a rule matches or not whereas with a dynamic trace whether or not a rule matches is actually a binary decision. However, relying on dynamic traces is also a weakness of this approach as it verifies only those paths in code actually executed in the verification runs. The effectiveness of Dynamine has been shown by a qualitative discussion of patterns found in Eclipse and JEdit.

Error Specs [AX09] statically mines API error-handling specifications for C code from API client code. Procedural languages ,like C, that lacks explicit error handling mechanisms, frequently use return values to indicate success or failure of a procedure call. Consequently, checking these return values and handling error conditions are essential for writing proper working code. Error specs leverages this information to mine so called error-handling specifications. It identifies error paths (i.e, paths in the program execution that either return a negative or return values from preceding procedure call or even exit the main program) to identify sections in code that potentially deal with error-handling (called API cleanup methods). It uses static analysis tools to create error-traces and normal traces and then applies sequence mining on these traces to find probable error-handling specifications. The results produced by error specs have be examined manually and discussed by the authors.

Jadet [WZL07] leverages code examples to automatically infer legal sequences of method calls. The resulting patterns can then be used to detect anomalies such as "Before calling next(), one normally calls hasNext()". It follows a three-step approach. First, it mines finite state automata with anonymous states and transitions labeled with feasible method calls on per-object-level. Second, it extracts temporal properties of the kind `start()` $\prec$ `stop()` (expressing the temporal property "start *can* be called before stop") from these state automata, and then, third, applies closed frequent itemset mining on these temporal properties. These closed frequent itemsets are then used as input into a classifier which identifies locations in code that violates these patterns. Hereby, a violation is given if an object usage is rare, i.e., thinking of closed itemsets as nodes of a concept lattice, when neighboring pattern nodes have almost the same support value. A violation, however, does not necessarily imply a bug. Jadet applies several filtering criteria before a violation is classified as an anomaly. First, the base pattern itself needs to exceed a minimum support of 20. Second, the confidence that this violation is actually a anomaly must exceed 90% and must not miss more than two temporal properties. With these settings Jadet was able to find two previously unknown bugs in AspectJ and several code smells. Recently, this approach has been extended to support mining of temporal specifications not just for Java but also for C, C++, PHP and similar language using a mostly language independent lightweight parser and generalized mining approach. The resulting tool is accessible under `checkmycode.org` [GWZ10].

CP-Miner [LLMZ06] uses frequent subsequence mining to efficiently identify copy-pasted code in large software suites and detects copy-paste bugs. It mines these sequences inside basic blocks (i.e., a straight line piece of code without any jumps or jump targets in the middle) and merges these sequences afterwards to build larger sequences. The authors enhance the basic mining algorithm CloSpan to allow arbitrary interleaving gaps in the sequences to deal with newly introduced statements in code. For sequence mining, each identifier are discarded at mining time and later at matching time reconstructed. For copy and paste bug-detection CP Miners searches for sequences that are *mostly similar* to other frequent sequences but differ in single method calls or identifier mismatches in single statements which are likely to be caused by incomplete renaming refactorings. With the aid CP Miner the authors found more than 80 bugs in Linux, FreeBSD, Apache, and PostgreSQL, which were fixed afterwards.

Alattin [TX09] analyzes, similar to PR-Miner, code examples to find frequent API usage patterns for bug detection. Alattin extends PR-Miner in three ways: (i) it additionally leverages control-flow statements (such as `if`, `for`, or `while` statements) and checks (such as `check-null`, `check-constant`) to create more fine-grained usage patterns, (ii) allows *OR*ing of several frequent patterns to express alternative rules such as $P_1 \vee P_2 \vee \ldots \vee P_n$, and (iii) mines so called neglected (i.e., infrequent) patterns that can be combined with frequent patterns as in $P_1 \vee \ldots \vee P_n \vee N_1 \vee \ldots \vee N_m$. Neglected patterns are alternative patterns that (despite infrequent in an entire set of code examples) are frequent among the code examples that do not support a frequent pattern P_i. The authors show that the application of neglected patterns reduces the false-positive rate for bug detection by 28%. The tool was evaluated by a manual study of the bug detection patterns found for 6 example APIs (Java Util, Java SQL, Java Transaction, Hibernate, Apache BCEL, and HSQL DB) and a quantitative evaluation of several factors, such as different mining thresholds and the effect of neglected patterns on the quality of the bug detection system.

6.5 Others

eRose [ZWDZ04] applies data mining to version histories in order to guide programmers along related changes. For example, the tool may give programmers recommendations like "Programmers who changed these functions also changed...". Given a set of existing changes, such rules suggest and predict likely further changes, which may even be in-detectable by program analysis, and thus prevents errors due to incomplete changes. eRose is tightly integrated into the Eclipse IDE. It tracks changed elements (the context) and updates recommendations in a view after every save operation. For evaluation of eRose the an automated approach has been applied. Given a set of recent commits (say, the last 500), the set of modified entities were extracted and divided into a query and an expectation set. The elements in the query were used to exercise eRose while the expectation was used to evaluate its recommendation quality. For

evaluation precision and recall was used. Different aspects such as its ability to recommend all relevant entities or its potential to prevent errors has been evaluated.

GrouMiner [NNP+09] mines usage patterns of objects and classes using graph-based algorithms. The usage of a set of objects is represented as a labeled DAG, called a "Groum" (graph-based, object usage model), which nodes represent object's constructor calls, method calls, field accesses, and branching points of control structures, and edges represent temporal usage orders and data dependencies among them. In other words, Groums describe the usage orders of objects' actions and also capture control structure and data dependencies among them. A Groum is created for each method in a client system. GrouMiner then finds object usage patterns by looking for frequently appearing subgraphs occurring in the extracted set of Groums. Like PR-Miner, GrouMiner can then use the extracted patterns to find violations as a means for locating possible bugs, or as input for framework migration guidance.

MUDABlue [KGMI04] aims to solve the software categorization problem prevalent in large-scale project hosting services like SourceForge, Google Code etc. Therefore, MUDABlue takes a list of applications, extracts the identifiers used in source code (i.e., variable names, method names etc.) and puts them into (identifier x application) matrix, where the rows indicate identifiers, columns indicate applications, and cells (i,j) express how often an identifier i occurred inside the an application j. It then uses Latent Semantic Indexing to find groups of similar identifiers, clusters the applications in the corpus according to these groups, and assigns each cluster the top-10 most frequently used identifiers. For evaluation the authors picked 41 C programs in five categories from SourceForge and run MudaBlue on these programs. The resulting classification for each program was checked and its overlap with the origin classification was measured using a adapted version of precision, recall, and f-measure.

Exemplar [GFX+10] finds relevant software projects from large archives of applications (e.g., SourceForge) given some natural-language query that contains high-level concepts (e.g., MIME, data sets). In contrast to existing search engines Exemplar does not only search the code of the example code to match the query but also the API documentation of third-party APIs used by the applications. The underlying assumption is that the key phrases a developer is looking for are more likely contained in these help documents than in the application code. Furthermore, it facilitates data-flow analysis between these concepts, i.e., if a developer searches for terms like compress and encrypt examples are higher ranked that use APIs related to these terms and there is a data-flow between these API calls. Grechanik et al. evaluated Exemplar in a case study with 39 participants. The participants were organized into three groups; every participant had to use a different search engine to solve several tasks: SourceForge built-in search engine as state-of-the-art, Exemplar without dataflow links, and Exemplar with dataflow links. Each participant had to judge the relevance of each returned recommendation on a four-level Likert scale. The

overall performance was evaluated by counting the number of relevant recommendations.

Hipikat [vM03] supports newcomers to an project to quickly come-to-speed by recommending relevant artifacts like documents, issues, or source code revisions for the task at hand. From project information available in mailing lists, forums, websites, revision control and bug trackers, and the links between these artifacts, Hipikat builds what the authors call a group's memory. This group memory can then be queried by a newcomer either by selecting an existing resource in the workspace and triggering a search query for related elements or by using plain keyword searches. Hipikat is tightly integrated into Eclipse. Čubranić and Murphy conducted a qualitative user study with 7 participants and walked the reader through a use case of extending Eclipse CVS plug-in. The subjects had to accomplish an extension to some mid-size software developed at the research group and were interviewed after they finished the extension.

Expertise Browser [MH02]. Finding the right software experts to consult can be difficult, especially when they're geographically distributed. Expertise Browser is a tool that recommends people by detecting past changes to a given code location or document. It assumes that developers who changed a method have expertise in it. Mockus and Herbsleb performed a study of two projects with more than 100 developers each. They tracked the how developers of these projects used Expertise Browser. Based on these usage profiles they identified different user groups and information needs. In addition to the quantitative usage data, the authors collected qualitative user feedback to support their findings.

Fraser and Zeller [FZ11] present an approach that leverages common object usages for systematically generating more readable and meaningful test cases. In a first step, they use Jadet [WZ09] to extract common object usage patterns from test code. Next, they adapt the test case generation to follow the patterns of common object usage, as mined from code examples. A quantitative study of the test generation results for the Joda-Time framework gives evidence for the benefits of the approach.

Dhruv[ASH+06] recommends people and artifacts relevant to a bug report. It operates chiefly in the open source community, which interacts heavily via the Web. Using a three-layer model of community (developers, users, and contributors), content (code, bug reports, and forum messages), and interactions between these, Dhruv constructs a Semantic Web that describes the objects and their relationships. It recommends objects according to the similarity between a bug report and the terms contained in the object and its metadata. To best of our knowledge, there are no reports on the effects of using Dhruv.

Kagdi et al. [KCM07] apply closed sequence mining on C code to identify common call usages. In addition to mining call sequences, they also encode control-flow information such as if-statements into their sequence mining approach. The authors discuss the benefits of their approach by four sequence patterns mined from the Apache HTTPD code base. The authors point out the benefits of their approach for bug detection and documentation. However, to the best of our knowledge, no tool has been published yet that leverages their approach.

Marie [SJM08] mines framework migration rules from client code. It takes as input two API versions (v_1 and v_2) and two versions of clients (c_1 and c_2) where c_1 uses v_1 and was then updated to use v_2 in c_2. Given this, the approach mines API-migration rules from the two client versions. Rules are of the form ($u_1 \rightarrow u_2$), where u_1 is an API usage w.r.t. v_1 and u_2 is an API usage w.r.t. to v_2. API usage update patterns are one to one and must follow certain predefined patterns (e.g., a call can be replaced by an object instantiation or another call, but not by a field access). Rules are harvested from individual client methods whose signatures did not change between c_1 and c_2 or from type and methods signatures (for extensions/implementations and overridings) whose signatures also remained fixed. Rules are generated using association rule mining. Sch"afer et al. evaluted their system by comparing the evolution rules found by Marie with those found by RefactoringCrawler [DDN00]. To assess the overall quality of these systems, both sets were compare with a reference rule set manually extracted from the documentation of Eclipse UI, JHotdraw, and Struts framework.

7 Conclusions and Future Work

7.1 Conclusions

The goal of this thesis was to identify whether the concepts of Web 2.0 can be successfully adapted to improve today's IDE services. We showed how code completion engines, API documentation, bug detection, and even code search engines can benefit from these concepts. Each tool (except code search) has been published on leading conferences such as Foundations of Software Engineering, the European Conference on Object Oriented Programming, Mining Software Repositories, and ACM Recommender Systems.

Although important for pursuing a doctorate, publishing on research conferences is only one criterion to assess the work. Being a tool for developers, evaluating the concepts of IDE 2.0 in-the-large was another important goal of this project.

From its beginning, the project had a strong focus to produce tools used by developers in their day-to-day work, and thus, aimed to leave the space of a pure research prototype. During this thesis, more than 60 students were engaged in this project; most of them during various hands-on trainings but 15 students also wrote their bachelor and master theses in this topic. With the aid of these students, we evaluated many different paths (some of which turned out as dead ends) and created a full-functional prototype of considerable quality. During this phase, the project received continued attention of several Eclipse committers and users which finally led to promoting Code Recommenders as Eclipse project hosted at eclipse.org.

However, the exciting part of all the work begins now: after the thesis is written. Evaluating the concepts of IDE 2.0 in-the-large was not possible during the years of this thesis. However, research goes on. There are quite a lot more tools in the pipe...

7.2 Future Work

Intelligent Code Completion In chapter 2 we presented an intelligent code completion that leverages the knowledge how other programmers used a given API before. It uses this knowledge to recommend relevant method calls and to filter the irrelevant ones. This system has proven to work well with Eclipse APIs such as SWT, JFace or Eclipse UI. However, it is unclear whether such systems perform equally well for other APIs too. Large-scale evaluations are needed to verify the advantage such systems may bring to software development in the long-term.

Despite its precision, it is unclear whether such systems actually improve a developer's productivity. Does is make development faster? If so: because less typing is needed or because relevant items are found faster? Does it help to reduce the number of bugs? Some of these questions have been tried to be answered by the small user study conducted for the paper. However, this study rather states a potential trend than allows to draw robust conclusions from it. Long-term user studies are required to verify whether such completion systems are actually beneficial for developers—something we currently begin to evaluate with the Eclipse Code Recommenders project.

Another issue with the current approach is that it learns its models from mining code repositories. This approach has been used by many researchers before. Yet, we argue that mining code repositories cannot cope well with the demand to support virtually every framework since only small fraction of application code is available in public repositories. Instead, we think that such data collection routines should move into the developer's IDE and thus enable a developer to share the knowledge about how to use an API directly from his own code. The advantages of such an approach are obvious: No need to set up classpaths for analyzing a project's contents, no need to continuously crawl code repositories for changes etc. Everything is inside the IDE and can be analyzed and shared. Such an approach, however, imposes it's own challenges like how to deal with evolving example code, how to deal with different versions of an API, how to deal with bad example code etc. Problems we have to deal with if intelligent code completion should become mainstream. In it's recent version, Code Recommenders already enables developers to share their knowledge by uploading their usage data to a server. Nightly build jobs then create new recommender models from that data and offer them for download and reintegration into the developer's IDE. This way, a continuously evolving, self-improving system can be built around the concepts of Code Recommenders.

Another dimension of future work are improved recommendation models. The approach presented in this thesis leverages the knowledge which methods have been invoked on a given object and where the object is used. Here, various improvements are conceivable. For instance, a recommender may take into account how an object reference was obtained (e.g., as method return, as parameter etc.), or in which parameter call sites it participates, or even how it was used in other parts of the class to create something like a complete picture of how to use it.

Finally, futher systems may include other kinds of completion engines, for instance, for parameter guessing or even complex multi-object code templates. Code completion is a developer's primary input device for low-level instant recommendations and as such it's potential future uses are manifold—also offside the known.

Extended JavaDocs In chapter 3 we presented an approach to mine documentation from example code. A platform for visualizing such mined documentation and conducting further studies is currently developed as part of a Google Summer of Code project and close to be finished.

Although considered useful, the quality of mined documentation cannot be judged automatically. Comprehensive user studies are missing for our work presented here but also for related work. Yet, evaluating such approaches could be straight-forward by leveraging user feedback. For instance, users may judge the quality of mined documentation via explicit ratings - or even by implicit feedback by viewing it or just "make it disappear" if it's not what she was looking for. Based on such developer behaviors, evaluation models could be created that allow much more rigor statements. Also, competitive approaches can be evaluated in parallel by showing results of several approaches at the same time letting the user decide which one is superior to the other.

However, mining documentation still offers a lot room for improvements. For instance, one could mine common usages from code including multi-object protocols and combine these snippets with state-machines to synthesize code examples from it. Another area of future work may be the combination of Amazon's "People who viewed at this (java) element also looked at ..." feature with Mylyn's task focus idea for recommending documentation. A system that tracks a developer's click-through behavior through documentation may recognize that users that visit `Wizard.addPage` very frequently visit the documentation of `IWizardPage.createContents`. Thus, the system may conclude that these elements are somehow related even if there is not direct link in code between both, and thus, may recommend a developer to have a look at these methods, or more Mylyn like, would mark those methods as relevant and display them more prominent inside the IDE.

Detecting Missing Method Calls The approach to finding bugs in code by comparing object usages with common usages patterns observed in example code, presented in chapter 4, has proven that it is capable to find bugs even in mature code bases like that of Eclipse 3.5. However, as with code completion, long-term evaluations and user studies are required to prove the effectiveness of this and related approaches. In addition, it may be of interest for API developers to know which usage patterns actually exist for their APIs. Generating rule-sets from mined models may be an interesting area of future work. According to the IDE 2.0 thought it also may be interesting to see how developers actually deal with potential bugs reported by these tools. For illustration consider a warning that has been ignored or even discarded by several developers in different projects. Based on such an information it may be likely that the rule that caused this warning may not be good enough and needs an overhaul. The information for such a feedback may be provided directly from within the IDE or taken from build reports of a continuous integration server.

Another area of future work is to apply the concept of *almost-similarity* not only to method calls but to other parts of software. For instance, searching for *almost-similar* traces could yield major improvements in the area of runtime defect detections. Also, searching for *almost-similar* conditional statements is worth further investigation to improve the resilience of software w.r.t. incorrect inputs.

Code-Search Engines Code-search engines sprang up like mushrooms in the past five years but it seems that there is no provider that dominates code-search like Google does for web-search. Top search engines like Koders for instance claim to have 30,000 users a day - a rather small number compared to 400 million requests per day Google web-search gets. Even Google code-search has only 80.000 request per day; that are roughly only 20 requests per minute. What's the reason for such small numbers?

And when using state-of-the-art code-search interfaces, it immediately becomes apparent that there is a lot room for improvements. Used to the way how web-search engines work, developer queries are rather short in nature and it is hard for a search engine provider to figure out in a user is actually interested in. In this thesis, we presented an approach that is capable to learn what is relevant based on implicit user feedback. This approach, however, could be generalized to personalized code-search engines that take into account which APIs a user is already familiar with and thus would be able to recommend only those examples which probably contain new knowledge to her.

Our evaluation also revealed that it different APIs may need different feature sets or feature weights. Advanced scoring systems are needed that take into account characteristics of a framework to and adopt the scoring according to these charcteristics.

Another challenging research are code summarizations. Existing approaches often fail to highlighted the actually interesting information but select rather useless lines like import statements for their summaries. Clearly, better code summarization techniques are needed. But how could we evaluate which one is better than another one? The Code Recommenders project may also help to gain new insights into this by research topic due to its growing community.

Summary In the past (almost) five years, we developed many different tools; each of them aimed to improve state-of-the-art of how programmers develop software. However, research in this area is far from being done. Software repositories offer a wide variety of data. But interestingly, it won't be the repositories that offer us the data we need to improve IDEs; it will be the user and the way how she interacts with it: by pressing buttons, navigating between code elements, searching code examples, or rating the quality of a (maybe mined) code example... Explicit and implicit user feedback will be an interesting area of future research—and Code Recommenders is going to be the connective link between research and the IDE.

Bibliography

[AIS93] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami. Mining association rules between sets of items in large databases. In *Proceedings of the SIGMOD International Conference on Management of Data*, pages 207–216. ACM Press, 1993.

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2 edition, August 2006.

[ASH+06] Anupriya Ankolekar, Katia Sycara, James Herbsleb, Robert Kraut, and Chris Welty. Supporting online problem-solving communities with the semantic web. In *Proceedings of the 15th international conference on World Wide Web*, WWW '06, pages 575–584, New York, NY, USA, 2006. ACM.

[AX09] Mithun Acharya and Tao Xie. Mining api error-handling specifications from source code. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, FASE '09, pages 370–384, Berlin, Heidelberg, 2009. Springer-Verlag.

[BFN+06] G. Baxter, M. Frean, J. Noble, M. Rickerby, H. Smith, M. Visser, H. Melton, and E. Tempero. Understanding the shape of java software. In *Proceedings of OOPSLA*. ACM, 06.

[Bis06] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[Blo08] Joshua Bloch. *Effective Java (second edition)*. Addisson-Wesley, 2008.

[BMM09] Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *ESEC/FSE '09: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 213–222, New York, NY, USA, 2009. ACM.

[BMM10] Marcel Bruch, Mira Mezini, and Martin Monperrus. Improving the quality of framework subclassing directives. In *7th Working Conference on Mining Software Repositories*, 2010.

[BOL10] Sushil Bajracharya, Joel Ossher, and Cristina Lopes. Searching api usage examples in code repositories with sourcerer api search. In *SUITE '10*. ACM, 2010.

[BSM06] Marcel Bruch, Thorsten Schäfer, and Mira Mezini. FrUiT: IDE support for framework understanding. In *Proceedings of the OOPSLA Workshop on Eclipse Technology Exchange*, pages 55–59. ACM Press, 2006.

[BSM08] Marcel Bruch, Thorsten Schäfer, and Mira Mezini. On evaluating recommender systems for api usages. In *Proceedings of the International Workshop on Recommendation Systems in Software Engieering*, pages 16–20, New York, NY, USA, 2008. ACM.

[BYRN99] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.

[CDMS08] Prabhakar Raghavan Christopher D. Manning and Hinrich Schtze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[CH67] T. Cover and P. Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 1967.

[CJS09] Shaunak Chatterjee, Sudeep Juvekar, and Koushik Sen. Sniff: A search engine for java using free-form queries. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, FASE '09, pages 385–400, Berlin, Heidelberg, 2009. Springer-Verlag.

[CS01] Koby Crammer and Yoram Singer. Pranking with ranking. In *NIPS*. MIT Press, 2001.

[CSS99] William W. Cohen, Robert E. Schapire, and Yoram Singer. Learning to order things. *J. Artif. Int. Res.*, 10(1):243–270, 1999.

[DDN00] Serge Demeyer, Stephane Ducasse, and Oscar Nierstrasz. Finding refactorings via change metrics. In *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 166–177. ACM Press, 2000.

[DO07] Barthelemy Dagenais and Harold Ossher. Mismar: A new approach to developer documentation. In *Proceedings of the International Conference on Software Engineering*, 2007.

[dR01] Jim des Rivires. How to use the eclipse api. Technical report, Eclipse Foundation, 2001.

[DR08] Barthélémy Dagenais and Martin P. Robillard. Recommending adaptive changes for framework evolution. In *Proceedings of the International Conference on Software Engineering*. ACM Press, 2008.

[ECH+01] D. Engler, D.Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of SOSP'01*, volume 35, pages 57–72, 2001.

[Ecl06] Eclipse Foundation. SWT: The standard widget toolkit. `http://www.eclipse.org/swt/`, 2006.

[FSJ99] M. Fayad, D.C. Schmidt, and R.E. Johnson. *Building application frameworks: object-oriented foundations of framework design*. Wiley, 1999.

[FZ11] Gordon Fraser and Andreas Zeller. Exploiting common object usage in test case generation. In *IEEE Fourth International Conference on Software Testing, Verification and Validation, ICST 2011*, pages 80–89, 2011.

[GFX+10] Mark Grechanik, Chen Fu, Qing Xie, Collin McMillan, Denys Poshyvanyk, and Chad Cumby. A search engine for finding highly relevant applications. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 475–484, New York, NY, USA, 2010. ACM.

[gooa] Google code search. `http://www.google.com/codesearch`.

[goob] Google codesearch. `http://www.google.com/codesearch`.

[GWZ10] Natalie Gruska, Andrzej Wasylkowski, and Andreas Zeller. Learning from 6,000 projects: lightweight cross-project anomaly detection. In *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA '10, pages 119–130, New York, NY, USA, 2010. ACM.

[HFW07] Raphael Hoffmann, James Fogarty, and Daniel S. Weld. Assieme: finding and leveraging implicit references in a web search interface for programmers. In *UIST '07*. ACM, 2007.

[HM05a] Reid Holmes and Gail C. Murphy. Using structural context to recommend source code examples. In *Proceedings of the International Conference on Software Engineering*, pages 117–125. ACM Press, 2005.

[HM05b] Reid Holmes and Gail C. Murphy. Using structural context to recommend source code examples. In *Proceedings of the International Conference on Software Engineering*, pages 117–125. ACM Press, 2005.

[HR04] R. Hill and J. Rideout. Automatic method completion. In *Proceedings of the International Conference on Automated Software Engineering*, pages 228–235, 2004.

[HW08] Reid Holmes and Robert J. Walker. A newbie's guide to eclipse apis. In *Proceedings of the 2008 international working conference on Mining software repositories*, MSR '08, pages 149–152, New York, NY, USA, 2008. ACM.

[HWM11] Sangmok Han, David Wallace, and Robert Miller. Code completion of multiple keywords from abbreviated input. *Automated Software Engineering*, pages 1–36, 2011. 10.1007/s10515-011-0083-2.

[Joa02] Thorsten Joachims. Optimizing search engines using clickthrough data. In *KDD*. ACM, 2002.

[Joh92] R.E. Johnson. Documenting frameworks using patterns. In *Proceedings of Object-oriented programming systems, languages, and applications (OOPSLA'1992)*, page 76. ACM, 1992.

[KCM07] Huzefa Kagdi, Michael L. Collard, and Jonathan I. Maletic. An approach to mining call-usage patterns with syntactic context. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 457–460, New York, NY, USA, 2007. ACM.

[KE07] Sunghun Kim and Michael D. Ernst. Which warnings should i fix first? In *Proceedings of ESEC/FSE*, pages 45–54, New York, NY, USA, 2007. ACM.

[KGMI04] Shinji Kawaguchi, Pankaj K. Garg, Makoto Matsushita, and Katsuro Inoue. Mudablue: An automatic categorization system for open source repositories. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference*, APSEC '04, pages 184–193, Washington, DC, USA, 2004. IEEE Computer Society.

[KLwHK10] Jinhan Kim, Sanghoon Lee, Seung won Hwang, and Sunghun Kim. Towards an intelligent code search engine. In *AAAI Conference on Artificial Intelligence 2010*, 2010.

[KM06] Mik Kersten and Gail C. Murphy. Using task context to improve programmer productivity. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 1–11. ACM Press, 2006.

[KMCA06] Andrew J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Trans. Softw. Eng.*, 32:971–987, December 2006.

[kod] Koders. `http://www.koders.com`.

[Koh95] Ron Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1137–1145, 1995.

[kru] Krugle code search. `http://www.krugle.com/`.

[KRW07] D. Kirk, M. Roper, and M. Wood. Identifying and addressing problems in object-oriented framework reuse. *Empirical Software Engineering*, 12(3):243–274, 2007.

[LBN+09a] Erik Linstead, Sushil Bajracharya, Trung Ngo, Paul Rigor, Cristina Lopes, and Pierre Baldi. Sourcerer: mining and searching internet-scale software repositories. *Data Min. Knowl. Discov.*, 18(2):300–336, 2009.

[LBN+09b] Erik Linstead, Sushil Bajracharya, Trung Ngo, Paul Rigor, Cristina Lopes, and Pierre Baldi. Sourcerer: mining and searching internet-scale software repositories. *Data Mining and Knowledge Discovery*, 18:300–336, 2009. 10.1007/s10618-008-

0118-x.

[Lew95] T.G. Lewis. *Object-oriented application frameworks*. Manning Publications Co. Greenwich, CT, USA, 1995.

[LH68] P.F. Lazarsfeld and N.W. Henry. *Latent structure analysis*. Houghton, Mifflin, 1968.

[Lie89] KJ Lienberherr. Formulations and benefits of the law of demeter. *ACM SIGPLAN Notices*, 24(3):67–78, 1989.

[LLBO07] Otávio Augusto Lazzarini Lemos, Sushil Krishna Bajracharya, and Joel Ossher. Codegenie:: a tool for test-driven source code search. In *OOPSLA*. ACM, 2007.

[LLMZ06] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32:176–192, 2006.

[LWC09] Fan Long, Xi Wang, and Yang Cai. Api hyperlinking via structural overlap. In *ESEC/FSE '09: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 203–212, New York, NY, USA, 2009. ACM.

[LZ05a] Zhenmin Li and Yuanyuan Zhou. Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. *SIGSOFT Softw. Eng. Notes*, 30(5):306–315, 2005.

[LZ05b] Benjamin Livshits and Thomas Zimmermann. Dynamine: finding common error patterns by mining software revision histories. *SIGSOFT Softw. Eng. Notes*, 30(5):296–305, 2005.

[Mar03] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.

[MBM10] Martin Monperrus, Marcel Bruch, and Mira Mezini. Detecting almost correct variables in object-oriented software and their missing method calls. In *ECOOP*, 2010.

[MH02] Audris Mockus and James D. Herbsleb. Expertise browser: a quantitative approach to identifying expertise. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 503–512, New York, NY, USA, 2002. ACM.

[Mic00] Amir Michail. Data mining library reuse patterns using generalized association rules. In *Proceedings of the International Conference on Software Engineering*, pages 167–176. ACM Press, 2000.

[MKF06] Gail C. Murphy, Mik Kersten, and Leah Findlater. How are java software developers using the eclipse ide? *IEEE Softw.*, 23(4):76–83, 2006.

[MXBK05] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: helping to navigate the api jungle. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 48–61. ACM Press, 2005.

[NNP+09] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. Graph-based mining of multiple object usage patterns. In *ESEC/FSE '09: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 383–392, New York, NY, USA, 2009. ACM.

[RL08] R. Robbes and M. Lanza. How program history can improve code completion. In *Proceedings of ASE*, 2008.

[RL10] Romain Robbes and Michele Lanza. Improving code completion with program history. *Automated Software Engineering*, 17:181–212, 2010. 10.1007/s10515-010-0064-x.

[Rob05] Martin P. Robillard. Automatic generation of suggestions for program investigation. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 11–20, New York, NY, USA, 2005. ACM.

[Rob08] Martin P. Robillard. Topology analysis of software dependencies. *ACM Trans. Softw. Eng. Methodol.*, 17(4):1–36, 2008.

[Rob09] M.P. Robillard. What makes apis hard to learn? answers from developers. *IEEE Software*, 26(6):27–34, 2009.

[SC06] Naiyana Sahavechaphan and Kajal Claypool. Xsnippet: Mining for sample code. In *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 413–430. ACM Press, 2006.

[Sca06] C. Scaffidi. Why are apis difficult to learn and use? *Crossroads*, 12(4):4, 2006.

[Sch97] Hans-Albrecht Schmid. Systematic framework design by generalization. *Commun. ACM*, 40(10):48–51, 1997.

[SFDB07] Zachary M. Saul, Vladimir Filkov, Premkumar Devanbu, and Christian Bird. Recommending random walks. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 15–24, New York, NY, USA, 2007. ACM.

[Sin06] Renuka Sindhgatta. Using an information retrieval system to retrieve source code samples. In *Proceedings of the International Conference on Software Engineering*. ACM, 2006.

[SJ07] Karen Spärck Jones. Automatic summarising: The state of the art. *Inf. Process. Manage.*, 43(6), 2007.

[SJM08] T. Schaefer, J. Jonas, and M. Mezini. Mining framework usage changes from instantiation code. In *Proceedings of the 30th international conference on software engineering*, pages 471–480, 2008.

[SM06] Jeffrey Stylos and Brad A. Myers. Mica: A web-search tool for finding api components and examples. In *Proceedings of the Visual Languages and Human-Centric Computing*, pages 195–202, Washington, DC, USA, 2006. IEEE Computer Society.

[SMY09] Jeffrey Stylos, Brad A. Myers, and Zizhuang Yang. Jadeite: improving api documentation using usage information. In *Proceedings of the 27th international conference extended abstracts on Human factors in computing systems*, CHI EA '09, pages 4429–4434, New York, NY, USA, 2009. ACM.

[tre90] *Overview of the Third Text REtrieval Conference (TREC-3)*, Gaithersburg, MD, USA, 1990. NIST.

[TX07] Suresh Thummalapenta and Tao Xie. Parseweb: a programmer assistant for reusing open source code on the web. In *Proceedings of the International Conference on Automated Software Engineering*, pages 204–213. ACM Press, 2007.

[TX08] S. Thummalapenta and T. Xie. Spotweb: Detecting framework hotspots and coldspots via mining open source code on the web. In *Procedings of ASE'2008*, pages 327–336, 2008.

[TX09] Suresh Thummalapenta and Tao Xie. Alattin: Mining alternative patterns for detecting neglected conditions. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 283–294, Washington, DC, USA, 2009. IEEE Computer Society.

[Vil03a] J. Viljamaa. Reverse engineering framework reuse interfaces. In *Proceedings of ESEC/FSE*, pages 217–226, 2003.

[Vil03b] Jukka Viljamaa. Reverse engineering framework reuse interfaces. *SIGSOFT Softw. Eng. Notes*, 28:217–226, September 2003.

[vM03] Davor Čubranić and Gail C. Murphy. Hipikat: recommending pertinent software development artifacts. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 408–418, Washington, DC, USA, 2003. IEEE Computer Society.

[vR79] C. J. van Rijsbergen. *Information retrieval*. Butterworths, London, 1979.

[WKB09] Markus Weimer, Alexandros Karatzoglou, and Marcel Bruch. Maximum margin matrix factorization for code recommendation. In *RecSys '09: Proceedings of the third ACM conference on Recommender systems*, pages 309–312, New York, NY,

USA, 2009. ACM.

[WZ09] Andrzej Wasylkowski and Andreas Zeller. Mining temporal specifications from object usage. In *ASE 2009: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 295–306, Los Alamitos, CA, November 2009. IEEE Computer Society.

[WZL07] Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. Detecting object usage anomalies. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 35–44, New York, NY, USA, 2007. ACM.

[YF02] Yunwen Ye and Gerhard Fischer. Supporting reuse by delivering task-relevant and personalized information. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 513–523, New York, NY, USA, 2002. ACM.

[YFR00] Yunwen Ye, Gerhard Fischer, and Brent Reeves. Integrating active information delivery and reuse repository systems. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 60–68. ACM Press, 2000.

[ZWDZ04] Thomas Zimmermann, Peter Weisgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 563–572, Washington, DC, USA, 2004. IEEE Computer Society.

[ZXZ+09] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. Mapo: Mining and recommending api usage patterns. In *ECOOP*, pages 318–343, 2009.