Utilizing static and dynamic software analysis to aid cost estimation, software visualization, and test quality management

Gergő Balogh

University of Szeged
Department of Software Engineering

Supervisor: Dr. Árpád Beszédes
Szeged, 2020.

A thesis submitted for the degree of doctor of philosophy of the University of Szeged

University of Szeged
PhD School in Computer Science

To whom it may concern:

to mentors, colleagues, fellow researchers and customers
who made me an experienced professional and helped me finish those projects,
if there had not been any of them then this thesis would have already been published years ago.

Preface

The *goal of software development* was rephrased several times during the history of the profession. Most people widely agree on what this term means, while the exact meaning of software engineering is still being debated. They usually emphasize the creative processes that yield a new or improved version of the software systems. However, we should not forget that eventually these products will be created and used by humans.

The development process is initiated by the *customers,* who would like to solve their problems or accomplish their tasks efficiently. These problems and tasks are translated into high-level requirements by *project managers.* The *senior developers* further specify these by adding details about technical requirements. The *developers* use these specifications to implement the appropriate features, while *testers* ensure that they meet the expected quality and functionality.

However, all of these stakeholders strive for the common goal of creating or improving the software. Their objectives may differ from each other. For example, one of the primary motivations of managers is to deliver new features as soon as possible, while developers would like to enforce certain technical (de facto) standards, which usually increase development time, but later decrease maintenance costs.

My responsibility as a researcher is to aid them in achieving their common goals without hindering their objectives: by improving their processes and tools, and by helping juniors achieve their full potential.

Contents

List of Figures

List of Tables

List of Emphesized Statements

Chapter 1

Introduction

1.1 Motivation

Software development utilizes several abstract concepts. The intricate mesh and finely tuned properties of these entities embody the software systems. There are several properties of these which are well-known for general audiences. They are usually strongly related to their end-user features. Some of these describe the product itself, while others capture the implementation process. For example, users able to asses the number of available features and the amount of time (and eventually money) required to implement them. The layout of the GUI (and other similar attributes) is usually just the publicly visible surface of the enormous set of software system properties. The interconnected entities could reach virtually infinite number and complexity.

The overall goal of customers is to decrease the price, while maintaining (or even increasing) the quality and the number of available features. The duration of development and the produced gain of the process are just two of the main factors of the cost estimation. Managers have to choose various measurement methods and metrics to capture these attributes of the development process. Furthermore, naive approaches like purely quantitative metrics are not able to express properties of intellectual work – for example, we can not measure the value of a book by counting its pages. These issues present enormous pressure for the managers to decrease the net development time, hence lower the cost of the software systems. This eventually leads to the emergence of rapid and agile development models that offer accelerated shipping of new versions for these highly complex systems.

The responsibility of researchers is more pronounced because stakeholders should rely on methodologies devised by these scientists to address the above-detailed issues. The new and improved methods (and the tools based on them) should take into account the complexity of the software systems and their rapid changes. In practice, it means we have to find an efficient way to navigate in the system and locate those parts that could yield some (unexpected) failures. Automatic or semi-automatic corrective and preventive

techniques (like refactoring) can presumably improve the development time by reducing the amount of manual labor required to inspect and fix these error-prone structures. Finally, the efficiency of resource management and task assignment should be improved by examining adequate measurements of the development process.

1.1.1 Impact on Stakeholders' Daily Work

These considerations have a widespread effect on almost all the different stakeholders. Any participant has to understand and interact with the abstract construct, called the software system, to some degree.

Students have to get familiar with not just the strictly functional concepts, but the underlying principles used during their construction. For example, it is not enough to know the definition of classes, they also have to understand how to design a set of them that respect the guidelines of the object-oriented paradigm. The importance of learning from practical examples cannot be called into question. Students have to inspect and analyze several real-life systems during their study.

The comprehension of software systems is not only essential for the students but also for developers. Juniors have to understand complex solutions and advanced technologies, which are considered common practice for senior colleges. The necessity of inspecting previously written code base also concerns non-junior developers. For example, newcomers have to build their own mental image, which represents the structure of the system.

These mental representations tend to capture a larger and larger portion of the software over time, but even the most experienced developers cannot grasp the whole structure, since the software system could achieve virtually infinite complexity. The navigation in this extensive multidimensional data is a challenging task both for the developers and testers. They have to find relevant entities (e.g. classes, namespaces) and inspect their functional and non-functional properties to achieve their goal more efficiently.

Finally, some stakeholders do not understand technical details because their work is only related to source code transitively. Managers do not have to execute implementation related tasks, but they have to assess the impact of these actions. They are usually interested in high-level non-functional metrics, like various cost estimations.

On the other hand, customers do not want to analyze development related concepts. Their goal is to get the required features as soon as possible and for a reasonable price. But there are several phases of the development process that yield no new features. Hence the accomplished work of these phases is invisible to the client. To maintain customer confidence leaders have to present non-functional changes to the client.

Understanding the abstract structures and properties of the software is not sufficient to achieve the common goals of stakeholders, namely, to create or improve the system. Improvements can be accomplished either by adding new features or by fixing errors present in already implemented ones. The comprehension of the system is imperative in both cases, but developers and testers have to execute additional tasks to eliminate errors.

They have to find those parts of the software that will presumably yield some unintended effects. These error-prone chunks could be located in either the software or in any of the related artifacts, like tests. The connection among these entities is rarely marked explicitly, which could yield confusion among testers and developers during the development process.

Managers do not deal with technical details, but they have other objectives to help achieve their common goal. They strive to perfect the development process itself. This task often involves social skills, like tasks assignment, cost estimation, and scheduling various activities of developer teams.

These scenarios only represent some of those challenges that arise from my daily experiences as a software developer. But I hope that my research, presented in this thesis, will help others to understand some of the underlying connections and concepts of software development, which will eventually result in more elaborate methods to aid various stakeholders.

These challenges could be grouped by their beneficiary, i.e. those stakeholders who get most of the gain for their own agenda.

1.1.2 Challenges

Challenge 1: Software comprehension. *Students and newcomers have to get familiar with the large, previously created code base and understand abstract concepts of software development, while senior developers and testers have to navigate efficiently in a usually highly complex software structure.*

Challenge 2: Fault localization. *Developers and testers have to locate those parts of the software and test suite, which could cause failures, i.e. those parts that violate well-established principles.*

Challenge 3: Cost estimation. *The manager has to monitor the properties of the development process to improve it, while quantitative measurements are not able to capture intellectual and creative work, like software developments.*

Challenge 4: Program structure analysis. *Software development research often utilizes a comparison of various interconnected entities. For example, software analysis frequently relies on test-code connections, which are not always noted explicitly.*

1.2 Structure of the Dissertation

The main results presented in the thesis are related to semi- or fully-automated analysis of the software and its development processes. My overall research goal is to provide meaningful insights, methods, and practical tools to help the work of stakeholders during various phases of software development. Some of the methods and tools presented in the

thesis have been utilized in Hungarian and international R&D projects as well as by the industrial partners of the Software Engineering Department of the University of Szeged.

The thesis statements have been grouped into three major thesis points. The structure of these and their connection to various stakeholders and their main topic are shown in fig. 1.1. Thesis and sub-thesis points are marked with a yellow hue, while the central is theme noted with cyan colors. The relevant chapters are depicted as gray rectangles, and various groups of stakeholders are represented with different icons.

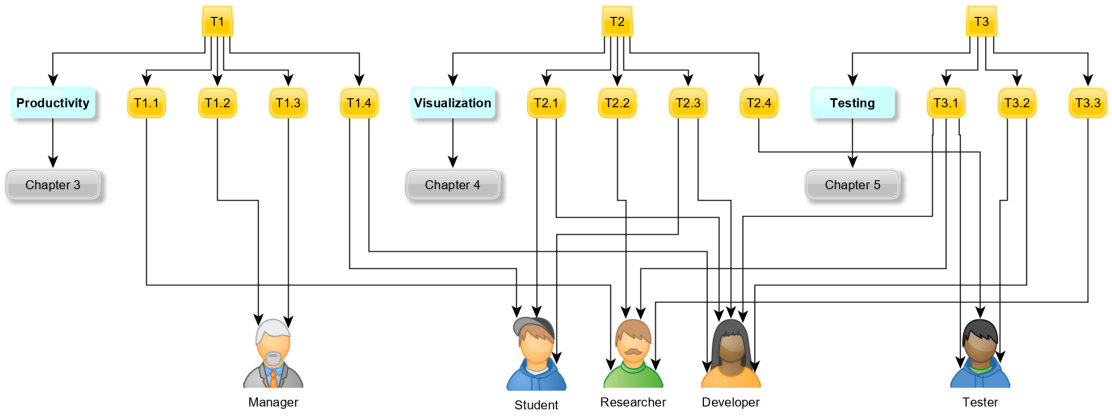


Figure 1.1: Structure of thesis points

T1: Measuring, predicting, and comparing the productivity of developer teams. The contributions of this thesis point are related to the measurement and prediction of the developer team's productivity and will be discussed in Chapter 3.

T1.1: ***Productivity metrics that incorporate types of modifications possess more expressive power.*** I present two new metrics [19] for productivity measurement, namely Typed Modification (TMod) and Modification Effort (MEff). These highly customizable source code metrics are more expressive than the count of changed source code lines, which is commonly used to measure the productivity of developer teams or individual developers.

T1.2: ***The effectiveness of productivity prediction can be increased by taking account of different types of modifications.*** I was able to increase the efficiency of the previous modification cost prediction method based on product and process metrics using the previously mentioned novel approach to productivity measurement. I found that my productivity estimation model [19] can achieve a significant improvement in the overall efficiency of the prediction, from around 50% to 70% (F-measure).

T1.3: ***Measurement of wasted effort via developer interaction data helps managers to improve the development process.*** In a field study [22], I analyzed the aspect of productivity dynamics in a medium-sized J2EE project with 17 developers for

seven months. Based on the experiments, project stakeholders identified several points to improve the development process.

T1.4: ***The average of software quality (a factor of productivity) of the students and the professional developers' work does not show significant differences in classroom exercise.*** I conducted a case study [13] where several student's work were compared to works created by professional developers by using non-functional properties like software quality. The results suggest that there are not any significant differences between the average performance of the two groups. Although, the quality of source code produced by experts had less fluctuation.

T2: Providing immersive methods for software and unit test visualization. This thesis point is related to the visualization of software system architectures and their connected entities. A detailed description could be found in Chapter 4.

T2.1: ***Sandbox game-based techniques can enhance the visualization of software as a virtual city.*** My main contribution is to connect data visualization with end-user graphics capabilities of games. My enhanced visualization method (and its supporting toolset) aided the developers and students to comprehend complex software systems, by constructing a virtual city, which represents abstract, software development related concepts like source code metrics.

T2.2: ***The degree of realism for the city metaphor in software visualization can be measured automatically.*** I presented three low- and one high-level metric that expresses various features of a virtual city used to visualize software systems to capture the differences between a realistic and an unrealistic city. Both high- and low-level metrics were validated by a user survey [15]. The results show that it is possible to construct methods that can estimate the degree of realism of a generated city.

T2.3: ***Integration of integrated development environment and software visualization can aid developers to understand software systems.*** I present an approach to integrate our visualization tool, CodeMetropolis, into the Eclipse IDE. A set of plug-ins were implemented that were able to connect these two pieces of software. Hence we became capable of integrating an elaborated visualization technique without disturbing the daily routine of developers.

T2.4: ***The city metaphor is able to visualize test-related metrics and test-code connection.*** I extended the metaphor to include properties of the tests related to the program code using a novel concept [20]. This allowed the combining two previous approaches: a method to express test quality in terms of metrics, and visualization of code related metrics in the CodeMetropolis framework.

T3: Spotting the structures in the package hierarchy that required attention using test coverage data. In this thesis point, I summarize my results considering test

coverage analysis and its usage to improve the quality of (unit) tests and their subjects. For further details see Chapter 5.

T3.1: ***Community detection is able to cluster test cases and code elements simultaneously.*** To automate various test and code analysis tasks, I employ a clustering algorithm that can group test and code items. This method allowed the simultaneous inspection of tests and their subjects and aided researchers to conduct further analyses.

T3.2: ***Classification of structural discrepancies of tests and code elements helps developers to improve test and code quality by providing contextual data to restore test-code traceability links.*** This work addressed the quality of unit test suites from a novel angle. My approach was to compare the physical organization of tests and tested code in the package hierarchy to what can be observed from the dynamic behavior of the tests. Guidelines through examples for refactoring the problematic tests were provided based on measurements of large open-source systems with notable test suites.

This data was interpreted as contextual information for a semi-automatic method for recovering test-to-code traceability links. It is based on computing connections using static and dynamic approaches, comparing their results and presenting the discrepancies to the user, who will determine the final traceability links based on the given information.

T3.3: ***Providing a methodology for unified graph's discrepancy analysis.*** I presented a methodology for a unified graph's discrepancy analysis, named UniGDA. It is based on the previously defined domain-specific discrepancy detection techniques, which were extended to arbitrary graphs by providing several domain-independent similarity functions and patterns.

The structure of the rest of my thesis is as follows. Basic concepts and terms are elaborated on along with the related works in chapter 2. I present my detailed results in chapters 3 to 5. Finally, I conclude my thesis with the list of publications in chapter 7, summarize my results, and briefly introduce my plans for further work in chapter 6.

Chapter 2

Preliminaries

This chapter contains a brief introduction to the main topics detailed in subsequent chapters. My goal is not to give an in-depth evaluation of the state of the art techniques and research, but to provide (informal) definitions of core concepts and general notions. The knowledge presented in this section aims to help the reader to understand the technical details discussed in chapters 3 to 5.

In the first section, I introduce the core concepts required to understand the elaborations in chapter 3, which will support my results for thesis point 1. I listed all the general notions related to chapter 4 in the next section. These will help the reader to understand the research connected to thesis point 2. Finally, the definitions in the last part of this chapter aim to cover concepts related to chapter 5 and thesis point 3.

2.1 Productivity Measurement and Prediction

As stated earlier, the first main topic of this thesis is related to cost estimation, more precisely, productivity measurement. Productivity describes various measures of the efficiency of production. To understand the compound concept of productivity, we have to inspect its constituents and their relations. Often (yet not always), a productivity measure is expressed as the ratio of an aggregated output to a single input or an aggregate input used in a production process, i.e., output per unit of input [81]. If they are interpreted correctly, these components are indicative of productivity development and approximate the efficiency with which inputs are used in an economy to produce goods and services.

The Organisation for Economic Co-operation and Development (OECD)[1] defined [68] so-called workforce productivity as "the ratio of a volume measure of output to a volume measure of input".

[1] OECD is an intergovernmental economic organization with 36 member countries, founded in 1961 to stimulate economic progress and world trade.

2.1.1 Measuring Productivity

The simple and straightforward definition masks the underlying complexity of productivity measurements: that these metrics rely on the assessment of resources used and on the amount of gain achieved during the process. Finding the exact definition for these underlying metrics is hard for those professions, which mostly yield non-physical results (like software systems). Hence, the area of cost and productivity estimation is a recurring topic in the software engineering literature. Productivity research is mainly centered around productivity influence factors. Traditional factor-based models for measurement and prediction include Putnam's SLIM, Albrecht's FP method of estimation, the COnstructive COst MOdel (COCOMO and COCOMO II) [32]. One may distinguish technical and soft factors that influence productivity [103]. We refer the interested reader to the survey of Trendowicz and Münch [95]. Different researchers and stakeholders use various definitions of input (required effort) and output (achieved gain) in practice. For example, Tóth, G. et al. [93, 94] used a quantitative approach. They defined input as the *net development time* required to change source code lines, while the output was the number of changed lines. This definition can be interpreted as a domain-specific version of the general one defined by OECD.

We will use a similar definition during our research. For the rest of this thesis, *productivity means workforce productivity, i.e. the ratio of input and output for the process (the software development itself), where the input is measured as the time spent to accomplish the result.* These considerations are depicted with the following informal equation.

$$\text{productivity} = \frac{\text{output}}{\text{input}} = \frac{\text{gain}}{\text{effort}} = \frac{\text{gain}}{\text{time spent}} \tag{2.1}$$

2.1.2 Impact Analysis

Productivity is highly connected to changes made on the software. For example several factors identified by Finnie, Wittig, and Petkov [40] are directly affected by source code modifications.

Impact analysis aims to capture the underlying principles and properties of these modifications. It is defined by Bohner and Arnold [27] as identifying the potential consequences of a change, or estimating what needs to be modified in order to accomplish a change. In contrast, Pfleeger and Atlee [71] focus on the risks associated with changes and state that impact analysis is the evaluation of the many risks associated with the change, including estimates of the effects on resources, effort, and schedule.

Both of these are associated with change management processes. Each change to the software is expensive and risky, but it also has the potential for generating revenue because of desired new functionality or cost savings in future maintenance. Hence the goal of much research is to develop more reliable impact analysis methods and tools [77, 28, 33].

The work presented in this thesis is marginally connected to impact analysis. We were using the modifications in the source code to asses one of the underlying metrics of productivity called the output or gain. More precisely, we applied the same categorization of changes according to the component they affect [3].

2.1.3 Genetic and Evolutionary Algorithms

During our experiment, we used various machine learning algorithms to improve the performance of the productivity prediction model. Machine learning is the scientific study of algorithms and statistical models that computer systems use in order to perform a specific task effectively without using explicit instructions, relying on patterns and inference instead [58][2].

We applied two types of supervised and reinforcement learning, namely a decision tree [93] and an evolutionary algorithm (backed with a genetic representation) [93, 19]. Supervised learning was used to predict future productivity for a development team, because it builds a mathematical model of a set of data that contains both the inputs (various process and product metrics) and the desired outputs (the value of productivity). At the same time, reinforcement learning aims to fine-tune this prediction model since the algorithm ought to take actions in an environment to maximize some notion of cumulative reward (performance of prediction model).

Decision tree learning is a method commonly used in data mining [73]. The goal is to create a model that predicts the value of a target variable based on several input variables, in our case, to predict future productivity based on various metrics. A tree is built by splitting the set of observations, constituting the root node of the tree, into subsets - which constitute the successor children. This process is repeated on each derived subset recursively. The recursion is completed when the subset at a node has all the same values of the target variable (i.e.the future productivity), or when splitting no longer adds value to the predictions [80, 72].

In artificial intelligence, an evolutionary algorithm (EA) is a subset of evolutionary computation, a generic population-based metaheuristic optimization algorithm [101]. It is a candidate solution to the optimization problem. The algorithm uses mechanisms inspired by biological evolution, such as reproduction, mutation, recombination, and selection. The evolution of the population then takes place after the repeated application of the above operators [64]. The genetic algorithm is the most popular type of EA. It is also inspired by nature, namely genetics. It represents each individual with a set of genes, its genotype, which could be indicating something about the problem being solved, and it applies operators such as recombination and mutation to modify the genotype of the individuals [83].

[2]The definition "without being explicitly programmed" is often attributed to Arthur Samuel, who coined the term "machine learning" in 1959, but the phrase is not found verbatim in this publication, and may be a paraphrase that appeared later.

2.2 Game-based Software Visualization

People use different mental processes to comprehend the world. Some of them need numbers, others use abstract formulas, but most of us like to see the information visualized as colors, shapes, and figures. Visualization is any technique for creating images, diagrams, animations, or any other visual content to communicate a message or information. The use of visualization to present information is not a new phenomenon. It has been used in maps, scientific drawings, and data plots for over a thousand years [97, 96].

A lot of data visualization techniques and tools were designed and implemented in software engineering research and practice, as well. There are traditional visualization tools like Rigi [107], sv3D [63] and SHriMP Views [86], which are built on innovative ideas, but often it is difficult to interact with them by current standards, and they usually fall behind in terms of graphics compared to today's computer games, for instance.

Besides that, there already exist a number of sophisticated software tools that are able to visualize the huge amount of data collected by collaborative tools (like SourceForge[] and Github[]), for instance, Gource [46], Logstalgia [60] and StarGate [62]. However, most of these tools use abstract shapes and simple graphical primitives like charts and vertex graphs.

Finally, several methods use artificially generated copies of real-life entities to encode their message. The most closely related approaches to our method and its supporting tool discussed in this thesis are CodeCity [106] and EvoSpace [59], which use the analogy of skyscrapers in a city. Despite their appealing appearance and great potential in general, these tools still use relatively low fidelity graphics compared to today's most advanced computer games. CodeCity simplifies the design of the buildings to a box with height, width, and color. The quantitative properties of the source code – called metrics – are represented with these attributes. In particular, each building represents a class where height shows the number of methods, width indicates the number of attributes, and color shows the type of the class. The buildings are grouped into districts as classes are tied together into namespaces. The diagram itself resembles a 3D bar chart with grouping. EvoSpace uses this analogy in a more sophisticated way. The buildings have two states: closed – when the user can see the large scale properties like width and height, and open – when we can examine the low, small scale structure of the classes, see the developers and their connections. It also provides visual entity tagging and quick navigation via the connections and on a small overview map.

There are several types of classifications and grouping of the various visualization techniques. The research presented in this thesis describes methods that yield interactive, non-static representations of the software. It means that the user can change the visual appearance of the moving images in real-time to inspect various parts of it. Our technique has several applications starting with scientific, through educational, to product visualization. Either of these could aid further visual analysis of the data.

2.2.1 Underlying Metaphors

The central role of metaphors in human languages, which are emphasized by several authors like Deutscher [36], reveals deeper insights about how we process information. Similarly to phrase and word formation, all of these visualization methods use some kind of underlying representation to encode their information. We call these the metaphors on which the visualization builds.

Simple metaphors are sometimes more difficult to notice since they are more entwined with our thought processes. For example, consider a simple bar chart that represents the temperature for each day in a week. We could say that "today the temperature is higher (or lower) than yesterday". However, there is not any fundamental connection between cold-low and warm-high. These relations are made by our abstract representation and eventually get encoded into the underlying metaphor of the chart itself.

More complex metaphors like location and the interconnectivity of various shapes are present in several graph-based visualizations, like flow-charts and UML diagrams. Realistic metaphors are commonly used to present the results of different engineering simulations, for example, thermograms, which represents temperature with false coloring. Real-object based visualization is often used to capture abstract concepts and properties frequently used in software development [57, 106, 59].

2.2.2 Phases of Visualization

We used the 4-step division of visualization introduced by Upson et al. [98]. These steps, which are common for each visualization process, are *filtering* the data, *mapping* it to the appropriate graphical items, *rendering* the image and finally *displaying* it to the user.

Filtering step Software analysis usually produces large quantities of information, which is hard to understand and display even with multi-dimensional visualization techniques. During this step, we select the data that should be inspected during the visualization. Our goal was to provide a general yet practical input data structure.

Mapping step There are several stakeholders in software development interested in different aspects of the process. For example, developers tend to care about the low-level connection between source code items, while leaders are usually interested in higher-level metrics like maintainability. During the mapping step, the user can specify relationships between data and metaphor level items and their properties, based on their own agenda.

We will use the term *geometry* first in a broader sense, then in the commonly accepted one. We will refer to the collection of the object description, which can determine the visual appearance of the graphical items as a geometry. For example, in the case of a simple bar chart, we will store the index, the color, and the height of the bars. For more complex metaphors (like the city-metaphor), the geometry could contain the object graphs of the whole city.

Rendering step In the rendering step, we use the description of graphical items produced by the mapping step to generate the visualization, which is ready to display for the user.

Displaying step The final step of visualization is to display the result to the user, who can inspect and understand its meaning. This step usually produces more data and open questions, which could trigger a new cycle of the visualization process.

2.2.3 Extending Third-party Application

It is not cost-efficient (and probably impossible) to meet all the needs of the users with a single software system. There will always be special requirements and unorthodox use cases. Fortunately, several techniques allow the end-users to extend the set of the available functionality of the system. There are not any legal obstacles to do so. For example, a popular game, called Minecraft [2], explicitly allows and encourages the users to extend the game in its license agreement [1].

There are various methods to provide end user-level extensions. During this, we will discuss two of them. The first is the so-called modification (or mod for short). It means that the user needs to disassemble the internal structure of the software, modify it, and re-assemble the extended version. A good example is the previously mentioned game, Minecraft, but several other games offer this kind of extendibility. Note that in this case, the vendor does not provides very little to no additional support, besides declaring its legal status in the license agreement.

The other technique is more common among non-game software. In this case, the vendor provides a full-blown plug-in architecture, which allows users to add and modify features without disassembling the system. For example, Eclipse [38], which is a well-known integrated development environment, has a thin core, and almost every other feature is provided by external plug-ins.

2.2.4 Video Game Genre

The video game genre is a classification assigned to a video game based on its gameplay interaction rather than visual or narrative differences.[9] The video game genre is defined by a set of gameplay challenges and is classified independently of their setting or game-world content, unlike other works of fiction such as films or books. For example, a shooter game is still a shooter game, regardless of where or when it takes place.[5, 52]

From the myriad of genres, we will concentrate on two of them: *role-playing* and *open word or sandbox* games. In role-playing video games (RPG), the player controls the actions of a character (and/or several party members) immersed in some well-defined world. For example, in the case of Minecraft, the player impersonates an explorer, who has to survive in the virtual world by crafting new tools and defeating monsters. Other significant

similarities with pen-and-paper games include developed story-telling and narrative elements, player character development, complexity, as well as replayability and immersion. In Minecraft, there are various achievements that the player can reach by obtaining certain items or executing specific actions.

However, these achievements do not have a strict order, and all of them are optional, which highlights another essential property of Minecraft from our point of view, namely its open-world features. In video games, an open world is a virtual world in which the player can explore and approach objectives freely, as opposed to a world with more linear gameplay. An open world is a level or game designed as nonlinear, with open areas many ways to reach an objective. In Minecraft could be anything from building a tree-house or to constructing an automated monster butchery.

These qualities of Minecraft make it ideal for extension with new features, like compatibility with other software [11], which is a similar approach to our visualization related research.

2.3 Clustering Test and Code Elements

There is a large body of work in the area of code smells, and researchers only recently started to apply similar concepts to check software tests and test code for quality issues. For tests that are implemented as executable code, Van Deursen et al. introduced the concept of *test smells*, which indicates poorly designed test code [35], and listed 11 test code smells with suggested refactorings. Our work best relates to their *Indirect Testing* smell. Some follow-up researches use these ideas in practice. For example, Breugelmans and Van Rompaey present TestQ, which allows developers to explore test suites and quantify test smelliness visually. They also demonstrate its use on test suites for both C++ and Java systems [29].

Our work significantly differs from these approaches because we are not concerned about code-oriented issues in the tests but about their dynamic behavior and relationship to their physical placement. We achieved this by comparing different groupings (clustering) of test cases and code elements.

2.3.1 Best practices of unit test writing

Unit testing is a low-level testing activity that has a close relation to the source code of the system under test. During this test, we search for defects in and verify the correct functioning of software components (modules, programs, objects, classes, etc.), which are separately testable [25]. There are many guidelines for how to write and organize unit tests (e.g.[50, 29, 99]), but most of them start by emphasizing two basic test design concepts: “unit tests should be isolated” and “test only one code unit at a time” [50]. Besides, unit testing frameworks – such as JUnit[55], which is part of the *de facto* unit test family of frameworks – have naming and packaging conventions on how unit tests should be

organized with respect to their intended goal (unit to test). Also, since different build systems and development environments suggested similar conventions, these eventually became best practices.

One of these conventions is about how tests are placed into logical or physical modules (such as packages in the case of Java and folders on the file system). Most environments logically group test and program code together, while physically separating them from each other. This can mean, for instance, that program code and the associated tests are put in the same logical package or namespace, while they are in separate folders on the file system.

For the purposes of the research presented in this thesis, we thus assume that a well-designed unit test has the following two essential properties.

1. A unit test should exercise the unit and only the unit it was designed for. Execution of code in other units on which the tested one is dependent should be eliminated using *stubs* and *mocks*.

2. Unit tests should follow a clear naming and packaging convention, which reflects both the purpose of the test and the structure of the tested system, providing clear traceability between the test cases and the tested units.

2.3.2 Code Coverage

The term *code coverage* in software testing denotes the amount of program code which is exercised during the execution of a set of test cases on the system under test. This indicator may simply be used as an overall *coverage percentage*, a proxy for test completeness, but typically more detailed data is also available about individual program elements or test cases. Code coverage measurement is the basis of several software testing and quality assurance practices including white-box testing [69], test suite reduction [75], or fault localization [75].

It includes various granularity levels of the analysis (such as component, method, or statement) and different types of "exercised parts of program code" (for instance, individual instructions, blocks, control paths, data paths, etc.). The term *code coverage* without further specification usually refers to statement level analysis and denotes *statement coverage*. Statement coverage shows which instructions of the program are executed during the tests and which are not touched. Coarser granularity level coverage criteria (such as methods, classes, or components) are also common, for instance, when the system size and complexity do not allow for a fine-grained analysis. Also, often it is more useful to start the coverage analysis in a top-down fashion by starting from the components that are not executed at all, extend the tests to cover that component at least once, and then continue the analysis with lower levels. In particular, *method level coverage* is a good compromise between analysis precision and the ability to handle large systems.

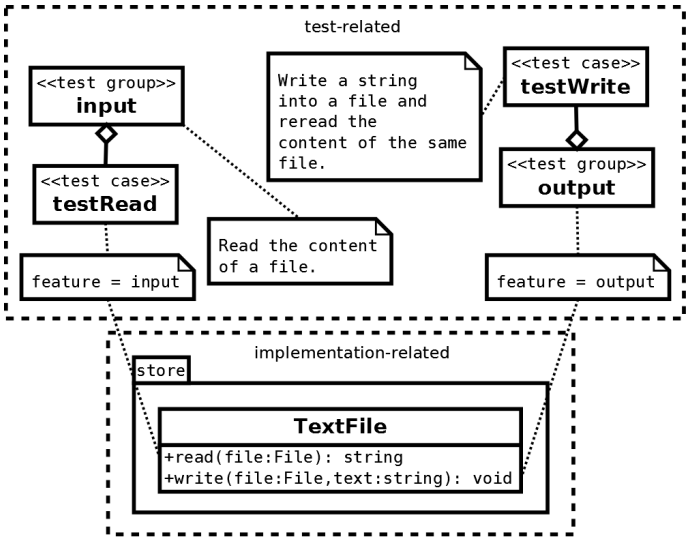


Figure 2.1: An example of functional units

In our research, we primarily deal with this granularity, that is, we treat procedures (Java methods in particular) as atomic code elements that can be covered. At this level "covered" means that the method has been executed at least once during the tests, but we do not care about what instructions, paths, or data have been exercised in particular.

Code coverage has applications with high significance in academic researches. For example, coverage-driven test case generation and code coverage-based fault localization. In the second case, the program elements are ranked according to how suspicious they are to contain the fault based on test case coverage and pass/fail status.

Functional Unit

Similarly to code metrics, test-related metrics can be defined for the different code and test artifacts [49, 39]. For example, a popular test-related metric is the previously defined *code coverage*, which expresses the percentage of how well the code elements are covered by the test cases. Code coverage can be computed at different levels: a single global value can express to what extent all the test cases are able to check the whole code base; a value can be assigned to method-test case pairs to show detailed coverage; or it can be assigned to *functional units* formed from pairs of code and test groups [89].

Functional units are organized around the functionalities (features) of the software. For each feature there are test cases created to test the given functionality, we call these the *test groups*. Similarly, the features were implemented in certain classes and methods, which constitute the *code group*. A *functional unit* consist of the code group and the test group of the same feature, while a *cross-functional unit* consists of a code group and a test group of two different funcionalities [89].

As an example, consider a class called `TextFile` with only two methods, `read()` and `write()` implementing *input* and *output* features, which are tested by the test cases

testRead and **testWrite**, respectively (Figure 2.1). This enables us to define two functional units: *input-input* with **testRead** and `read()`; *output-output* with **testWrite** and `write()`; as well as two cross-functional units: *input-output* with **testRead** and `write()`; *output-input* with **testWrite** and `read()`.

In a functional unit, a code group implements some functionality and the associated test group is intended to verify it. Analyzing the system and the tests with this kind of division can be an aid in test selection, prioritization, and test suite reduction activities [89, 54].

2.3.3 Traceability recovery in unit tests

Several methods have been proposed to recover traceability links between software artifacts of different types, including requirements, design documentation, code, test artifacts, and so on ([85, 33]). The approaches include static and dynamic code analysis, heuristic methods, information retrieval, machine learning, and data mining based methods.

We used the previously introduced comparison of various tests and code clustering to recover *test-to-code* traceability. The purpose of recovering this is to assign test cases to code elements based on the relationship that shows which code parts are tested by which tests. This information may be critical in development, testing, or maintenance, as already discussed.

Our work concentrated on unit tests, in which case the traceability information is mostly encoded in the source code of the unit test cases, and usually, no external documentation is available for this purpose. Traceability recovery for unit tests may seem simple at first ([23, 34, 43]), however, in reality it is not ([42, 56]).

Bruntink and Van Deursen et al. [30] illustrated the need and complexity of the test-to-code traceability. They investigated factors of the testability of Java systems. The authors concluded that the classes dependent upon other classes required more test code, and suggested the creation of composite test scenarios for the dependent classes. Their solution heavily relies on test-to-code traceability relations.

Rompaey and Demeyer et al. [74] evaluated the potential of six traceability resolution strategies (all are based on static information) for inferring relations between developer test cases and units under test. The authors concluded that no single strategy had high applicability, precision, and recall. However, combining these approaches with strategies relying on developer conventions (e.g.naming convention) and utilizing program-specific knowledge (e.g.coding conventions) during the configuration of the methods provided better overall results.

In summary, most of the mentioned related works emphasize that strong test-to-code traceability links are difficult to derive from a single source of information and combined or semi-automatic methods are required. Our research follows this direction, as well.

2.3.4 Clustering and Classification

Tengeri et al. proposed an approach to group related test and code elements together, but this was based on manual classification done by the testers and developers [89]. In the method, various metrics are computed and used as general indicators of test suite quality, and later it has been applied in an in-depth analysis of the WebKit system [100].

There are various approaches and techniques for automatically grouping different items of software systems together based on their structural or behavioral properties. Mitchell and Mancoridis [65] examined the Bunch clustering system, which, unlike other software clustering tools, uses search techniques to perform clustering. Schwanke's ARCH tool [78] determined clusters using coupling and cohesion measurements. The Rigi system [66], by Müller et al., pioneered the concepts of isolating omnipresent modules, grouping modules with common clients and suppliers, and grouping modules that had similar names. The last idea was followed up by Anquetil and Lethbridge [6], who used common patterns in file names as a clustering criterion.

The concept of community structure arises from the analysis of social networks in sociology. Community structures can be identified in many other real-world graphs and have applications in biology, economics, and engineering, among others. Recently, efficient community detection algorithms have been developed, which can cope with extensive graphs with millions of nodes and potentially billions of edges [26]. The application of these algorithms to software engineering problems is emerging. Hamilton and Danicic [51] introduced the concept of *dependence communities* on program code and discussed their relationship to program slice graphs. They found that dependence communities reflect the semantic concerns in the programs. Šubelj and Bajec [87] applied community detection on classes and their static dependencies to infer communities among software classes.

We performed community detection on method level, using dynamic coverage information as relations between production code and test case methods, which we believe is a novel application of the technique.

Chapter 3

Measuring, Predicting, and Comparing Productivity of Developer Teams

In this chapter, we present the researches related to the productivity of development teams. The structure of the related thesis point and its connections to various stakeholders and topics are shown in fig. 3.1.

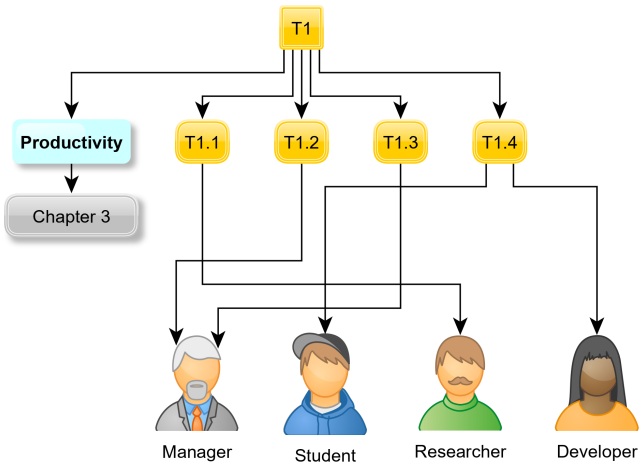


Figure 3.1: Structure of thesis points

Productivity describes various measures of the efficiency of production. We used the previously introduced definition of productivity, where it is expressed as the ratio of aggregate output to a single input or an aggregate input used in a production process, i.e., output per unit of input.

Two major issues that need to be addressed during software development from the manager's point of view: cost prediction and wasted effort handling. During the planning, development, and maintenance of software projects, one of the main challenges is

to accurately predict the modification cost of a particular piece of code. Furthermore, several parts of the source code are usually re-written due to imperfect solutions before the code is released. This wasted effort is of central interest to the project management to assure on-time delivery. Both of these issues are related to challenge 3. An overview of our research phases are shown on fig. 3.2.

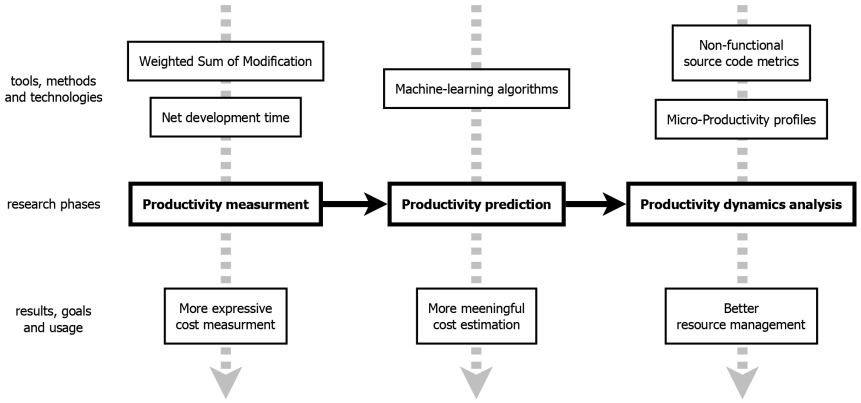


Figure 3.2: Phases of our productivity related researches

Several methods are traditionally applied to address these issues, and many of them are based on static code investigation. We experimented with the combined use of product and process metrics to improve cost prediction. The method depends on several important parameters that can significantly influence the success of the prediction model. We applied machine learning to increase the accuracy of the prediction and fine-tune our model. In the following sections, we describe the usage of search-based methods (one genetic algorithm in particular) to calibrate these parameters.

To address the second issue, we propose a productivity analysis method where productivity is expressed through dynamic profiles – the so-called Micro-Productivity Profile (MPP). They can be used to characterize various constituents of software projects, such as components, phases, and teams. We present and evaluate profiles of two important axes of the development process: by milestone and by application layers. MPP can be an aid to identify wasted effort, to take project control actions, and to help in planning future projects.

For the first set of experiments, four industrial projects were analyzed, and the accuracy of the predictions was compared to previous results. We found that by calibrating the parameters using search-based methods, we could achieve significant improvement in the overall efficiency of the prediction, from about 50% to 70% (F-measure). During the second phase of the experiments, we measured the productivity of a medium-sized J2EE project. We collected detailed traces of developers' actions using an Eclipse IDE plug-in for seven months of software development throughout two milestones. Based on the experimental results, project stakeholders identified several points to improve the development process. It is also acknowledged that profiles show additional information compared to a naive

diff-based approach.

Besides measuring the productivity of professional development teams, it is crucial to understand various aspects of their roots, like educational background. There is a widely accepted belief that education has a positive impact on the improvement of expertise in software development. The studies in this topic mainly focus on the product, more closely the functional requirements of the software. Besides these, they often pay attention to the individual so-called basic skills like abstract and logical thinking. However, we cannot find any references where the final products of classroom exercises were compared by using non-functional properties like software quality. However, these attributes are often used during real-life projects to asses their values. To address this issue, we introduced a case study where several students' works are compared to works created by professional developers. The model that is used to measure the various aspects of software quality, is also known in the industrial sector. Hence it provides a well-established base for our research.

The notion of productivity is strongly related to cost estimation and resource management. One of the tasks in software cost estimation, especially in the evolution phase, is to predict the cost (required effort) of modifying a piece of code. A possible approach for such modification effort prediction is to use various software attributes from historical, development data and from the current version of the software. The attributes can be expressed in the form of software metrics, both product, and process. Product metrics are calculated by performing the static analysis of the software (a simple example is the logical lines of code), while process metrics can represent time-related quantities collected during project implementation (for example, the net development time of the modifications).

Since these two kinds of metrics capture different aspects of the software development (i.e., the product and the process), our assumption was they can encode more information when utilized together. In the case of productivity analysis, our experiments showed that by combining these two types of metrics into a single model, managers were able to increase the accuracy of their modification cost predictions.

Our research combines several granularity levels: it is built upon fine-grained productivity data to model it as Micro-Productivity Profile, but enables to reason about varying levels of software development productivity observation.

3.1 Defining Weighted Modification-based Productivity Measure

In this section, we elaborate on the experiments that let us define a more expressive productivity measurement then the previously used ones, which are mostly based on counting the lines of source code. We used these new metrics to investigate further productivity dynamics, which will eventually aid the manager during cost analysis (challenge 3).

We used the following process (shown in Fig. 3.3) to define and fine-tune productivity

measures. The process contains two steps: in the measurement phase, we collect all necessary data to calculate productivity metrics; then, we applied a genetic algorithm to fine-tune these metric definitions.

The experiment starts with a measurement phase where the data is collected from various sources: the metrics about the evolution of the software, the source code from the SVN version controlling system, and the metrics estimations which were given by the project manager. This phase has two main tasks; to collect and calculate the process and product metrics and to detect and group the modifications of the source code between the revisions. The metrics and the modification groups are sent to the genetic algorithm, which prepares a population of individual entities. During the initial set-up of the population and the evolution steps, two metrics are calculated: Typed Modification (TMOD), which was defined as the weighted count of modifications between two revisions; and Modification Effort (MEFF), the ratio of TMOD and the net development time of these modifications. This latter one is used to measure productivity.

$$\text{productivity} = \frac{\text{output}}{\text{input}} = \frac{\text{gain}}{\text{effort}} = \frac{\text{TMOD}}{\text{net development time}} = \text{MEFF} \tag{3.1}$$

Afterward, the prediction model (targeting MEFF) is evaluated, and its F-measure value is used as fitness to rank the individuals in the population and select the best entities for breeding. As the final step of the evolution cycle, the new weights of the modification are calculated, and the model is updated. When the precision reaches an appropriate value, the GA stops, and a new, enhanced model is built using the weights of the best entity in the final population. This MEFF prediction model is the output of the execution of the framework.

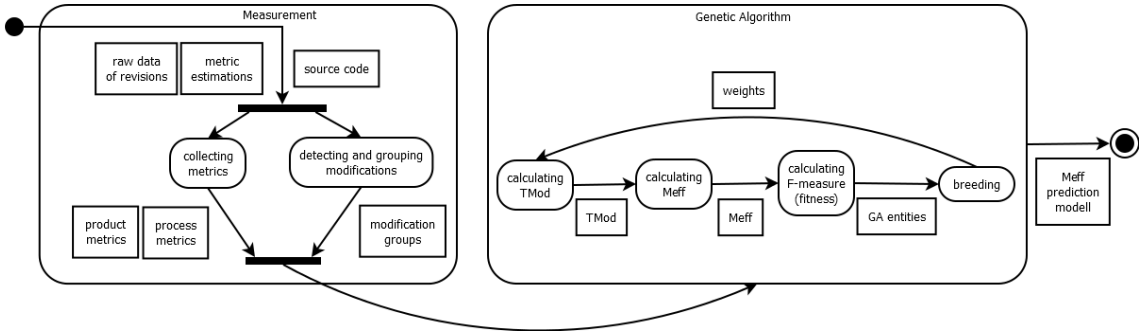


Figure 3.3: Overview of the experiment

The initial data was collected during the experiment from about 800 revisions, in an approximately 75 days long period.[1] Both R&D and industrial projects were analyzed. The majority of their codebases were written in Java language using the Java EE 6 virtual machine and the Seam 2 framework.

[1] Altogether 2200 records were collected as a learning set.

3.1.1 Modification Effort Prediction

In this phase of research, our goal was to predict the level of productivity (high, medium, low) based on various product and process metrics of the system.

Our research is based on the work of Tóth et al.[93]. Both prediction models [19] used the same process and product metrics together per source file basis as separated entities.

Effectively Changed Lines Of Code (ECLOC) defined as the delta calculated from the SVN, the number of added, deleted, or modified lines by comparing the previous version of the class with the current version. Tóth et al.[93] defined the metric Level of Modification Complexity (LMC) as the ratio of DT and ECLOC for the next change of the file or class, which embodied the target of prediction in the previous model. We can rephrase their definition by using the common notion of productivity.

$$\text{productivity} = \frac{\text{output}}{\text{input}} = \frac{\text{ECLOC}}{\text{DT}} = \text{LMC} \tag{3.2}$$

To extend the previous framework [93], a new metric called Modification Effort (MEFF) was defined and was calculated as follows. At first, the modifications were grouped, based on the target entity (e.g.: method) and the preformed action (e.g.: creation) [3]. The weighted count of these modifications was called Typed Modification (TMOD). This expresses the different amounts of the developer's effort used in the modifications. Finally, the ratio of the net development time with the TMOD is the MEFF metric. The weight was defined based on the groups of modifications. We inspected the modification groups listed in table 3.1.

	class	**method**	**data member**
creation	•	•	•
deletion	•	•	•
accessibility change	Line 1	Line 5	Line 3
prototype change		Line 5	
return type change		Line 10	
size change		Line 7	
type change			Line 2

Table 3.1: Inspected modification per code element

We also provided some examples in listing 3.1 and listing 3.2.

Listing 3.1: Original version

```
1 public class LogOnlyModeControl extends ViewlessControl<LogOnlyModeModel> {
2     public static String NAME = "log-only";
3     private PartAccessor partAccessor;
4
5     public LogOnlyModeControl(LogOnlyModeModel model, PartAccessor accessor) {
6         super(model);
```

```
7           this.partAccessor = accessor;
8       }
9
10      public void logDenied() {
11          activityMonitor.log(new ModeEvent(NAME, ModeEvent.DENIED));
12      }
13  }
```

Listing 3.2: Modified version

```
1  private✍ class LogOnlyModeControl extends ViewlessControl<LogOnlyModeModel> {
2      public static Double✍ NAME = 42;
3      public✍ PartAccessor partAccessor;
4
5      protected✍ LogOnlyModeControl(LogOnlyModeModel model✍) {
6          super(model);
7          this.load()✍
8          this.init()✍
9      }
10
11     public Boolean✍ logDenied() {
12         return activityMonitor.log(new ModeEvent(NAME, ModeEvent.DENIED));✍
13     }
14 }
```

Instead of the LMC, MEFF was used as the target function. In doing so, we could distinguish details about the modifications, which was impossible in the previous model.

$$\text{productivity} = \frac{\text{output}}{\text{input}} = \frac{\text{TMOD}}{\text{DT}} = \text{MEFF} \tag{3.3}$$

Similarly to the previous experiment, we used a machine-learning algorithm to construct the prediction model. The Weka framework [53] machine learning and the 10-fold cross-validation utility were used to implement and evaluate this model. We chose the F-measure as the fitness value, which is the harmonic mean of precision and recall.

3.1.2 Measuring Developer Productivity with Modification Effort

A crucial component of measuring the overall developer productivity is to define a comparable measure of the effort spent on various modifications. We modeled Modification Effort during software development as the ratio of profit (program code) and time spent to produce it [19]. There are several ways to express the profit, possibly the most trivial metric is the count of produced lines of code. However, it hides key differences between modifications. To overcome this disadvantage, we chose to replace this metric. We calculated profit using the number of higher-level modifications, like method creation or deletion. This added an abstraction level that made a difference between code constructs,

which required a different effort but were written in the same number of lines. This metric provides a more detailed view of the various modifications in the source code than other traditional metrics based on the changed lines of code [93].

Listing 3.3: Previous version (1)

```
1 class IntSet {
2     protected double FindGreater( double limit ) {
3         for (int _i = 0; _i < Items.Count(); _i++) {
4             double _current = Items[ _i ];
5             if ( _current > limit ) {
6                 return _current;
7             }
8         }
9     }
10 }
```

Listing 3.4: Current version (2)

```
1 class IntSet {
2     protected int ✍ FindGreater( double limit )   {
3         for (int _i = 0; _i < Items.Count(); _i++) {
4             int ✍ _current = Items[ _i ];
5             if ( _current > limit ) {
6                 return _i; ✍
7             }
8         }
9     }
10 }
```

To better understand the meaning of these new metrics, let us consider the following example. For the formal definition of MEff and TMod see appendix A.2. MEff is a number that express the average amount of performed modification during a unit of time. The code example in listing 3.3 will be used to illustrate the measures for expressing programmer productivity.

The modified code in listing 3.4 includes two changes over the previous version, occurring in three separate lines. The first change refers to a "return type change" in line 2, while the second one is a "method implementation change" in line 4 and 6. For illustration purposes, let us assume that it takes 8 minutes for the programmer to implement both modifications together.

Based on these values, the modification effort can be calculated by taking the ratio of the sum of the modification and the net development time:

$$\frac{1 \text{ return type ch.} + 1 \text{ method imp. ch.}}{8 \text{ min}} = 0.25$$

Notice that it is different from the naive method, which only counts the changed lines. We chose to use the modification effort because, during the implementation, developers

consider methods and classes as logical units and not individual lines of source code.

3.1.3 Determining the Weights

During the next phase of the research, we defined two variations of the original prediction model. These only differ in the weights of the modification groups set to calculate the MEff metric. In the case of the base (or initial) model, these parameters are preset and do not change. However, in the case of the enhanced version, we use a genetic algorithm to fine-tune the weights of the modification. The initial weight-vector was set by our developer experience. We assumed that the genetic algorithm should converge to the suitable weights, which should provide a more accurate estimation [31].

The individuals were identified by their chromosome, which is a vector over the real numbers with the same dimension. In the model, each chromosome represents a weight-vector, and every element determines the weight of a single modification group.

The fitness value is calculated for each individual by evaluating the prediction model with the weights defined in that particular individual. The final goal of the GA is to improve the precision of the model. The prediction model in previous experiments [93] was not enhanced with the GA, but it was evaluated using the F-measure metric. Thus the F-measure was chosen to be the fitness value of the GA.

We used the later described mutation operator to produce an initial population. This operation was repeatedly applied to the weight vector of the base model to create the appropriate amount of random elements.

Evolution starts with the breeding phase. The GA selects the two best entities with its fitness value. The crossover operator will only apply to this pair. Every call of the crossover operator produces exactly one offspring. The algorithm repeats the operation to produce more than one child.

We used a uniform crossover logic. During the crossover, the algorithm iterates via the elements of the chromosome (vector) and randomly chooses an element from one of the two parents. Every element has the same chance to be copied into the child's chromosome [88].

The chromosomes of the children are subject to mutation. During the mutation, some elements (weights) of the chromosome change. A lower limit and an upper limit were preset for the weights of the groups. The algorithm gets half of the distance between the limits and the currently selected weight and sets the current value either to the lower or to the upper half point. This way, the two limits are never exceeded. Then, the mutated child is inserted into the population.

The individuals with the worst fitness value are killed (removed from the population) to maintain the size of the population. This way, the current evolution step is completed, and the algorithm proceeds to the next generation [64, 64].

The above mentioned GA parameters and their values are shown in table A.1 located in the appendix.

	Project1	Project2	Project3	Project4
base model	59,8000%	47,0000%	44,2000%	45,3000%
enhanced model	75,2630%	66,6425%	67,4835%	60,2791%

Table 3.2: Fitness values of prediction per project

3.1.4 Evaluation

As shown in fig. 3.4 and table 3.2, the fitness value (F-measure) of the prediction grows in every case. The average grows about 18 percent (table 3.3). It is also relevant that in the worst case, our model proves to be better by about 16 percent.

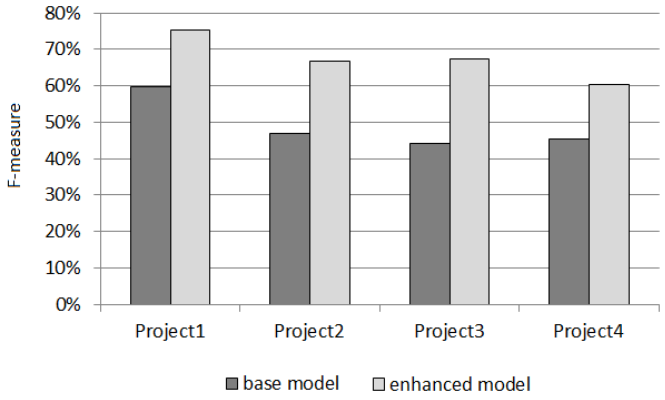


Figure 3.4: Fitness values of prediction per project

	worst	best	average	median
base experiment	44,2000%	59,8000%	49,0750%	46,1500%
enhanced model	60,2791%	75,2630%	67,4170%	67,0630%
difference	16,0791%	15,4630%	18,3420%	20,9130%

Table 3.3: Comparison of models

These data support the sub-thesis points 1.1 and 1.2, since our model gives a better estimation from the beginning of the evolution and the population average fitness value is higher in every generation, and GA can further improve its precision.

3.2 Estimation and Reduction of Superfluous Effort

During this phase, we utilized the previously introduced MEFF metrics to measure productivity and analyze its dynamics [22]. To characterize the relationship between the small and large scale productivity changes, we defined a modified version of the Micro-Productivity Profile(MPP) published by Tóth G. et al. [94].

We investigated the development of a medium-sized web application. The development of the application was carried out iteratively with some agile elements, so the project managers wanted to see the effects of the changing requirements to the productivity of the development in some measurable way to refine the further iterations of the project. The technical leaders of the project were interested in the productivity of different application layers to see the sensitivity of the layers related to changes in the application. The analyses conducted during the experiment are related to challenge 3.

3.2.1 Subject System

Our subject system is based on the Java Enterprise Edition and the Seam 2.3 platforms, and it contains approximately 2200 classes and around 119k logical lines of code. The application is a part of a home security system developed by AENSys Informatics Ltd., which is responsible for the management of various security sensors installed at the end user's apartment, and handling security alerts sent by the sensors. The architecture of the system is divided into the following five layers.

User interface layer it contains the implementation of composite user interface components and general, complex operations related to the user interface.

Business logic layer is responsible for the management of complex business processes and transactions. This layer establishes a connection between the persistence layer and the user interface.

Integration layer is responsible for communication with external systems and sensors.

Utility classes this layer provides general, common functionality used by many other components and layers.

Persistence layer it contains the entities to be managed in the system and the high-level implementation of database operations related to the entities.

3.2.2 Measured Development Phases

We investigated seven months (from 3 April 2013 to 7 November 2013) in the early stage of the development. This period consisted of three main development phases.

Phase 1 (customer UI) development of user interfaces for customer users. It ended with Milestone 1 on 3 June 2013.

Phase 2 (provider UI) development of user interfaces for service provider users. It ended with Milestone 2 on 1 September 2013.

Phase 3 (Release) development tasks related to the preparation for the first release of the application.

During the investigated period, 17 developers worked on the project: 8 developers with at least four years of development experience, five developers with 2-3 years of experience, and four junior developers with less than two years of experience. All developers committed their work to the SVN version control system at least once a day, therefore, approximately

2200 revisions were created by the developers.

Figure 3.5 shows an overview of the measured properties of the project. Productivity data were collected from all three phases. We identified the endpoints of each phase by an SVN revision. Unfortunately, we had to ignore the last one (labeled as the first release). Some developers did not use the productivity measurement tool properly, so too few events were collected from their work. Most of the data loss occurred in the third phase, which meant that we could not collect enough productivity data to analyze that phase properly.

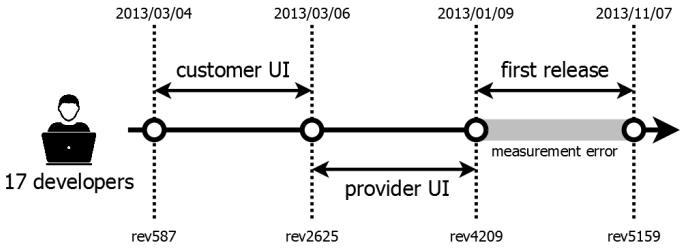


Figure 3.5: Overview of the history of the measured project

3.2.3 Productivity Measurement Process

Our productivity measurement method relies on development data, including various developer actions in IDE, file modifications, and time logs. In order to accumulate important project information, detailed traces are logged in the IDE. Figure 3.6 depicts our productivity measurement process. At the beginning of the development process, the project manager defines the tasks of the project on the productivity data collector server. The developers work with the Eclipse IDE with the productivity plug-in included, which monitors the detected activities and uploads the collected events and data to the server. The developers commit their source code modifications to the SVN version control server. An internally developed *productivity data analysis toolkit* processes and analyzes the collected events, and calculates the real development time for files in the project. A source code analyzer toolkit analyzes the source code of revisions and compares them to each other to find modifications between them. By using these two data sets, productivity information can be calculated for the project.

During the experiment, developers used the productivity measurement plug-in [7], which monitored the following types of events and characteristics of the development.

File events: opening, closing, creating, deleting, saving, switching.

Project events: creating, deleting, opening, closing.

Events related to the user interface: editors, views, perspectives, dialogs, windows, etc. in the IDE.

Code execution events: starting, stopping, debugging, profiling.

Code editing events: cut, copy, paste.

Keystroke and shortcut events from the keyboard.

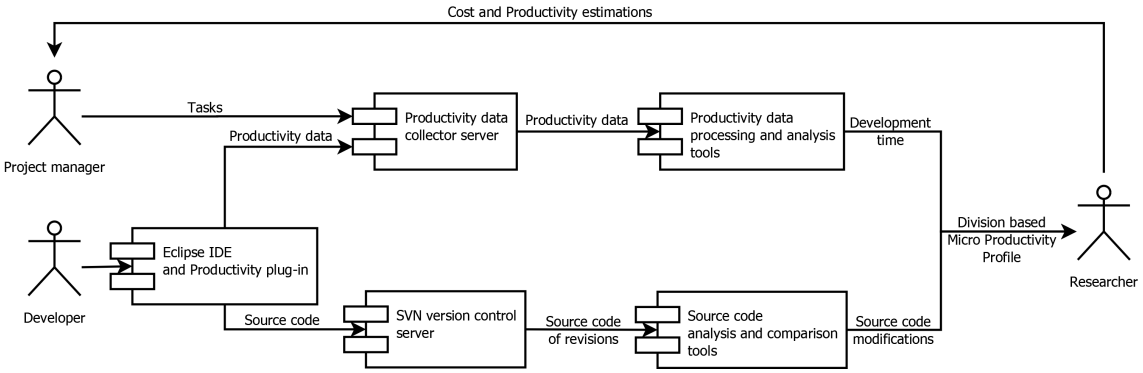


Figure 3.6: Measurement architecture

Detecting idle time intervals and interruptions. After a predefined time limit is exceeded without any interactions with the IDE, an idle time interval is detected, and a special file event is raised, indicating that the opened file is left unchanged by the developer. After another interaction is performed with the IDE, the developer can select whether he/she worked on that file or not.

The actual task of developer. The plug-in can download a predefined list of tasks for the project, and the developer can select his/her actual task and switch between tasks.

Save event. Every time a Java source file is saved, the structure of the source file and some code quality metrics are logged.

The collected productivity data can be used to calculate the net development time of files in the project, by iterating over the file events for each developer in the ascending order of event timestamps.

During our experiments, we collected the modifications groups defined in section 3.1. In this study, we did not add weights to modification types to reduce the bias of inaccurate weight vector assigned as determining proper weights requires further project-specific research.

3.2.4 Applying Micro-Productivity Profile

The central concept during this phase of the research is the Division based Micro-Productivity Profile (MPPD for short), which measures the frequency of changes in productivity at various granularity levels. To understand the basic concept, consider the following scenario. Let us suppose we can measure the productivity of the developer, i.e. the ratio of produced output, and required effort. The measured productivity depends on the sample size: productivity measured on the whole development considers only the final program code, while measurements on weekly samples consider thrown away program code as well. Thus, repeating productivity measurement with various sample sizes lets us estimate the wasted effort, i.e., where developers modified the same code again. Informally, plotting

these numbers as a curve is what we call productivity profile.

Figure 3.7 shows the history of the source code, with its revisions. We used a division based approach instead of the approach with the gradually growing sampling window [94]. The figure illustrates both sampling methods: our division based method (at the bottom) and the related window-based one (on the top). The window-based method uses windows with various sizes to swipe along with history. This lets them capture the wasted effort independently from the frequency of the commits, but there is always a part of the history which can not be measured, due the window extending over the last revision.

To measure the neglected parts of history, we introduced another technique for sampling the changes. To calculate the initial value of the MPPD for the whole history (zero division points), we compared the first and last revisions of the system, i.e., the range is divided into one single part (P(0,0)) with no intermediate points. The modifications were aggregated into the MEFF metric. After that, the algorithm moves on to the next step, when we take one division point in the middle of the range. It divides the history into two parts, P(0,1) and P(1,1). In this iteration, we compare each division point with the subsequent ones – i.e. rev0-rev3, rev3-rev6 – and compute the Modification Effort for these pairs. The value of the MPPD is the sum of these values. As we continue with two, three, four, or more divisions, the range will be cut into three, four, five, or more parts, and the productivity will be the sum of more and more parts. Notice that this method depends on the frequency of the revisions; however, in this particular case, the distribution of the commits allows us to use it without any serious side effects. A formal definition of MPPD can be found in appendix A.4.

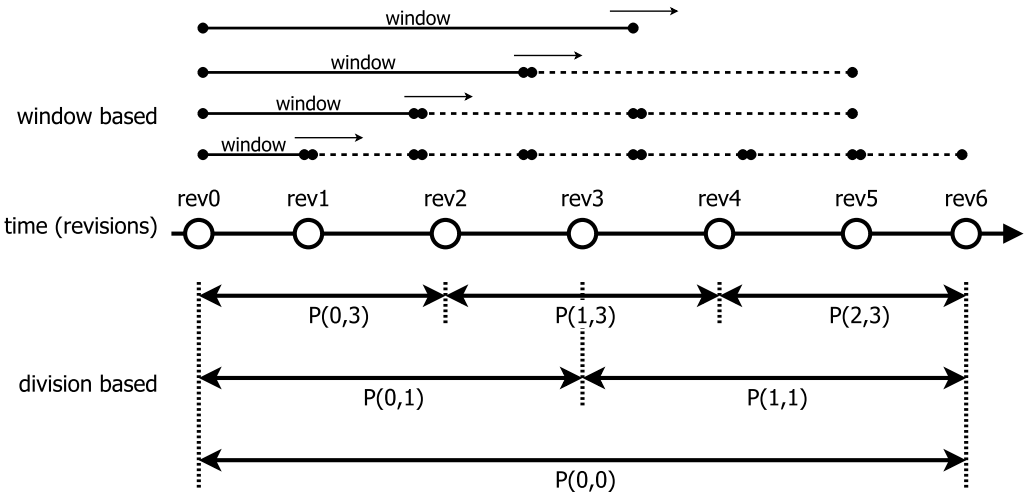


Figure 3.7: Equal division of revisions

A resulting curve (fig. 3.8) shows the superfluous effort spent by developers during the implementation. In an ideal case, these would be zero, and the MPPD would be a flat line. In real life, these values are affected by incomplete specifications and requirements, which are changing over time. For example, if the customer changes the plans of the user

interfaces, the developer has to modify the parts of the code that are already written. The first version is irrelevant compared to the final release, hence the modifications in the first set were unnecessary, and the MPPD will increase. The steepness of the MPPD curve can be interpreted as the ratio at which the developers re-modify the same code again. Using these profiles instead of the naive approach where only the most fine- and coarse-grained divisions were compared, shows not just the amount but the distribution (the frequency) of the wasted effort. This reveals some aspect of the developers' practices.

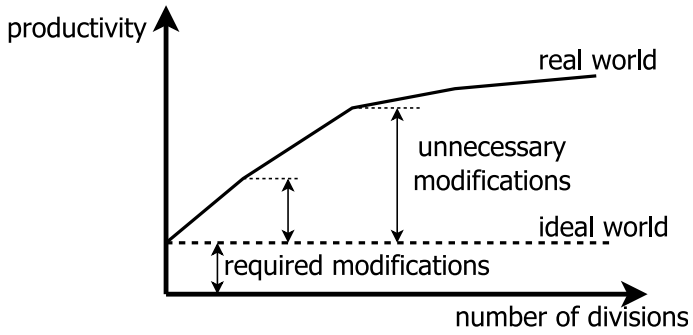


Figure 3.8: The underlying concepts of MPPD

While fig. 3.8 illustrates the underlying concept of MPPD, fig. 3.9 shows a concrete example of the curve itself. The measured productivity values are represented in the right, vertical axis. As previously stated, these values increase for a higher number of divisions. There are nine distinct points each for every sum of equal distance division parts.

Figure 3.9 shows a concrete curve based on repository commits in the subject project. Besides the MPPD curve itself, the figure compares the time-based and revision-based division of the project history. On the bar-chart, at the top, we displayed the average number of SVN additions, deletions, and modifications. At the bottom part, one can inspect the median and the average elapsed time between the division points. Both the number of SVN changes and elapsed times approximate a hyperbolic function as it is expected from a gradually increasing division. These facts confirm that our revision based approach provides approximately equal divisions as dividing the development phase based on elapsed time.

3.2.5 Evaluation

Using the measurement architecture presented above, we monitored the development activities of the developers in the presented project, and examined the productivity data of the developer team using the MPPD profiles produced by our analysis tool-chain. We calculated MPPD profiles for the following examination aspects: comparison of profiles of different development phases and different application layers, examining profiles of the developer team during the whole 7-month period of the project. We present our findings in the following sections. These results support sub-thesis point 1.3.

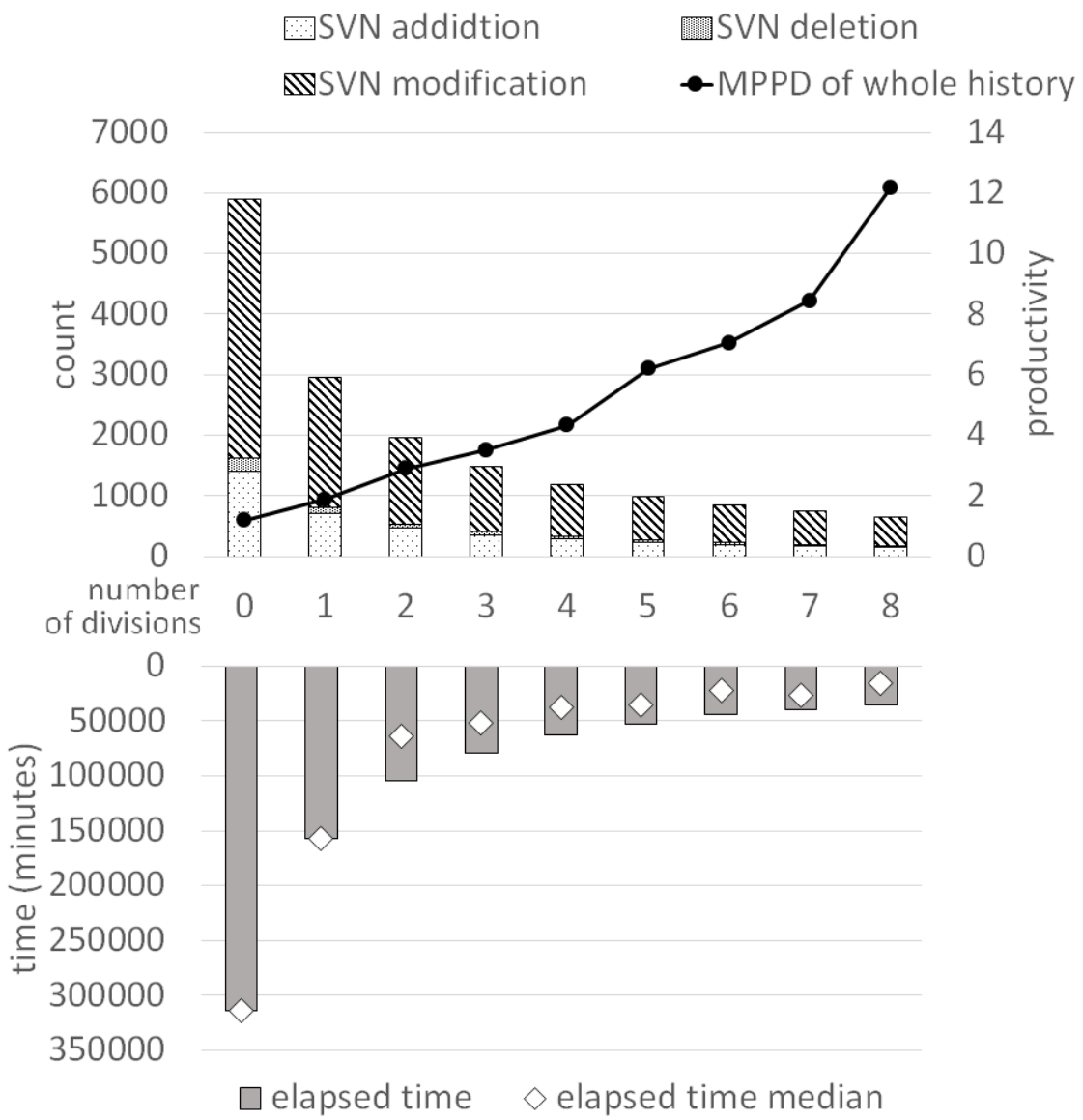


Figure 3.9: Overview of MPPD over the whole history and its statistics

The MPPD curves provide details about subtle productivity changes over time. To assign precise meaning to the shape of these curves requires further analysis, but some practical suggestions can be concluded already. These hints concern mainly the development process and provide help for the managers. For example, the shape of the MPPD can be used to plan the time of various activities during the project, like code reviews and milestones.

Productivity over Development Phases

We investigated productivity over two phases of the development. During these phases, two main components were implemented: the customer user interface in the first, and the

provider user interface in the second. The MPPD-s calculated for different development phases are shown in Figure A.2. The collected productivity data and the calculated MPPD curves show different shapes. The developers create more modifications during the implementation of provider UI hence it has higher MPPD values. Furthermore, there is a slight increase in its steepness which denotes that there are more unnecessary modifications (i.e. possibly wasted effort) during this phase than the previous one.

These differences can be explained by the fact that there was a more rigid specification for the customer UI than the provider UI, as reported by the project manager. This means that the developers of provider UI had to discover the possibilities considering the technical details of the implementation. By doing this, they produce more code and change more components. They also need to adapt the existing solutions to the new requirements, which results in more rewritten parts of the source code and more unnecessary modifications. In this particular case, it means that managers should rearrange their resources and provide a more detailed specification for the provider UI. The slightly higher steepness of this curve shows a manageable amount of wasted effort, but we suggest that it should address to prevent further growth.

Productivity over Application Layers

Figure A.3 shows the MPPD-s for the development productivity of the developer team related to the five layers of the application. The MPPD of the utility layer has very high steepness; therefore, differences between MPPD-s of the other four layers are not clearly visible. For this purpose, Figure A.4 shows their differences without the utility layer.

The higher productivity values near the right-hand side of the curve and steepness in the MPPD of the utility layer can be explained by the fact that this layer has to provide the most reusable solutions for the most general problems. Its components should be easily usable from any of the other layers; therefore, the requirements related to the interface of this layer changes very often. This may result in frequent modifications in the source code of the layer; in addition, many changes do not appear in the final revision of the application. The developers also verified that most of the unnecessary modifications were related to utility classes. However, we do not suggest that these modifications are strictly wasted effort, and developers should stop writing utility classes. However, they have to be aware of the nature of this layer and try to reduce the amount of rewritten code. It can be achieved by careful planning of the common functionalities and inspecting the feature specification of other layers.

The user interface layer in this context contains only the Java implementation of general, composite components and operations used by the web pages of the application. This layer also has to provide general solutions for different types of pages, and the developers also stated that several components in this layer needed many modifications to follow the changing requirements. This fact explains that the MPPD of the user interface layer has the second-highest steepness.

Some slight increase can be observed in the MPPD of the business logic layer, which can be originated from the changing requirements of the service provider related functions. The MPPD curves of the other two layers are quite flat, which can be verified by the fact that the persistence layer has been well designed, and the integration layer depended only on the fixed interfaces of the external systems and sensors to be integrated. Therefore these layers did not need a significant number of modifications after the implementation.

3.3 Comparison of Software Quality in the Work of Children and Professional Developers

As stated earlier, productivity can be influenced by several factors, one of them is the developer's level of expertise. Both the practical and theoretical knowledge are gathered (among others) during the time spent in some educational institute, like schools and universities.

Our opinion is that there is a growing need for a well-defined model that can evaluate the performance of students in such a way that it is acceptable for the industrial sector as well. In this phase of our research, we seek for similarities or contrasts between the implementations of students and experts, which can be measured objectively. During the research, a software quality model was used to evaluate the high-level properties of the solutions.

We used data gathered on special classroom sessions to compare the software quality of students and experts. The sessions took place in a single afternoon in three distinct parts, each with the duration of one and a half hour. The high-school student used the provided development environment in groups of two or three. After they had finished their tasks, we collected all solutions and analyzed them with an automated software quality model.

3.3.1 Original and reduced quality model

We use a modified version of the source code quality model implemented by FrontEndART Ltd. It is an IT company located in the southern region of Hungary. It is a medium-sized company that specializes in developing and implementing software quality measurement models as well as using them to asses software quality for various customers, like monetary bodies, for example.

The original model, which we used, is based on the research at the University of Szeged, conforms to the ISO/IEC 25010 standard and is capable of qualifying the source code of a software system. Figure 3.10 shows the original quality model. The computation of the ISO/IEC 25010 high-level quality characteristics, together with the maintainability of the system, is based on a directed acyclic graph whose nodes correspond to quality properties that can be either internal (low-level) or external (high-level). Internal quality

properties characterize the software product from an internal (developer) view and are usually estimated by using source code metrics. External quality properties characterize the software product from an external (end-user) view and are usually aggregated somehow by using internal and other external quality properties. The nodes representing internal quality properties are called sensor nodes as they measure internal quality directly. The other nodes are called aggregate nodes as they acquire their measures through aggregation. The edges of the graph represent dependencies between an internal and an external or two external properties. [12]

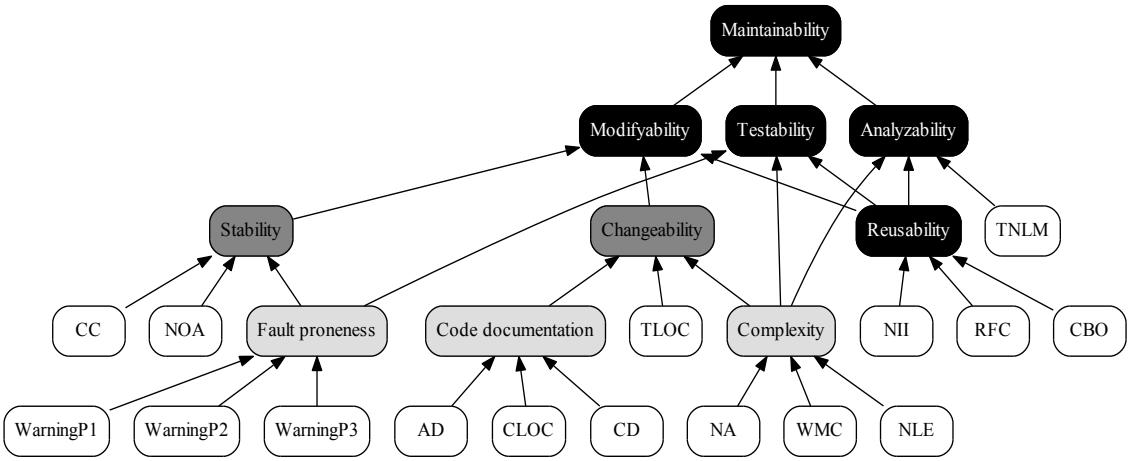


Figure 3.10: Original quality model

We modified the previously described model and used this reduced model in our experiment. During the modification we only deleted existing nodes. In particular, reusability, documentation, and all of their child nodes were removed to create a reduced quality model, which provides better measurement capabilities. In general, these nodes were deleted, because their value was irrelevant, in the context of the classroom exercises analysed in the following sections. We applied the following changes.

Number of Incoming Invocations (NII) *Reason of exclusion:* The children do not use inter-class calls and rarely use inter-method calls.

Response set For Class (RFC) *Reason of exclusion:* The students only need to implement methods in the same class and do not modify other classes.

Coupling Between Object classes (CBO) *Reason of exclusion:* While solving the tasks, only the provided API classes were used.

API Documentation (AD) *Reason of exclusion:* Students do not write any comments or documentation.

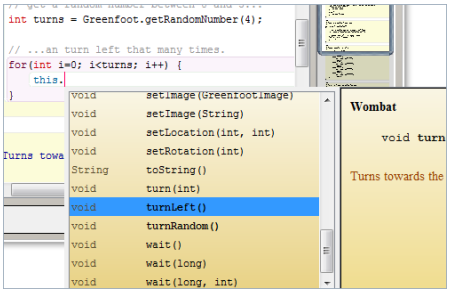
Comment Lines of Code (CLOC) *Reason of exclusion:* Students do not write any comments or documentation.

Comment Density (CD) *Reason of exclusion:* Students do not write any comments or documentation.

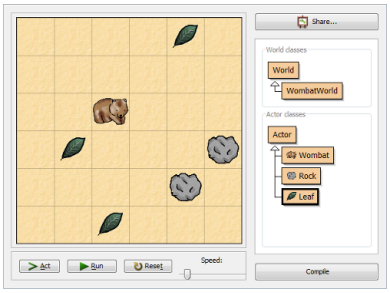
3.3.2 Development environment

We use a special integrated development environment called Greenfoot. Greenfoot is a project in the Programming Education Tools Group, part of the Computing Education Research Group at the School of Computing, University of Kent in Canterbury, UK. [48]

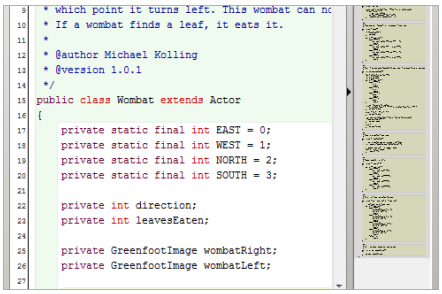
Its main goal is to provide a simple and easy to use user interface for students to acquire necessary programming skills. The users can interact with various elements of an object-oriented program via an intuitive and straightforward user interface. The interface is a full IDE that includes project management, auto-completion, syntax highlighting, and other tools common to most IDEs. A couple of these features are shown in fig. 3.11a.



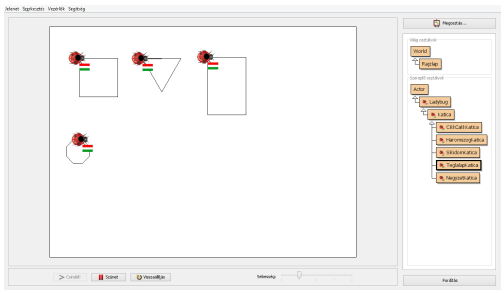
(a) Greenfoot integrated development environment features



(b) Greenfoot main window



(c) Greenfoot source code editor



(d) A solution made by an expert

Figure 3.11: Greenfoot integrated development environment for students

The graphical elements do not hide the underlying source code, so the users have to use the mouse, which is more natural for young children, and the keyboard together to accomplish their tasks. Its main concepts are *actors* who live in *worlds* to build games, simulations, and other graphical programs.

The main window is shown by fig. 3.11b. On the left side you can see the visual representation of a world object that acts as a canvas or scene for the whole project. The classes and their relations are shown on the right.

The creators also provide a basic class library for Greenfoot, which is highly customizable by the teachers to their needs. The objects are programmed in standard textual Java code, providing a combination of programming experience in a traditional text-based language (fig. 3.11c.) and visual execution.

3.3.3 Implemented classroom exercises

In Hungary, teachers tend to use the *Logo programming language* in primary and secondary schools. Logo is an educational programming language designed in 1967 by Daniel G. Bobrow, Wally Feurzeig, Seymour Papert and Cynthia Solomon. Today, the language is mainly remembered for its use of *turtle graphics,* in which commands for movement and drawing produced line graphics either on the screen or with a small robot called a turtle. The language was originally conceived to teach concepts of programming related to LISP and later to enable what Papert called body-syntonic reasoning where students could understand (predict and reason about) the turtle's motion by imagining what they would do if they were the turtle. There are substantial differences between the many dialects of Logo, and the situation is confused by the regular appearance of turtle graphics programs that mistakenly call themselves Logo. From the many implementations and IDE-s our teachers use IMAGINE.

To ease the transition from a toy language (Logo) to a programming language used in the real-world (Java), we reimplemented the underlying logic of turtle graphics in Java. We integrated it into the Greenfoot development environment. With this API, students could create new Java classes to control the turtle and draw some simple graphics.

Two base classes were provided, namely `Ladybug` and `Katica.` The latter is a subclass of the first, it wraps the original English instruction with their Hungarian equivalent.[2] `Ladybug` implements the following methods and properties.

`turn()` Turns the ladybug.
`moveTo()` Moves the ladybug to the given position.
`move()` Moves the ladybug to the given distance.
`penUp()` Takes the pen up.
`penDown()` Puts the pen down.
`setColor()` Sets the color of the pen.
`setLocation()` Sets the location of the ladybug.

The students and the experts accomplished the same classroom exercises. Each task implemented in a unique class inherited either the `Ladybug` or the `Katica` classes. The following four exercises were solved and their results were analysed.

1. Create a ladybug that can draw a square with a specific size.
2. Create a ladybug that can draw a rectangle with a specific size.
3. Create a ladybug that can draw a triangle with a specific size.
4. Create a ladybug that can draw a polygon with the given number and length of sides.

Figure 3.11d shows a solution made by an expert developer. In this example, each ladybug was placed on the world, and the program was already executed.

[2]We renamed turtles to ladybugs, because the original word *'teknős'* contains a diacritics on the second-to-last letter, while *'katica'* does not.

3.3.4 Evaluation

After each student and expert had solved the above mentioned tasks, we measured the quality of their source code with the previously introduced modified model. We collected altogether 37 solutions from the students and from 3 experts. During the initialization of the measurement phase, all solutions which did not solve the task, i.e. they did not contain any source code, were eliminated. We measure the quality of the code of each user as if it was a separate system. These results were aggregated into the four well known descriptive metrics: minimum, maximum, average, and the median of a given quality or source code metric.

Comparison of high-level metrics

We measured the following high level software quality metrics.

Maintainability Maintenance cost of the software system due to its source code

Testability Resources needed to test and verify the modifications made in the software

Fault proneness The probability that a failure occurs during the operation of the system

Complexity The general complexity of the software source code

Modifiability Risk of altering the source code without causing side effects

Stability Probability of operational failures caused by modifications of the software

Comprehensibility How difficult it is to understand the source code

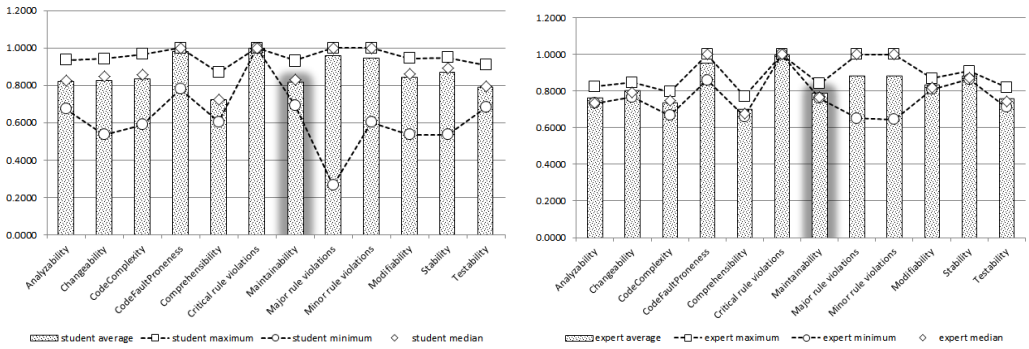
Changeability Resources needed to alter the behavior of the software

Analyzability Expected cost of detecting faults and their causes during operation

Minor rule violations Minor issues in the code that, e.g., decrease the code readability.

Major rule violations Major issues in the code that can cause, e.g., performance issues.

Critical rule violations Critical issues in the code that can cause bugs and unintended behavior.



(a) Code quality metrics of students

(b) Code quality metrics of experts

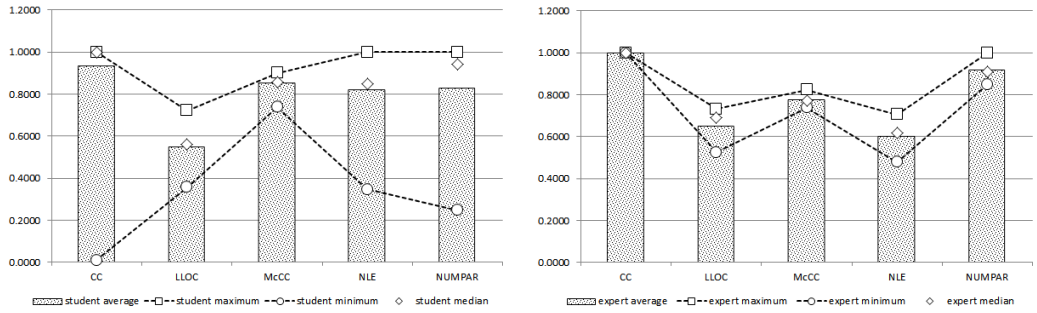
Figure 3.12: High level code quality metrics

Figure 3.12b and Figure 3.12a show the aggregations of the high level metrics. The average value was represented on the bar-charts. The minimum, the maximum and the

median were also displayed as points. The metric of top-level – called maintainability – was highlighted to emphasize the fact that it aggregates all other metrics.

We cannot find any significant differences in the average performance of developers and students. But the ranges of code quality are much more extensive in the case of students than in the case of experts.[3]

Comparison of low-level metrics



(a) Code metrics of students

(b) Code metrics of experts

Figure 3.13: Low level code quality metrics

The following low level metrics were measured and evaluated.

CC The real value between 0 and 1 expresses which amount of the item is covered by code duplication.

LLOC The metric counts all non-empty, non-comment lines. Lines of nested classes or packages are not counted.

McCC The number of decisions within the specified method plus 1, where each if, for, while, do...while and ?: (conditional operator) counts once, each N-way (switch) counts N+1, and each try block with N catch counts N+1.

NLE NLE for a method is the maximum of the control structure depth. Only if, switch, for, foreach, while and do...while instructions are taken into account but if...else if does not increase the value. NLE for a class is the maximum of the NLE values of its methods.

NUMPAR The number of parameters of a method (the ellipsis is counted as one parameter).

Low-level metrics follow the same pattern, as seen with high-level metrics. There are not any significant differences between the average and median values of students and developers, but the performance of the first ones tends to fluctuate more wildly.

[3]To emphasize these differences; we connected the lower and upper limits with dashed lines. However, these lines solely added to make the previously noted differences more easy to see. The order of these high-level metrics is irrelevant, and these data points represent discrete values.

3.4 Contributions

The research presented in this chapter is divided into three phases, according to the central issues they addressed.

3.4.1 Defining Weighted Modification-based Productivity Measure

In this phase I strove to define a more expressive productivity metric for software development, which could be used to increase the accuracy of cost prediction models. I used a compound metric for expressing the modification effort, which was the aggregation of different kinds of modifications like creation, deletion, and type change. To express the effort across modification types, I used different parameters (weights) for the various kinds of modifications. The choice of these parameters was crucial for the accuracy of prediction. In this phase, we reported on my early experiences in applying search-based methods to determine these parameters (a basic genetic and evolutionary algorithm (GA) was used for this purpose). A typical improvement of 20 percentage points was achieved in the combined prediction accuracy (F-measure) when comparing the model with initial parameters to the one obtained after running the search-based method.

To achieve the mentioned goals I implemented a framework which is capable of collecting and aggregating product and process metrics from various sources including the source code and the integrated development environment. The framework detects the modifications between revisions, and tries to predict the effort of further changes.

3.4.2 Estimation and Reduction of Superfluous Effort

During this phase, we utilize Micro-Productivity Profile to characterize low and high grain productivity changes. We applied this method in an empirical experiment during the development of a middle-sized J2EE project to aid the project management with detailed productivity information. MPP was used to identify the amount of wasted effort related to various entities of the project, like developer teams and application layers. I characterized these entities to aid the project manager in decision making related to cost estimation. Especially, I used the collected data to help the project managers and lead developers to understand the rhythm of the project better and help them plan meeting sessions.

The framework that we used to measure the project was composed of several parts. However, I aided the development, and the integration of these my main task was to implement the productivity calculation and MPPD constructing subsystems.

3.4.3 Comparison of Software Quality in the Work of Children and Professional Developers

In this phase, I used a simplified version of the quality model based on the researches at the University of Szeged that conforms to the ISO/IEC 25010 standard and is capable of

qualifying the source code of a software system to measure and compare the quality of source code created by students and experts. The subjects of my analysis were distinct solutions for predefined classroom exercises. The results suggest that there are not any significant differences between the average performance of the two groups. These similarities can be explained by the fact that students were guided by an expert i.e. the teacher.

On the other hand, the quality of source code produced by experts has less fluctuation. They tend to provide a more stable performance. Outliers can be found in either direction from the average or median among the solutions of the students. I suggested that these represent the children who have more or less affinity for abstract thinking and logical problem-solving.

3.5 Advantages and Disadvantages of These Methods

In this section, we elaborate on the threats that could invalidate our results. We also mention several potential use cases for our foundings.

3.5.1 Internal Validity

Based on the experiences of the developers participating in this research, we can conclude that the causal relationships being tested during the experiments are reliable but may be influenced by other factors or variables.

Construct Validity

The usage of a well known and accepted approach to measuring productivity ensures the soundness of our high-level theoretical constructs. However, the usefulness and validity of these measures are highly dependent on the underlying metrics used to capture input and output.

The used method is highly sensitive to the predefined modification types and their weights. The current measurement used the trivial unit weight function; however, this may blur some aspect of the development process (e.g., addition is more complex than deletion). The modification detection component of the model was designed for object-oriented languages; hence it can not be applied in the case of systems with other paradigms. However, we believe that with necessary modifications, the concept can be easily adapted to other paradigms as well.

Despite the careful design, interaction-based measurements, in general, require additional effort compared to solutions solely based on version control systems. On the other hand, the ease of information extraction may be a trap in this latter case. Rough approximations in the base data – like time estimation based on commit timestamps – may provide uncertain results. Interaction data contains more accurate and detailed information. The plug-in collects per user and per-task data as well, which enables a low-level

evaluation of the work in progress. This is invaluable when the stakeholders aim to make evidence-based decisions.

Another internal threat is the sensitivity of the MPPD curves to the homogeneity of measurement points in time. To eliminate this dependency, we plan to introduce a new sampling algorithm over the history of the source code.

Criterion Validity

There are several approaches to measure the unnecessary work of developers. The most simple and naive methods use some kind of historical data about the development to calculate the differences between the number of changes. For example, one can measure the number of changes of code in every single step of the implementation, then subtract the number of changes between the first and last state of the system. The underlying concepts of these types of algorithms are independent of the method of change detection. However, in practice, the precision of these methods highly depends on the unit of the measurement.

While these approaches can capture the total amount of unnecessary work, they fail to give any insights about the processes that generate these superfluous changes. Those methods that can give useful help for the participants to improve the processes are more successful in practice. The MPPD curves provide details about subtle productivity changes over time. To assign precise meaning to the shape of these curves requires further analysis, but some practical suggestions can be concluded already. These hints concern mainly the development process and provide help for the managers. For example, the shape of the MPPD can be used to plan the time of various activities during the project, like code reviews and milestones (see discussion below).

Another difference between the two approaches is that the naive diff-based concept gives a very inaccurate approximation for the development time of the changes. The time elapsed between two commits of the same developer is necessarily much higher than the real development time of the changes by the developer. For example, the intervals collected from logs of version control systems often contain parts that are not related to working time (nights, weekends, holidays, etc.). These can be approximated by the daily working time of developers. However, there are further problems caused by other parts of the working time, which are not related to the implementation of the software: meetings, activities related to documentation or time spent on other parallel projects, etc. Our approach collects events related to interactions with the IDE to give a more accurate approximation for the real development time of the changes in the software.

3.5.2 External Validity

The presented Micro-Productivity Profile related experiment is conducted on a single project; thus, it is appropriate for introducing the advantages and usefulness of the pro-

posed method, and not for modeling productivity or drawing general conclusions on productivity factors for other systems. Likewise, the measurement strongly relied on the fact that the investigated application was developed in Java programming language with Eclipse IDE.

The success of the productivity measurement depends on the active and proper use of our tools by the developers. The amount of extra effort due to measurements is critical. Although the task information is only a small plus, we experienced that some programmers did not use the plug-in properly, which caused a significant data loss in the third development phase. Based on the analysis of the experiences of the team after the project, the reason for improper use of the tool was not the high amount of extra effort, but insufficient motivation by the management and also some technical issues.

Based on the analysis of factors affecting productivity, we conclude that the comparison of students and professional developers' non-functional source code metrics suggests some exciting ideas. However, we are aware that this is just a stepping stone for further research.

Chapter 4

Providing Immersive Methods for Software and (Unit) Test Visualization

The main topic of this chapter is the visualization of software systems and their connected items. The structure of the related thesis point and its connections to various stakeholders and topics are shown in fig. 4.1.

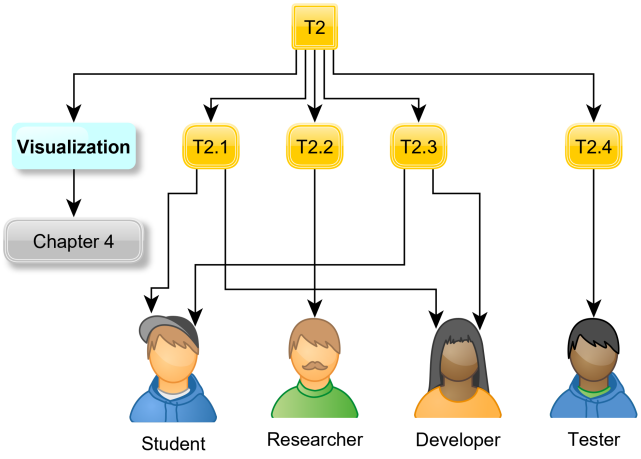


Figure 4.1: Structure of thesis points

The rapid developments in computer technology have made it possible to handle a large amount of data. New algorithms have been invented to process data, and new ways have emerged to store their results. However, the final recipients of these are still the users themselves, so we have to present the information in such a way that human beings can easily understand it. One of the many possibilities is to express that data in a graphical form. This conversion is called visualization.

The importance of visualization techniques is undeniable. Diagrams, charts, and other

graphical elements are often used to present quantitative and qualitative properties and their relations. These tools use simple and abstract graphical primitives that could not be found in the real world like straight lines, points, and circles. They can express some attributes of the software successfully, but are less useful in presenting more complex many-dimensional contexts. To address this issue, in this chapter, we introduce our novel software visualization tool and its related research. It utilizes an enhanced version of the city metaphor, which provides higher expressive power by allowing the user to display several abstract concepts simultaneously.

Data visualization with high expressive power plays an important role in several software development-related activities. Recent visualization tools try to fulfill the expectations of the users by using various analogies. For example, in a city metaphor, each class is represented by a building. Buildings are grouped into districts according to the structure of the namespaces. We think that these unique ways of code representation have great potential. However, in our opinion, they use very simple graphical techniques (shapes, figures, low resolution) to visualize the structure of the source code.

On the other hand, computer games use high-quality graphics and have a good expressive power. A good example is Minecraft, a popular sand-box or role-playing game with high extensibility and interactivity from another (third party) software. It supports both the high definition, photo-realistic textures, and long-range 3D scene displaying.

Furthermore, software systems could reach virtually infinite complexity by their nature. In theory, there is no limit of control flow embedding, or the number of methods, attributes, and other source code elements. In practice, these are only bound to computational power, time, and storage capacities. To comprehend these systems, developers have to construct a detailed mental image. These images are gradually built during the implementation of the system. Often, these mental images are realized as physical graphics with the aid of data visualization software. For example, different kinds of charts are used to emphasize the difference among various measurable quantities of the source code or UML diagrams, which can visualize complex relations and connections among various entities in the system. However, there are certain situations when developers do not have sufficient time to construct this mental landscape, for example, when they are not present during the early stages of the software's life cycle. The issues and solutions discussed in this chapter are related to challenge 1.

4.1 Enhancing the City Metaphor with Game-based Visualization

To address these issues while taking into account the considerations as mentioned earlier, we enhanced the already known city metaphor, by connecting data visualization with high end-user graphics capabilities. To achieve this, a visualization tool was implemented. It processes structured data related to the source code (for example, product metrics) as

input and generates a Minecraft [2] world with buildings, districts, and gardens to provide a detailed representation of the mental landscape populated with abstract concepts and their underlying connections. The tool is called CodeMetropolis. We used it to investigate the possibilities of this kind of data visualization. Works, detailed in this section and summarized in sub-thesis point 2.1, provide a solid base for further investigation.

CodeMetropolis is a set of the command line and GUI tools written in Java. It can generate a playable Minecraft world, which represents the properties of the original data set. The current version supports SourceMeter [61] or SonarQube server [76] directly as a data source, but end-users are encouraged to integrate other output types. The generated world uses the city metaphor, which means that the source code metrics are represented with the various properties of the different kinds of buildings. For example, Figure 4.2 shows an example world, which represents a small Java program.

Figure 4.2: JUnit project visualized by CodeMetropolis

We used two main levels to represent data and entities of our visualization process. On the data level, each item has its own property set – for example, metrics. These data are displayed on the metaphor level. All buildings in the metropolis belong to this level. The buildings and the world (city) itself has a couple of attributes which control its visual appearance. The properties are mapped to the attributes in order to visualize the data with a sophisticated mapping language.

As a visualization tool, CodeMetropolis executes the common steps of constructing such a graphical representation.

4.1.1 The Embodiment of Visualization's Phases in CodeMetropolis

We used the 4-step division of visualization introduced by Upson, Craig et al. [98]. These steps, which are common for each visualization processes, are *filtering* the data, *mapping*

it to the appropriate graphical items, *rendering* the image and finally *displaying* it to the user.

Filtering Step - Converters

In the case of CodeMetropolis, we provide several converters that can produce a unified XML-based input format. Currently, we are directly supporting SourceMeter and SonarQube as a data source, but users are encouraged to write their converter using the shared libraries of CodeMetropolis. For a detailed definition of input format, please check the official repository of the project, here we are only describing a common use case.

There are two mandatory properties for each data entry: their unique name and their type. In the case of the SourceMeter based analysis, they are usually mapped to the fully qualified name or signature of the source code items and their types, like "class" or "method". All other properties are not obligatory and express the values of various source code metrics calculated by SourceMeter, for example, LLOC and CBO. In this case, child nodes represent the containment relation between source code items.

Mapping step

In this step, we assign properties and entities from the data level to items and their attributes on the metaphor level. The current version of CodeMetropolis uses the entities and attributes to visualize the source code listed in appendix B.1.

To create a visualization with sufficient expressive power, the structure of the system has to be displayed beside their properties. We encoded this information into the containment relation between different graphical items (e.g., buildings); for example, a garden could contain other gardens and skyscrapers.

Rendering step

In this step, we convert the annotated tree of graphical items into a 3D matrix of blocks, which is the underlying structure of every Minecraft world and will be detailed in the displaying step. To do this, we utilized the open world format of Minecraft that encodes our interactive virtual city. There are several binary formats used to describe the scene – called world – which are either open standards or free formats.

Displaying step

In the case of CodeMetropolis, we use a well-known game to display our visualization, called Minecraft [2]. It is written in the Java language and uses the OpenGL graphical engine to display the scenes. Both of these technologies are widely supported on major platforms. It is distributed as commercial software with support.

Due to its extensibility, its simple yet sophisticated functions, and its rich palette of possibilities, Minecraft can display complex structures with low overhead.

The game itself does not have a strict game-flow. Its primary focus is creativity and the joy of creation. Only the available computational power and the storage capacity can limit the fantasy of the player. The central concept in the game is the block. It is a cube with sides about one meter long, when compared to the player. Almost everything is built out of it, so the whole world is a 3D matrix filled with blocks of various types. The player can collect the blocks, create (craft) new ones, and interact with them. The game is similar to a virtual Lego™ with an infinite playground and an infinite number of building blocks.

4.1.2 Integration of Eclipse IDE and CodeMetropolis

The graphical representation of the source code could provide new viewpoints that are crucial for creative work and problem-solving. However, the world of source code is still highly dominated by textual representation. Our goal was to build a bridge between coding and visualization. We chose Eclipse among the IDEs because it is a standard tool for Java developers. Software visualization is embodied by our previously introduced tool, CodeMetropolis. We implemented a set of plug-ins which were able to connect these two software, hence it became capable of integrating an elaborated visualization technique without disturbing the daily routine of developers.

These tools enable developers to launch visualization and initialize the buildings of the virtual city. To help find the most relevant parts of the visualization, a manual and an automatic navigation were included. As a result, developers can get customized visual information about their system fast and without leaving their familiar environment. Besides that, we would like to provide an easy way of navigation in the city to avoid wasting time searching for the place representing the inspected part of the source code.

The implementation has three interlinked components, shown in fig. 4.3. The first is *Eclipse,* the IDE itself, the second is *Minecraft,* which displays the generated city, and *SourceMeter* [61], a static code analyzer, which provides the metrics and the structures of the source code. These are connected via *CodeMetropolis* that converts the data into a visual representation using the given mapping and city metaphor.

There are two small extensions, an *Eclipse plug-in* and a *Minecraft modification* (or mod for short). These are integrated into Eclipse and Minecraft, respectively, and provide communication with the parts of the CodeMetropolis toolchain. The developers interact directly with the game and the IDE.

Modification of Minecraft

In the year 2015, the Minecraft End User Licence Agreement [1] allowed users to change the game once they have bought the license, with the condition that they will not sell those changes as original features.[1] This made the formation of global and local communities

[1]Since the 2013 release of CodeMetropolis toolkit, it has undergone significant changes, while the publishing of Minecraft also changed. Currently, we are in the process of upgrading both the technical and

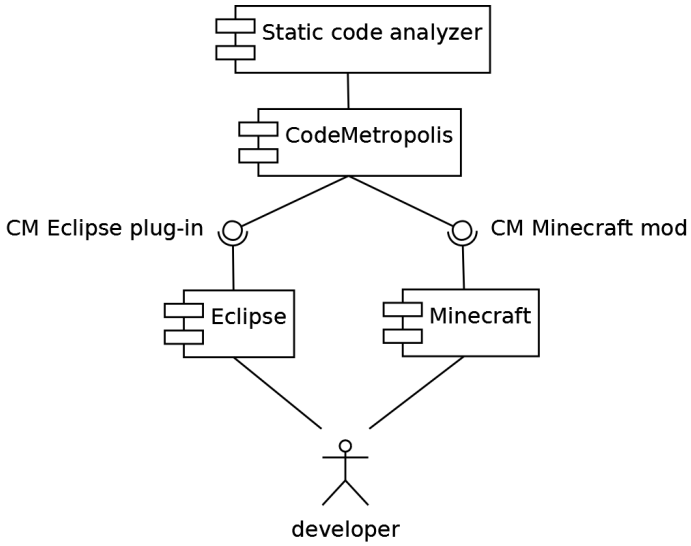


Figure 4.3: Overview of integration

possible, whose members are continuously seeking new ways to extend the features of the game with modifications, or mods for short.

One of the many ways to implement such a modification is to decompile the JAR files of Minecraft, make the necessary changes in the source, and then rebuild the executable file. The process results are several pre-compiled Java files, which have to be inserted into the original Minecraft client in order to install the mod. It only supports a single version of Minecraft, in our case version 1.8.

These mods can have a wide range of goals, from introducing new types of blocks or capabilities to integrating with other third-party tools, like ours, the *CodeMetropolis mod*. It is a collection of recompiled Java classes which provides the following features and functions.

Synchronizing To prevent any concurrent modification with the game, it disables the user interface while building the generated city. After the conversion, the target world is reloaded. We also provide informative messages to notify the user about the state of the process.

Positioning the Player It allows the external processes to set the position and orientation of the player. It is used to redirect the attention of the developer to different components by automatically moving him to a new part of the city.

CodeMetropolis Plug-In for Eclipse

As stated earlier, Eclipse is an Integrated Development Environment or IDE for short. It is one of the most commonly used tools by Java developers. Its main functions are grouped

the legal parts of CodeMetropolis. During this phase some of the features will not be available.

around source code editing, compiling, and running the binary code either in debug or release mode and project management. Since it is beyond the scope of this paper, we do not present an elaborate list of its features. We simply highlight the most important ones for our purposes. Starting with project management, providing basic file, library, and source code management, Eclipse utilizes the common tree view to display the structure of the program. The developer can open the file for editing by double-clicking on it. Afterward, the content of the file becomes visible in the main area. This pane supports multiple opened files by displaying them in a tab control. Functionalities are also available via toolbars and standard menubars.

All these components can be extended with third-party tools called plug-ins. The plug-in infrastructure plays a crucial role in Eclipse; in fact, some of its basic features are also implemented as plug-ins. The API lets the external code collect information about the development process and change the layout of the graphical user interface. Our *CodeMetropolis plug-in* utilizes these possibilities by detecting the name of the edited source file and adding new buttons and menu items to the GUI. To implement the following features, which are available via the menu- and toolbar as well, we used the Eclipse PDE framework 4.5 [**pde**]. It contains a specific version of the Eclipse development environment equipped with several plug-ins, which aid the creation of new plug-ins.

Building This functionality initiates a complete rebuild of the virtual city representing the source code. During this process, the code is analyzed with SourceMeter, and the result is forwarded to the CodeMetropolis toolchain, which generates the city and renders it with the help of Minecraft. The user is continuously notified about the state of the conversion. In the current version, the developer has to initiate the building manually, because the time it takes highly depends on the size of the codebase.

Jumping The size of the generated city could be too large to search for points of interest manually. To overcome this, our plug-in lets the user quickly navigate to the building representing the currently open and active file by using the jump feature. With this, the developer can spend more time with the true exploration of the source code without clueless wandering.

Following We also provide automation over the jumping function, called the following. When users turn this feature on, the system will be continuously checking the open and active file, and update the position of the player accordingly. This means that the player will always be near the building representing the currently edited file.

Changing the Settings The integrated tools required some basic configuration. These contain the location of SourceMeter and Minecraft, and also the path to the mapping file of CodeMetropolis which specifies the meaning of the visual attributes in the city.

Integration of Components

The previously presented components have to work in tandem to provide the following high-level features.

Navigation in the visualization is achieved by setting the position of the player to coordinates specified by the jump or follow functions.

Generation of virtual cities are initiated directly by the developer from the IDE with the preset settings.

The introduction of this connection between the IDE and CodeMetropolis helps the developer (or student) eliminate the wasted time of switching from one application to the other. However, several other factors affect the overall usefulness of these visualization techniques; for example, the graphical and topological properties of the generated cities.

4.2 Assessing Degree of Realism for the City Metaphor in Software Visualization

Our brains are hard-wired to grasp the meaning of real objects. To make navigation easier in a virtual environment and to help interpret the underlying connections and concepts for the user, the generated world has to be quite similar to the real world. In our case, it means we have to generate realistic cities to represent the abstract concepts and relations among the properties of the source code. To create such a city without human intervention, we need a way to connect the low-level properties of the city with its degree of realism.

4.2.1 Low-Level Metrics of Virtual Cities

In order to define low-level metrics, we have to specify the exact model of a generated city. In our study, we mainly focused on cities that only contain buildings like skyscrapers. These buildings could be represented by their bounding box, which is a box with the same width, length, and height as the maximal width, length, and height of the building itself. The current model does not represent the inner structure of the buildings. The buildings are grouped into various types, which could be districts at the metaphor level or namespaces and packages at the data level. Buildings also have a position on the plain. Based on the above statements, the current model defines a building as a box with the following properties.

Width the maximal size of the building along the x-axis.

Length the maximal size of the building along the y-axis.

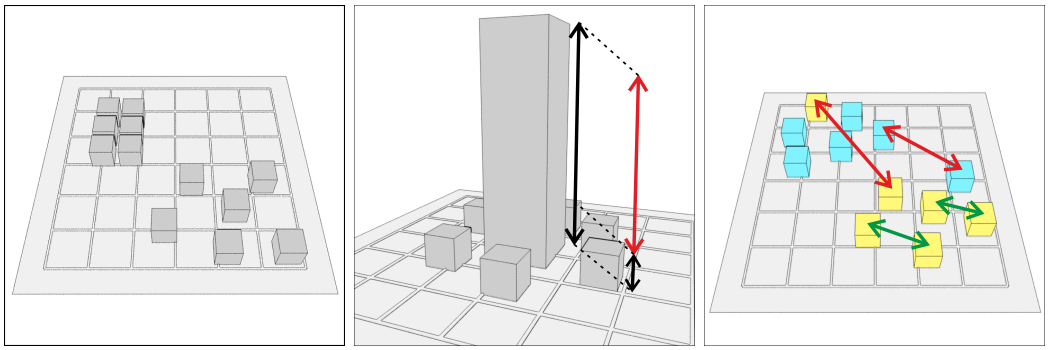
Height the maximal size of the building along the z-axis.

Position an ordered pair of numbers that represents the location of the pivot point of a building on the plain.

Type the unique identifier of the set that the building belongs to.

A city or metropolis is a set of buildings. The buildings cannot be rotated or have any intersecting region. Three low-level metrics were constructed during the study. Our main design the goal was that these metrics have to be independent of the following properties since they show high variations depending on the visualized software system.

- the area of the city
- the height of the city, i.e., the height of the tallest building
- the number of buildings in the city
- the size of the buildings



(a) Compactness (b) Homogeneity (c) Connectivity

Figure 4.4: Low-level metrics

With these in mind, the following metrics were constructed. The formal definition of these properties and metrics can be found in appendix B.2.1.

Compactness

This expresses the density of the buildings in the city (fig. 4.4a). It is the ratio of the total area of the buildings over the convex hull of the city.

Homogeneity

This metric expresses the smoothness of the small scale scenery (fig. 4.4b). It is the distance between any two buildings weighted by the difference in their height.

Connectivity

Connectivity describes the spatial coherence among buildings (fig. 4.4c). It is the distance between any two buildings guarded by their type.

4.2.2 Construction of a High-Level Metric

To create a new high-level metric that could express the similarity between a generated city and a real one, a user survey was used. The target audience contained mainly students and coworkers from the IT field. Altogether, *51 complete and 20 partial surveys were*

processed. The survey contained several questions with a predefined list of choices. We asked the users to rank the cities according to their degree of realism. Furthermore, they had to decide which of the two given cities could be used as an example from a specific point of view. For example, they had to choose the more compact one.

A high-level metric was defined, which can describe the similarity between a real and a generated city. This metric is the weighted sum of the above-defined compactness, connectivity, and homogeneity metrics. We used the answers for the ranking questions of the survey mentioned above. Users had to rank several predefined cities, from the most realistic to the least realistic. The rankings were compared and processed by various methods and algorithms to determine the weights of the low-level metrics.

We utilized the Kendall tau correlation coefficient and community detection algorithm to select a ranking, which reflects the opinions of most of the users. Then we solved a relaxed version of the inequality-system representing this particular opinion to calculate the weights to construct the high-level metric. For more details about the formal method can be found in Balogh, G. [15]. Using this method, we can construct a sub-optimal, high-level metric, which can express the degree of realism of a generated city. Using the notion introduced in appendix B.2.1 the formula for this high-level metric is the following.

$$D \in \mathbb{B} \text{ a city i.e. a subset of buildings} \tag{4.1}$$

$$\underset{\zeta,\gamma_1,\gamma_2}{\operatorname{Rlsm}} D = -0.27 \operatorname{Comp} D - 0.73 \underset{\gamma_1}{\operatorname{Conn}} D + 4.43 \underset{\gamma_2,\zeta}{\operatorname{Hom}} D \tag{4.2}$$

As can be seen in eq. (4.2), users almost completely disregard the compactness of the city and the spatial connectivity of the buildings compared to the homogeneity of their heights. In the future, we plan to compare the low- and high-level metrics of various real-life and generated cities. Nevertheless, we could make some preliminary assumptions. We suppose that the regional distribution of people who submit their opinions may cause this difference in magnitude. For example, cities of Europe are not as much crowded as, for example, cities in China. Furthermore, we used the data captured in Szeged, a city of Hungary. There is not a strict functionality-based district boundary in this city, which could explain the negligent of the connectivity metric.

4.3 Test Visualization with CodeMetropolis

During this phase, we extended the metaphor by using additional metrics computed from the test-related artifacts of the inspected software system. Our approach to combine code and test metrics in CodeMetropolis is to build separate objects corresponding to the code and the associated tests on physical proximity. Also, suitable mapping is used between the metrics and the physical properties, such as building dimensions and building materials. This way, code will become *houses* and tests will turn to *outposts* “defending the code.”

Physical attributes of the outposts such as height, density, and the material will indicate, for example, how thoroughly the associated code is tested (covered) or how specialized the tests are to this code or they test other objects as well.

4.3.1 Measuring test-related metrics

We defined several concrete metrics for the previously mentioned functional units (see, section 2.3.2). For example, the *specialization* metric shows how specialized a test group is to a code group in terms of the ratio of other test groups. A lower value shows that other test groups intensively test the code group in question, while a high value reflects greater specialization. A related metric is the *uniqueness* metric, which measures the portion of the elements that are covered only (uniquely) by a particular test group.

These metrics apply to cross-functional code and test groups (where the tests do not intend to check the methods), not only to functional units. For example, we can compute how the test group of functional unit A covers the code group of functional unit B. These additional measurements can reveal the properties of the test suite and its parts. Hence they may contribute to e.g., the changeability, or maintainability of the test suite.

We used the *SoDA library and toolset* [91] to compute different test-related metrics. SoDA uses detailed coverage information and other metadata (e.g., functionalities tested or implemented by a group of items) to compute the above mentioned metric values as well as others such as the tests to code element ratio.

4.3.2 Test Visualization in CodeMetropolis

As explained in previous sections, CodeMetropolis could assign gardens to classes, cellars to attributes, rooms to methods, buildings to method sets, and uses elevated platforms to denote namespaces (or packages) and express inclusion. Different metrics can be assigned to properties of various components of the virtual city. For example, the physical dimensions of cellars and rooms, the amount of flowers, trees, or mushrooms in a garden can represent various metrics like complexity, size, coupling, code style issues, etc. The assignment between the metrics and the visualization attributes is easily configurable during the mapping step.

Our main goal during this phase was to extend this metaphor and include the visualization of functional and cross-functional units, and test-related metrics, but also preserve visibility of existing static code attributes.

The first step in our method is source code analysis and code metrics computation. Next, functional units are formed by assigning code and test case groups to the different feature sets of the program. This process can be performed in various manual, semi-automatic, or automatic ways, but this is not a topic of the present research. After that, tests are executed, which generates the detailed code coverage from which test-related

metrics are calculated.[2]

During the visualization, we mapped code and test entities and their properties to architectural or landscape objects, to their attributes, and other visually observable phenomena. Then, these objects are constructed from the building blocks of the Minecraft world and placed in it according to their relations. Finally, the game loads the created world, and the developers can get around the city and examine different objects.

The existing visualization concepts in CodeMetropolis remained the same; thus, all source code elements (namespaces, classes, methods, attributes) and their properties (source code metrics) are visualized as before. The additions are *functional* and *cross-functional units* and their metrics.

We could not directly visualize functional units as simple objects in the virtual space, as this combination of code and test groups does not fit in the existing hierarchical approach based on the source code. So, we decided to represent a functional unit as some visible properties of the corresponding objects. Similarly, test cases do not appear in the visualization space as individual objects; instead, our metrics were computed for the *code item–test case* relations. Since the granularity of the metrics is not individual pairs of these items, but functional code and test groups, the base of visualization will be *code group–test group* relations. More concretely, test-related metrics computed for a given functional or cross-functional unit will appear as objects, and their visual properties will reflect the corresponding metrics. We call these objects the *outposts*.

However, if we placed outposts directly in the visualization space, we would lose the connection between them and the source code. Therefore, we placed an outpost for each (test metric, code group, class) triple. This decision implied an additional property to be visualized: the *completeness* of a feature regarding a class. This metric expresses the ratio of the concerned code elements (methods in the same class that belong to the assigned code group) compared to all code elements that are assigned to the feature implemented by the given code group. For example, if the class `TextFile` has a `read()` method assigned to the *input* feature, which has two other assigned methods in addition from other classes, then `TextFile` provides $\frac{1}{3}$ feature completeness for the *input* feature.

4.3.3 Side by side visualization of code and tests

Two objects whose source code elements are close to each other in the code structure (and hence appear also close in the virtual city) may implement different features. Similarly, the same feature may be assigned to distant objects, so mapping functionalities to object placement would raise several problems. Therefore, instead of representing features as objects, they will be mapped to object properties. Fortunately, in Minecraft, we can use many kinds of building blocks, so we assigned different blocks to different features. Then, the outlook (color and texture) of the objects represents the assigned feature.

[2]More details of the process can be found in our other work [89].

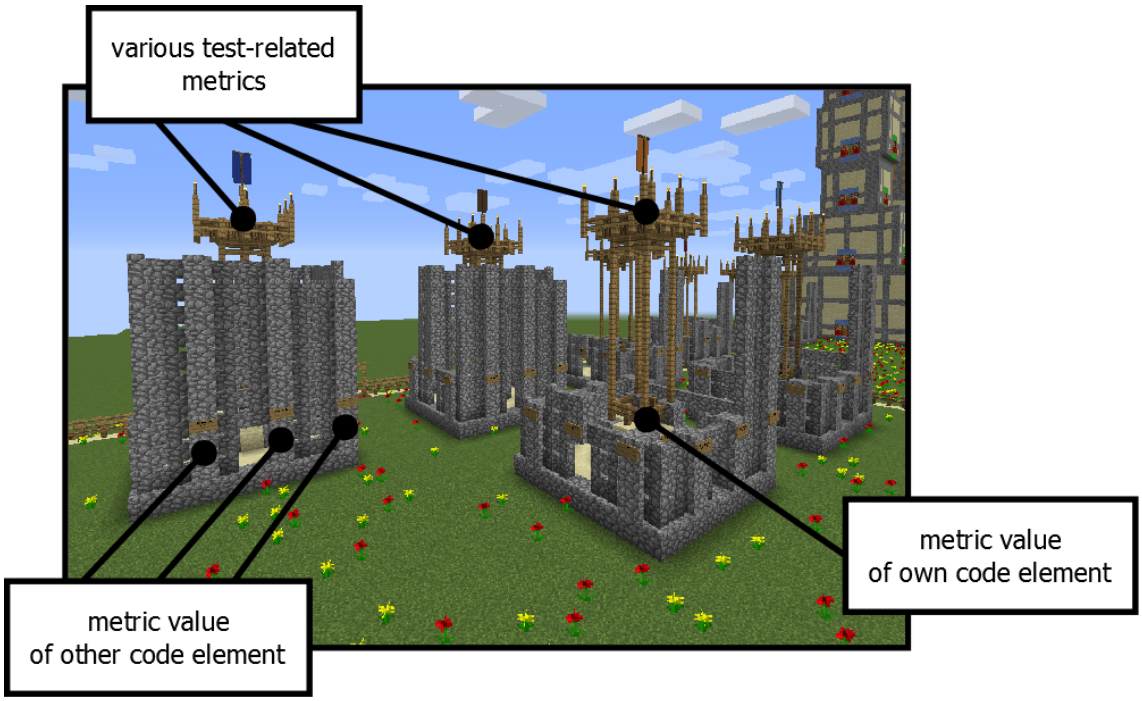


Figure 4.5: Parts of outpost of test-related metrics

To visualize the concept of (cross-) functional units and their test-related metrics, we are using the aforementioned outpost objects. What we want to see is how well the code elements are tested along with the features, so we had to find a way to visualize the metric values of functional and cross-functional units. Outposts are placed inside the gardens of classes. Each outpost has a central watchtower and a surrounding fence, as shown in Figure 4.5. The height attribute is used to represent the metric value. Also, one of its two building materials reflects the assigned feature. In addition, the outposts are equipped with explanatory signs and a colored flag on the top.

The basic concept behind using outposts is that tests are "guarding" the code. Tests of a certain functional unit are created to check the quality of the code of the same unit, but they are not intended to test code from cross-functional units. Based on this consideration, each outpost of the class is assigned to a *metric–code group* pair and represents a single metric of multiple (cross-) functional units. The central tower of the outpost is assigned to the test group of the same feature that the code group of the outpost was assigned to, while segments of the surrounding fence of the outpost represent the other test groups. Thus, the central tower represents the metric value of the functional unit. The tower also has scaffolding up to the top, which represents the actual metric value. The name of the functional unit is shown on a wall sign, but is also encoded into the building material of the tower walls and outpost ground. This provides a strong visual connection between the methods and the outposts of their test-related data.

The surrounding fence of the tower is divided into segments, and each segment is assigned to a cross-functional unit (cross-functional test group of the given code group).

The height of each fence segment represents the metric value of the same metric (which was assigned to the outpost) for the cross-functional unit assigned to the segment. For example, if the outpost stands for coverage and is assigned to the *input* code group, one segment of the fence will represent the *output* test group, i.e. the *output-input* cross-functional unit.

This construction of the outpost lets us visualize some common shortcomings of the tests. For example, consider the outposts in Figure 4.5. A high fence around a low tower in the leftmost outpost assigned to the coverage metric (and some feature) shows that tests intended to check the implementation of a feature are not performing well. While other tests (not intentionally created to do so) will do the job instead. This scenario violates the modularity of the system.

4.4 Application

In some cases, developers need to step away from the source code and inspect the system from a different perspective. We believe that CodeMetropolis will be able to maintain motivation without sacrificing productivity thanks to its intuitive and, for many people, already known graphical user interface. The provided metropolis metaphor has enough expressive power to represent the complex items of the source code. Combined with high-quality graphical techniques provided by today's computer games, it can offer a rich graphical interface, easy to learn controls, and a productive user experience.

Currently, CodeMetropolis is probably easier to fit in classrooms than in a commercial project. However, with Eclipse IDE integration, we think it might be useful outside the classrooms. For example, it could help developers during refactoring sessions – when neither are new features implemented, nor bugs fixed – only the underlying structure of the code is changed to improve the quality of the software. With the proper mapping of metrics, it could guide the developers to detect the possible weak spots of the system.

The software system usually contains several types of artifacts beside source code, for example, elaborated test suits and their source code. Understanding the structure of large test suites and the relation of their constituent test cases to the code of a system is hard, and there are not many tools to aid this activity. This work combines two previous approaches: a method to express test quality in terms of metrics and visualization of code related metrics in the CodeMetropolis framework. The city metaphor employed by CodeMetropolis seems to be useful for test metrics as well, and we believe that the side by side presentation of code and tests will enable the developer to obtain a more global picture of their software.

4.4.1 Scenarios of Practical Usage

We have identified two major use cases for our tool. Exploration tasks of software comprehension consist of the actions that need to be performed to comprehend source code, which was written by someone else. We assume that developers need to execute these tasks dur-

ing their daily routine. The other potential use of the tool is in education. Visual analogies can make learning a lot easier for most of the students.

Software Comprehension

Developers, either juniors or experts, often have to join ongoing projects. In these cases, the codebase already contains the implementation of some features. The size, the importance, and the quality of these show a wide range of variation. The developers have to find the location of the important parts of the code and need to gather general knowledge about the properties of various code entities, like classes and methods. In other words, confidently navigating trough the codebase can speed up the implementation of further features.

The integration can help to explore the code by combining an intriguing and rich visual representation with the familiar environment of the Eclipse IDE. The use case begins with the opening of Eclipse and then launching Minecraft right from the CodeMetropolis menu. The next step is to generate a virtual city, which is going to represent the source code. To do this, developers use the Build feature of the CodeMetropolis Eclipse plug-in. Afterward, the users can open the generated world in Minecraft and begin the exploration task itself. This usually contains a series of repeated steps, during which various code entities are inspected. Activating the Follow feature in Eclipse ensures that the player is always in the garden, which represents the edited class. This synchronous navigation lets the developers compare the values of source code metrics which are challenging to see from the code, but are displayed as various visual properties of the buildings in Minecraft. This might be less tiring than manually comparing a bunch of raw metric values, especially in the case of large systems.

Education

For students, it usually takes much time to fully understand the concepts and advantages of object-oriented design. They need to learn a new perspective on programming tasks to be able to design the structure of their systems properly. By visualizing the structural parts of code, they can see programs in a new way. They can comprehend the structure of the source code just by walking around in a virtual metropolis. The relationship between packages, classes, methods, and attributes can easily be presented through the buildings of the city. They can understand underlying properties (metrics) and their connections. This kind of visualization is also a great way to present programming to younger children. Real-life analogies can make them feel more comfortable while talking about abstract things like classes or metrics.

During a learning session, students should perform the following actions. First, they need to start both tools: the IDE and the game. After opening the selected project, they can build the virtual city and enter into the visualization. Then, they should investigate

and understand the connections between the objects of the city and the code entities. The jump function can be beneficial during this phase. Implementing new features or modifying existing ones affect the structure and quality of the code. It is recommended to rebuild the visualization after the changes so students can examine the effects of their actions. We assume that by repeating these steps multiple times during the life cycle of the project, they can monitor their coding quality and recognize structural weaknesses in the system.

4.4.2 Demo Scenarios

In this section, we present two demo scenarios. Both of these use the open-source project *tutorial-refactoring-rectangles* [84], which is a small Java program. It serves as a classroom exercise for students to practice various refactoring techniques. To help the reader understand the scenarios, we only specify the relevant metrics and properties. The elements of the source code are assigned to the same type of building in all cases, so classes are represented as gardens, their methods are displayed as the floors of buildings, and the stone plates stand for namespaces or packages. However, the properties of those which are linked to various metrics are different and will be explained later. More demo scenarios can be found in the supplement material.

Inspecting Various Visual Properties of a Single Building

In the first scenario, the *logical lines of code* are mapped to the *height of the floors*, and the *number of statements* is visualized as the *material the walls are made of*. This means that if a method has more lines that are neither empty nor comments, the related floor will be higher. On the other hand, if a method contains fewer statements, the floor will be built from lighter materials like sandstone or glass instead of the darker ones like stone or obsidian. The minimal height of a floor is nine blocks, and the materials range from glass to obsidian.

Figure 4.6 shows the constructor of the Rectangle class and its visualization. The assigned elements are highlighted and connected. In this case, the code from line 41 to line 46 is represented with the floor in the middle. This is made from sandstone, and it is neither extremely tall nor short. Its visual appearance suggests that it cannot contain too many logical lines of the code or statements. Furthermore, the lightness of the material and the moderate height indicate that the value of these two metrics is relatively close to each other. The implementation of the constructor contains four logical lines and four statements. It means that the average ratio is 1 statement per line.

These values of metrics can be calculated or compared manually, but the time it takes depends on the size and the complexity of the code. The use of CodeMetropolis as visualization and its plug-in for Eclipse could speed up this process by providing a rich and interactive visual representation. Inspections like this are the core step when the

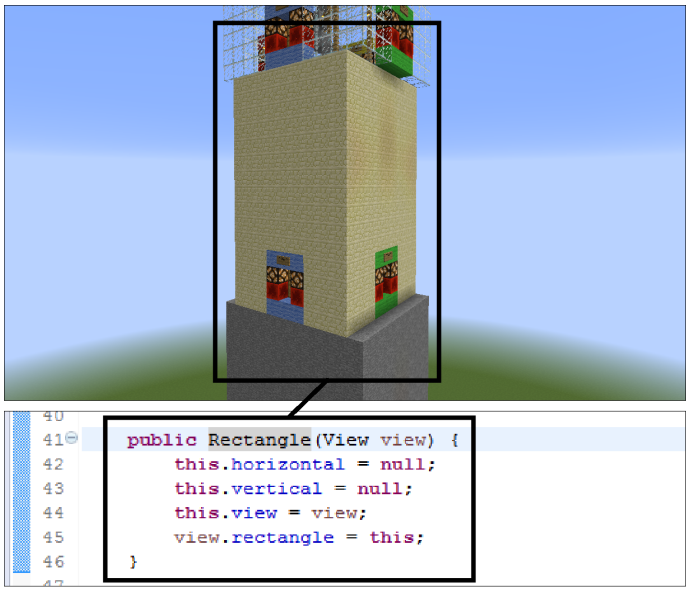


Figure 4.6: Inspection of the constructor of the Rectangle class and its visualization

developer needs to explore new source code.

Compare Two Different Buildings

In this case, two buildings are compared, namely two methods of the *Rectangle* class: the *contains* and the *equals* methods. We use the same ranges for building material, as in the previous case. The *number of statements* is represented by the material, but the height is assigned to the *cyclomatic complexity* used to indicate the complexity of a program. It is a quantitative measure of the number of linearly independent execution paths through the source code of a program.

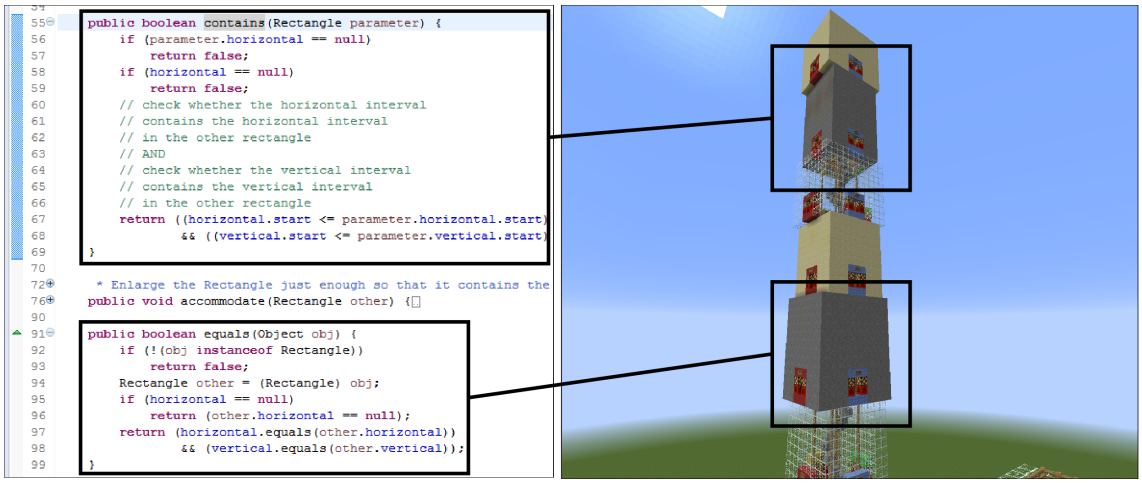


Figure 4.7: Comparison of two methods and their corresponding floors in the virtual city

Figure 4.7 shows the relevant parts of the source code and its graphical representation.

The assigned elements are highlighted and connected. In this case, the code from line 55 to line 69 and from line 91 to line 99 is represented with the two stone floors. Both of these are made of the same material, so they contain a similar amount of statements. The one above is higher, so it is more complex than the other.

To decrease the overall complexity of the class and improve readability, developers may refactor the most complex parts of the class, in this case, the *contains* method. After finding the appropriate method, the modification can be applied in Eclipse. In this case, the last two lines (lines 67 and 68) can be extracted into two new methods, which will compare the vertical and the horizontal size of the rectangles. Then, the developers rebuild the virtual city to see the effect of their changes. These steps can be repeated to accomplish the required refactoring tasks.

4.5 Contributions

My software visualization related research consisted of three phases. At first, I improved the expressive power of an already existing city metaphor. In the second phase, I constructed several metrics to asses the degree of realism. Finally, I contributed to the development of a novel approach to visualize test-related properties.

Enhancing the City Metaphor with Game-based Visualization My main contribution in this phase was to connect data visualization with high end-user graphics capabilities. To achieve this, a conversion tool was implemented. During the research, I utilized the city metaphor to visualize abstract concepts and properties related to software systems. It processes the source code metrics as input and generates a Minecraft world with buildings, districts, and gardens, which represent abstract properties of the software. The tool is in the prototype state, but it can be used to investigate the possibilities of this kind of data visualization. The research conducted in this phase forms the basis for several sub-thesis points, but it has the strongest relation to sub-thesis points 2.1 and 2.3.

Assessing the Degree of Realism for the City Metaphor in Software Visualization In the previous phase, I focused on enhancing the city metaphor, which represents information as buildings, districts, and streets. To allow the users to navigate freely in the artificial environment and to understand the meaning of the objects, we had to learn the difference between a realistic and unrealistic city. To do this, we had to measure how similar it is to reality. I designed and presented three metrics that express various features of a city. These metrics are compactness for measuring space consumption, connectivity for showing the low-level coherence among the buildings, and homogeneity for expressing the smoothness of the landscape. I analyzed the connections between the high-level measure of realism and these low-level metrics. I wanted like to capture the subjective opinions of users with an online survey. These data were used to construct and fine-tune

the high-level metrics. These experiments support sub-thesis point 2.2.

Test Visualization with CodeMetropolis Before this phase, CodeMetropolis was limited to represent only code related artifacts. I contributed to the extension of the metaphor to include properties of tests related to the program code using a novel concept, which covers sub-thesis point 2.4.

I participated in all parts of the project, but my main contribution was to design the graphical elements (called outpost) used to display test-related information. It includes the ecstatic and functional details of these. The test suite and the test cases are also associated with a set of metrics that characterize their quality, but also reveal new properties of the system itself. In a new version of CodeMetropolis, gardens representing code elements give rise to *outposts* that characterize properties of the tests and show how they contribute to the quality of the code. Former methods presented either code or test-related objects individually, but not both in a common space.

4.6 Advantages and Disadvantages of These Methods

4.6.1 Internal Validity

The usefulness of a tool like CodeMetropolis depends on various factors including the experience and personal values of the users. For people who naturally use similar metaphors to understand the world this could be a straightforward way of visualization, while for many others the approach would merely be an interesting but generally experimental idea.

Construct Validity

The dominant benefit of the enhanced city metaphor, namely that it can represent many different properties, could also be its biggest drawback. The resulted visualization could be overwhelming. The increased amount of encoded information may reduce the chance of locating the point of interest. Moreover, the multitude of configuration options make it hard to set up a visualization which aids the completion of a specific task. These problems could be addressed by evaluating the generated cities and automatically fine-tuning the settings of CodeMetropolis.

The evaluation of visualization is, by nature, a subjective endeavor. This makes it hard to automatically preset the optimal configuration based on a set of measurements, like the previously described low- and high-level metrics of the city. One possible solution could be to build a benchmark of real-life cities to approximate the optimal solution. However, there are several legal concerns to take into account when acquiring the necessary layout data.

Content Validity

We do not doubt that there are several factors of software quality that cannot be displayed using the city metaphor. The current version of CodeMetropolis is using augmented trees as an underlying data structure. Any information which cannot be captured with this datatype is currently out of reach of our visualization techniques. For example, we are unable to display intricate web of interconnecting classes and methods. However, in this case, the original data could be converted (maybe in lossless fashion) to the appropriate representation.

Furthermore, the metrics for generated cities are designed to help the layout of the metropolis. Hence these measures ignore the internal properties and shapes of the buildings. We could address this issue in two ways: by extending (changing) the low-level metrics or by adding new ones to the set.

4.6.2 External Validity

The generalization of CodeMetropolis and the techniques it represents can be inspected from two perspectives. We have high confidence that these can be used to teach abstract concepts excitingly and intriguingly. Neither CodeMetropolis nor its underlying concept is demanding any strict constraint to the data that need to be visualized. Hence it could be useful during the education of other natural sciences.

In the case of industrial application, the city metaphor could be useful in communicating non-functional improvements towards non-professional stakeholders, like customers. On the other hand, developers and testers are more reluctant to disturb their routine with emerging techniques and presently unknown tools. In the future, we plan to ease the learning curve by providing ready-to-use settings and scenarios to complete several everyday tasks for various stakeholders; for example, software comprehension (challenge 1).

In either case, the success of these techniques strongly depends on, the user being able to perform their tasks, while staying motivated; in other words, “making software metrics such fun that you want to do it”, as one of our users put it into words.

Chapter 5

Using Test Coverage to Analyze Structures in the Package Hierarchy

This chapter contains a detailed elaboration of the researches related to test and code quality measurement and improvement. The structure of the related thesis point and its connections to various stakeholders and topics are shown in fig. 5.1.

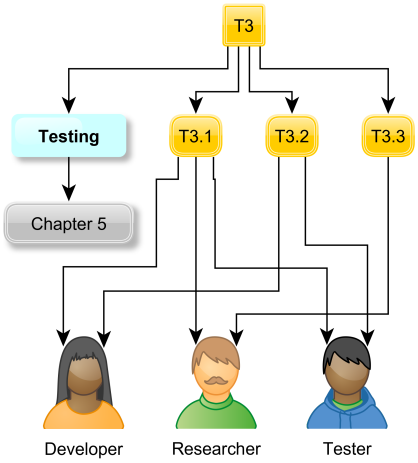


Figure 5.1: Structure of thesis points

During software analysis, researchers and IT experts often rely on the comparison of datasets. They also frequently draw conclusions based on differences between two representations of the same item's set. For example, developers may examine the densely connected parts of method call graphs in the context of their location in the package hierarchy tree to find error-prone parts of the system. These kinds of analyses could be aided with a generalized methodology for graphs, which can be used to unify the underlying process of discrepancy analysis. In this chapter, we present a *methodology for*

unified graph's discrepancy analysis, named UniGDA, which is a generalized technique to compare the arbitrary graph-based representation of the inspected entities and analyze the differences (discrepancies) between them. It is based on our previously defined domain-specific discrepancy detection technique for cluster comparison. Our generalized methodology is using different types of characteristic functions to capture the similarity structures between vertices of arbitrary graphs. We provide several domain-independent options for the free parameters of UniGDA.

We also present two possible *use cases of* UniGDA*: the classification of structural test smells and the clustering of test-code traceability discrepancies* to showcase the usage of our methodology. We propose a semi-automatic method for these concrete tasks, which are based on static and dynamic software analysis approaches, comparing their results and presenting the discrepancies to the user, who will determine the final action to take based on the differences and contextual information. We define a set of domain-specific discrepancy patterns, which can help the user in this task. Additional outcomes of analyzing the discrepancies are structural unit testing issues (suspicious parts of the system and test suite) and related refactoring suggestions.

For the static test-to-code analysis, we rely on the physical code structure, while for the dynamic, we use code coverage information. In both cases, we compute combined test and code clusters, which represent sets of tests and their subject items. We also present an empirical study of the method involving eight non-trivial open-source Java systems. This chapter mainly address the challenges 1 and 2.

Software systems consist of several interlinked items; the results of these analyses are often presented in the form of graphs.

For example, the internal structure of the systems is usually captured with various diagrams, like control flow graphs or several types of UML diagrams [70], which could be interpreted as directed graphs with special labels. There are several ways to represent the relationships between different kinds of development-related entities, like users, developers, bugs and test cases [104, 79, 24, 4].

The wide variety of items and their associated meta-information lead to very distinct graph-based representations. There are several ways to encode the same information, and there are various tools that can provide the requested data. For example, both the static and dynamic program analysis could yield information about the consecutive function or method calls, and there are several tools (using either technique) that could retrieve these data.

Researchers usually use some domain-specific heuristics to asses the similarity between these representations to aid their analysis. During our research, we address two particular issues, namely *the classification of structural test smells and the clustering of test-code traceability discrepancies.* We conduct manual analyses of the discrepancies to evaluate their findings. However these methods [44, 21, 8] contain several common steps and utilize similar concepts. Based on these experiences, we provide a general methodology for the

graph's comparison, which can be used to aid the researchers during their work, by unifying the underlying process of discrepancy analysis.

5.1 Simultaneous Clustering of Test Cases and Methods

In section 2.3.1, two important properties of well-designed unit tests have been put forth: their executions are restricted to the tested unit, and they are appropriately named and structured. To check whether this holds for a particular test suite, the manual option would be to verify what pieces of code each test case exercises (directly or indirectly) and compare this to the physical location of the test. This is, however, impractical for real-word test suites.

To automate this task, we employ two clustering algorithms that can group together test and code items. The first one is based on code coverage and captures dynamic relations between the test suite and the system under test. This is then compared to the other, trivial clustering, that works from static information and captures the structural properties of the tests and program code.

5.1.1 Package Hierarchy Based Clustering

Through package based clustering, we aim to detect groups of tests and code that are connected by the *intention* of the developer or tester. The placement of the unit tests and code elements within a hierarchical package structure of the system is a natural classification according to their intended role. When tests are placed within the package that the tested code is located in, then it helps other developers and testers to understand the connection between tests and their subjects. However, it might happen that the developers did not follow unit testing guidelines, or the system evolved in such a way that due to package reorganization, the package structure does not reflect the actual role of the tests.

Our package based clustering simply means that we assign the fully qualified name of the innermost containing package to each test and method, and treat tests and methods belonging to the same package as members of the same cluster. Names of the test and code elements are not considered; the naming of a particular piece of code (either a unit test or regular code) is determined by the rules of JUnit (such as the special annotations), our unit testing framework.

5.1.2 Test-Code Coverage Based Clustering

In order to determine the clustering of the tests and code based on the dynamic behaviour of the test suite, we will apply *community detection* [26, 41] on the detailed code coverage information. Detailed code coverage, in this case, is that, for each test case, we record individually what code elements (methods, in our case) it executed. This forms a binary

matrix (called *coverage matrix*), with test cases in its rows and methods in the columns. A value of 1 in a matrix cell indicates that the method is executed at least once during the execution of the corresponding test case (regardless of the actual statements and paths took within the method body), and 0 indicates that that test case has not covered it.

The concept of clustering based on dynamic behavior used in this work can be illustrated by investigating different regions in the coverage matrix. Groups of tests and methods that form "dense regions" in the matrix may be grouped, indicating that there is a close correspondence between them from a dynamic point of view. These regions contain more "one" values in the cells, while outside of them in their rows and columns, the 0 values are more common. The property of the members of such groups (or clusters) is that their test cases more likely to cover their methods than other methods and that their methods are more likely to covered by their test cases than by other test cases.

There might be different approaches to detect these clusters, but they are based on some kind of heuristic that tries to maximize the coverage within a cluster and minimize them outside. Our choice for cluster identification was to use *community detection* [41]. This set of algorithms is originally defined on (possibly directed and weighted) graphs that represent complex networks (social, biological, technological, etc.), and recently have also been suggested for software engineering problems [65, 51].

Community structures are detected based on statistical information about the number of edges between sets of graph nodes. So, in the next step, we construct a graph from the coverage matrix, whose nodes are the methods and tests of the analyzed system (hereafter referred to as the *coverage graph*). A method and a test node in this graph are connected with a single unweighted and undirected edge if and only if the method was covered during the execution of that particular test, i.e.there is a 1 in the corresponding matrix cell. This way, we define a bipartite graph over the method and test sets because no edge will be present between two methods or two tests. The actual algorithm we used for community detection is the Louvain Modularity method [26] (also used by Hamilton and Danicic [51]), a greedy optimization method based on internal graph structure statistics to maximize modularity [21].

5.2 Similarity Pattern Detection

The previously introduced package and test-code coverage based clusterings could be depicted as graphs where each node represents a single cluster without any connection between them. We propose a unified process (UniGDA) to compare two arbitrary graphs and also present a domain-specific implementation for these two graphs of clusters [14][1]. In further sections, we will show that; however, UniGDA was deduced from this simple representation, it can capture arbitrary edge-related properties.

UniGDA contains two main phases. During the preparation, we calculate the sim-

[1]This paper was submitted for publication, but it was not accepted yet

ilarity between each pair of the inspected graphs and construct a generic description of the discrepancies among them. We use this information to classify or further analyze the discrepancies during the evaluation phase. These phases are shown in Figure 5.2.

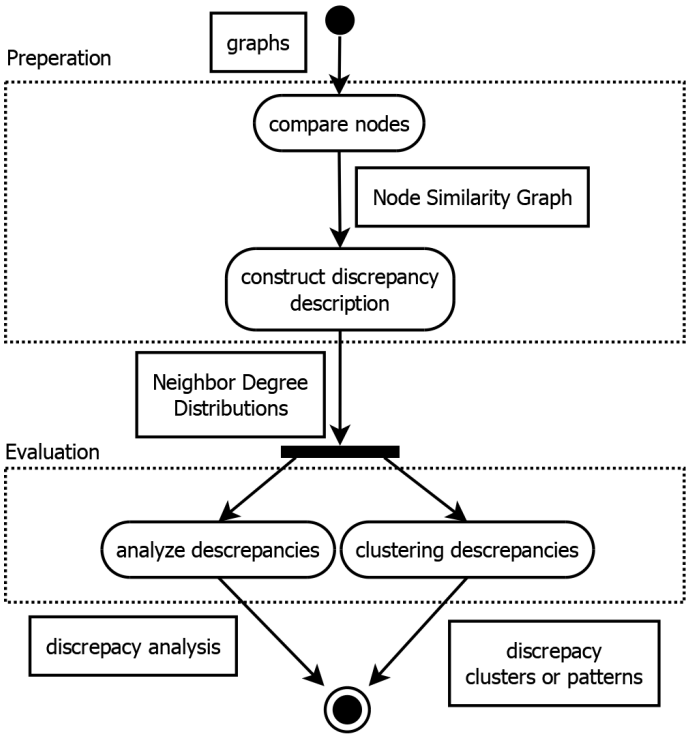


Figure 5.2: Overview of the UniGDA Process

The first phase consists of two steps: the comparison of subject nodes and the construction of discrepancy descriptions. These steps are responsible for extracting the relevant properties of nodes and collecting information about their differences.

The evaluation phase could encapsulate several parallel steps, for example, the analysis or the grouping of discrepancies. These steps could be executed manually or with various degrees of automation. Their goal is to provide data that could be used to answer the research questions or solve the concrete problems under investigation.

In the following subsections, we will elaborate on the theoretical (and some technical) details about these steps.

Comparison of Nodes

In this step, we compare the nodes of the two subject graphs and capture the differences (and similarities) between them. In other words, our methodology requires a node level comparison to provide the relevant fine-grained information, which will allow us to inspect the local discrepancies of graphs under inspection.

This comparison is made by computing the similarity between each pair of nodes and denoting the similarities with a value from zero to one (for a formal definition of similarity

function see definition C.1.1).

General (plain) graphs are rarely used to describe various aspects of software systems or their related processes and artifacts. Researchers usually represent their data as graphs that have several properties associated with their nodes and edges. These could be weights, labels, ids, names, and several other attributes rooted in the domain of the original problem. During the comparison, we use the similarity function to capture domain-specific details about the resemblance of these properties.

The details of this comparison highly depend on the concrete problem, so our methodology does not enforce any unnecessary constraints of the similarity calculation method. To define the discrepancy descriptors (introduced in a subsequent section), we bound the range of the similarity function ($\sim$). The lowest value ($g \sim h = 0$) means that there is no similarity between the inspected nodes, while the highest value represents that they are identical (they have maximum similarity, $g \sim h = 1$).

Note that our process does not require any specific distribution or granularity of similarity values. However, the properties of the selected similarity function should be considered during the evaluation phase.

Domain Independent Similarity Functions

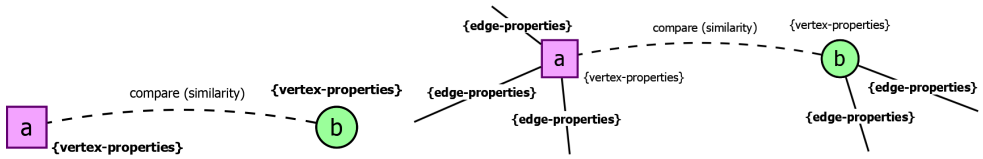
Based on our experience, we can suggest some domain-independent candidates, which fulfill the previously mentioned requirements and could be used as a similarity function between nodes. Each of these has some prerequisite conditions. See appendix C.1.1 for formal definitions.

Vertex-property Based Similarity encodes the aggregated similarity of properties for each vertex (fig. 5.3a). The prerequisite condition for this similarity function is that there should be an equivalence operator over the set of property values of vertices.

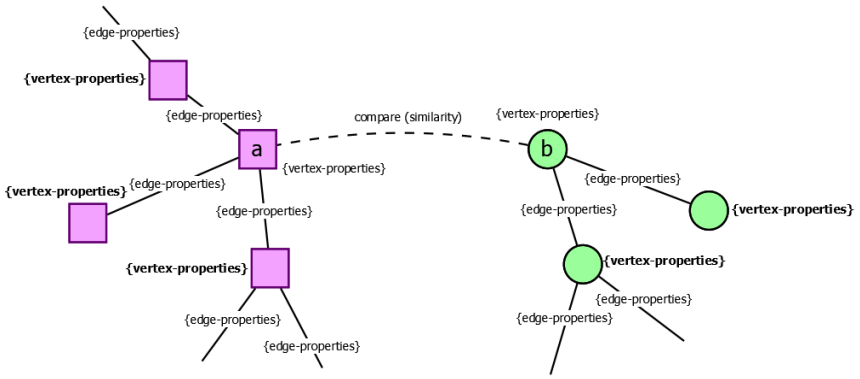
Edge-property Based Similarity captures the aggregated similarity of the node's edges based on their properties (fig. 5.3b). The prerequisite condition for this similarity function is that there should be an equivalence operator over the set of property's values of edges, and a meaningful aggregation over the edge similarities should be specified, which could be easily interpreted by experts of that domain.

Adjacent Vertex's Similarity-Based Similarity can express the aggregated similarity of adjacent vertexes based on their properties or the properties of their edges (figs. 5.3c and 5.3d). Both of these similarity functions inherit the prerequisite condition of the previous similarity computation method. Furthermore, we have to choose a meaningful aggregation for the similarities of the adjacent vertices.

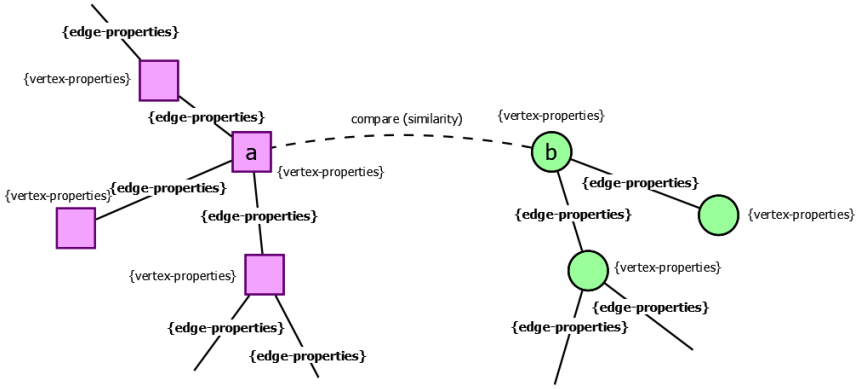
Compound, Property-Based Similarity is constructed by taking the aggregation of any number of previously listed similarity functions. This compound similarity function inherits all previously mentioned prerequisite conditions.



(a) Vertex-property Based Similarity (b) Edge-property Based Similarity



(c) Adjacent Vertex's Similarity Based Similarity (vertex-property based)



(d) Adjacent Vertex's Similarity Based Similarity (edge-property based)

Figure 5.3: Domain Independent Similarity Functions

For aggregation, we could use various statistical functions like the means, median, and deviation. Normalized set comparison metrics [102], for example Jaccard similarity index [92] are useful for encoding underlying similarities, like similarity of property sets.

Constructing the Node Similarity Graph

During the second step of our comparison methodology, we construct a graph-based representation, which contains all the required information to locate and process the discrepancies.

We call this compound graph the Node Similarity Graph (NSG, definition C.2.1). It contains all nodes from both inspected graphs. These vertices are connected if their similarity is greater then zero (i.e., they are at least a bit similar to each other). The edges of NSG are directed since our methodology does not require that the similarity has

the same magnitude in both directions. These edges are weighted with the magnitude of similarity of their endpoints. The NSG encodes all information into the similarities. Hence it does not denote the edges of the inspected (original) graphs.

In Figure 5.4, we show an example of the construction of NSG, marked with S. Edges in the two subject graphs (G and H) are marked with a dashed, gray line, while solid, black lines represent connections in the similarity graph.

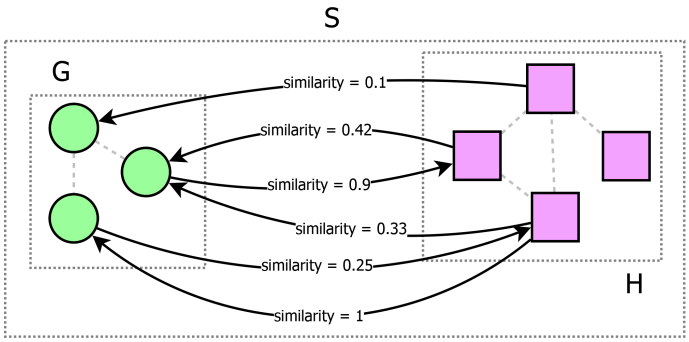


Figure 5.4: Node Similarity Graph construction

Discrepancy Pattern Description To enable further analysis of the discrepancies, we require a unified way to characterize them. Informally, we need to create tools, which could be used to answer the questions: *How similar entities relate to other entities?* To address this issue, we introduce a special feature vector and function to describe the inspected vertex and its neighbors in the NSG, namely the *Neighbor Degree Distribution (*NDD*) vector and curve*. As their name suggests, they encode the distribution of the inspected node's adjacent vertices according to their degree. These descriptors can capture information about the adjacent vertexes and their neighbor (i.e., 2-edge wide context) of the inspected vertex. When applied to the NSG, they provide an easy-to-use tool to describe the local similarity relations.

During the following sections, we will use the notions presented in Figure 5.5.

The label a represents the vertex under inspection, while b_i stands for its adjacent vertices and their neighbors are marked with $c_j^{(i)}$. We use α_i and $\beta_j^{(i)}$ to denote weight for edges $\overrightarrow{(a;b)}$ and $\overrightarrow{(b;c)}$ respectively. Please note that some of the vertices may have more than one label, for example, the inspected vertex could be referred to as a or c, since it is a neighbor of its neighbors.

Discrete Case

In the discrete case, NDDs are capable of encoding the presence of similar vertices, but they do not store information about the magnitude of similarity between the inspected vertex and its neighbors.

In this case, NDDs are represented as vectors of natural numbers. We named these

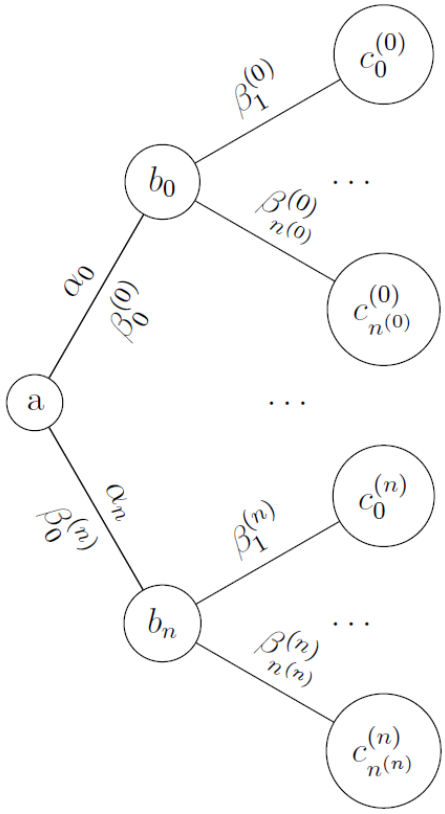


Figure 5.5: Sample graph for NDD definition

Discrete Neighbor Degree Distribution vectors, or DNDD for short. They give an accurate representation of the distribution of neighbors with a specific degree. The counting of adjacent vertices is preceded by a filtering step, where all edges with zero weight are eliminated.

Informally, we seek answers to the following questions: How many adjacent vertices have a specific number of neighbors? The resulting vectors can be represented as a bar chart, where the horizontal axis denotes the number of adjacent vertices, and the vertical axis shows the number of neighbors with that particular degree. This vector bears a resemblance to the commonly used histogram, which is an accurate representation of the distribution of numerical data, in our case, the number of neighbors with a specific degree. It is an estimation of the probability distribution of this quantitative variable. The general and formal definition of DNDD vectors can be found in definition C.3.1.

For example, the DNDD vector of the middle vertex in Figure 5.6 is $(0, 1, 1, 2, 0, 0, \dots)$. We usually simplify this notion by removing the trailing 0-s, [0 1 1 2]. The red circle represents the scope of the DNDD vector since it is only capable of encoding information in a 2-edge wide context. This DNDD vector means that the inspected vertex in the middle has the following neighbors.

$d_1 = 0$ There is not any adjacent vertex with only one connection.

$d_2 = 1$ It has one neighbor with two adjacent vertices, i.e. the one on its left side.

$d_3 = 1$ There is also a single adjacent vertex with three connections, i.e. the one on its right side.

$d_4 = 2$ Finally, there are two neighbors with the degree of 4, the top and bottom ones.

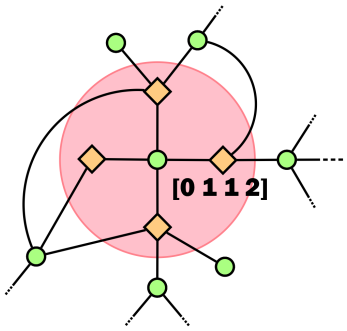


Figure 5.6: Example graph for DNDD calculation

Note that the above described DNDD vectors are unable to express the differences between the weights of edges. Hence they are not encoding any information on the magnitude of similarity. The NDD vectors could still be sufficient to analyze similarity in various cases (see, sections 5.3.1 and 5.3.2). However, a more in-depth inspection requires taking the magnitude of similarity into account (see, section 5.3.3).

Continuous Case

In this step of the research, our goal was to construct a descriptor which can capture information about the magnitude of similarity between vertices, instead of only being able to express the existence of similar ones. To address this issue, we extended the previously discussed DNDD vectors, and defined a new construct, namely CNDD curves ($\text{cNDD}(a) : \mathbb{R} \mapsto \mathbb{R}$). We assign a CNDD curve to each inspected vertex of the subject graph (i.e., the NSG). These curves are an aggregation of several Gaussian functions meant to describe the neighbors of the inspected nodes. The free parameters of the bell-curves are based on the magnitude of similarity of the adjacent vertexes (and their neighbors). For the formal definition of CNDD curves, including these free parameters, see appendix C.3.2.

We used the core concept of *kernel density estimation* [82] to define the characterizing function of the neighboring vertexes. In statistics, kernel density estimation (KDE) is a non-parametric way to estimate the probability density function of a random variable, in our case, the number neighbors with various degree. These estimates are closely related to histograms, but can be endowed with properties such as smoothness or continuity by using a suitable kernel. In the case of the histogram, first, the horizontal axis is divided into sub-intervals or bins which cover the range of the data; this is analogous to the items of DNDD vectors. For the kernel density estimate, we place a kernel function on each of

the data points, i.e., on the neighbor with a specific degree. The kernels are aggregated to make the cNDD function, similarly to the kernel density estimate.

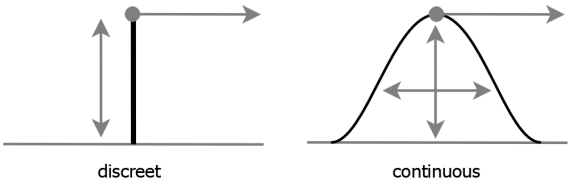


Figure 5.7: Characteristic function

By definition, cNDD curves can express more information about the inspected vertex and its surroundings than dNDD vectors. These differences arose from their underlying function used to characterize neighbors of an inspected vertex. As illustrated in Figure 5.7, in the discrete case, the characterizing function has two independent properties to encode information (informally, its offset and height); while in the continuous case, we can represent one more additional dimension of the data (informally, the width of the bell-curves). For more details, see appendix C.3.2 and definition C.3.3.

Evaluation

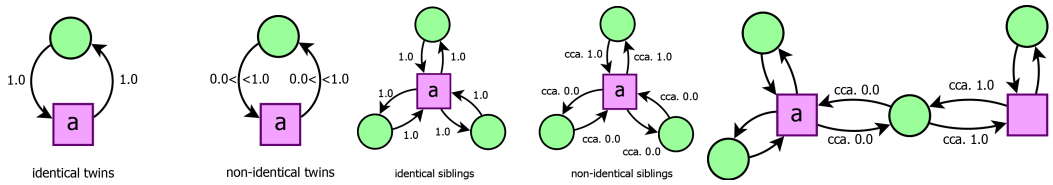
During the evaluation, we used the previously described constructs (dNDD or cNDD) to detect and analyze various discrepancies between the two subject graphs, namely the graphs of interrelated unit tests and methods captured by static and dynamic analyzes (section 5.1).

The usage of NDD vectors and curves made it possible to conduct an in-depth analysis of node similarity. We could use these descriptors to define *similarity patterns*. These are sub-graphs of the NSG, which describe the relation of the inspected node in respect of its similarity to other entities.

Researchers could use either the continuous or the discrete version during the analyses to construct these patterns, but they have to take into account the differences between them.

For example consider the *twins pattern* shown in fig. 5.8a. The figure shows a sub-graph of the NSG constructed from two subject graphs. The inspected vertex is always marked with an a. There are two items, one from each graph, represented as green dots and purple squares. The similarity between these two and any other items are zero (they are only similar to eachother). If the magnitude of their similarities, i.e., the weights of edges in the NSG, is equal to one, we call them *identical twins*.

Informally, the identical version of this pattern suggests that these two vertices represent the same item in the two graphs (based on the given similarity function). While in the non-identical case, when the magnitude of similarity is less than one, these represent the "best possible choice," since there are not any other similar items to choose from. The dNDD vectors are unable to distinguish identical and non-identical cases. So, the dNDD



(a) Twins (b) Siblings (c) Fostering

Figure 5.8: Similarity Patterns

vector of these vertices in both case are $\text{DNDD}(a_{\text{id.}}) = \text{DNDD}(a_{\text{non-id.}}) = (1, 0, 0, \dots)$. While CNDD curves are $\text{CNDD}(a_{\text{id.}}) = g(x, 1, 1, 0)^2$ and $\text{CNDD}(a_{\text{non-id.}}) = g(x, \alpha, 1, 0)^2 : 0 < \alpha < 1, x \in \mathbb{R}$ for the identical and the non-identical case, respectively. In these examples x denotes the input value of the function. To better understand this aggregation of a single function, let us inspect the parameters of the underlying characteristic function ($g(\dots)$). For more details see appendix C.3.2.

Magnitude of similarity (second parameter) is the weight of the edge from the inspected node to its lonely neighbor.

Count of similar nodes (third parameter) is expressing the current number of similar nodes of the single adjacent vertex.

Standard distribution of further similarities (fourth parameter) captures the magnitudes of similarity between the single adjacent vertex and its neighbors.[2]

Please note that in this simple case, it could be sufficient to record only these parameters, but the number of properties to capture will grow with the number of neighbors, which demands some kind of aggregation like what we did in the case of CNDD curves.

The definition of meaningful similarity patterns are highly dependent on the domain of the actual problem; however, we can identify the following domain-independent patterns.

Siblings The previously introduced twins patterns are a special case of the *siblings similarity patterns*. In these cases, the inspected vertex's neighbors do not have any more adjacent vertices. An example involving three siblings and one inspected vertex is shown in Figure 5.8b. The DNDD vectors for these patterns are $(3, 0, 0, \dots)$. We distinguish two types of this pattern based on the magnitude of similarity between the inspected vertex and its neighbors. It is called identical when all of its siblings are highly similar to the inspected vertex, and non-identical types when only one of the siblings shows the higher magnitude of similarity.

The interpretation of these patterns are also domain-specific, but identical siblings could indicate that the methods, which produced the two subject graphs have a "different resolution". For a simple example, let us consider two representations of the same file

[2]This is zero because there is not any such vertex.

system, where one uses the full path to identify the files, and the other relies on the name of files. The non-identical type is a possible sign of noise in the subject graphs. For example, if we would build the type hierarchy of the same object-oriented system, but with different tools. One of the tools stores interfaces as properties on the vertex representing the class, while the other uses individual vertices to encode them. There should be some interfaces that are quite similar to the inspected classes.

Fostering There are three main components of the *fostering similarity pattern*. There is the inspected vertex, which shows a low magnitude of similarity to its foster brother or sister, and there are siblings by blood, which are highly similar to the inspected (foster) vertex. This pattern is shown in Figure 5.8c. The DNDD vector for this instance of the pattern is $(2, 1, 0, 0, \dots)$.

The number and connection of adjacent vertices of these three are not defined for fostering similarity patterns. For example, a foster vertex could have more than one step-brother and also more than one blood-brother.

It is exceedingly difficult to give a domain-independent meaning to more complex patterns, but fostering patterns could be an indicator of a poorly chosen similarity function. For example, if we try two pair methods by using all of their attributes, including accessibility (such as private, public, and protected), there should be many methods that are slightly similar to others.

5.3 Comparison of Static and Dynamic Clusterings

In this section, we will elaborate on two practical use cases for the previously described theoretical tool-set. The construction of the UniGDA methodology is based on our previous research of Balogh, G.et al. [21, 44, 45]. We successfully used a similar methodology to analyze discrepancies between two kinds of test cases and their subject method clusterings. Since these experiments preceded the development of a methodology for unified graph's discrepancy analysis, we will show that our methods are the domain-specific versions of the more general UniGDA. The fig. 5.9 shows the informal connection between the components of the theoretical tool-set and their domain-specific implementations.

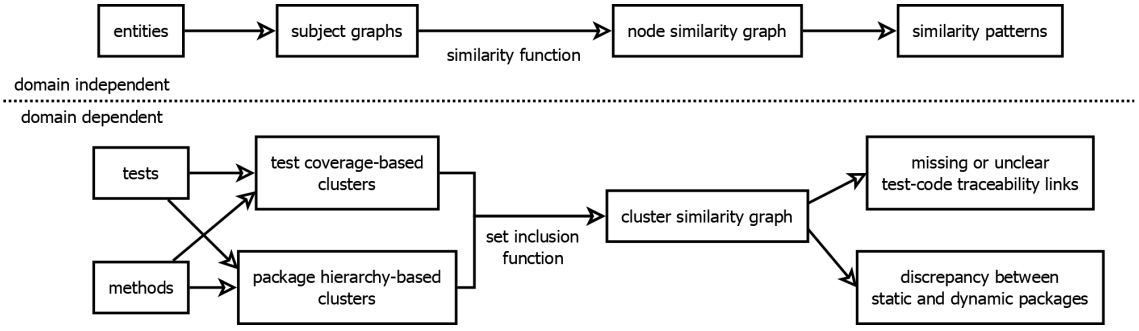


Figure 5.9: Various domain specific implementations of UniGDA

During our experiments [21, 45] we applied a community detection algorithm on a code coverage matrix to detect sets of closely related unit tests and code elements (section 5.1.2). We performed these measurements on real open source systems and compared the identified clusterings with the trivial clustering based on physical code structure (i.e. package hierarchy, section 5.1.1). Finally, we categorized the discrepancies between the two into typical cases, quantified their amount in the subject programs, and provided guidelines to how these can be used as actual bad smells and associated refactorings to improve the structures of existing tests.

We used the *inclusion metric* to asses the similarity of any two clusters. This similarity function was used to construct a weighted, bi-partite graph, where vertices represent clusters, and edges are weighted by their inclusion metric. This *cluster similarity graph (*CSG*)* is a domain-specific version of NSG in the UniGDA methodology. We updated (rephrased and regrouped) some of our previously defined patterns to fit into our current frame of mind. Several patterns are simple domain-specific cases of previously defined similarity patterns, but there are others which only share some of their properties. Since we can use DNDD vectors to detect these, the later ones could also be generalized to find their similarity patterns. The relations among these domain-dependent implementations and the general similarity patterns are shown in fig. 5.10.

Our subject systems were medium to large sized open-source Java programs which have their unit tests implemented using the JUnit test automation framework. Table 5.1 shows some of their basic properties. We chose these systems because they had a reasonable number of test cases compared to the system size. We modified the build processes of the systems to produce method level coverage information using the Clover coverage measurement tool [10]. This tool is based on source-code instrumentation and gives more precise information about source code entities than tools based on bytecode instrumentation ([90]).

Program	tag / hash	LOC	Methods	Tests	P	C
checkstyle	*checkstyle-6.11.1*	114K	2 655	1 487	24	47
commons-lang	*#00fafe77*	69K	2 796	3 326	13	276
commons-math	*#2aa4681c*	177K	7 167	5 081	71	39
joda-time	*v2.9*	85K	3 898	4 174	9	22
mapdb	*mapdb-1.0.8*	53K	1 608	1 774	4	7
netty	*netty-4.0.29.Final*	140K	8 230	3 982	45	35
orientdb	*2.0.10*	229K	13 118	925	130	39
oryx	*oryx-1.1.0*	31K	1 562	208	27	40

Table 5.1: Subject programs and their basic properties

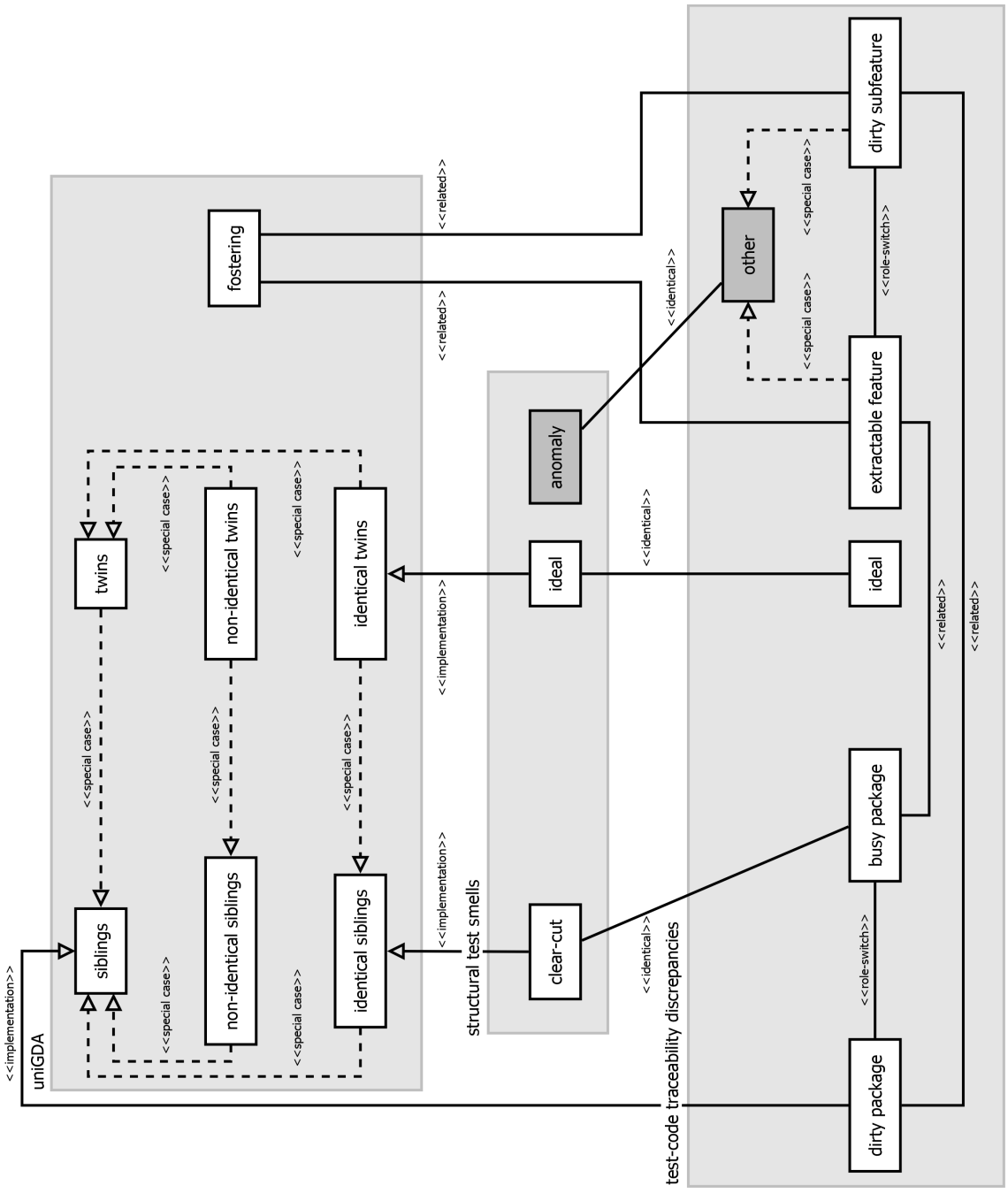


Figure 5.10: Relation among similarity pattern

5.3.1 Classification of Structural Test Smells

The discrepancies found in the results of two clusterings can be seen as some sort of smell, which indicates potential problems in the structural organization of tests and code. For tests that are implemented as executable code, the paper [35] introduced the concept of *test smells*, which indicates poorly designed test code, and listed 11 test code smells with suggested refactorings. We can best relate our work to their *Indirect Testing* smell.

We used our experiences to define the following discrepancy patterns. Our research utilized the previously detailed DNDD and CSG (NSG), which are key concepts of the more general UNIGDA methodology to find and identify these patterns.

Ideal pattern In the ideal case, a single unit that is observable along the test suite structure is also observable from the actual behavior of the tests. The methods and tests are the same in both clusters. Hence they are alternative manifestations of the same entity. The inclusion measures between the two clusters are 1 in either direction in these cases. This pattern will be referred to as *Ideal* in the following. The ideal discrepancy pattern is the domain-specific equivalent of the identical twins similarity pattern.

Clear-cut pattern In the case of this pattern there is one *package based* cluster that consists of more *community based* clusters. In this case, the tests of the single unit are partitioned, and different test cases are testing (covering) different parts of the unit, and the partitions together correspond to the cluster as a whole. This pattern will be called in the following the *Clear-cut* scenario. A clear-cut discrepancy pattern could be interpreted as the domain-specific version of the siblings similarity pattern.

Anomaly pattern We treat anomaly to be present in the clustering comparison when package based clusters do not correspond precisely to a set of associated community clusters as in the case of the Clear-cut pattern. In other words, an anomaly is when a package based cluster does not fully include a community-based cluster. In this case, the remaining elements will be included in other package based clusters. This pattern will be referred to as an *Anomaly*. This scenario does not represent any distinct similarity patterns. It is a collection of unclassified cases.

We automatically searched for the presence of the patterns in our subject systems using a custom script. It examines each package or community cluster and their surroundings on the cluster similarity graph, and using the inclusion measure determines one of the three cases. The corresponding statistics are shown in Table 5.2. The table is divided into three regions, representing the three cases elaborated earlier. For the *Ideal* and *Clear-cut* patterns we base our measurements on the number of package based clusters that are involved in such scenarios, while in the case of the *Anomaly* pattern, the number of affected community clusters will be used. The first column in each region shows the actual number of the given pattern occurrence found in the corresponding subject. In

contrast, *possible occurrences* denotes the total number of clusters of the corresponding type. The last column in each region shows the ratio of these elements. We can observe that, although there are ideal and clear cut scenarios for some programs, most of the clusters – furthermore, in four systems (commons−lang, commons−math, joda−time, and netty) all of them – are involved in anomalies.

smell	Ideal			Clear-cut			Anomaly		
	occurrences	possible occ.	occurrences %	occurrences	possible occ.	occurrences %	occurrences	possible occ.	occurrences %
checkstyle	0	24	0%	1	24	4%	13	45	29%
commons-lang	0	13	0%	0	13	0%	15	275	6%
commons-math	0	71	0%	0	71	0%	15	41	37%
joda-time	0	9	0%	0	9	0%	8	21	38%
mapdb	0	4	0%	0	4	0%	4	7	57%
netty	4	45	9%	1	45	2%	12	34	35%
orientdb	2	130	2%	0	130	0%	13	42	31%
oryx	8	27	30%	4	27	15%	8	39	21%

Table 5.2: The number of cluster comparison patterns in the subject systems

Our goal was not to investigate each pattern occurrence in detail in this phase of the research; instead, we manually checked the systems with the most detected anomalies. This indicates a need for the reorganization of units or the re-implementation of the unit tests, especially for programs like joda−time and mapdb. For example, this manual inspection revealed that joda−time uses a very small number of packages to group the tests and methods. The intention of the developers is encoded into pre- and postfixes of the class names, for example TestDateMidnight_Basics, TestDateMidnight_Constructors and TestDateMidnight_Properties. This information could be expressed by moving the relevant items into different packages. However, to analyze the causal relationship between these properties and the high number of anomalies, further investigation is required.

5.3.2 Clustering of Test-Code Traceability Discrepancies

In our previous work [45], we propose a semi-automatic method, which is based on computing traceability links using static and dynamic approaches, comparing their results and presenting the discrepancies to the user, who will determine the final traceability links based on the differences and contextual information. We define a set of discrepancy patterns, which can help the user in this task. Each pattern describes a setting of related coverage- (C) and package-based (P) clusters with a specific set of inclusion measures, hence specific DNDD vectors as follows. In a sense, this is a more fine-grained classification

of discrepancies between the static (package hierarchy) and dynamic (test-code coverage based) clusterings than the previously described structural test-smells (section 5.3.1).

Ideal Here, the pair of C and P clusters contain the same elements, and there are no other clusters that include any of these elements (the inclusion measures are 1 in both directions). This case is the ideal situation, which shows that there is an agreement between the two clusterings.

Ideal patterns has an DNDD vector of $(1, 0, 0, \dots)$ for both P and C clusters. As noted earlier, these are domain-specific cases of identical twins similarity patterns.

Busy Package This discrepancy describes a situation in which a P cluster splits up into several C clusters, and each C cluster is entirely included in the P cluster.

Busy Package patterns are composed of a P cluster with $(x, 0, 0, \dots)$ DNDD vector, where $x > 1$. These patterns are analogous to the previously defined Clear-cut patterns. Hence they are domain-specific cases of the siblings similarity patterns.

Dirty Packages This pattern is the opposite of the *Busy Package*: one C cluster corresponds to a collection of P clusters, and there are no other clusters involved.

Dirty Packages can be identified in the same way as *Busy Package*, but with the roles of P and C exchanged, hence their DNDD vectors are equal ($(x, 0, 0, \dots)$ DNDD vector, where $x > 1$). They also represent the general siblings patterns like *Busy Package*.

Extractable Feature This pattern refers to a case when there are C clusters that are parts of a pattern that resembles *Busy Package*, but the related P package has some other connections as well, not qualifying the pattern for *Busy Package*.

In the case of *Extractable Feature*, we are looking for P clusters with DNDD vectors of the form $(x, d_1, d_2, \dots)$, where $x \gg d_i, i < 0$, for example $(12, 0, 0, 0, 1, 0, 0, \dots)$, $(21, 1, 2, 2, 0, 0, \dots)$ or $(42, 3, 0, 0, \dots)$. This pattern is related to the general fostering similarity pattern; however, they are not identical. *Extractable Feature* requests that the inspected vertex has more neighbors with no other connection and some adjacent vertices which have several neighbors. While fostering similarity patterns prescribe that the foster brother has only two connections: a weaker and a stronger one. These requirements are neither mutually exclusive, nor are they sub-cases of each-other.

Dirty Subfeature These patterns can be treated as special cases of the *Dirty Packages* pattern. Besides the fully included P clusters, there are semi-included ones, connected to a C cluster, which forms an imperfect *Dirty Packages* pattern.

Dirty Subfeature can be identified in the same way as *Extractable Feature*, but with the roles of P and C exchanged, hence their DNDD vectors are equal ($(x, d_1, d_2, \dots)$, where $x \gg d_i, i < 0$). This pattern is also similar to the fostering similarity pattern, likewise *Extractable Feature*.

Program	*Ideal*pattern count	*Busy Package* count	*Busy Package* C count	*Dirty Packages* count	*Dirty Packages* P count
checkstyle	0	1	{4}	0	
commons-lang	0	0		0	
commons-math	0	0		0	
joda-time	0	0		0	
mapdb	0	0		0	
netty	4	1	{2}	0	
orientdb	2	0		1	{2}
oryx	9	5	{4,4,4,2,2}	0	

Table 5.3: Pattern counts – *Ideal*, *Busy Package*and *Dirty Packages*; columns 'count' indicate the number of corresponding patterns, columns 'C count' and 'P count' indicate the number of C and P clusters involved in each identified pattern

Other The last category is practically the general case when neither of the above more specific patterns could be identified. This typically means a mixed situation and requires further analysis to determine the possible cause and implications.

Finally, we performed a pattern search with the help of the vectors to locate the *Ideal* pattern and the four specific discrepancy patterns, *Busy Package*, *Dirty Packages*, *Dirty Subfeature* and *Extractable Feature* (for the *Other* pattern, we consider all other clusters not present in any of the previous patterns). The second column of Table 5.3 shows the number of *Ideal* patterns the algorithm found for each subject (every instance involves one cluster of each type). As expected, generally, there were very few of these patterns found. But purely based on this result, we might consider the last three programs better in following unit testing guidelines than the other five programs. For instance, one third of the packages in `oryx` include purely isolated and separated unit tests according to their code coverage. These instances can be treated as reliable elements of the final traceability recovery output.

Table 5.3 also shows the number of *Busy Package* and *Dirty Packages* patterns found in the subjects. Columns 3 and 5 count the actual instances of the corresponding patterns, i.e. the whole pattern is counted as one regardless of the number of participating clusters in it. The numbers in columns 4 and 6 correspond to the number of connected clusters in the respective instances. That is, for *Busy Package*, it shows the number of C clusters connected to the P cluster, and in the case of *Dirty Packages*, it is the number of connected P clusters. We list all such connected cluster numbers in the case of `oryx`, which has more than one instance of this type.

The biggest hit was the set of five *Busy Package* instances for `oryx`, and this, together with the nine *Ideal* patterns for this program, leaves only 13 and 15 clusters to be present in the corresponding *Other* categories.

Table 5.4 shows the number of different forms of *Other* discrepancy patterns found. Still, in this case, each participating cluster is counted individually (in other words, each

Table 5.4: Pattern counts – *Other*; columns *P Other* and *C Other* indicate the number of clusters involved in these specific patterns, columns 'all' indicate the number of all involved clusters (including the specific ones)

Program	*P Other* count		*C Other* count	
	all	*Dirty Subfeature*	all	*Extractable Feature*
checkstyle	23	3	43	29
commons-lang	13	1	276	260
commons-math	71	22	39	26
joda-time	9	1	22	14
mapdb	4	0	7	3
netty	40	30	29	17
orientdb	126	48	36	25
oryx	13	6	15	7

cluster is individually treated as one pattern instance). Clusters participating in the *Other* pattern instances are divided into two groups, *P Other* and *C Other*, consisting of the package and coverage cluster elements, respectively. *Dirty Subfeature* and *Extractable Feature* are the two specific subtypes of *Other*, and as explained, the former are subsets of *P Other* clusters and the latter of *C Other* clusters.

5.3.3 Interpretation of cNDD Curves for Clusters Comparison

The DNDD vectors alone are sufficed to describe the above mention patterns. However, we suggest that CNDD curves could be used to improve these results in two major aspects.

Anomalies, *Dirty Packages* and *Dirty Subfeature* scenarios are the most common patterns present in the analyzed systems [21, 45]. Inspecting the magnitude of similarity between clusters could help researchers to distinguish meaningful sub-cases of the *Dirty Packages* and *Dirty Subfeature* scenarios. These could eliminate (or reduce the size of) unclassified, hence not properly analyzed parts of software systems.

Manual analysis of method clusters by the developers could be time-consuming. Usage of CNDD curves could help to reduce the number of these tasks by automatically providing changes to address the already identified issues. For example, in the case of the *non-identical twins similarity pattern*, we can select the most similar cluster to the inspected one and merge any other vaguely similar ones into it. Similarly, in the case of *fostering similarity pattern*, we could identify the foster cluster's true brother, then rearrange items to lower the similarity (inclusion) between the foster and fostering cluster.

Some of these high-level actions could be deduced to source code editing steps, for example, moving test cases and methods into different packages, or removing method calls from the test cases. However, several modifications of clusters cannot be interpreted so easily. In these cases, we could suggest potential steps for the developers to address these issues.

The detailed evaluation and analysis of these automatic source code transformations are out of the scope of this research since our goal is to extract a general methodology for graph comparison.

5.4 Contributions

As stated earlier, our research in discrepancy analysis consists of two major phases: a theoretical one, where we defined a new methodology for unified graph's discrepancy analysis (UniGDA); and a practical one, where we use several underlying concepts of this methodology to present possible solutions for two real-life problems. I contributed to both of these, but undoubtedly I played a more significant role in the construction of the theoretical tool-set.

Introduction of UniGDA In the case of the theoretical phase (which was preceded by our practical investigations), I define a methodology named UniGDA. It serves as a foundation for domain-specific variants, which can aid further research to investigate discrepancies between items, represented as vertices.

- The previous cluster-specific comparison technique was extended for arbitrary graphs.
- Several domain-independent similarity functions were defined based on the properties, the edges, and the neighbors of inspected vertices.
- The characteristic function of similarity patterns was extended to take into account the magnitude of similarity, not just the presence of similar items.
- I suggested various domain-independent similarity patterns, which could be used as prototypes to define other domain-specific ones.

I also rephrased a previously published technique for the mindset of UniGDA to serve as a practical use case.

- We inspected the relation between the domain-specific static and dynamic clustering discrepancy detection algorithm and the more general UniGDA methodology.
- All of the previously defined discrepancy patterns were assigned to one or more general similarity patterns.
- Finally, I provided some ideas on how it could lower the number of unclassified cases by taking account of the magnitude of cluster similarity.

Classification of Structural Test Smells In this work, we addressed a specific type of issue related to unit tests. We seek to automatically uncover violations of two fundamental rules. 1) unit tests should exercise only the unit they were designed for, and 2) they should follow a clear packaging convention. However, I participated during the whole phase; my main role was to define the basic structural patterns of the package hierarchy to be detected. I rely on my developer experiences and the lessons learned from manually inspecting several software systems.

We used these patterns as a basis for further investigations. Our approach was to use code coverage to investigate the dynamic behavior of the tests with respect to the code elements of the program and use this information to identify highly correlated groups of tests and code elements (using community detection algorithm). This grouping is then compared to the trivial grouping determined by package structure, and any discrepancies found are treated as bad smells.

Effectively, we want to compare two *clusterings* of the test cases and code elements: one "as implemented" and the other "as behaving". To address this issue, I defined a data structure, named *cluster similarity graph*, to store containment relations between clusters. The *as implemented* classification is simply the physical structure of the program and test code, organized into language packages. The other classification is derived from the coverage matrix by applying an algorithm to detect such tightly connected groups of tests and code based on their dynamic relationship of code coverage. It will output sets of nodes (mixed tests and code) that are tightly bound together.

In this work, we present our approach to automatically compare these two clusterings by using suitable measures and pinpointing the discrepancies between them. These discrepancies can be thought of as "bad smells", so we also elaborate on the possible refactorings to bring the *as intended* and *as behaving* structures closer together. As it turned out, it is not simple to determine specific parts of the tests that should be refactored and work out how they should be modified.

The concepts above have been empirically investigated on a set of real size open-source Java programs with significant test suites. To summarize, we provide the following contributions.

- I applied a community detection algorithm on a code coverage matrix to detect sets of closely related unit tests and code elements.
- I contributed to the measurements performed on real open source systems, and the comparison of the identified clusterings with the trivial clustering based on physical code structure.
- We categorized the discrepancies between the two into typical cases, quantified their amount in the subject programs, and provided guidelines on how these can be used as actual bad smells and in associated refactorings to improve the structures of existing tests.

Clustering of Test-Code Traceability Discrepancies Recovering test-to-code traceability links may be required in virtually every phase of development. This task might seem simple for unit tests thanks to two fundamental unit testing guidelines: isolation (unit tests should exercise only a single unit) and separation (they should be placed next to this unit). However, practice shows that recovery may be challenging because the guidelines typically cannot be fully followed.

In this work, we present a semi-automatic method for unit test traceability recovery.

In the first phase, we compute the traceability links based on two fundamentally different but very basic aspects: 1) the static relationships of the tests and the tested code in the physical code structure and 2) the dynamic behavior of the tests based on code coverage. In particular, we compute *clusterings* of tests and code for both static and dynamic relationships, which represent coherent sets of tests and tested code. These clusters represent sets whose elements are mutually traceable to each other and maybe beneficial over individual traceability between units and tests, which is often harder to precisely express. For computing the static structural clusters we use the packaging structure of the code (referred to as *package based clusters*), while for the dynamic clustering, we employ community detection [26] on the code coverage information (called the *coverage based clusters*).

In the next phase, these two kinds of clusterings are compared to each other. If both approaches produce the same clusterings, we conclude that the traceability links are reliable. However, in many cases, there will be discrepancies in the produced results, which we report as inconsistencies. There may be various reasons for these discrepancies, but they are usually some combination of violating the isolation and separation principles mentioned above.

The final phase of the approach is then to analyze these discrepancies and, based on the context, produce the final recovered links. During this analysis, it may turn out that there are structural issues in the implemented tests and code, hence refactoring suggestions for the tests or code may be produced as well.

This work is an extension of my previous study, [21], which introduced our concept on structural test smells, which are strongly related to test-code traceability. We extended the previous study with a detailed manual analysis phase, additional discrepancy patterns, and their enhanced detection method using Neighbor Degree Distribution.

5.5 Advantages and Disadvantages of These Methods

5.5.1 Internal Validity

As all theoretical toolsets and frameworks, UniGDA has its advantages and drawbacks. These threats could be associated with the soundness of formalism and to the applicability of the method for real-life problems. In this section, we discuss these issues and suggest a possible solution for them.

Construct Validity

However, our main contribution is to define a unified graph comparison methodology; there is a point where the benefits of a general solution are lower than the cost of creating domain-specific variants. In the case of UniGDA, both the definition of meaningful similarity function and the interpretation of similarity patterns are challenging tasks. These

aspects require in-depth knowledge of the field, which could rarely be addressed by a unified methodology. Our experience suggests that these can be done by manually analyzing several sample cases. It is a well-known fact, that the gathering of expert opinion is a time consuming, hence costly phase. But these tasks have to be completed only once, and subsequent analyses could use these data.

On the other hand, the sets of graph manipulation frameworks are extensive. These libraries and programming languages could be used to implement general frameworks based on the previously introduced UniGDA methodology. For example graph databases like Neo4J [67] and OrientDB [47] could store the analyzed data structures, while domain-dependent knowledge is injected, by using Python callback functions (e.g., various similarity functions).

By using the already defined and tested construct of dNDD vectors (see sections 5.3.1 and 5.3.2), we ensure that UniGDA captures at least some aspects of the relevant information about similarity. However, there could be more ways to extend this construct by choosing different neighbor characteristics or aggregation functions. The evaluation of other extensions will be one of the topics of our future research.

Content Validity

Our method, UniGDA, does not restrict the complexity of the similarity function or the patterns. Still, there is a practical limit on how much information can be encoded into these constructs. We assume that there will always be several aspects of the data sets that similarity functions and patterns will not be able to capture. Hence the user of UniGDA has to prioritize their goals because a poorly chosen similarity function could sabotage the usage of UniGDA. Similarity patterns could lose their meaning if this function fails to encode real life similarity. Moreover, these patterns could overlap, i.e., there are one or more vertices that are part of more than one pattern, which may make further analyses more complicated.

5.5.2 External Validity

There are several threats to validity, which may effect the usage of UniGDA. For example, during the previously discussed use-cases (sections 5.3.1 and 5.3.2), we assume that the developers intended to organize their unit tests in a certain way, which might not hold true for some projects. Furthermore, it can be seen that there are relatively few discrepancy pattern instances in several predefined categories and that the connected cluster numbers are relatively small as well. This suggests that the definitions of some patterns might be too strict because they require a complete inclusion of the connected clusters. For example, currently, in cases where the corresponding pattern is present, but some outliers will currently not be detected. This might be improved in the future by allowing a certain level of tolerance in the inclusion values on the CSG edges, i.e., the similarity values on the

NSG. For instance, by introducing a small threshold value below which the edge would be dropped, we would enable the detection of more patterns in these categories.

Finally, at the present moment, we only know of two interrelated researches [45, 21], which used part of UniGDA directly. These rely on the discrete version of NDD. However, the number of uncategorized cases, like "anomalies" and "others" suggests that there are some unknown, underlying factors in these datasets. We suggest that by taking into account the magnitude of similarity between items (with cNDD curves) researchers could identify these factors. Since we only used these practical examples to construct our domain-independent UniGDA methodology, there could be some unknown aspects that could hinder the practical usability of these techniques. We plan to fine-tune our framework by incorporating lessons learned from future experiments.

Chapter 6

Conclusion

The main results presented in the thesis are related to the semi- or fully-automated analysis of the software and its development processes. My overall research goal was to provide meaningful insights, methods, and practical tools to help the work of stakeholders during various phases of software development. The thesis statements have been grouped into three major thesis points, namely "Measuring, predicting, and comparing the productivity of developer teams"; "Providing immersive methods for software and unit test visualization"; and "Spotting the structures in the package hierarchy that required attention using test coverage data".

Two major issues that need to be addressed during software development from the manager's point of view are: cost prediction and wasted effort handling. During the planning, development, and maintenance of software projects, one of the main challenges is to accurately predict the modification cost of a particular piece of code. Furthermore, several parts of the source code are usually re-written due to imperfect solutions before the code is released. This wasted effort is of central interest to the project management to assure on-time delivery. Both of these issues are related to challenge 3.

The managers could use the two novel metrics (Typed Modification, Modification Effort) and the related methodology (detailed in section 3.1) to get a more accurate measure of the developer's productivity, hence also get a more reliable cost prediction. Moreover, the Division based Micro-Productivity Profile (section 3.2) could provide detailed insight about the wasted resources (and productivity dynamics) for different components and development phases. The managers could use this information to reallocate resources more precisely.

The importance of visualization techniques is undeniable. Diagrams, charts, and other graphical elements are often used to present quantitative and qualitative properties and their relations. These tools use simple and abstract graphical primitives that could not be found in the real world like straight lines, points, and circles. They can express some attributes of the software successfully, but are less useful in presenting more complex many-dimensional contexts. Data visualization with high expressive power plays an important

role in several software development-related activities too. Recent visualization tools try to fulfill the expectations of the users by using various analogies. We think that these unique ways of code representation have great potential. However, in our opinion, they use very simple graphical techniques (shapes, figures, low resolution) to visualize the structure of the source code.

We introduced our novel software visualization tool and its related research (chapter 4). It utilizes an enhanced version of the city metaphor, which provides higher expressive power by allowing the user to display several abstract concepts simultaneously. We have identified two major use cases for our tool (introduced in section 4.1). Exploration tasks of software comprehension consist of the actions that need to be performed to comprehend source code that was written by someone else. We assume that developers need to execute these tasks during their daily routine. See section 4.4.1 for more details. The other potential use of the tool is in education (section 4.4.1). Visual analogies can make learning a lot easier for most of the students.

Source level testing is an integral part of most software quality assurance approaches. Unit tests are often implemented as parts of the source of the system under test, written in the language of the system, and usually with the help of specialized frameworks. Consequently, these tests might be the subject of source code analysis, just as the system code itself. Source code analysis may then be used for various purposes, including test code quality assessment, test comprehension, refactoring, re-documentation, and others. During software analysis, researchers and IT experts often rely on the comparison of datasets. They also frequently draw conclusions based on differences between two representations of the same item's set. Researchers usually use some domain-specific heuristics to asses the similarity between these representations to aid their analysis.

During our research, we address two particular issues, namely the classification of structural test smells (section 5.3.1) and the clustering of test-code traceability discrepancies (section 5.3.2). Developers and testers could use these techniques to check the structure of unit tests and source code items. The information retrieved by these domain-specific versions of methodology for unified graph's discrepancy analysis (UniGDA) could be used as contextual details to restore test-code traceability links. The general methodology was detailed in section 5.2. Several aspects of UniGDA require in-depth knowledge of the field. However, our experience suggests that these can be done by manually analyzing several sample cases. It is a well-known fact, that the gathering of expert opinion is a time consuming, hence costly phase. But these tasks only have to be completed once, and subsequent analyses could use these data, thanks to Methodology for Unified Graph's Discrepancy Analysis.

6.1 Further Works

I do not consider these research topics final and complete. There are several open questions to address and problems to solve.

My productivity measurement methods and profile inspection techniques are based on the fine-grained analyses of the developer activities. The required resolution is usually much higher than the one captured by the various version control systems. One of my future research will investigate the possibility of relying on a system which is already in use and data which are collected by that system (for example, Git).

There are several properties of Micro-Productivity Profile that are not analyzed. For example, the local steepness of these curves could indicate various phases of software development, which may or may not coincide with the rhythm dictated by the project management.

I already used the city metaphor to illustrate the various abstract concepts for students and children. I would like to continue this research by introducing ready-to-use settings and scenarios for various stakeholders. These results will aid the integration of CodeMetropolis into the daily workflow of software development.

In the case of test quality analysis, we plan to investigate the situations in which the violations of clustering indicate the need for refactoring, and whether we should suggest moving test cases to different packages or modify the internal working of the test case instead. This way, we would obtain a real bad smell and refactoring catalog for this particular kind of test code quality issue. Our plans for the continuation also include a more detailed analysis of the anomaly patterns, to define more specific cases.

Finally, I would like to analyze discrepancies between various types of graphs (like, those generated with the Dorogovtsev-Mendes algorithm[37]) with the methodology for unified graph's discrepancy analysis. My assumption is that these investigations will lead to a more comprehensive collection of domain-independent similarity patterns, detected either with Discrete Neighbor Degree Distribution or Continuous Neighbor Degree Distribution.

Chapter 7

Publications

The main results presented in the thesis are related to the semi- or fully-automated analysis of the software and its development processes. My overall research goal was to provide meaningful insights, methods, and practical tools to help the work of stakeholders during various phases of software development. Some of the methods and tools presented in the thesis have been utilized in Hungarian and international R&D projects as well as by the industrial partners of the Software Engineering Department of the University of Szeged.

The thesis result statements have been grouped into three major thesis points, where the author's contributions are clearly shown. The relation between thesis points and supporting publications are shown in table 7.1.

	Thesis point 1				**Thesis point 2**				**Thesis point 3**		
	1.1	**1.2**	**1.3**	**1.4**	**2.1**	**2.2**	**2.3**	**2.4**	**3.1**	**3.2**	**3.3**
[19]	1.1	1.2									
[22]			1.3								
[13]				1.4							
[17]					2.1						
[16]					2.1		2.3				
[15]						2.2					
[18]							2.3				
[20]								2.4			
[21]									3.1	3.2	
[44]										3.2	3.3
[45]										3.2	3.3
[14][1]											3.3

Table 7.1: Thesis contributions and supporting publications

[1] This paper was submitted for publication, but it was not accepted yet

In the following sections I will briefly discuss my publications and their relevance to the previously detailed researches. In addition, I remark that although the results presented in this thesis are my major contribution, the term "we" is used instead of "I" for self reference to acknowledge the contribution of the co-authors of the papers this thesis is based on.

7.1 Measuring, predicting, and comparing the productivity of developer teams

"Prediction of Software Development Modification Effort Enhanced by a Genetic Algorithm"

[19] *Gergő Balogh, Ádám Zoltán Végh, and Árpád Beszédes. "Prediction of Software Development Modification Effort Enhanced by a Genetic Algorithm". In:* SSBSE Fast Abstract track *(2012), pp. 1–6*

During the planning, development, and maintenance of software projects one of the main challenges is to accurately predict the modification cost of a particular piece of code (challenge 3). We experimented with a combined use of product and process metrics to improve cost prediction, and we applied machine learning to this end.

In this paper, we present two new metrics (sub-thesis point 1.1) – to measure productivity more precisely – and a new procedure with which we can increase the effectiveness of our productivity prediction method. Our previous results have been improved with the introduction of new metrics, namely Typed Modification and Modification Effort. Furthermore, we found that by calibrating the free parameters using genetic-algorithms we could achieve an improvement in the F-measure of the prediction model, from about 50% to 70% (sub-thesis point 1.2).

To conduct a preliminary validation, we manually investigated the final parameter values. These parameters seem to be valid based on our own developer experience, but further analysis will be needed to validate the results.

"Identifying wasted effort in the field via developer interaction data"

[22] *Gergő Balogh et al. "Identifying wasted effort in the field via developer interaction data". In:* Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on. *IEEE. 2015, pp. 391–400*

During software projects, several parts of the source code are usually re-written due to imperfect solutions before the code is released. This wasted effort is of central interest to the project management to assure on-time delivery (challenge 3). Although the amount of thrown-away code can be measured from version control systems, stakeholders are more interested in productivity dynamics that reflect the constant change in a software project.

In this paper we present a field study of measuring the productivity of a medium-sized J2EE project. We propose a productivity analysis method where productivity is expressed through dynamic profiles – the so-called Micro-Productivity Profiles (sub-thesis point 1.3). They can be used to characterize various constituents of software projects such as components, phases, and teams. We collected detailed traces of developers' actions using an Eclipse IDE plug-in for seven months of software development throughout two milestones. We present and evaluate the profiles of two important axes of the development process: by milestone and by application layers. Based on the experiments, project stakeholders identified several points to improve the development process.

"Comparison of Software Quality in the Work of Children and Professional Developers Based on Their Classroom Exercises"

[13] *Gergő Balogh. "Comparison of Software Quality in the Work of Children and Professional Developers Based on Their Classroom Exercises". In:* International Conference on Computational Science and Its Applications. *Springer, Cham. 2015, pp. 36–46*

As stated earlier, productivity can be influenced by several factors, one of them is the developer's level of expertise. Both the practical and theoretical knowledge are gathered (among others) during the time spent in some educational institute, like schools and universities. There is also a widely accepted belief that education has a positive impact on the improvement of expertise in software development, But the studies in this topic mainly focus on the product, more precisely the functional requirements of the software. Besides these, they often pay attention to the individual so-called "basic skills" like abstract and logical thinking. We could not find any references where the final products of classroom exercises were compared by using non-functional properties like software quality.

In this paper, we introduce a case study where several children's works are compared to works created by professional developers and not qualified adults. We used a simplified version of the quality model based on the researches at the University of Szeged that conforms to the ISO/IEC 25010 standard and is capable of qualifying the source code of a software system to measure and compare the quality of source code created by students and experts. The subjects of our analysis were distinct solutions of predefined classroom exercises.

The results suggest that there are not any significant differences between the average performance of the two groups based on non-functional properties. These similarities can be explained with the fact that students were guided by an expert i.e. the teacher. On the other hand, the quality of source code produced by experts has less fluctuation (sub-thesis point 1.4). They tend to provide a more stable performance. Outliers can be found in either direction, form the average or median among the solutions of the students. We suggest that these represent the children who have more or less affinity for abstract

thinking and logical problem solving.

In general, we conclude that these data and the results of their analysis suggest some interesting ideas. However, we are aware that this is just a stepping stone for further research.

7.2 Providing immersive methods for software and unit test visualization

"CodeMetropolis-code visualisation in MineCraft"

[17] *Gergő Balogh and Arpad Beszedes. "CodeMetropolis-code visualisation in MineCraft". In:* Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on. *IEEE. 2013, pp. 136–141*

In some cases, developers need to step away from the source code and inspect the system from a different perspective. Our main contribution in this paper was to connect data visualization with high end-user graphics capabilities. To achieve this, a conversion tool was implemented. The tool and its background are connected to sub-thesis point 2.1. It processes the basic source code metrics as input and generates a Minecraft world with buildings, districts, and gardens. The tool was in the prototype state in the year of publication, but it could be used to investigate the possibilities of this kind of data visualization.

We believe that CodeMetropolis will be able to maintain motivation without sacrificing productivity thanks to its intuitive and, for many people, already known graphical surface. Our main goal was to aid the process of software comprehension (challenge 1). The provided metropolis metaphor has enough expressive power to represent the complex items of the source code. Combined with high quality graphical techniques provided by today's computer games, it is capable of offering a rich graphical interface, an easy way of learning the controls, and a rich user experience. It is probably easier to fit in classrooms than in a commercial project. However, we will continue its development to integrate the functionalities which are useful for developers, for students, and for teachers.

"CodeMetrpolis—A minecraft based collaboration tool for developers"

[16] *Gergo Balogh and Arpad Beszedes. "CodeMetrpolis—A minecraft based collaboration tool for developers". In:* Software Visualization (VISSOFT), 2013 First IEEE Working Conference on. *IEEE. 2013, pp. 1–4*

Classical visualization techniques have proven to be useful in many situations, but they fail to maintain the motivation of developers in some circumstances. The provided metropolis metaphor, combined with high quality graphical techniques and the advanced

collaborative features of today's computer games, has enough expressive power to represent the complex items of the source code and hopefully maintain motivation.

In this paper, we introduced our mission to create a virtual world of source code in which developers and other stakeholders could explore and evaluate their project collaboratively in a virtual Minecraft world (sub-thesis points 2.1 and 2.3). Code properties are represented by graphical primitives offered by the game engine. Besides challenges of the implementation, there are some fundamental research issues considering the selection of a set of visual elements and mapping to source code properties. These elements have to be compatible not only with the visualization and with the data model, but also with the way of developers think (challenge 1).

As stated earlier, we created a proof of concept implementation for this metaphor. The current prototype implements various basic functionalities, but the more advanced collaborative features overviewed above will be implemented in the future. Eventually, we want to offer a useful tool in the future, not only for enthusiastic developers who are gamers in their spare time, but also for fulltime developers and managers in the software industry.

In our opinion, software development could be made more interesting and motivating if we united the solid engineering practices and technologies from the industrial segment with the endless fantasy and joy of creation found in games. As one of our developers said: "It makes software metrics such fun that you want to do it."

"Validation of the city metaphor in software visualization"

[15] *Gergő Balogh. "Validation of the city metaphor in software visualization". In:* International Conference on Computational Science and Its Applications. *Springer, Cham. 2015, pp. 73–85*

We live in an age of information explosion where grasping large amounts of data as quickly as possible is a basic requirement. One of the many possibilities is to convert the data into some clear graphical form, such as data that represents elements of a virtual city. In this study, we presented three computable metrics which express various features of such a city. These are compactness for measuring space consumption, connectivity for showing the low-level coherence among the buildings, and homogeneity for expressing the smoothness of the landscape (sub-thesis point 2.2). These metrics were defined in both a formal and an informal way. We also constructed a high-level metric that is able to express the similarity between a generated metropolis and a real one. Both high- and low-level metrics were validated by a user survey. The opinions obtained in the survey were much like as we had anticipated. The results show that it is possible to construct methods which are able to estimate the degree of realism of a generated city. This method embodied as a software-system could provide a full- or semi-automatic way of creating a life-like virtual environment within a reasonable time. In such a world we could use our everyday senses

to perceive the data represented in a clear graphical way (challenge 1).

"CodeMetropolis: Eclipse over the city of source code"

[18] *Gergő Balogh, Attila Szabolics, and Arpád Beszédes. "CodeMetropolis: Eclipse over the city of source code". In:* Source Code Analysis and Manipulation (SCAM), 2015 IEEE 15th International Working Conference on. *IEEE. 2015, pp. 271–276*

The graphical representations of software (code visualization in particular) may provide both professional programmers and students only learning the basics with support in program comprehension (challenge 1). Among the numerous proposed approaches, our research applies the *city metaphor* to the visualization of such code elements as classes, functions, or attributes by the tool CodeMetropolis. It uses the game engine of Minecraft for the graphics, and is able to visualize various properties of the code based on structural metrics. In this work, we presented our approach to integrate our visualization tool into the Eclipse IDE environment (sub-thesis point 2.3). Previously, only standalone usage was possible, but with this new version the users can invoke the visualization directly from the IDE, and all the analysis is performed in the background. The new version of the tool now includes an Eclipse plug-in and a Minecraft modification in addition to the analysis and visualization modules which have also been extended with some new features. Possible use cases and a detailed scenario are presented.

"Using the City Metaphor for Visualizing Test-Related Metrics"

[20] *Gergo Balogh et al. "Using the City Metaphor for Visualizing Test-Related Metrics". In:* 1st International Workshop on Validating Software Tests. *2016*

Understanding the structure of large test suites and the relation of its constituent test cases to the code of a system is hard, and there are not many tools to aid this activity. This work combined two previous approaches: a method to express test quality in terms of metrics, and visualization of code related metrics in the CodeMetropolis framework (sub-thesis point 2.4). The city metaphor employed by CodeMetropolis seems to be useful for test metrics as well, and we believe that the side by side presentation of code and tests will enable the developer to obtain a more global picture of their software (challenge 1).

Currently, the approach has been tried on systems that we developed, about which we have in depth knowledge. In the future we plan to perform additional experiments, possibly involving human evaluation, on other software. Our long term goal is to enhance the metaphor to include additional information sources (such as defects or process data) because we believe that a successful visualization needs to feed from multiple sources.

7.3 Spotting the structures in the package hierarchy that required attention using test coverage data

"Are My Unit Tests in the Right Package?"

[21] *Gergő Balogh et al. "Are My Unit Tests in the Right Package?" In:* Source Code Analysis and Manipulation (SCAM), 2016 IEEE 16th International Working Conference on. *IEEE. 2016, pp. 137–146*

The software development industry has adopted written and de facto standards for creating effective and maintainable unit tests. Unfortunately, like any other source code artifact, they are often written without conforming to these guidelines, or they may evolve into such a state (challenge 2).

This work addressed the quality of unit test suites from a novel angle. Our approach was to compare the physical organization of tests and tested code in the package hierarchy to what could be observed from dynamic behaviour of the tests. The application of community detection algorithms for the latter is a viable approach, and we believe that this kind of analysis of unit tests may reveal knowledge about them not investigated earlier (sub-thesis point 3.1). Our results indicate that for realistic systems, there are a quite lot of discrepancies between the package based and community based structures. But it does not necessarily mean that each of these need to be fixed in the first place by some kind of refactoring of test code. Furthermore, it is not generally possible to decide if there is a problem with the placement of test cases in the package structure or with the way test cases invoke elements of the tested code. Hence any discrepancies found are treated as "bad smells" (sub-thesis point 3.2).

"Analysis of Static and Dynamic Test-to-code Traceability Information"

[44] *Tamás Gergely et al. "Analysis of Static and Dynamic Test-to-code Traceability Information". In:* Acta Cybernetica *23.3 (2018), pp. 903–919*

In this study, we carried out an analysis of test-to-code traceability information. Unit test development has some widely accepted rules that support things like the maintenance of these tests suites. Some of them concern the structural attributes of these tests. These attributes can be described by traceability relations between the test and code. Previous studies demonstrated that fully automatic test-to-code traceability recovery is difficult, if not impossible in the general case. There are several fundamental approaches that have been proposed for this task, based on, among other things, static code analysis, call-graphs, dynamic dependency analysis, name analysis, change history, and even questionnaire based approaches. However, there seems to be general agreement between researchers that no single method can provide accurate information about test and code relations (challenge 4).

Following this line of thinking, we developed a method that is able to detect Structural Unit Test Smells, i.e.locations in the code where unit test development rules are violated. This method foreshadows the definition of a unified comparison methodology related to sub-thesis point 3.3. In particular, we compute test-to-code traceability using two relatively straightforward automatic approaches, one based on the static physical code structure and the other on the dynamic behavior of test cases in terms of code coverage. Both can be viewed as objective descriptions of the relationship of the unit tests and code units, but from different perspectives; hence, each location where they disagree about traceability can be treated as a Structural Unit Test Smell. Our approach is to use clustering and hence form mutually traceable groups of elements (instead of atomic traceability information), and this makes the method more robust because minor inconsistencies will probably not influence the overall results.

Here, we investigated the results of this method applied on four subject programs. Our goal was to manually check the reported Structural Unit Test Smells to see whether at least a part of these are real problems that need to be examined. Experience indicates that most of the reported Structural Unit Test Smells point to parts of the test and code that could be reorganized to better follow unit test guidelines. However, in some situations it might not be worth modifying the tests and the code (e.g. for technical reasons). Overall, we found several typical reasons that could form the basis of future studies and this might lead to an automatic classification of the Structural Unit Test Smells.

These findings have several implications. First, the method has a potential to find Structural Unit Test Smells, but the results will probably contain a large number of false positives (sub-thesis point 3.2). To filter them out, we need to carry out an investigation of the given situation. Fortunately, it seems that there are similar situations that can provide a basis for the automatic classification of the identified smells, and it may assist the developers in their refactoring activities. However, it is also clear from our manual analysis that automatic classification requires additional knowledge (i.e. simply relying on the currently used static and dynamic data is not enough). Furthermore, we found several intricate Structural Unit Test Smell patterns in the CSGs, for which we could not make informed refactoring suggestions because of their complexity and size.

"Differences between a static and a dynamic test-to-code traceability recovery method"

[45] *Tamás Gergely et al. "Differences between a static and a dynamic test-to-code traceability recovery method". In:* Software Quality Journal *(2018), pp. 1–26*

Recovering test-to-code traceability links may be required in virtually every phase of development. This task might seem simple for unit tests thanks to two fundamental unit testing guidelines: isolation (unit tests should exercise only a single unit) and separation (they should be placed next to this unit). However, practice shows that recovery may

be challenging because the guidelines typically cannot be fully followed. Furthermore, previous works have already demonstrated that fully automatic test-to-code traceability recovery for unit tests is virtually impossible in a general case (challenge 4).

In this work, we proposed a semi-automatic method for this task, which is based on computing traceability links using static and dynamic approaches, comparing their results and presenting the discrepancies to the user, who will determine the final traceability links based on the differences and contextual information (sub-thesis point 3.3). We defined a set of discrepancy patterns, which could help the user in this task (sub-thesis point 3.2). Additional outcomes of analyzing the discrepancies were structural unit testing issues and related refactoring suggestions. For the static test-to-code traceability, we relied on the physical code structure, while for the dynamic, we used code coverage information. In both cases, we computed combined test and code clusters which represent sets of mutually traceable elements. We also presented an empirical study of the method involving 8 non-trivial open source Java systems.

"First Steps towards a Methodology for Unified Graph's Discrepancy Analysis"

[14] *Gergő Balogh. "First Steps towards a Methodology for Unified Graph's Discrepancy Analysis". submittted for review to 13th International Conference of Graph Transformation, (part of STAF 2020)*

During software analysis, researchers and IT experts often rely on the comparison of datasets. They also frequently draw conclusions based on differences between two representations of the same item's set (challenge 4). For example, developers may examine the densely connected parts of method call graphs in the context of their location in the package hierarchy tree to find error-prone parts of the system. These kinds of analyses could be aided with a generalized methodology for graphs, which could be used to unify the underlying process of discrepancy analysis. In this paper, we present a *methodology for unified graph's discrepancy analysis,* named UniGDAsub-thesis point 3.3. It is based on the previously defined domain-specific discrepancy detection technique for cluster comparison. Our generalized methodology is using different types of characteristic functions to capture the similarity structures between vertices of arbitrary graphs. We provided several domain independent options for the free parameters of UniGDA. We also presented two possible use cases of UniGDA: the classification of structural test smells and the clustering of test-code traceability discrepancies to showcase the usage of our methodology.

Appendix A

Measuring, Predicting, and Comparing Productivity of Developer Teams

A.1 General Notions and Definitions

In this chapter, we use the following notions and definitions.

Definition A.1.1. *For a given software system we define* $R = \langle r_0, \ldots, r_n \rangle$ *to be the* ***ordered set of revisions*** *of the source code.*

During the experiment, the various modifications were collected to grasp the effort spent by developers.

Definition A.1.2. *A* ***modification*** m *is any difference between any two revisions,* $m \in \text{diff}(r_i, r_j)$ *where* $i < j$*. We assign one from a predefined set of types to each modification, based on the affected source-code element and its affected property if any,* $t(m) \in T$*.*

Definition A.1.3. $\delta_t(r_i, r_j) \in \mathbb{N}$ *is the* ***count of modifications of type*** t*, between the revisions* r_i, r_j*. In other words* $\delta_t(r_i, r_j) = |M|$ *where* $M \subseteq \text{diff}(r_i, r_j)$ *and* $m \in M, t(m) = t$*. Furthermore,* $\Delta(r_i, r_j) \in \mathbb{N}^n$ *is a vector over natural number contains the counts of all predefined modification types between the revisions* r_i, r_j*.*

Definition A.1.4. *Furthermore we use* $\text{devtime}_{r_i \to r_j}$ *to represent the* ***net development time*** *between* r_i *and* r_j *revisions, where* $i < j$*.*

A.2 Formal Definition of Modification Effort and Typed Modification

In this section, we give a formal definition of the underlying metrics used to measure productivity during our research.

Definition A.2.1. *We use the previously introduced notions to define the* ***Typed Modification (*TMod*)*** *metric. Let* $w = (w_0, w_1, \ldots, w_{|T|})$ *vector of weight for each possible type of modification. For each revision pair* $r_i, r_j \in R$ *we define as follows.*

$$\text{TMod}(r_i, r_j) = \sum_{k=0}^{|T|} w_k \Delta(r_i, r_j)_k \tag{A.1}$$

Definition A.2.2. *The definition of* ***Modification Effort (*MEff*)*** *is the following for each revision pair* $r_i, r_j \in R$ *we define as follows.*

$$\text{MEff}(r_i, r_j) = \frac{\text{TMod}(r_i, r_j)}{\text{devtime}_{r_i \to r_j}} \tag{A.2}$$

We used MEff to express and measure productivity between two-state (revision) of the analyzed system.

$$\text{productivity} = \frac{\text{output}}{\text{input}} = \frac{\text{gain}}{\text{effort}} = \frac{\text{TMod}(r_i, r_j)}{\text{devtime}_{r_i \to r_j}} = \text{MEff}(r_i, r_j) \tag{A.3}$$

A.3 Determining the Weights of Modification Groups

The table A.1 shows the parameters used with the genetic algorithm to fine-tune the weights of the TMod metric.

Table A.1: GA parameters

initial mutation rate	100%
mutation rate	50%
mutation lower limit	0.5
mutation upper limit	100
birth count	2 child per evolution step
crossover rate	2 crossover per evolution step
population size	200 individuals
generation count	50 generation

During the evaluation, the weight of groups was aggregated from all four inspected projects and weighted with the size of the learning set. We used two aspects to examine the validity of the weights calculated by the GA.

These aspects are shown in fig. A.1. The values can be interpreted as the "importance" of modification, i.e., how much gain will be achieved by applying the modification. The diagram on the left shows an aggregation by action. As can be seen, the creation and deletion are "more important" than the type and visibility changes. On the right side, a subject-based aggregation can be seen. The "most important" modification was applied to the method elements, which included the method body modifications as well.

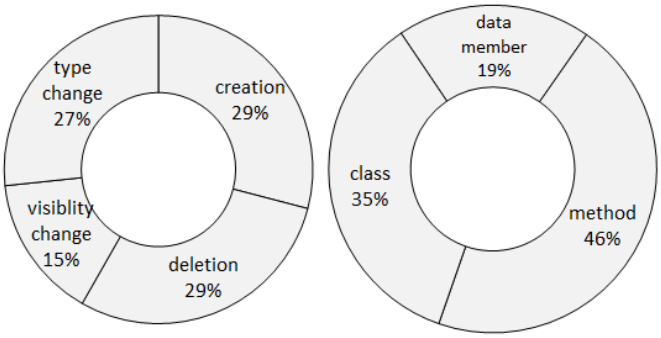


Figure A.1: Aggregated weights of groups

A.4 Formal Definitions of Division based Micro-Productivity Profile

In this section, we specify MPPD and the kind of data used to compute it more formally.

An equal distance division was used to determine points of comparison.

Definition A.4.1. *We define the* ***equal distance divisions*** *for an ordered set as a list of indices:*

$$j = \left\lfloor i \cdot \frac{n}{d+1} \right\rfloor$$

Where $n \in \mathbb{N}$ *is the number of revisions,* $i = 0, \ldots, d+1$ *is the index of parts and* $d \in \mathbb{N}$ *is a predefined number of divisions.* $R_i^{(d)}$ *is also used to simplify further definitions, which is the* i^{th} *revision of the equal distance division with* d *dividing point.*

Definition A.4.2. *Productivity* $P_i^{(d)}$ *for a given equal distance division is*

$$P_i^{(d)} = \frac{\Delta(R_i^{(d)}, R_{i+1}^{(d)})}{\text{devtime}_{R_i^{(d)} \to R_{i+1}^{(d)}}}$$

.

Definition A.4.3. *The* ***division based micro-productivity profile*** *is defined as a function over natural numbers,* $\text{mppd} : \mathbb{Z} \to \mathbb{Q}$*. It assigns the sum of all productivity values for a given equal distance division:*

$$\text{mppd}(x) = \sum_{i=0}^{x+1} P_i^{(x)}$$

Notice that in a perfect world the mppd is a constant function, $\text{mppd}(i) = \text{mppd}(0)$; however in real-life software development it is always increasing ($\text{mppd}(i) \leq \text{mppd}(i+1)$) because of re-written code. Productivity values may incorporate wasted effort, so a higher $P_i^{(d)}$ value does not necessarily mean better overall productivity.

A.5 mppds of Analyzed Project

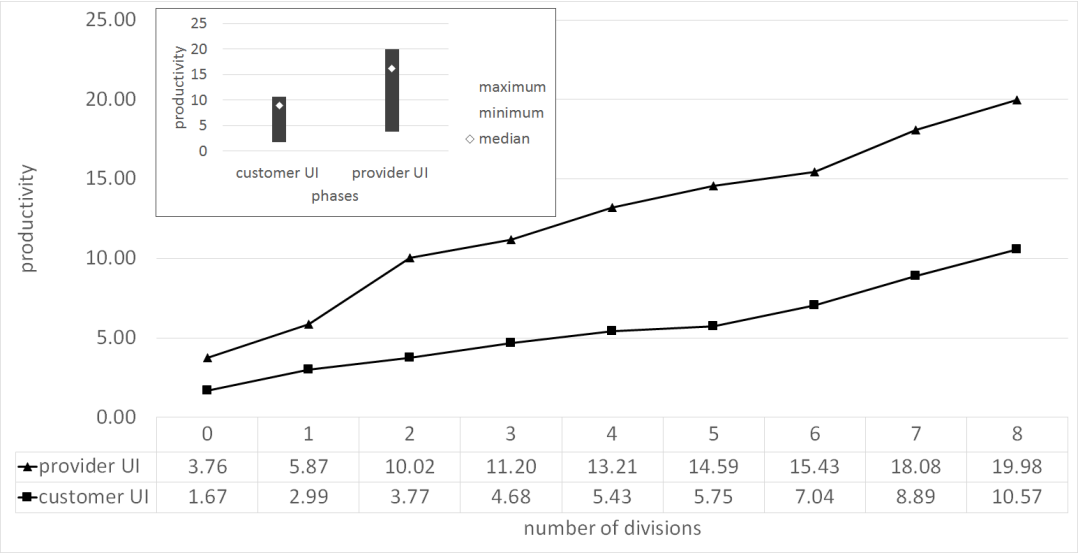
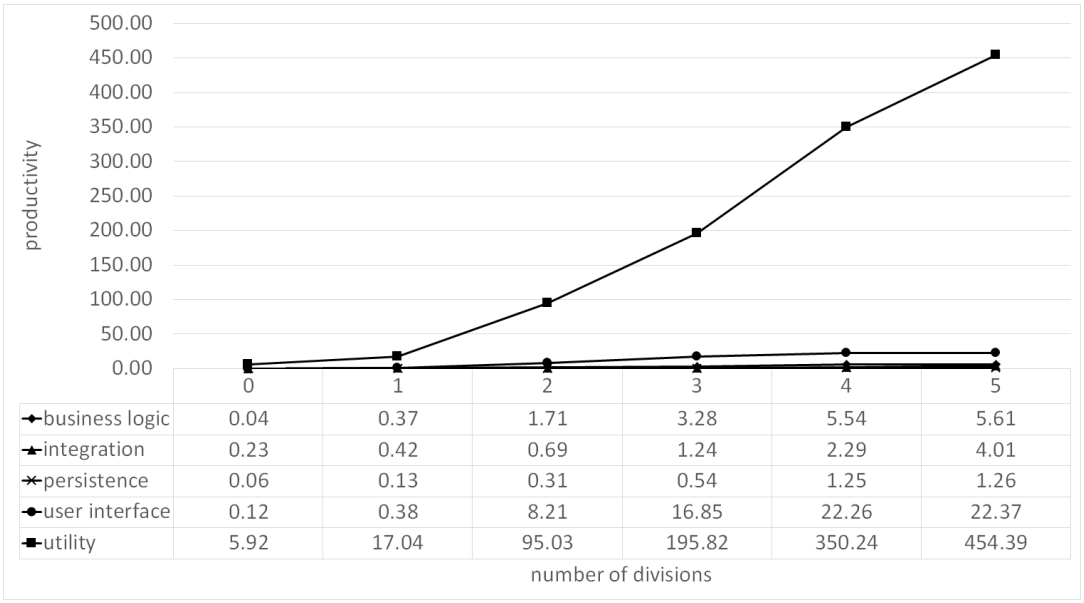
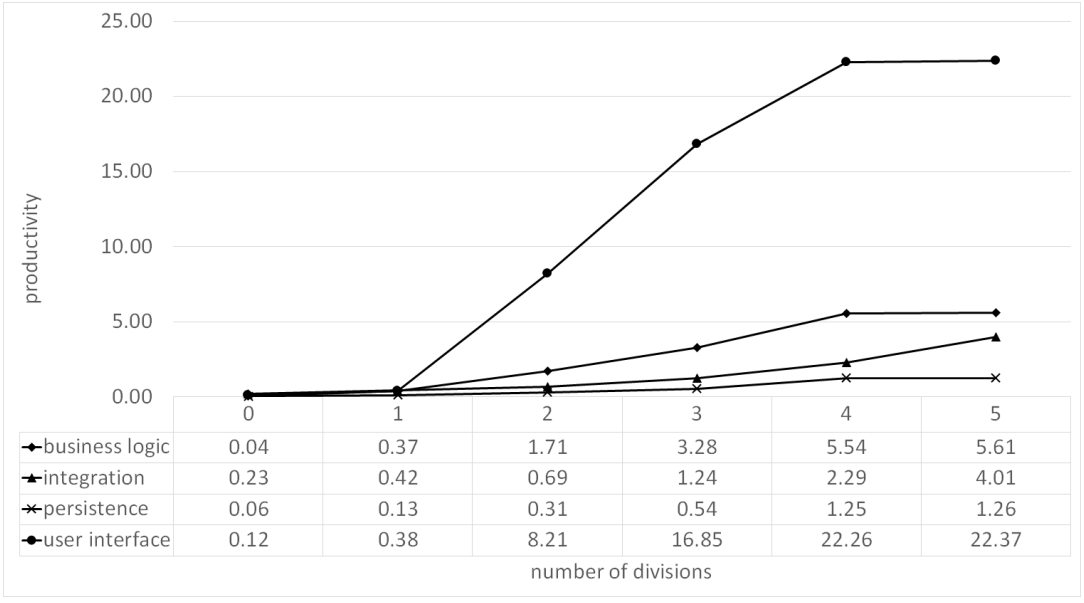


Figure A.2: MPPD over development phases



	0	1	2	3	4	5
business logic	0.04	0.37	1.71	3.28	5.54	5.61
integration	0.23	0.42	0.69	1.24	2.29	4.01
persistence	0.06	0.13	0.31	0.54	1.25	1.26
user interface	0.12	0.38	8.21	16.85	22.26	22.37
utility	5.92	17.04	95.03	195.82	350.24	454.39

Figure A.3: MPPD over application layers (all)



	0	1	2	3	4	5
business logic	0.04	0.37	1.71	3.28	5.54	5.61
integration	0.23	0.42	0.69	1.24	2.29	4.01
persistence	0.06	0.13	0.31	0.54	1.25	1.26
user interface	0.12	0.38	8.21	16.85	22.26	22.37

Figure A.4: MPPD over application layers (exluding utility)

Appendix B

Providing Immersive Methods for Software and (Unit) Test Visualization

B.1 CodeMetropolis Technical Details

The current version of CodeMetropolis uses the following entities and attributes to visualize the source code. These items are highlighted on Figure B.1 and their properties are listed in appendix B.1. We do not force any predefined mapping between the metrics of source code items and the visual properties of graphical items, which allows CodeMetropolis to be useful for various stakeholders.

(a) Elevated grounds to group items

(b) Gardens with various flower-ratio

(c) Buildings surrounded with gardens

(d) Floors with various materials

Figure B.1: Items of the metaphor level

Ground It can group various types of entities, including other grounds. It is usually used to display the namespace hierarchy like a tree-map. In the generated world, it is displayed as a solid rectangle of stone blocks. Its width and length were adjusted automatically to fit its contents.

Garden They are similar to grounds. It is commonly used to represent individual classes. It is displayed as a plate of grass blocks surrounded by fences.

House A house is another compound entity. It consists only of floors and cellars which are placed on the top of each other ordered according to their width and length. The converter uses this entity to group floors and cellars; however, it has no meaning on the data level.

Floor Floor is a hollow box with lattice, which is located over the ground level It usually represents a single method.

Cellar They are the underground equivalents of floors. They commonly stand for data members of the classes.

attribute	**type**	**targets**	**description**
width	integer	floor, cellar	size along X-axis
height	integer	floor, cellar	size along Y-axis
length	integer	floor, cellar	size along Z-axis
character	string	floor, cellar	primary material of the structure
external_character	string	floor, cellar	secondary material of the structure
torches	integer (0 to 5)	floor, cellar	quantity of torches
flower-ratio	float (0 to 1)	garden	quantity of flowers
tree-ratio	float (0 to 1)	garden	quantity of trees
mushroom-ratio	float (0 to 1)	garden	quantity of mushrooms

Table B.1: Graphical attributes of items

B.2 Metrics for Generated Cities

B.2.1 Low-level Metrics

We will use the following formalism in the rest of this chapter. Let us define the buildings as a tuple with six items and the collection of these as an unordered set.

$$\mathbb{B} = \{buildings\} \tag{B.1}$$

$$b \in \mathbb{B} \tag{B.2}$$

$$b = \begin{pmatrix} x_b & x_b & z_b \\ |x|_b & |y|_b & |z|_b \end{pmatrix} \tag{B.3}$$

$$D \subseteq \mathbb{B}d \qquad \in \mathbb{B} \tag{B.4}$$

where

$$(x_b, y_b, z_b) \in \mathbb{N}^3 \text{ is a predefined pivot point of the building} \tag{B.5}$$

$$|x|_b, |y|_b, |z|_b \in \mathbb{N} \text{ is the width, the length and the height of the building} \tag{B.6}$$

$$|\hat{z}|_b = \frac{|z|_b}{\max_{d \in \mathbb{B}} |z|_d} \text{ is the normalized height of the building} \tag{B.7}$$

We will define the distance between any two buildings as the Euclidean distance between their pivot points, and we will also use the convex hull of a set of buildings, Conv D.

$$\|b; d\| = \|(x_b; y_b); (x_d; y_d)\| \in \mathbb{R} \tag{B.8}$$

The following notation will be used to denote some basic properties of the buildings.

$$A_b = |x|_b \cdot |y|_b \text{ is the area of the building} \tag{B.9}$$

$$A_D \in \mathbb{R} \text{ is the area of the convex hull of buildings in } D \tag{B.10}$$

$$P_D \in \mathbb{R} \text{ is the perimiter of the convex hull of buildings in } D \tag{B.11}$$

We define a classification over the set of buildings; this is the type of the building. The type of building is given with the following relation. It is equal to 1 if and only if two buildings are of the same type.

$$t(b), t(d) \in \mathbb{N} \text{ is the type of the buildings} \tag{B.12}$$

$$\delta(b; d) = \begin{cases} 1 & \text{if } t(b) = t(d) \\ 0 & \text{otherwise} \end{cases} \tag{B.13}$$

Compactness

$$\operatorname{Comp} D = \frac{A_D}{\sum\limits_{d \in D} T_d} \tag{B.14}$$

Compactness could be defined as the ratio of the area of the convex hull of a set of buildings (i.e. the convex hull of the set of points of the buildings) over the total area of these buildings. Because our model does not allow any intersecting buildings, the lower limit will be 0, and the upper limit will be 1.

Homogeneity

$$\underset{\gamma}{\operatorname{Conn}} D = \frac{1}{\binom{|D|}{2}} \sum_{\substack{d,b \in D \\ d \neq b}} \left(\underbrace{\delta(d;b)}_{\text{connection guard}} \underbrace{\left(\frac{\left\| d;b \right\|}{\max\limits_{e,f \in D} \left\| e;f \right\|} \right)^{\gamma}}_{\text{distance part}} \right) \tag{B.15}$$

Homogeneity is specified as the arithmetic mean of buildings weighted with the difference of their normalized height. As in the case of connectivity, an overall normalization is added to ensure that the values are bounded. A gamma correction is applied to the height difference and the distance part as well, to be able to fine-tune its sensitivity.

Connectivity

$$\underset{\gamma,\zeta}{\operatorname{Hom}} D = \frac{1}{\binom{|D|}{2}} \sum_{\substack{d,b \in D \\ d \neq b}} \left(\underbrace{\left| |\hat{z}|_d - |\hat{z}|_b \right|^{\zeta}}_{\text{height delta part}} \underbrace{\left(\frac{\left\| d;b \right\|}{\max\limits_{e,f \in D} \left\| e;f \right\|} \right)^{\gamma}}_{\text{distance part}} \right) \tag{B.16}$$

Connectivity is defined as the sum of normalized distances between each pair of buildings. We introduced a connection guard and a final normalization part. With this formula, we only charge a fee for the buildings with the same type. The size of the fee is greater if the buildings are farther away from each other. Gamma correction was applied to the distance part, to be able to fine-tune its sensitivity.

Appendix C

Using Test Coverage to Analyze Structures in the Package Hierarchy

C.1 Formal definitions of methodology for unified graph's discrepancy analysis

Definition C.1.1. *Let $G = (V_G, E_G)$ and $H = (V_H, E_H)$ be the two subject graphs and their vertices and edges sets respectively. In this case, we have to compare the pairs $(g, h) : g \in V_G, h \in V_H$. We capture the node level similarity with the following* ***similarity function.***

$$G = (V_G, E_G) \qquad H = (V_H, E_H) \tag{C.1}$$

$$g \in V_G \qquad h \in V_H \tag{C.2}$$

$$\tag{C.3}$$

$$\mathbb{R} = \{real\ numbers\} \tag{C.4}$$

$$g \sim h \in [0; 1] \subset \mathbb{R} \qquad similarity\ function \tag{C.5}$$

C.1.1 Domain Independent Similarity Functions

We adopt several notions from set theory to construct these functions. Let us define the set of properties for each node and for each edge as $\mathcal{P}(x)$, where $x \in V$ or $x \in E$. Furthermore, let $p \in \mathcal{P}(x)$ and let $p(x)$ denote the current value of p on x.

To simplify our notion, we define the union and the intersection of any two arbitrary property sets $(P_x = \mathcal{P}(x), P_y = \mathcal{P}(y))$.

$$P_x \cap P_y = \{p \mid p \in P_x \wedge p \in P_y \wedge p(x) = p(y)\} \tag{C.6}$$

$$P_x \cup P_y = \{p \mid p \in P_x \vee p \in P_y\} \tag{C.7}$$

As a final step we could use (normalized) set comparison metrics [102], for example the Jaccard similarity index [92].

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{C.8}$$

Vertex-property Based Similarity We could use the previously explained notions to construct the following *domain independent vertex-property similarity function.* Let v_i and v_j be any two vertex.

$$v_i \overset{\text{d.i.v}}{\sim} v_j = J(\mathcal{P}(v_i), \mathcal{P}(v_j)) \tag{C.9}$$

The prerequisite condition for this similarity function is that there should be an equivalence operator over the set of property values of vertices.

Edge-property Based Similarity Using the aggregation (for example, arithmetic mean) of edge similarity, we can define the following *domain independent edge-property similarity function.* Let $E(v)$ be the set of edges of the v vertex.

$$v_i \overset{\text{d.i.e}}{\sim} v_j = \underset{\substack{e_i \in E(v_i) \\ e_j \in E(v_j)}}{\text{aggregation}} J(\mathcal{P}(e_i), \mathcal{P}(e_j)) \tag{C.10}$$

The prerequisite condition for this similarity function is that there should be an equivalence operator over the set of property's values of edges, and a meaningful aggregation over the edge similarities should be specified, which could be easily interpreted by experts of that domain.

Adjacent Vertex's Similarity-Based Similarity Both of the previously described similarity functions could be applied to all pairs of vertices adjacent to the inspected ones. Using a proper aggregation function over the similarities of the adjacent vertices we can construct two kinds of *domain independent adjacent vertex's similarity-based similarity functions.* Let $V(v)$ be the set of adjacent vertices of v (if $v' \in V(v)$ then there is an edge $\overrightarrow{(v; v')}$ in the graph).

$$v_i \overset{\text{d.i.a.v}}{\sim} v_j = \text{aggregation } v_i' \overset{\text{d.i.v}}{\sim} v_j' \quad v_i' \in V(v_i) \quad v_j' \in V(v_j) \tag{C.11}$$

$$v_i \overset{\text{d.i.a.e}}{\sim} v_j = \text{aggregation } v_i' \overset{\text{d.i.e}}{\sim} v_j' \quad v_i' \in V(v_i) \quad v_j' \in V(v_j) \tag{C.12}$$

Both of these similarity functions inherit the prerequisite condition of their underlining similarity computation method. Furthermore, we have to choose a meaningful aggregation for the similarities of the adjacent vertices.

Compound, Property-Based Similarity All of the previously described functions could be used to construct a *compound, property-based similarity function* with a properly chosen aggregation, for example, a normalized, weighted, arithmetic mean. Let w_t be the weight of the similarity of type t.

$$T = \{\text{d.i.v}, \text{d.i.e}, \text{d.i.a.v}, \text{d.i.a.e}\} \tag{C.13}$$

$$v_i \overset{\text{d.i.}}{\sim} v_j = \frac{\sum_{t \in T} w_t \cdot (v_i \overset{\text{t}}{\sim} v_j)}{\sum_{t \in T} \alpha_t} \tag{C.14}$$

This compound similarity function inherits all previously mentioned prerequisite conditions.

C.2 Node Similarity Graph

Definition C.2.1. *Let $G = (V_G, E_G)$ and $H = (V_H, E_H)$ the graphs to be compared and their set of vertices and edges, respectively. Using this notion, the **Node Similarity Graph** is a directed,[1] bipartite graph $S = (V_S, E_S)$, where its set of vertices contain the vertices of the two subject graphs ($V_S = V_G \cup V_H$). There is an edge connecting $g \in V_G$ and $h \in V_H$ if and only if the similarity of these vertices is greater than zero ($g \sim h > 0$). Edges are weighted with the degree of similarity between their endpoints ($w(\overrightarrow{(g,h)}) = g \sim h$).*

[1] Note that our methodology does not require that $g \sim h = h \sim g$.

C.3 Neighbor Degree Distribution

C.3.1 Discrete Neighbor Degree Distribution

Definition C.3.1. *The general definition of the* **dNDD *vector*** *is the following. Let* $G = (V_G, E_G)$ *be a graph and let* $\deg(x)$ *denote the degree of vertex* x*.*

$$a \in V_G \text{ the inspected vertex} \tag{C.15}$$

$$\overrightarrow{(a;b)} \in E_G \tag{C.16}$$

$$\deg(b) = \left|\left\{c : w(\overrightarrow{(b;c)}) > 0\right\}\right| \tag{C.17}$$

$$\text{DNDD}(a) = (d_1, d_2, \ldots, d_n) \tag{C.18}$$

$$d_i = \left|\left\{a : w(\overrightarrow{(a;b)}) > 0 \wedge \deg(b) = i\right\}\right| \tag{C.19}$$

C.3.2 Continuous Neighbor Degree Distribution

In this section, we summarize the formal notions related to CNDD curves and their definition. We are using $\sigma(x)$ to denote the standard deviation of x.

Definition C.3.2. *We define the* $\boldsymbol{g(x, h, o, w)}$ ***function*** *based on the well-known Gaussian function.*

$$g(x, h, o, w) = \begin{cases} he^{-\frac{(x-o)^2}{2w^2}} & if\, w \neq 0 \\ g'(x, h, o) & otherwise \end{cases} \tag{C.20}$$

$$g'(x, h, o) = \begin{cases} h & if\, x = o \\ 0 & otherwise \end{cases} \tag{C.21}$$

The parameter h is the height of the curve's peak, o is the position of the center of the peak, and w controls the width of the bell. We modified the original definition of Gaussian function in such a way that $g'(x, h, o, 0)$ represents an infinitely narrow bell. We used this modified version of the function to characterize each neighbor (b_i) of the inspected node.

Definition C.3.3. *We define* ***Continuous Neighbor Degree Distribution (*cNDD*) curves*** *as an aggregation of the following neighbor characteristic functions.*

$$\text{cNDD}(a, x) = \sum_{i=0}^{n} p\left(b_i, x\right)^2 \tag{C.22}$$

$$p\left(b_i, x\right) = g(x, \alpha_i, n^{(i)}, \sigma(\beta^{(i)})) \tag{C.23}$$

Informally, the Gaussian function of various edge weights is used to characterize the

current neighbor of the inspected vertex. Then these functions are aggregated using the sum of squared values.

Formal construction and properties of cNDD curves

In this section, we will give a detailed description of this definition and its properties.

Note that the above described dNDD vectors are unable to express the differences between the weights of edges. Hence, they are not encoding any information on the magnitude of similarity. In other words, during the calculation we discard the weights of NSG by using the $w(\overrightarrow{(a;b)}) > 0$ constraint.

The loss of information could be expressed with the following formalism. Let $\mathrm{sgn}(G)$ be a simplified version of the original G graph, where we discard the weight of edges using the following method. (sgn is a function that extracts the sign of a real number.)

$$\operatorname{sgn} x = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x = 0 \\ -1 & \text{if } x < 0 \end{cases} \tag{C.24}$$

$$e \in E_G \qquad \text{an edges of the orignal graph} \tag{C.25}$$

$$\mathrm{sgn}(e) \in E_{\mathrm{sgn}(G)} \qquad \text{an edge in the simplified graph} \tag{C.26}$$

$$w(\mathrm{sgn}(e)) = sign(w(e)) \tag{C.27}$$

The weight of edges in the simplified graph is either 0 or 1. In the discrete case, NDD yields the same result for the original and the simplified graphs as well.

We use a similar method to KDE to encode this lost information into a cNDD curve, which can capture information about the degree of similarity between nodes. The definition of these curves is easier to understand if we follow our steps during the research to construct these descriptors.

At first, we identified two main points where the definition of the dNDD vectors could be extended.

$$\left(\Xi_{i=0}^{n}\,\alpha_i \;\middle|\; \Xi_{j=0}^{n^{(i)}}\,\beta_j^{(i)} = k\right)_k = q_{i=0}^{n}\left(p\left(\beta_i, k\right)\right) \tag{C.28}$$

$$\Xi_{i=0}^{n}\,x_i = \sum_{i=0}^{n} \mathrm{sgn}(x_i) \tag{C.29}$$

The rephrased definition of the dNDD vector is presented on the left side. If all weights are greater or equal to zero, then $\sum_{i=0}^{n} \mathrm{sgn}(\alpha_i)$ is equal to the number of edges with a non-zero weight of the inspected vertex. We used the same operator (Ξ) to calculate the degree of these neighbors, excluding zero weighted edges. The constrains $\Xi_{j=0}^{n^{(i)}}\,\beta_j^{(i)} = k$ is

used to "group the neighbors" by a specific degree.

Generally, the neighbors are characterized by a feature function (conditional count of their adjacent vertices), and these characteristics are aggregated to compute the DNDD vector (summation). These two sub-functions are denoted with $p(\dots)$ and $q(\dots)$, respectively. If we extend the domain and the range of these functions to the real numbers, we can rephrase the definition as shown on the right side of the formula.

Our primary (extension) requirement was that the CNDD curves have to yield the same result in the same position for the simplified graph as the DNDD vectors for the original one.

$$a \in V_G \tag{C.30}$$

$$\operatorname{sgn}(a) \in V_{\operatorname{sgn}(G)} \tag{C.31}$$

$$\text{CNDD}(\operatorname{sgn}(a), x) = \Big(\text{DNDD}(a)\Big)_x \qquad x \in \mathbb{N} \tag{C.32}$$

This requirement ensures that the CNDD curves are real extensions of DNDD vectors since they only store more information. The implicit effect of ignoring the weight of edges (which are encoded into the definition of DNDD vectors) are simulated by using the simplified graph.

In the last step of the CNDD curve construction, we evaluated several aggregation functions. We seek a function that does not violate the extension requirement like, for example, arithmetical mean. Our choice fell on the previously mentioned sum of squared values, since it preserves more information than, for example, the more common maximum function.

To summarize, the previously identified extension points of NDD functions are the following.

$$p\left(b_i, x\right) = \alpha_i e^{-\frac{\left(x-n^{(i)}\right)^2}{2\sigma\left(\beta^{(i)}\right)^2}} \qquad \text{neighbor characteristic} \tag{C.33}$$

$$q_{i=0}^{n} x_i = \sum_{i=0}^{n} x_i^2 \qquad \text{aggregation} \tag{C.34}$$

By choosing these extension points as previously described, we get the definition of the CNDD curves (definition C.3.3) presented earlier in this section.

Appendix D

Summary in English

In this chapter, I aim to accomplish the most challenging task as a Ph.D. student: squeezing many years of research and results into a few intriguing pages. The success of this endeavor can only be measured by the willingness of the reader to inspect the details presented in earlier parts of this thesis. I encourage everybody to do so since those are the chapters where you could find answers for your "Why?"'s and "How?"'s.

D.1 Summary of the Topics

The main results presented in the thesis are related to the semi- or fully-automated analysis of the software and its development processes. My overall research goal is to provide meaningful insights, methods, and practical tools to help the work of stakeholders during various phases of software development. The thesis statements have been grouped into three major thesis points, namely "Measuring, predicting, and comparing the productivity of developer teams"; "Providing immersive methods for software and unit test visualization"; and "Spotting the structures in the package hierarchy that required attention using test coverage data".

D.1.1 Measuring, Predicting, and Comparing Productivity of Developer Teams

This part mainly dwelt with the concept of productivity and the challenges of cost prediction related to it.

Using types of modifications to define more expressive productivity metric (effort made by developers). During my research, I presented [19] two new metrics to measure productivity. My goal was to eliminate common shortcomings of currently used ones, namely, they were unable to capture fine-grained productivity changes and distinguish the amount of effort taken by the developer in certain situations. These highly customizable source code metrics provide more expressive power than the commonly used

number of logical line based versions. The productivity measurement methodology was evaluated with middle size industrial systems, and the results were validated with the lead developers of the projects.

Predicting and fine-tuning productivity metrics enhanced by a genetic algorithm. I was able to increase the effectiveness of the previous modification cost prediction method based on product and process metrics. The traditionally used changed lines of code metric were replaced by the previously mentioned weighted count of modified source code entities. With this, I was able to increase the success of the prediction model significantly. I used machine learning algorithms, namely genetic and evolution algorithms, to fine-tune the free parameters. During the empirical evaluation phase, four industrial projects were analyzed, and the accuracy of the predictions was compared to previous results. I found that my productivity estimation model can achieve significant improvement in the overall efficiency of the prediction, from around 50% to 70% (F-measure).

Identifying wasted effort via developer interaction data. Although the amount of throw-away code can be measured from version control systems, stakeholders are more interested in productivity dynamics that reflect the constant change in a software project. In a field study, we analyzed this aspect of productivity in a medium-sized J2EE project [22] with 17 developers for seven months. We proposed a productivity analysis method where productivity is expressed through dynamic profile – the so-called Micro-Productivity Profile (MPP). They can be used to characterize various constituents of software projects, such as components, phases, and teams. These properties let the management fine-tune the schedule of the project and aid the leaders in reassigning resources to the most sensitive tasks. Based on the experiments, project stakeholders identified several points to improve the development process.

Comparison of software quality in the work of children and professional developers. I conducted a case study where several children's work was compared to works created by professional developers by using non-functional properties like software quality. The model used to measure the various aspects of software quality, also known in the industrial sector; hence, it provides a well-established base for our research.

The subjects of my analysis were distinct solutions for predefined classroom exercises. The results[13] suggest that there are not any significant differences between the average performance of the two groups. These similarities can be explained by the fact that an expert, i.e., the teacher guided students. On the other hand, the quality of source code produced by experts had less fluctuation. They tend to provide more stable performance. Outliers can be found in either direction from the average or median among the solutions of the students. I suggest that these represent the children who have more or less affinity for abstract thinking and logical problem-solving.

D.1.2 Providing Immersive Methods for Software and (Unit) Test Visualization

The main topic of this part is the visualization of software systems and their connected items. It addresses the challenges of software comprehension.

Using a sandbox game to visualize software as a virtual city My main contribution was to connect data visualization with high end-user graphics capabilities. To achieve this, a visualization tool [17, 16] was implemented which utilized the high expressive power of the well known sandbox game, Minecraft [2].

In CodeMetropolis, different physical properties of the city and the buildings are related to various code or test metrics. I continued the legacy of CodeCity [105] and EvoSpace [59] which use the analogy of skyscrapers in a city to represents the structure of the source code. Despite their appealing appearance and great potential in general, these tools still use relatively low fidelity graphics compared to today's most advanced computer games. I introduced our approach for visualizing source code using the same metaphor, but employing a sophisticated game called Minecraft. The set of visual properties was extended to support higher dimensional data visualization since the software systems could reach virtually infinite complexity by their nature.

Our tools were utilized during various educational sessions, ranging from primary to secondary schools. We also use this method to help the university students to understand abstract software development related concepts like source code metrics.

Assessing the degree of realism for the city metaphor in software visualization To allow the users to navigate freely in the artificial environment and to understand its meaning, the difference between a realistic and unrealistic city has to be expressed. I presented three low- and one high-level metrics that express various features of a virtual city used to visualize software systems to capture this difference. These are compactness for measuring space consumption, connectivity for showing the low-level coherence among the buildings, and homogeneity for expressing the smoothness of the landscape. The constructed high-level metric can express the similarity between a generated metropolis and a real one.

Both high- and low-level metrics were validated by a user survey. The results show that it is possible to construct methods that can estimate the degree of realism of a generated city.

Integration of Eclipse and CodeMetropolis We presented an approach to integrate our visualization tool into the Eclipse IDE environment. Previously, only standalone usage was possible, but with this integrated version, the users can invoke the visualization directly from the IDE, and all the analysis is performed in the background. The new version of the tool now includes an Eclipse plug-in and a Minecraft modification in addition

to the analysis and visualization modules, which have also been extended with some new features.

These tools enable developers to launch visualization and initialize the buildings of the virtual city. We also described two possible, high-level use cases and detailed scenarios, for educational and professional usage.

Using the city metaphor for visualizing test-related metrics We extended the metaphor to include properties of the tests related to the program code using a novel concept [20]. The test suite and the test cases were associated with a set of metrics that characterize their quality (such as coverage and specialization), which allowed us to combine two previous approaches: a method to express test quality in terms of metrics, and visualization of code related metrics in the CodeMetropolis framework.

In this version of CodeMetropolis [18], gardens representing code elements will give rise to outposts that characterize properties of the tests and show how they contribute to the quality of the code.

D.1.3 Using Test Coverage to Analyze Structures in the Package Hierarchy

This part contains a detailed elaboration of the researches related to test and code quality measurement and improvement, addressing the challenges of quality management and software analysis.

Simultaneous Clustering of Test Cases and Code Elements To automate various tests and code analyses tasks, I employ a clustering algorithm that can group test and code items. In order to determine the clustering of the tests and code based on the dynamic behavior of the test suite, I applied *community detection* [26, 41] on the detailed test-code coverage information. Groups of tests and methods that form "dense regions" may be grouped, indicating that there is a tight correspondence between them from a dynamic point of view. This method allowed the simultaneous inspection of tests and their subjects and aided us in conducting further analysis.

Classification of Structural Test Smells This work addressed the quality of unit test suites from a novel angle. Our approach was to compare the physical organization of tests and tested code in the package hierarchy to what can be observed from the dynamic behavior of the tests.

Our results indicate that for realistic systems, there are quite a lot of discrepancies between the package-based and community-based structures. However, it does not necessarily mean that each of these needs to be fixed in the first place by some kind of refactoring of test code. Furthermore, it is not generally possible to decide if there is a

problem with the placement of test cases in the package structure or with the way test cases invoke elements of the tested code.

Clustering of Test-Code Traceability Discrepancies We proposed a semi-automatic method for recovering test-to-code traceability links, which is based on computing connections using static and dynamic approaches, comparing their results and presenting the discrepancies to the user, who will determine the final traceability links based on the differences and contextual information.

We defined a set of discrepancy patterns, which can help the user in this task. Additional outcomes of analyzing the discrepancies were structural unit testing issues and related refactoring suggestions. For the static test-to-code traceability, we relied on the physical code structure, while for the dynamic, we used code coverage information, as mentioned in previous paragraphs. In both cases, we computed combined test and code clusters, which represent sets of mutually traceable elements. We also presented an empirical study of the method involving eight non-trivial open-source Java systems.

Providing a Methodology for Unified Graph's Discrepancy Analysis During software analysis, researchers and IT experts frequently draw conclusions based on differences between two representations of the same item's set, like the above mentioned dynamic and static clustering of the tests and their subjects. These kinds of analyses could be aided by a generalized methodology for graphs, which could be used to unify the underlying process of discrepancy analysis. I presented a methodology for a unified graph's discrepancy analysis, named UniGDA. It is based on the previously defined domain-specific discrepancy detection techniques.

My generalized methodology is using different types of characteristic functions to capture the similarity structures between vertices of arbitrary graphs. I extended the previous detection technique to arbitrary graphs by providing several domain-independent similarity functions and pattern. All of the previously defined discrepancies were assigned to one or more general similarity patterns. These results ensure that the UniGDA methodology does not reduce the number of identifiable cases. I also introduced a continuous version of the characteristic and aggregation function, which takes account of the magnitude of similarity between inspected items, instead of only being able to express the existence of similar ones.

D.2 Future Work

I do not consider these research topics final and complete. There are several open questions to address and problems to solve.

My productivity measurement methods and profile inspection techniques are based on the fine-grain analyses of the developer activities. The required resolution is usually much

higher than the one captured by the various version control systems. One of my future research will investigate the possibility of relying on a system which, is already in use and data which are collected by that (for example, Git).

There are several properties of Micro-Productivity Profile that are not analyzed. For example, the local steepness of these curves could indicate various phases of software development, which may or may not coincide with the rhythm dictated by the project management.

I already used the city metaphor to illustrate the various abstract concept for students and children. I would like to continue this research by introducing ready-to-use settings and scenarios for various stakeholders. These results will aid the integration of CodeMetropolis into the daily workflow of software development.

In the case of test quality analysis, we plan to investigate the situations in which violations of clustering indicate the need for refactoring, and whether we should suggest moving test cases to different packages or modify the internal working of the test case instead. This way, we would obtain a real bad smell and refactoring catalog for this particular kind of test code quality issue. Our plans for the continuation also include a more detailed analysis of the anomaly patterns, to define more specific cases.

Finally, I would like to analyze discrepancies between various types of graphs (like, those generated with Dorogovtsev-Mendes algorithm[37]) with the methodology for unified graph's discrepancy analysis. I plan that these investigations will lead to a more comprehensive collection of domain-independent similarity patterns, detected either with Discrete Neighbor Degree Distribution or Continuous Neighbor Degree Distribution.

Appendix E

Magyar nyelvű összefoglaló

Ebben a fejezetben a Ph.D. tanulmányaim egyik legnehezebb feladatát tűztem ki célul: hogyan sűrítsük bele sok év kutató munkáját néhány figyelemfelkeltő oldalba. A sikeresség emet csak azzal az egyetlen ténnyel lehet mérni, hogy a tisztelt olvasó hajlandó-e továbblapozni és megismerni, az e tézisben korábban leírtakat. Személy szerint mindenkit erre bíztatok, hiszen ezekben a korábbi fejezetekben fogják megtalálni a válaszokat a ,,Hogyan?"-okra és a ,,Miért?"-ekre.

E.1 Témák összefoglalása

E dolgozat fő eredményei kapcsolódnak a részben vagy egészben automatizált program elemzéshez és a fejlesztési folyamatokhoz. A célom az volt, hogy hasznos eszközökkel, módszerekkel és technológiákkal segítsem a különböző szoftverfejlesztéssel foglalkozó szakemberek munkáját. A téziseimet három nagy csoportra osztottam: ,,A szoftverfejlesztői csapatok produktivitásának mérése és előrejelzése"; ,,Izgalmas és magával ragadó szoftver és teszt vizualizációs technikák biztosítása"; és ,,Figyelmet érdemlő helyek azonosítása a csomaghierarchiában lefedettségi adatok alapján".

E.1.1 A szoftverfejlesztői csapatok produktivitásának mérése és előrejelzése

E rész a produktivitás fogalma és hozzá kapcsolódó költségbecslés köré csoportosuló kutatásaimat ismerteti.

A módosítások típusainak felhasználása egy kifejezőbb produktivitás metrika definiálása során. A kutatásom során [19] két új metrikát definiáltam a produktivitás mérésére. Ezek lehetővé tették, hogy kiküszöböljem a korábban használt változatok hátrányait. Pontosabban, a korábbi megoldások nem voltak képesek különbséget tenni az alacsony szintű produktivitás változások során, vagyis nem tudták kifejezni a fejlesztők által tett erőfeszítések kis léptékű változásait. Az általam bevezetett metrikák nagy-

obb kifejező erővel rendelkeznek mint a korábban használt sorok számosságán alapuló változatok. Az új metrikákat közepes méretű, ipari projekteken értékeltük ki és az eredményeket összevetettük a vezető fejlesztő által adott becslésekkel.

Produktivitás mértékének előrejelzése genetikus algoritmus segítségével finomhangolt metrikák alapján. Sikeresen növeltem a korábbi folyamat és termék metrikákon alapuló módosítási költség előrejelző modell teljesítményét. A tradicionálisan használt módosított sorok száma alapú metrikákat a korábban említett módosítások súlyozott összegére cseréltem. Ezzel a változtatással jelentősen megnövekedett a korábbi előrejelző modell sikeressége. A szabad paraméterek finomhangolásához gépi tanulási algoritmust, név szerint evolúciós és genetikai algoritmust használtam. Az empirikus kiértékelés során négy ipari projekten végeztünk méréseket és hasonlítottuk össze a kapott pontosságot a korábbi modellek értékeivel. A kutatás során kimutattam, hogy az általam kifejlesztett előrejelző modell teljesítménye jelentősen, mintegy 50%-ról 70%-ra (F-mérték) nőtt átlagosan.

Az elvesztegetett erőforrások azonosítása fejlesztők munkavégzési adatai alapján. Bár a többszörösen (vagyis legalább egyszer feleslegesen) módosított forráskód mennyisége megbecsülhető a verzió kezelő rendszerek segítségével, mégis a vezetők számára fontosabb a részletesebb és pontosabb erőforrásbecsélt lehetővé tevő a produktivitás időbeli változását vizsgáló mérések. Ezt az aspektust vizsgáltuk egy tanulmány során [22], melyet egy közepes méretű, J2EE projekten végeztünk 17 fejlesztő bevonásával hét hónapon keresztül. A produktivitás változásának elemzésére dinamikus profilokat vezettünk be, melyeket Mikro-Produktivitás Profiloknak (MPP) neveztünk. Ezek a görbék lehetővé teszik a szoftver fejlesztési projekt különböző elemeinek jellemzését, úgy mint a komponensek, fejlesztési fázisok, és csapatok. Ezáltal a projekt vezető megfelelőbb időrendet alakíthat ki és lehetővé válik az erőforrások pontosabb kiosztása. A kísérlet alapján a vizsgált projekt szakmai vezetője több javításra érdemes pontot is azonosított.

Diákok és szakemberek kódminőségének összehasonlítása. Egy általam végzett tanulmány keretében több, diákok által készített programkód minőségét vetettem össze tapasztalat fejlesztők megoldásaival. A vizsgálat során az ipari szektorban is elfogadott modellt alkalmaztam, mely lehetővé teszi különböző nem-funkcionális tulajdonságok kiértékelését és aggregálását.

Az elemzéseim tárgya különálló órai feladatok megoldásai voltak. Az eredményekből arra következtettem, hogy nincs számottevő különbség a két csoport átlagos teljesítménye között. Ezek a hasonlóságok részben magyarázhatóak azzal, hogy míg a gyerekek tanári felügyelet mellett dolgoztak addig a fejlesztők önállóan oldották meg a feladatot. A fejlesztők által végzett munka minősége sokkal kisebb szórást mutatott az átlag körül. A diákok között talált kirívó esetek, véleményem szerint a személyes affinitáshoz és képességekhez

kapcsolódó különbségekkel magyarázhatóak, úgy mint az absztrakt gondolkodás és a logikus probléma megoldás.

E.1.2 Izgalmas és magával ragadó szoftver és (egység) teszt vizualizációs technikák biztosítása

E rész fő témája a szoftverek és hozzá kapcsolódó elemek vizualizációja köré csoportosul. A kutatás célja, hogy válaszokat adjon a szoftverek megértésével kapcsolatos kihívásokra.

Nyílt-terű játékok felhasználása a szoftverek virtuális városként történő megjelenítése során. Ebben a fázisban nagy hangsúlyt kapott az adat vizualizáció összekapcsolása valósághű és magával ragadó grafikus megjelenítéssel. Ennek elérése érdekében egy új eszköz csomag [17, 16] került kifejlesztésre, mely lehetővé teszi, hogy a program tulajdonságait Minecraft [2] világban generált városokkal reprezentáljuk.

A CodeMetropolis-nak nevezett rendszerben a generált város különböző elemei egy vagy több forráskód vagy teszt elemet jelképeznek, míg ezek fizikai tulajdonságai a megjelenített elem metrikáit fejezik ki. A program a korábban a CodeCity [105] és az EvoSpace [59] során már felhasznált város metafora módszerét alkalmazza. E kapcsolódó programok, bár képesek megjeleníti a szoftverek elvont szerkezetét, vizuális ábrázolásuk és interaktivitásuk messze elmarad a közismert számítógépes játékok szintjétől. A kutatásaim során e jól ismert vizualizációs módszert ötvöztem a modern grafikai lehetőségekkel egy népszerű játékon keresztül. A megjeleníthető tulajdonságok kibővített halmaza, lehetővé tette, hogy több, magasabb dimenziójú adatot is reprezentáljunk, melyek gyakoriak a szoftver rendszerek elemzése során.

A módszert és az eszközt sikeresen alkalmaztuk különböző oktatási platformok és események keretében, kezdve az általános iskolai szakkörtől a rendhagyó informatika órán át a középiskolákban tartott pályaválasztási napokig. A technológia integrálásra került az egyetemi oktatás során is, ahol segítette a hallgatókat az elvont fogalmak és köztük lévő kapcsolatok megértésében.

A generált városok valósághűségének mérése a szoftver vizualizáció során. Ahhoz hogy a felhasználó akadálytalan navigációját biztosítsuk egy virtuális térben, szükségünk van a generált világ valósághűségének mértékére. Kutatásaim során három alacsony és egy magas szintű metrikát definiáltam, melyek kifejezik a generált városok bizonyos jellemzőit. Ezek a kompaktság, ami a város térbeli kiterjedését méri; az összekapcsoltság, ami az épületek közötti kis-léptékű koherenciát fejezi ki; és a homogenitás, ami a látkép folytonosságát jelzi. Ezen alacsony szintű mérőszámok aggregálásával kapjuk a valósághűséget kifejező magas szintű metrikát.

Mind a négy metrika egy felhasználókkal végzett tanulmány keretében került ellenőrzésre. Az eredmények arra engednek következtetni, hogy lehetséges olyan metrika konstruálása, mely becslést ad egy generált város valósághűségére.

Eclipse fejlesztő környezet és CodeMetropolis szoftvervizualizációs eszközök integrációja. Ebben a fázisban egy új módszert mutattunk be, mely lehetővé tette a közismert Java nyelvű fejlesztéseket támogató rendszer, az Eclipse, és a korábban részletezett városmetaforát használó szoftvervizualizációs program csomag, a CodeMetropolis együttes használatát. A korábbi független használati esetekkel ellentétben, jelen verzió biztosítja, hogy a felhasználó navigálhasson a forráskód és az azt jelképező város elemei között. Ezeket a funkciókat egy Eclipse plug-in és egy Minecraft mod biztosítja.

Az eszközök és a hozzá kötődő módszer kiértékelése során több különböző ipari és oktatási fejlesztéshez kapcsolódó használati esetet is meghatároztunk.

Város metafora használata a teszt metrikák vizualizációja során. Ebben a fázisban kiterjesztettük a korábban használt város metaforát, mely ezáltal képessé vált különböző teszt metrikák megjelenítésére is [20]. A korábban a teszt esetekhez és a hozzájuk kapcsolódó kód elemekhez rendelt minőséget kifejező metrikák (mint a lefedettség és a specializáció) lehetővé tette számunkra, hogy a tesztek adatait megjelenítsük a korábban használt virtuális térben, mely a programok szerkezetét jelképezi.

A CodeMetropolis programcsomag ezen verziójában a forráskód elemeket jelképező kerteket őrposztok népesítik be, melyek tulajdonságai kifejezik a hozzájuk kapcsolódó teszt esetek minőségét több különböző nézőpontból.

E.1.3 Teszt lefedettség használata a csomaghierarchia szerkezetének vizsgálata során

Ebben a fejezetben részletezem a teszt és forráskód minőség javítása érdekében végzett kutatásaim eredményét, különös hangsúlyt fektetve a minőségmérés és a szoftver elemzés által támasztott kihívásokra.

Teszt és forráskód együttes klaszterezése. A különböző teszt és forráskód elemzések automatizálása érdekében, bevezettem egy új módszert mely képes a teszt és forráskód elemek együttes csoportosítására. A tesztek és a hozzájuk kapcsolódó kódrészletek futás közbeni viselkedése alapján ún. dinamikus csoportosítást hoztunk létre. Ehhez a végrehajtás során gyűjtött részletes lefedettségi adatokat közösség detektáló algoritmus használatával elemeztük. Az ilyen algoritmusok egy csoportba foglalják azokat az elemeket, melyek között nagyságrendileg több kapcsolat található mint a csoporton kívül. Az így kapott heterogén elemhalmazok vizsgálata lehetővé teszi, hogy a tesztek minőségét a hozzájuk kapcsolódó kódrészletekkel együtt elemezzük.

Gyanús struktúrájú teszt csomagok osztályozása. A kutatásunk során újszerű nézőpontból vizsgáltuk az egység tesztek minőségét. A megközelítésünk a fizikai csoportosítás és a futás közben tapasztalt viselkedés összehasonlításán alapszik.

Az eredményeink azt mutatták hogy a valós ipari projektek esetében jelentős eltérés figyelhető meg a csomag hierarchia és a lefedettség alapján detektált teszt és kód klaszterek között. Fontos hangsúlyozni, hogy ezen eltérések nem feltétlenül eredményeznek hibás viselkedést, vagy jelentenek nem megfelelő megvalósítást, azonban mindenképp kitüntetett figyelemmel kell bánni velük. A kutatásaink során végzett fél-automatikus elemzések rámutattak, hogy nem lehet általános javítási lépéseket megfogalmazni, vagyis az adott típusú gyanús struktúrák egymástól függetlenül kezelendők.

Teszt és kód összerendelés során tapasztalt eltérések klaszterezése. A munkánk során létrehoztunk egy fél-automatikus módszert, mely képes bizonyos mértékben helyre állítani az elveszett teszt-kód nyomon-követhetőségi kapcsolatokat. Ez a módszer a statikus és dinamikus elemzések közötti különbségek alapján lokalizálja a figyelmet érdemlő pontokat, azonban a helyreállítást már a fejlesztő végzi.

A felhasználó munkáját különböző eltéréseket osztályozó minták definiálásával segítjük. A statikus elemzés során jelentős információt szerzünk a program csomaghierarchiájából, míg a dinamikus adatokat a részletes lefedettség mérések szolgáltatják. A nyomon-követhetőségi kapcsolatokat az ezen adatok alapján konstruált teszt és kód klaszterek jelképezik. Az elméleti kutatást nyolc valós programon végzett mérés során értékeltük ki.

Egységesített módszer a gráfok közötti eltérés vizsgálatára. A kutatók, munkájuk során gyakran vonnak le következtetéseket egy adott elemhalmaz kétféleképpen végzett csoportosításának elemzéséből. Erre egy példa a korábban bemutatott teszt és kód elemek klaszterezése statikus és dinamikus analízis alapján. Kutatásom célja, hogy segítsük ezeket az elemzéseket, egy olyan általános összehasonlító módszertan definiálásával, mely képes a gráfok közötti eltérések vizsgálatára. Ezt az új módszert UniGDA-nak neveztem el, és a korábban bemutatott speciális eseteken alapszik.

Ez az általános módszertan különböző típusú függvények segítségével detektálja az egyes csomópontok közötti hasonlóságot. A korábban bevezetett szakterület specifikus különbségeket vizsgáló módszertant kiterjesztettem több terület-független hasonlósági függvénnyel és mintával. Az így kapott módszer lehetővé teszi tetszőleges gráfok közötti különbségek elemzését. Minden korábban definiált mintát hozzárendeltem egy vagy több általános mintához, mely biztosítja, hogy az UniGDA nem csökkenti a korábbi módszerekkel vizsgált elemtípusok számát. Sőt a folytonos karakterisztikus függvény bevezetésével lehetővé vált az elemek közötti hasonlóság mértékének mélyebb tanulmányozása, a korábbi, csak hasonló elemek létezésének vizsgálata helyett.

E.2 Jövőbeli tervek

Az eddig elért eredmények ellenére sem tekintem a kutatómunkámat lezártnak és teljesnek, természetesen még számtalan nyitott kérdés maradat, melyek közül néhányat említek a

továbbiakban.

Az általam bevezetett produktivitás értékét mérő és profilját vizsgáló technika a fejlesztői aktivitás nagy felbontású elemzésén alapszik. A szükséges felbontás általában jóval nagyobb mint az ipari szektorban is használt verziókövető rendszerek esetében. Egyik jövőbeli tervem, hogy a korábbi módszereket átültessem a gyakorlatban is széles körben használt és így sokkal több valós adatot biztosító verzió követő rendszerekre, mint például a Git.

A korábban részletezett produktivitást leíró profilok számos tulajdonságát figyelmen kívül hagytuk az elemzési folyamat egyszerűsítése érdekében. Ilyen tulajdonságok például a görbe meredeksége, mely segítséget nyújthat a fejlesztési projektek ütemezésének tervezése során.

Az oktatás során felhasznált város metafora alapú szoftver vizualizáció sikerei alapján, szeretném kiterjeszteni ezt az absztrakt fogalmak megismerését támogató módszert a szakemberek munkájára is. Ezt elsődlegesen a különböző területek számára előre elkészített és kiértékelt speciális beállítás csomaggal tervezem támogatni.

A tesztek minőségelemzésének területén további vizsgálatokat tervezünk a lehetséges javítások szükségességének megállapítására. Ezek alapján támogatni tudjuk majd a fejlesztőket annak a kérdésnek a megválaszolása során, hogy mely komponensek átszervezése és átalakítása szükséges. Ez lehetővé teszi teljes hibalokalizációs és javítási javaslatokat biztosító katalógus létrehozását. Ezzel párhuzamosan tervezzük a nem osztályozott esetek részletes vizsgálatát és további minták definiálását.

Végül, tervezem a különböző gráftípusok közötti különbségeket és ezek eloszlását vizsgálni a korábban bemutatott UniGDA módszerrel. Véleményem szerint az így gyűjtött adatok még részletesebb szakterület-független minta adatbázis építését teszik majd lehetővé, mind a folytonos és a diszkrét leírók esetében is.

Bibliography

[1] Mojang Synergies AB. *Minecraft End User Licence Agreement*. URL: `https://account.mojang.com/documents/minecraft_eula` (visited on 10/07/2015).

[2] Mojang Synergies AB. *Minecraft Official Website*. URL: `http://minecraft.net/` (visited on 06/24/2013).

[3] MK K Abdi, Hakim Lounis, and Houari Sahraoui. "Using Coupling Metrics for Change Impact Analysis in Object-Oriented Systems". In: *Proceedings of the 10th ECOOP Workshop on Quantitative Approaches in ObjectOriented Software Engineering QAOOSE 06*. 2006, pp. 61–70. URL: `http://www-ctp.di.fct.unl.pt/QUASAR/Resources/Papers/2006/QAOOSE2006%5C_Proceedings.pdf%5C#page=65`.

[4] Roberto Abreu and Rahul Premraj. "How developer communication frequency relates to bug introducing changes". In: *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*. ACM. 2009, pp. 153–158.

[5] Ernest Adams and Andrew Rollings. *Fundamentals of Game Design (Game Design and Development Series)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006. ISBN: 0131687476.

[6] Nicolas Anquetil and Timothy Lethbridge. "Extracting concepts from file names: a new file clustering criterion". In: *Software Engineering (ICSE 1998), Proceedings of the 20th International Conference on*. IEEE Computer Society. 1998, pp. 84–93.

[7] Gábor Antal, Ádám Zoltán Végh, and Vilmos Bilicki. "A methodology for measuring software development productivity using Eclipse IDE". In: *Proceedings of the 9th International Conference on Applied Informatics (ICAI 2014)*. Accepted. Eger, Hungary, 2015.

[8] Gábor Antal et al. "Static JavaScript Call Graphs: a Comparative Study". In: *Proceedings of the 18th IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2018, pp. 177–187.

[9] Thomas H. Apperley. "Genre and Game Studies: Toward a Critical Approach to Video Game Genres". In: *Simul. Gaming* 37.1 (Mar. 2006), pp. 6–23. ISSN: 1046-8781. DOI: 10.1177/1046878105282278. URL: http://dx.doi.org/10.1177/1046878105282278.

[10] Atlassian. *Clover Java and Groovy Code Coverage Tool Homepage*. last visited: 2016-05-27. URL: https://www.atlassian.com/software/clover.

[11] Steven P. Den Baars and Sander Meester. "CodeArena: Inspecting and Improving Code Quality Metrics in Java using Minecraft". In: 2019.

[12] Tibor Bakota et al. "A probabilistic software quality model". In: *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, Sept. 2011, pp. 243–252. ISBN: 978-1-4577-0664-6. DOI: 10.1109/ICSM.2011.6080791. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6080791.

[13] Gergő Balogh. "Comparison of Software Quality in the Work of Children and Professional Developers Based on Their Classroom Exercises". In: *International Conference on Computational Science and Its Applications*. Springer, Cham. 2015, pp. 36–46.

[14] Gergő Balogh. "First Steps towards a Methodology for Unified Graph's Discrepancy Analysis". submittted for review to 13th International Conference of Graph Transformation, (part of STAF 2020).

[15] Gergő Balogh. "Validation of the city metaphor in software visualization". In: *International Conference on Computational Science and Its Applications*. Springer, Cham. 2015, pp. 73–85.

[16] Gergo Balogh and Arpad Beszedes. "CodeMetrpolis—A minecraft based collaboration tool for developers". In: *Software Visualization (VISSOFT), 2013 First IEEE Working Conference on*. IEEE. 2013, pp. 1–4.

[17] Gergő Balogh and Arpad Beszedes. "CodeMetropolis-code visualisation in MineCraft". In: *Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on*. IEEE. 2013, pp. 136–141.

[18] Gergő Balogh, Attila Szabolics, and Arpád Beszédes. "CodeMetropolis: Eclipse over the city of source code". In: *Source Code Analysis and Manipulation (SCAM), 2015 IEEE 15th International Working Conference on*. IEEE. 2015, pp. 271–276.

[19] Gergő Balogh, Ádám Zoltán Végh, and Árpád Beszédes. "Prediction of Software Development Modification Effort Enhanced by a Genetic Algorithm". In: *SSBSE Fast Abstract track* (2012), pp. 1–6.

[20] Gergo Balogh et al. "Using the City Metaphor for Visualizing Test-Related Metrics". In: *1st International Workshop on Validating Software Tests*. 2016.

[21] Gergő Balogh et al. “Are My Unit Tests in the Right Package?” In: *Source Code Analysis and Manipulation (SCAM), 2016 IEEE 16th International Working Conference on*. IEEE. 2016, pp. 137–146.

[22] Gergő Balogh et al. “Identifying wasted effort in the field via developer interaction data”. In: *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*. IEEE. 2015, pp. 391–400.

[23] Kent Beck, ed. *Test Driven Development: By Example*. Addison-Wesley Professional, 2002. ISBN: ISBN 0321146530.

[24] Pamela Bhattacharya. “Using software evolution history to facilitate development and maintenance”. In: *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE. 2011, pp. 1122–1123.

[25] Rex Black, Erik van Veenendaal, and Dorothy Graham. *Foundations of Software Testing: ISTQB Certification*. Cengage Learning, 2012. ISBN: 9781408044056.

[26] Vincent D Blondel et al. “Fast unfolding of communities in large networks”. In: *Journal of statistical mechanics: theory and experiment* 2008.10 (2008), P1000.

[27] Shawn A Bohner and Robert S Arnold. *Software change impact analysis*. English. Includes bibliographical references (p. 361-374). Los Alamitos, Calif. : IEEE Computer Society Press, 1996. ISBN: 0818673842 (pbk.)

[28] Shawn A Bohner et al. “Extending Software Change Impact Analysis into COTS Components”. In: (2003).

[29] Manuel Breugelmans and Bart Van Rompaey. “TestQ: Exploring structural and maintenance characteristics of unit test suites”. In: *WASDeTT-1: 1st International Workshop on Advanced Software Development Tools and Techniques*. 2008.

[30] Magiel Bruntink and Arie Van Deursen. “Predicting class testability using object-oriented metrics”. In: *Source Code Analysis and Manipulation, 2004. Fourth IEEE International Workshop on*. IEEE. 2004, pp. 136–145.

[31] D.J. Cavicchio. *Adaptive search using simulated evolution*. Tech. rep. Engineering, College of - Technical Reports, 1970. URL: `http://deepblue.lib.umich.edu/handle/2027.42/4042`.

[32] Zhihao Chen et al. “Finding the Right Data for Software Cost Modeling”. In: *IEEE Softw.* 22.6 (Nov. 2005), pp. 38–46. ISSN: 0740-7459. DOI: `10.1109/MS.2005.151`. URL: `http://dl.acm.org/citation.cfm?id=1098520.1098575`.

[33] Andrea De Lucia, Fausto Fasano, and Rocco Oliveto. “Traceability management for impact analysis”. In: *Frontiers of Software Maintenance, 2008. FoSM 2008.* IEEE. 2008, pp. 21–30.

[34] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-oriented reengineering patterns*. Elsevier, 2002.

[35] A. van Deursen et al. “Refactoring Test Code”. In: *Extreme Programming Perspectives*. Ed. by G. Succi et al. Addison-Wesley, 2002, pp. 141–152.

[36] Guy Deutscher. *The unfolding of language : an evolutionary tour of mankind's greatest invention*. Henry Holt and Co, 2006. ISBN: 0805080120.

[37] S. N. Dorogovtsev and J. F. F. Mendes. “Evolution of networks”. In: *Advances in Physics* 51.4 (2002), pp. 1079–1187. DOI: 10.1080/00018730110112519. eprint: https://doi.org/10.1080/00018730110112519. URL: https://doi.org/10.1080/00018730110112519.

[38] *Eclipse IDE Homepage*. https://www.eclipse.org/. last visited: 2016-11-25.

[39] Rudolf Ferenc et al. “Columbus – Reverse Engineering Tool and Schema for C++”. In: *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2002)*. IEEE Computer Society, Oct. 2002, pp. 172–181.

[40] Gavin R Finnie, Gerhard E Wittig, and Doncho I Petkov. “Prioritizing software development productivity factors using the analytic hierarchy process”. In: *Journal of Systems and Software* 22.2 (1993), pp. 129–139. ISSN: 0164-1212. DOI: https://doi.org/10.1016/0164-1212(93)90091-B. URL: http://www.sciencedirect.com/science/article/pii/016412129390091B.

[41] Santo Fortunato. “Community detection in graphs”. In: *Physics reports* 486.3 (2010), pp. 75–174.

[42] Markus Gaelli, Michele Lanza, and Oscar Nierstrasz. “Towards a taxonomy of SUnit tests”. In: *Proceedings of 13th International Smalltalk Conference (ISC'05)*. 2005.

[43] Markus Gaelli, Rafael Wampfler, and Oscar Nierstrasz. “Composing Tests from Examples.” In: *Journal of Object Technology* 6.9 (2007), pp. 71–86.

[44] Tamás Gergely et al. “Analysis of Static and Dynamic Test-to-code Traceability Information”. In: *Acta Cybernetica* 23.3 (2018), pp. 903–919.

[45] Tamás Gergely et al. “Differences between a static and a dynamic test-to-code traceability recovery method”. In: *Software Quality Journal* (2018), pp. 1–26.

[46] *gource software version control visualization*. URL: https://code.google.com/p/gource/.

[47] *Graph Database — Multi-Model Database — OrientDB*. https://orientdb.com/. (Accessed on 02/09/2019).

[48] *Greenfoot homepage*. http://www.greenfoot.org/home. 2014. URL: http://www.greenfoot.org/home.

[49] Tibor Gyimóthy, Rudolf Ferenc, and István Siket. “Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction”. In: *IEEE Transactions on Software Engineering*. Vol. 31. IEEE Computer Society, Oct. 2005, pp. 897–910.

[50] Paul Hamill. *Unit Test Frameworks: Tools for High-Quality Software Development*. O'Reilly Media, Inc., 2004.

[51] James Hamilton and Sebastian Danicic. "Dependence communities in source code". In: *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE. 2012, pp. 579–582.

[52] Casper Harteveld. "Triadic Game Design - Balancing Reality, Meaning and Play". In: 2011.

[53] G. Holmes, A. Donkin, and I.H. Witten. "WEKA: a machine learning workbench". In: *Proceedings of ANZIIS '94 - Australian New Zealnd Intelligent Information Systems Conference*. IEEE, 1994, pp. 357–361. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=396988.

[54] Ferenc Horváth et al. "Test Suite Evaluation using Code Coverage Based Metrics". In: *Proceedings of the 14th Symposium on Programming Languages and Software Tools (SPLST'15)*. Tampere, Finland, Oct. 2015, pp. 46–60.

[55] *JUnit Java Unit Test Framework Homepage*. http://junit.org/. last visited: 2016-05-27.

[56] Teemu Kanstrén. "Towards a deeper understanding of test coverage". In: *Journal of Software: Evolution and Process* 20.1 (2008), pp. 59–76.

[57] P. Khaloo et al. "Code Park: A New 3D Code Visualization Tool". In: *2017 IEEE Working Conference on Software Visualization (VISSOFT)*. Sept. 2017, pp. 43–53. DOI: 10.1109/VISSOFT.2017.10.

[58] John R. Koza et al. "Automated Design of Both the Topology and Sizing of Analog Electrical Circuits Using Genetic Programming". In: *Artificial Intelligence in Design '96*. Ed. by John S. Gero and Fay Sudweeks. Dordrecht: Springer Netherlands, 1996, pp. 151–170. ISBN: 978-94-009-0279-4. DOI: 10.1007/978-94-009-0279-4_9. URL: https://doi.org/10.1007/978-94-009-0279-4_9.

[59] D Lalanne and J Kohlas. *Human Machine Interaction: Research Results of the MMI Program*. 2009.

[60] *logstalgia website access log visualization*. URL: https://code.google.com/p/logstalgia/.

[61] FrontEndART Ltd. *SourceMeter Static Source Code Analyzer Homepage*. https://www.sourcemeter.com/. Last visited: 2016-05-27.

[62] KL Ma. "StarGate: A unified, interactive visualization of software projects". In: *Visualization Symposium, 2008. PacificVIS'08. IEEE* VIDi (2008), pp. 191–198. URL: http://ieeexplore.ieee.org/xpls/abs%5C_all.jsp?arnumber=4475476.

[63] A. Marcus, D. Comorski, and A. Sergeyev. "Supporting the evolution of a software visualization tool through usability studies". In: *Proceedings of the 13th International Workshop on Program Comprehension* (May 2005), pp. 307–316. DOI: 10.1109/WPC.2005.34. URL: http://ieeexplore.ieee.org/xpls/abs%5C_all.jsp?arnumber=1421046%20http://dl.acm.org/citation.cfm?id=1058432.1059368.

[64] Mitchell Melanie. "An introduction to genetic algorithms". In: *Cambridge, Massachusetts London, England, Fifth* (1999). URL: http://www.boente.eti.br/fuzzy/ebook-fuzzy-mitchell.pdf.

[65] Brian S Mitchell and Spiros Mancoridis. "On the automatic modularization of software systems using the bunch tool". In: *IEEE Transactions on Software Engineering* 32.3 (2006), pp. 193–208.

[66] Hausi A Müller et al. "A reverse-engineering approach to subsystem structure identification". In: *Journal of Software: Evolution and Process* 5.4 (1993), pp. 181–204. DOI: 10.1002/smr.4360050402.

[67] *Neo4j Graph Platform – The Leader in Graph Databases*. https://neo4j.com/. (Accessed on 02/09/2019).

[68] OECD. *Measuring Productivity - OECD Manual*. 2001, p. 156. DOI: https://doi.org/https://doi.org/10.1787/9789264194519-en. URL: https://www.oecd-ilibrary.org/content/publication/9789264194519-en.

[69] Thomas Ostrand. "White-Box Testing". In: *Encyclopedia of Software Engineering* (2002).

[70] Marian Petre. "UML in practice". In: *2013 35th International Conference on Software Engineering (ICSE)*. IEEE. 2013, pp. 722–731.

[71] Shari Lawrence. Pfleeger and Joanne M. Atlee. *Software engineering : theory and practice*. Prentice Hall, 2010, p. 756. ISBN: 0136061699.

[72] J. R. Quinlan. "Induction of decision trees". In: *Machine Learning* 1.1 (Mar. 1986), pp. 81–106. ISSN: 1573-0565. DOI: 10.1007/BF00116251. URL: https://doi.org/10.1007/BF00116251.

[73] Lior Rokach and Oded Maimon. *Data mining with decision trees. Theory and applications*. Vol. 69. Jan. 2008. DOI: 10.1142/9789812771728_0001.

[74] B. V. Rompaey and S. Demeyer. "Establishing Traceability Links between Unit Test Cases and Units under Test". In: *Software Maintenance and Reengineering, 2009. CSMR '09. 13th European Conference on*. Mar. 2009, pp. 209–218. DOI: 10.1109/CSMR.2009.39.

[75] Gregg Rothermel and Mary Jean Harrold. "Empirical studies of a safe regression test selection technique". In: *IEEE Transactions on Software Engineering* 24.6 (June 1998), pp. 401–419.

[76] SonarSource S.A. *SonarQube homepage*. `https://www.sonarqube.org`. (Accessed on 07/23/2019).

[77] Lajos Schrettner et al. "Impact Analysis in the Presence of Dependence Clusters Using Static Execute After in WebKit". In: *Proc. of IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*. Riva del Garda, Trento, Italy, 2012, pp. 24–33.

[78] Robert W Schwanke. "An intelligent tool for re-engineering software modularity". In: *Software Engineering, 1991. Proceedings., 13th International Conference on*. IEEE. 1991, pp. 83–92.

[79] Robert Schwanke, Lu Xiao, and Yuanfang Cai. "Measuring architecture quality by structure plus history analysis". In: *2013 35th International Conference on Software Engineering (ICSE)*. IEEE. 2013, pp. 891–900.

[80] Shai Shalev-Shwartz and Shai Ben-David. "Decision Trees". In: *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, 2014, pp. 212–218. DOI: `10.1017/CBO9781107298019.019`.

[81] Robin C. Sickles and Valentin Zelenyuk. *Measurement of Productivity and Efficiency: Theory and Practice*. Cambridge University Press, 2019. DOI: `10.1017/9781139565981`.

[82] Bernard W Silverman. *Density estimation for statistics and data analysis*. Routledge, 2018.

[83] SN Sivanandam. "Introduction to genetic algorithms". In: (2007). URL: `http://dl.acm.org/citation.cfm?id=1557421`.

[84] Software Engineering for Smart Data Analytics & Smart Data Analytics for Software Engineering. *Small Refactoring Classroom Exercise Website*. URL: `https://sewiki.iai.uni-bonn.de/private/daniel/public/tutorials/small$%5C_$refactoring` (visited on 10/07/2015).

[85] George Spanoudakis and Andrea Zisman. "Software traceability: a roadmap". In: *Handbook of Software Engineering and Knowledge Engineering* 3 (2005), pp. 395–428.

[86] Margaret-anne Storey, Casey Best, and Jeff Michaud. "SHriMP views: an interactive environment for information visualization and navigation". In: *CHI '02 extended abstracts on Human factors in computing systems - CHI '02* (Apr. 2002), p. 520. DOI: `10.1145/506443.506459`.

[87] Lovro Šubelj and Marko Bajec. "Community structure of complex software systems: Analysis and applications". In: *Physica A: Statistical Mechanics and its Applications* 390.16 (2011), pp. 2968–2975.

[88] Gilbert Syswerda. "Uniform Crossover in Genetic Algorithms". In: *Proceedings of the 3rd International Conference on Genetic Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1989, pp. 2–9. ISBN: 1-55860-066-3. URL: http://dl.acm.org/citation.cfm?id=645512.657265.

[89] Dávid Tengeri et al. "Beyond Code Coverage – an Approach for Test Suite Assessment and Improvement". In: *Proceedings of the Testing: Academic & Industrial Conference – Practice and Research Techniques (TAIC PART 2015)*. Graz, Austria: IEEE Computer Society, Apr. 2015, ??–??

[90] Dávid Tengeri et al. "Negative Effects of Bytecode Instrumentation on Java Source Code Coverage". In: *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2016)*. Osaka, Japan, Mar. 2016, pp. 225–235.

[91] Dávid Tengeri et al. "Toolset and Program Repository for Code Coverage-Based Test Suite Analysis and Manipulation". In: *Proceedings of the 14th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'14)*. Victoria, City of Gardens, British Columbia, Canada, Sept. 2014, pp. 47–52.

[92] Vikas Thada and Vivek Jaglan. "Comparison of jaccard, dice, cosine similarity coefficient to find best fitness value for web retrieved documents using genetic algorithm". In: *International Journal of Innovations in Engineering and Technology* 2.4 (2013), pp. 202–205.

[93] Gabriella Tóth et al. "Adding Process Metrics to Enhance Modification Complexity Prediction". In: *Proceedings of the IEEE International Conference on Program Comprehension (ICPC 2011)*. 2011, pp. 201–204. URL: http://www.inf.u-szeged.hu/~gtoth/research/mpp%5C_ICPC2011.pdf.

[94] Gabriella Tóth et al. "Adjusting Effort Estimation Using Micro-Productivity Profiles". In: *Proceedings of the Estonian Academy of Sciences* 62.1 (2013), pp. 71–80.

[95] Adam Trendowicz and Jürgen Münch. "Chapter 6: Factors Influencing Software Development Productivity – State-of-the-Art and Industrial Experiences". In: *Social networking and the web*. Vol. 77. Advances in Computers. Elsevier, 2009, pp. 185–241. DOI: http://dx.doi.org/10.1016/S0065-2458(09)01206-6.

[96] Edward R. Tufte. *The visual display of quantitative information*. Graphics Press, 2001, p. 197. ISBN: 1930824130.

[97] Edward R. Tufte and Graphics Press. *Envisioning information*, p. 126.

[98] Craig Upson et al. “The application visualization system: A computational environment for scientific visualization”. In: *Computer Graphics and Applications, IEEE* 9.4 (1989), pp. 30–42.

[99] Bart Van Rompaey and Serge Demeyer. “Exploring the composition of unit test suites”. In: *Automated Software Engineering-Workshops, 2008. ASE Workshops 2008. 23rd IEEE/ACM International Conference on*. IEEE. 2008, pp. 11–20.

[100] László Vidács et al. “Assessing the Test Suite of a Large Scale System based on Code Coverage and Derived Metrics”. In: *1st International Workshop on Validating Software Tests (VST'16) – accepted paper*. Osaka, Japan, Mar. 2016, pp. 1–4.

[101] P. A. Vikhar. “Evolutionary algorithms: A critical review and its future prospects”. In: *2016 International Conference on Global Trends in Signal Processing, Information Computing and Communication (ICGTSPICC)*. Dec. 2016, pp. 261–265. DOI: 10.1109/ICGTSPICC.2016.7955308.

[102] Silke Wagner and Dorothea Wagner. *Comparing clusterings: an overview*. Universität Karlsruhe, Fakultät für Informatik Karlsruhe, 2007.

[103] Stefan Wagner and Melanie Ruhe. *A Systematic Review of Productivity Factors in Software Development*. Tech. rep. Technische Universität München, 2008.

[104] Lu Wang et al. “Construct Bug Knowledge Graph for Bug Resolution”. In: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE. 2017, pp. 189–191.

[105] Richard Wettel and Michele Lanza. “CodeCity”. In: *Proceedings of WAS-DeTT* (2008), pp. 1–13.

[106] Richard Wettel and Michele Lanza. “CodeCity: 3D Visualization of Large-scale Software”. In: *Companion of the 30th International Conference on Software Engineering*. ICSE Companion '08. Leipzig, Germany: ACM, 2008, pp. 921–922. ISBN: 978-1-60558-079-1. DOI: 10.1145/1370175.1370188. URL: http://doi.acm.org/10.1145/1370175.1370188.

[107] Kenny Wong. “Rigi user's manual”. In: *Department of Computer Science, University of Victoria* (1998). URL: http://www.rigi.cs.uvic.ca/downloads/pdf/rigi-5%5C_4%5C_4-manual.pdf.