Technical University of Košice
Faculty of Electrical Engineering and Informatics

Integrating Runtime Metadata with Source Code to Facilitate Program Comprehension

Dissertation

2018 **Ing. Matúš Sulír**

Technical University of Košice
Faculty of Electrical Engineering and Informatics

Integrating Runtime Metadata with Source Code to Facilitate Program Comprehension

Dissertation

Study program:	Informatics
Field of study:	Informatics
Department:	Department of Computers and Informatics
Supervisor:	doc. Ing. Jaroslav Porubän, PhD.

Košice 2018 **Ing. Matúš Sulír**

Abstract

The general goal of this thesis is to provide suggestions for the facilitation of program comprehension and maintenance. We focus on approaches utilizing dynamic analysis, integrating the data collected during program executions with source code.

Since a compilable software system is a prerequisite for many comprehension tasks, we first describe a study of software build failures. Next, we provide a literature overview of approaches labeling source code with various types of metadata, including data obtained from running programs. In a controlled experiment, we found that the presence of a specific kind of metadata, concern annotations, improves program comprehension correctness and time. Although concern annotations are traditionally inserted into the source code manually, we demonstrate and evaluate their semi-automated insertion. The problem of runtime metadata persistence in the code is also discussed, suggesting suitable formats and workflows.

An important task during program comprehension is to find source code parts relevant to a given element displayed in the user interface of a running program. In a small-scale study, we found that many strings displayed in the running applications were not found it their source code at all, or they had too many occurrences. Therefore, a static source code search is often an insufficient strategy for this task. We designed RuntimeSearch – an approach to runtime string searching. It searches the given text in all strings being evaluated in a running program and pauses it when a match is found, offering standard debugging features available in integrated development environments. The approach was validated using a controlled experiment with human participants, showing a positive effect on maintenance efficiency.

We also present DynamiDoc, a documentation generator utilizing information from dynamic analysis. During program executions, it records the arguments, return values and object changes. Then, it generates sentences containing examples of concrete values, describing individual methods in the source code. We mention advantages and disadvantages of this approach. In a preliminary quantitative evaluation, we found that the generated sentences are succinct.

As an alternative to documentation generation, we present an interactive approach to source code augmentation, RuntimeSamp. Collected sample values of variables are displayed next to each source code line in the editor. We discuss several questions which should be answered for this approach to be useful in practice. As a supplement, we provide a systematic mapping study of visual augmentation of source code editors.

Keywords

program comprehension, software maintenance, dynamic analysis, text search, documentation, integrated development environment

Abstrakt

Hlavnou témou tejto práce je poskytnúť návrhy na uľahčenie pochopenia a údržby programov. Zameriavame sa na prístupy využívajúce dynamickú analýzu, integrujúce údaje zbierané počas vykonávania programov so zdrojovým kódom.

Keďže kompilovateľný softvérový systém je predpokladom pre mnoho úloh porozumenia, najprv popisujeme štúdiu zlyhaní zostavenia softvéru. Ďalej poskytujeme literárny prehľad prístupov značenia zdrojového kódu rôznymi typmi metadát, vrátane dát získaných z bežiacich programov. Riadeným experimentom sme zistili, že prítomnosť špecifického druhu metadát, zámerových anotácií, zlepšuje správnosť a čas porozumenia programom. Hoci zámerové anotácie sú tradične vkladané do zdrojového kódu ručne, demonštrujeme a vyhodnocujeme ich poloautomatické vkladanie. Diskutuje sa tiež o probléme uchovávania behových metadát v kóde, navrhujúc vhodné formáty a pracovné postupy.

Dôležitou úlohou počas porozumenia programom je nájsť časti zdrojového kódu relevantné k danému elementu zobrazenému v používateľskom rozhraní bežiaceho programu. V štúdii malého rozsahu sme zistili, že mnoho reťazcov zobrazených v bežiacich aplikáciách nebolo vôbec nájdených v ich zdrojovom kóde alebo mali príliš veľa výskytov. Preto je statické vyhľadávanie v zdrojovom kóde často nedostatočnou stratégiou pre túto úlohu. Navrhli sme RuntimeSearch – prístup k behovému vyhľadávaniu reťazcov. Vyhľadáva daný text vo všetkých reťazcoch vyhodnocovaných v bežiacom programe a pozastavuje ho, keď je nájdená zhoda, poskytujúc štandardné ladiace funkcionality dostupné v integrovaných vývojových prostrediach. Prístup bol overený pomocou riadeného experimentu s ľudskými účastníkmi, ukazujúc pozitívny efekt na účinnosť údržby.

Predstavujeme tiež DynamiDoc, generátor dokumentácie využívajúci informácie z dynamickej analýzy. Počas vykonávaní programu zaznamenáva argumenty, návratové hodnoty a zmeny objektov. Potom generuje vety obsahujúce príklady konkrétnych hodnôt, popisujúce jednotlivé metódy v zdrojovom kóde. Spomíname výhody a nevýhody tohto prístupu. Predbežným kvantitatívnym vyhodnotením sme zistili, že generované vety sú stručné.

Ako alternatívu ku generovaniu dokumentácie predstavujeme interaktívny prístup k rozširovaniu zdrojového kódu, RuntimeSamp. Zozbierané vzorové hodnoty premenných sú zobrazené vedľa každého riadku zdrojového kódu v editore. Diskutujeme tiež o viacerých otázkach, ktoré by mali byť zodpovedané, aby bol tento prístup užitočný v praxi. Ako doplnok ponúkame systematickú mapovaciu štúdiu vizuálneho rozširovania editorov zdrojového kódu.

Kľúčové slová

porozumenie programom, údržba softvéru, dynamická analýza, textové vyhľadávanie, dokumentácia, integrované vývojové prostredie

Declaration

I declare this thesis represents my own work end effort, all used information sources are acknowledged. Some parts of the thesis are a result of collaboration. Each chapter encompassing such material contains a footnote listing co-authors and their contributions.

Košice, May 9, 2018 . . . . . .

Acknowledgement

I would like sincerely thank my advisor Jaroslav Porubän for his guidance throughout the study and for the inspiration I received. My thanks goes also to my colleagues and family which has always supported me.

Contents

List of Figures

List of Tables

List of Abbreviations

API	application programming interface
DBMS	database management system
DL	digital library
DSL	domain specific language
CI	continuous integration
FO	formatting objects
GUI	graphical user interface
GPL	general-purpose language
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IDE	integrated development environment
JAR	Java ARchive
Java ME	Java Micro Edition
JDK	Java Development Kit
JNI	Java Native Interface
JSON	JavaScript Object Notation
kLOC	thousands of lines of code
LOC	lines of code
PDF	Portable Document Format
POM	Project Object Model
REST	REpresentational State Transfer
UI	user interface
UML	Unified Modeling Language
URL	Uniform Resource Locator
VCS	version control system
XML	eXtensible Markup Language

Chapter 1

Introduction

Maintenance of existing computer program systems typically counts for about 60% of software development cost [78]. Before a part of a system can be changed, it must be comprehended [195]. Nowadays, software projects are built using large frameworks. Systems have an immense number of dependencies and their behavior often depends on results of asynchronous calls to remote servers. They consist of multiple layers and having a web or mobile interface becomes a standard. Project teams are large and dispersed over the world [238], which hinders traditional communication about intents in the source code. Without proper tool support, understanding of software systems may soon become an unattainable task. Therefore, this work aims to improve understanding of existing programs by developers, i.e., *program comprehension*.

A programming language should serve as a mean of communication between a human and a computer. A valid source code of a program, by definition, contains all required information to be unambiguously compiled and executed on a computer. On the other hand, the source code is often understood by humans with great difficulties, or not at all. This can be attributed to a lack of information useful for humans contained in the code. Many approaches aim to fill this gap by assigning additional *metadata* to source code elements. These metadata can come from diverse sources: from manually entered comments, through documentation generated by static analysis, to harvested instant messages. They can be of various types – from numeric values to texts and images – and presented in even more various ways. They can be permanently persisted in the source code or a database, or shown only for a split second in the integrated development environment (IDE).

A particularly interesting kind of metadata are data obtained by dynamic analysis, i.e., *runtime* metadata. During the execution of a program, inherently abstract source code becomes concrete. Individual runtime states and their transitions offer the developer exemplified views of the program.

Our ultimate goal is to improve program understanding efficiency, particularly by integrating the run-time information with the source code. In this thesis, we offer a few approaches trying to at least partially fulfill this ambitious goal. Along with them, we provide motivational empirical studies, evaluations and literature reviews about related topics.

In this chapter, we provide an overview of the dissertation. We briefly describe the discussed topics and list hypotheses and research questions which are elaborated in the next chapters. Additionally, we note the used research methods.

1.1 Program Comprehension

In chapter 2, we first provide a general overview of program comprehension theories, approaches and tools in the form of an *ad hoc literature review*. Then we outline rising and falling trends in program comprehension over the timeline of 15 years. We asked the research question:

- **RQ2.1**: What are the rising and falling trends in program comprehension?

An automated technique of *trend analysis* on titles, abstract and keywords of scientific papers was used.

During multiple experiments, we recognized the importance of the software build process. Many open source projects fail to be built from source code, which, besides being a practical problem, hinders program comprehension research. A successful build result is a prerequisite for almost all comprehension activities, including the assignment of dynamic analysis data to parts of source code. Although there exists limited evidence about build failures (e.g., by Neitsch et al. [175] or Seo et al. [224]), the studies are rarely large-scale and suffer from low external validity. To quantify the severity of the build failure problem, we tried to fully automatically build about 7,000 Java projects. In chapter 3, we ask the following research questions:

- **RQ3.1** What portion of projects fails to build?
- **RQ3.2** What types of build errors do occur most frequently?
- **RQ3.3** Is there an association between the build success/failure and other project's properties (e.g., the used build tool, age)?

To answer the first two questions, we use *quantitative methods*. *Statistical hypothesis testing* is used for the third question.

1.2 Source Code Labeling

Since understanding a program only by looking at its source code is difficult, many approaches label parts of the code with various metadata. They include human-written descriptions of methods, information obtained from static and dynamic analysis, IDE interaction data, information from version control systems and other communication artifacts. These metadata are then presented to the programmer by appropriate tools. We conducted a partially *systematic mapping study* with the aim of taxonomy construction in this broad research area. We combined manual searching of selected years from relevant conferences and journals, keyword search and references search, and a personal bibliography. The main purpose was to show the variety of the research area. The following questions were asked in chapter 4:

- **RQ4.1**: What approaches (and tools implementing them) do exist to label parts of source code with additional metadata and present them to a programmer in order to improve program comprehension?
- **RQ4.2**: How can these approaches be categorized?

Concern annotations represent a specific kind of source code labeling. They are human-written Java annotations describing the intentions behind a particular class or method. Using a *controlled experiment* with human participants, we tested the following hypothesis in chapter 5:

- **H5.1**: The presence of concern annotations in the code improves program comprehension and maintenance correctness, time and confidence.

To further support our hypothesis, we also performed an *experiment replication*, now on industrial developers instead of students.

Manual insertion and updating of concern annotations in the source code are tedious. Thus we designed a runtime-based, semi-automated approach of concern annotation. This method uses an existing dynamic analysis approach (software reconnaissance [278]) to extract sets of methods pertaining to the given feature. Then we build upon the work which compares the overlap of multiple independent human annotators of the same code [181] by asking the research question:

- **RQ5.1**: How does the overlap between semi-automatic and code author's annotations compare to the overlap among annotations of multiple non-authors?

The previously mentioned approach can be extended to any runtime metadata, not only concern annotations. In chapter 6, we describe a method which assigns information obtained from dynamic analysis to source code elements – e.g., a set of dynamic method callers or code coverage results. Thus we ask the question:

- **RQ6.1**: How can runtime metadata be persisted directly in source code files?

We discuss what formats are suitable for metadata persisted in source files and how such an approach fits into an existing software development workflow.

1.3 Searching Concepts in Running Programs

Programmers often interact with the UI (user interface) of a running program, putting themselves in the role of end users. They create mappings of UI elements to the source code in their minds [209]. However, anecdotal evidence says us that not every textual label displayed in the GUI (graphical user interface) can be easily found in the source code. Therefore, we decided to *quantitatively* examine the relationship between the terms occurring in the static source code and the labels displayed in the GUI of a running program. The GUIs of four open source Java applications were automatically traversed, producing a list of strings and words contained in these strings. In chapter 7, we ask three research questions:

- **RQ7.1**: What portion of strings and words displayed in the GUIs of running desktop Java applications are located in their static source code too?
- **RQ7.2**: Mainly in what types of files are these terms located?
- **RQ7.3**: If some strings are not located in the source code, what are common reasons?

This study presents a motivation for the approach we designed – RuntimeSearch. It allows a developer to find run-time values in a manner similar to traditional, static source code search. After entering the searched string, the program is run, while comparing all evaluated string expressions at runtime with the given text. When a match is found, the program is paused and a traditional debugger is open, with all standard operations available. This is useful as a lightweight form of feature location or to confirm hypotheses during debugging. In chapter 8, we had two research questions and one hypothesis regarding our new approach:

- **RQ8.1**: What are possible use cases of RuntimeSearch?
- **RQ8.2**: What is the performance overhead of searching strings in the runtime?
- **H8.1**: RuntimeSearch improves the efficiency of search-focused maintenance tasks.

They were answered using *performance evaluation*, a *case study*, and a *controlled experiment*, respectively.

1.4 Generating Documentation from Runtime Values

There exist multiple approaches which label the source code by automatically generated documentation [174]. However, they traditionally process the static source code of a program

and associated collaboration artifacts. Our new approach – DynamiDoc, described in chapter 9, generates documentation sentences for methods using information from dynamic analysis. For each method, string representations of concrete arguments, return values and object states before and after the method execution are recorded. Then, sentences utilizing sample values are generated. For example, for a list reversal, the sentence can look like: "When called on [1.5, 2], the object changed to [2, 1.5]." We perform both a *qualitative* and preliminary *quantitative* evaluation:

- **RQ9.1**: What are the strengths and weaknesses of DynamiDoc?
- **RQ9.2**: What is a typical length of documentation sentences generated by DynamiDoc?
- **RQ9.3**: What portion of objects contained in the generated sentences has a custom string representation?

1.5 Augmenting Code Lines with Variable Values

Instead of inserting metadata into source code files, it is also possible to develop an interactive approach in the form of an IDE plugin. In recent years, we noted the advance of visual enhancements of source code editors. The IDEs for textual languages are no longer purely textual – they display graphs, charts, texts and other information directly next to the related code. We would like to use this fact with advantage and design an IDE augmentation approach using dynamic analysis.

First, we performed a *systematic mapping study* about visual augmentation of source code editors in general. In chapter 10, the following questions are answered:

- **RQ10.1**: What source code editor augmentation tools are described in the literature?
- **RQ10.2**: How can they be categorized?

Finally, in chapter 11, we describe RuntimeSamp – our technique of visual source code augmentation using runtime information, namely the values of variables. First, a few values of each variable are recorded during program execution. Next, at the end of each line, the sample values are displayed thanks to an IDE plugin. The approach has a similar base idea as DynamiDoc – to help the programmer by presenting concrete examples of values; however, this time they are assigned to each line instead of the method as a whole. The following research questions were asked:

- **RQ11.1**: How to present complicated objects succinctly on a small space?
- **RQ11.2**: When exactly should we capture the values of variables?
- **RQ11.3**: How to decide which iteration to display?
- **RQ11.4**: How to define an iteration in a way which is easy to detect and present?
- **RQ11.5**: How to collect enough data for sample values presentation while keeping the overhead reasonable?
- **RQ11.6**: Which variable values should be displayed and which not?
- **RQ11.7**: When and how to invalidate the collected variable values?

Since RuntimeSamp is a preliminary approach, we answered some of these questions only naively, leaving space for further research.

Chapter 2

Program Comprehension Overview

In this chapter[1], we will provide a brief general overview of the research field of program comprehension – its theories, methods, selected techniques and trends.

2.1 Research Field Definition

According to Biggerstaff et al. [25], a person comprehends a program if one understands its structure, behavior and connection to the application domain. We can say that program comprehension describes how developers understand existing programs (whether or not written by themselves) in order to improve their functional or non-functional qualities.

To delineate the research field, let us put program comprehension in the context of a software development process (Fig. 2.1). As a developer reads the specification, a mental model forms in his or her head. A mental model is defined as a representation of a program in the programmer's mind [239]. This model is refined until the developer is able to implement parts of the application and create manually produced artifacts – traditionally the source code. The code is compiled, producing generated artifacts like a runnable application. To comprehend the program, the developer inspects the source code and debugs the produced application. This further refines the mental model and the development continues. Occasionally, the specification must be adapted because not all requirements were implemented in a desired way.

It is important to note that the order of the mentioned processes is not fixed and they often interleave or are performed in parallel. In many cases the programmer starts to familiarize with the program by inspecting generated artifacts – the running application. Finally, the diagram depicts only one specific developer's perspective and does not consider team aspects of software engineering.

Two most related research fields are software maintenance and reverse engineering. Maintenance is a broader term which encompasses practically a whole software development process since it is difficult to distinguish between software creation and software modification.

The terms program comprehension and reverse engineering are often used interchangeably, although there is a slight difference. Reverse engineering focuses on techniques and tools for

[1] This chapter contains material from our articles: [244], [249]. The copyright symbols next to some figures and tables do not represent our opinion – they are a requirement of the publisher.

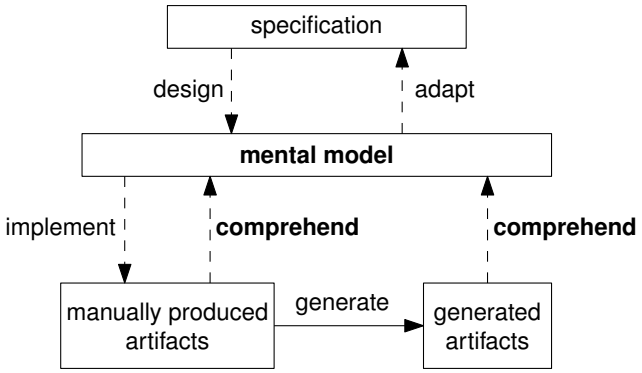


Figure 2.1: Program comprehension in the context of a software development process. Dashed arrows represent manual, human-performed processes; solid arrows are automatic, computer-performed operations. The research area of program comprehension is marked in boldface.

creation of higher-level abstraction documents from lower-level ones. Program comprehension research is also interested in the effect of using these tools. Therefore, reverse engineering techniques aid developers in program comprehension [168].

2.2 Theories and Research Methods

A cognitive model, also called a comprehension model, is a theory describing psychological processes involved in program comprehension which help to construct the mental model of a program [239]. There are multiple cognitive models described by researchers. Generally, we can divide them into top-down and bottom-up approaches.

Top-down program comprehension starts with the programmer's problem domain knowledge. The person forms hypotheses about the program and accepts or rejects them by inspecting the source code and other artifacts. The hypotheses are gradually refined into sub-hypotheses [35]. This type of comprehension is typically used when the program is familiar [274]. An example of a top-down cognitive model is the Brooks model [35].

Bottom-up comprehension is characterized by reading the source code line by line and incrementally grouping them to form higher-level abstractions [239]. An example is the Pennington model [188].

The fact that there are at least seven, relatively complex comprehension models described in literature (six in a review [274] and one recent [22]) suggests us an idea that this number is not finite. Each person's cognition is quite different and it depends on a specific situation. It is therefore difficult to generalize the theories and design useful program comprehension tools directly after them. For this reason, empirical studies are very common in the field of program comprehension. For an example of a controlled experiment in program comprehension, see section 5.1.

2.3 Techniques and Tools

We will now explore the process of program comprehension from the most high-level perspective to detailed inspection, providing an overview of relevant techniques and tools available, along with problems developers encounter.

2.3.1 Overall Comprehension

Architecture comprehension tools visualize the overall structure of a software system, its components and their relationship. Polymetric view tools like CodeCrawler [128] display code entities as nodes, relations between them as edges and arbitrary metrics as node shape attributes. For example, classes are displayed as rectangles with the width proportional to the method count and the height proportional to the line count. A disadvantage of such tools is the fact they usually focus just on simple quantitative metrics and perform only a static analysis of source code.

Metaphor-based visualizations transform code entities to 2D or 3D objects known from real life. For example, the famous city metaphor displays classes as buildings – like in the EvoSpaces tool [2]. An industrial application of such visualizations is questionable. A more promising approach is offered by ExplorViz [72], which combines them with so-called landscape view – a mixture of UML deployment and activity diagram produced by analyzing execution traces.

Domain analysis tools can be helpful for a new team member to become familiar with the system. The DEAL method [15] extracts a domain model from the GUI (graphical user interface) of a running application. However, manual user interaction is necessary and the method focuses only on a static structure of programs like relations between forms and their controls.

To gain the overview of domain knowledge contained in a program, it is possible to map program identifiers to an ontological dictionary like WordNet using subgraph homomorphism [199]. However, the approach is only semi-automatic and it is necessary to adjust the results manually. This is a result of representational defects like polysemy where one identifier in a program can represent multiple real-world concepts. Some types of defects can be detected and repaired, but the other are inherent to the nature of relationship between programs written in current general-purpose languages (GPLs) and the real world [200].

2.3.2 Feature Location

As software becomes more complicated, it is impossible for a programmer to understand it whole. Therefore partial, or as-needed comprehension is necessary [196].

Each software consists of problem concepts and features (like a shopping cart or shipping method selection) and solution features (e.g., a web presentation and database persistence). One problem domain feature is scattered across multiple solution features. This is called feature delocalization [82]. One of the key objectives of program comprehension is to tackle this phenomenon which impedes code navigation by using feature location (or concept location) tools.

Textual code search is a common feature location approach. To find syntactical patterns, programmers often use complicated regular expressions and the results are unsatisfactory as programming languages are rarely regular. An AST (abstract syntax tree) querying is more viable. By using an intuitive “query-by-example” language [186], developers can search an AST without realizing it. However, the approach is not usable for more complicated patterns.

The MuTT tool [143] implements feature location by execution trace collection. While debugging the program, just before utilizing the feature of interest like adding an item to the shopping cart, the programmer clicks a button in the tool to start collecting the trace. After performing the desired action, the button is clicked again. A list of all methods executed between the two clicks is displayed in an IDE (integrated development editor). It must be manually filtered since it probably contains many auxiliary methods, not related to the

tracked feature. Furthermore, to find multiple features, it is necessary to manually repeat the mentioned actions for each of them. The possibility of automation should be investigated.

After the features are located, it is desirable to mark their occurrences in the source code to avoid duplicate work in the future. This activity is called code labeling or concern tagging. In the previous example, we would mark all found methods with a tag "add to cart". An academic tool FLAT^3 [221] combines: full-text code search, feature location by execution trace collection using the MuTT tool, feature labeling with method-level granularity, and visualization of feature distribution across the source code files.

An another approach for feature location is differential code coverage [226]. If we run a program first selecting a feature of interest and then not, the difference of code coverage should be the source code of the feature. This approach suffers from similar limitations as execution trace collection.

It would be useful to combine execution trace collection of the MuTT tool with differential code coverage and assess the effectiveness of such approach. Furthermore, as bugs can be considered a kind of features (unwanted, of course), bug localization is a specific kind of feature localization. Differential code coverage could be used to localize the faults, too.

An interesting feature location tool is included directly in the core of an industrial IDE, NetBeans. During the runtime, it is possible to take so-called GUI Snapshot of the debugged program user interface and inspect the events to be performed after e.g., clicking a button or selecting a combo-box item.

2.3.3 Understanding the Details

The Theseus tool [140] shows how many times a particular method has been called during a web application execution. These numbers are displayed directly in the code editor and in real time. By clicking a given method, a retroactive log containing the argument values for each execution is shown. This allows for feature location along with a more thorough inspection of the program behavior. A similar tool [153], but for desktop Java applications, operates on individual source code lines instead of methods and thus focuses on detailed algorithm analysis.

Many difficult comprehension questions developers ask when debugging can be answered by asking reachability questions. A reachability question involves searching for all possible execution paths to find source code statements matching the given criteria [129]. However, developers' questions are often vague and difficult to formulate in mathematical terms. The overhead used to formulate such queries can exceed the benefits they provide.

As developers inspect the source code and associated artifacts using various techniques, they create mental models of the code which contain information far behind what is explicitly written. Despite there is a high demand for this implicit knowledge [147], it is not explicitly captured [130] or it is captured only in transient notes, not persisted across sessions [147]. One possible cause is that each developer's mental model is different and thus the personal notes created by one developer are not useful for an another one. This hypothesis should be tested by further research.

One form of temporary knowledge saving is the use of bookmarks in an IDE. To share such personal findings, collective bookmarks [85] can be used. In the study, they were found useful for marking the lines where to start with a given type of task, but not for detailed explanations.

2.3.4 Investigating the Rationale

Even if the code is thoroughly understood, programmers often ask why it was implemented this way [130]. One of the manually produced artifacts in Fig. 2.1 are version control system commits. Commit messages may contain invaluable information about intentions behind the source code.

Surprisingly, less developers actually read commit messages than write them [147]. A possible cause is that IDEs do not present them conveniently. Plugins like Deep Intellisense or Rationalizer [31] try to overcome this issue by displaying history information relevant to a given piece of code.

For version control analysis tools to be successful, each commit must contain code related to only one task. This is often not true in reality. A technique [115] can detect tangled changes before they are committed and suggest how they could be untangled. The disadvantage is a high ratio of false positives.

2.3.5 Making Programs Comprehensible

While analyzing existing software is useful, we must also learn lessons from it and design new systems to be more comprehensible.

Identifier naming is a significant factor of program comprehension. Method names can be evaluated for comprehensibility and more intuitive names can be suggested, using a database of commonly used n-grams [261].

Instead of using feature location approaches with often unsatisfactory results, there is a possibility to label classes and methods by their associated concerns using source code annotations immediately when writing the code. Then, source code projections [193] can be used to show classes and methods associated with a given concern in an IDE. While projections are perceived as useful, labeling the code manually is a time-consuming activity which does not directly affect the resulting program execution. Therefore, we can expect programmers to skip labeling or leave the annotations out of sync with the rest of the code, just as it often happens with comments.

Much work is done in the area of code clone detection and removal since there is a widespread opinion that duplicated code negatively affects comprehension. Surprisingly, a study [194] found out that fixing bugs affecting cloned code do not require more effort than fixing other bugs. Code with high regularity, which may be a result of cloning, is more easily comprehended by developers [102]. Furthermore, developers perceive code duplication in a broader sense than simple copy-pasting – for example, implementing the same method in multiple languages or in multiple version control branches [130].

2.4 Trends

It is beneficial for researchers not only to find out which approaches exist in the area, but also to gain insight about their popularity over time. Therefore, we ask the following research question:

RQ2.1 *What are the rising and falling trends in program comprehension?*

To analyze the development of trends in the research area of program comprehension, we decided to perform an automated trend analysis on the metadata of research articles published between the years 2000 and 2014 (inclusive).

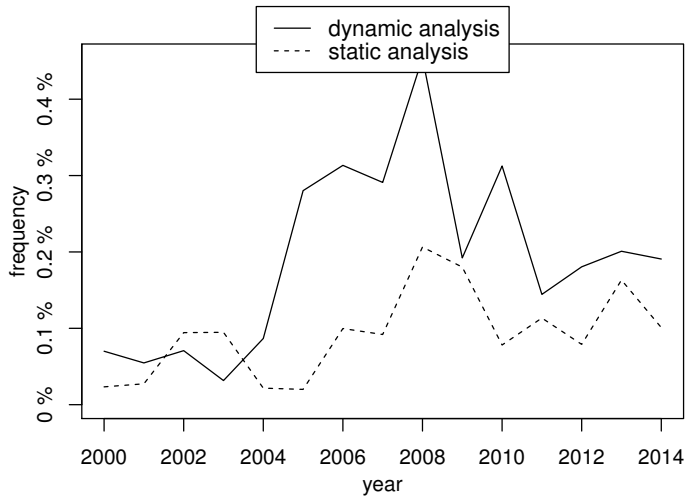


Figure 2.2: Static vs. dynamic analysis. © 2015 IEEE

We searched the citation databases Scopus[2], IEEE Xplore[3] and HCI Bibliography[4] for the following exact terms:

- program comprehension,
- program understanding,
- code comprehension.

Titles, abstracts and author-supplied keywords of 1885 articles were analyzed using a technique called n-grams [163]. For each year, a frequency of every phrase among all phrases of the given length was determined. For a detailed description of the method, see our article [249].

2.4.1 Techniques

Static program analysis (e.g., [122]) deals with the source code of an application without running it, whereas dynamic analysis [47] utilizes runtime information. As seen in Fig. 2.2, in 2004, the dynamic program analysis overtook the static one and this state lasts until today.

We can see a plot of two often used program comprehension techniques – feature location and visualization – in Fig. 2.3. While visualization is relatively steady, feature location rises rapidly from 2004.

Program slicing [284] is a technique used to find all code semantically related to the given statement or variable and produce a new program, containing only this related code. In Fig. 2.4, we can see a decreasing popularity of slicing.

Code clone detection [214] is a research field with a long tradition. However, from Fig. 2.4 it is obvious that it started to associate with program comprehension only in recent years.

2.4.2 Systems

In Fig. 2.5, we can see what types of systems the researchers study. Legacy systems have a clearly decreasing tendency during the last 15 years. One trend which immediately caught our attention is an extremely rapid increase of the n-gram "open source" between the years 2003 and 2009.

[2] http://www.scopus.com
[3] http://ieeexplore.ieee.org
[4] http://hcibib.org/ICPC14

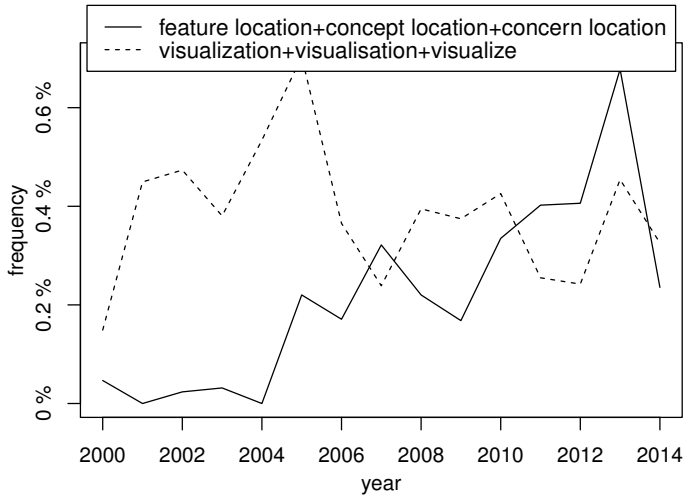


Figure 2.3: Feature location vs. visualization. © 2015 IEEE

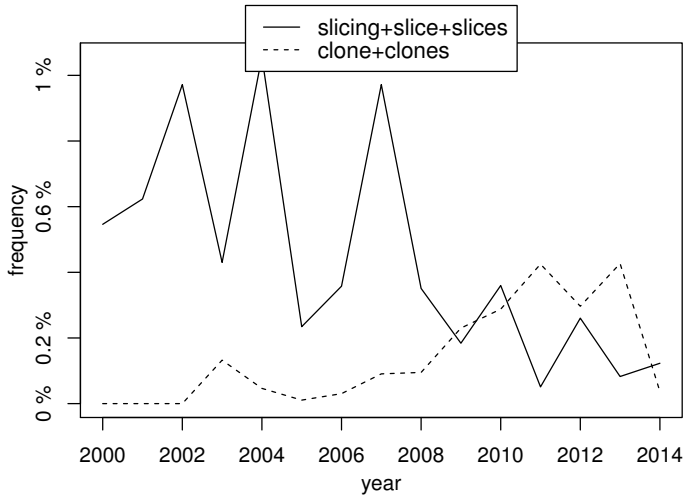


Figure 2.4: Program slicing, code clone detection. © 2015 IEEE

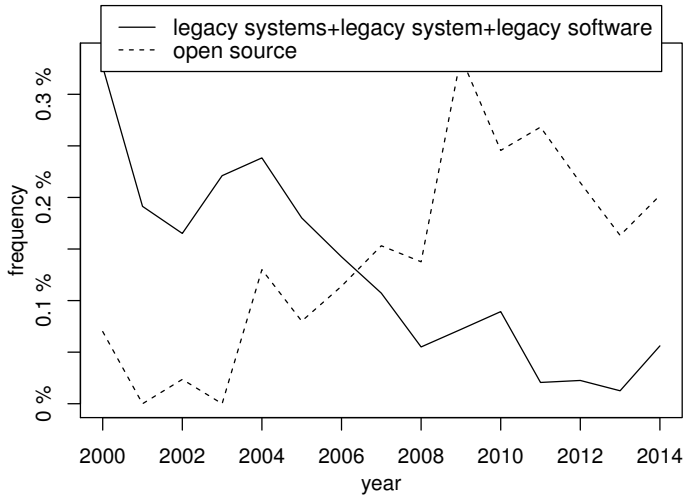


Figure 2.5: Studied systems – legacy vs. open source. © 2015 IEEE

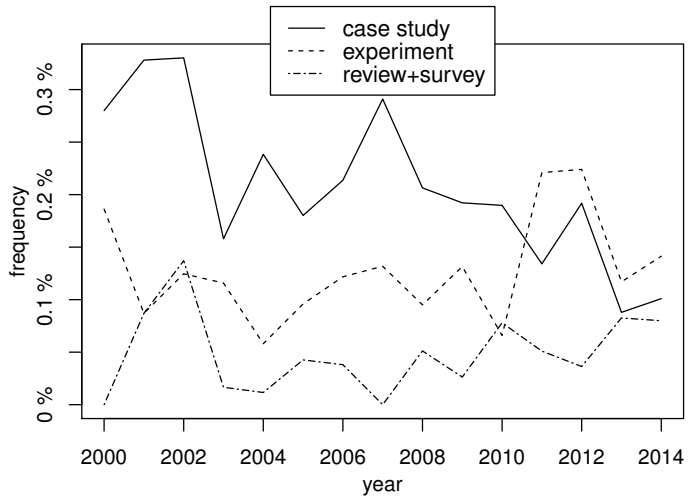


Figure 2.6: Types of research in program comprehension. © 2015 IEEE

2.4.3 Research Methods

We can see in Fig. 2.6 that case studies have a slightly decreasing tendency. There is a sudden rise of experiments in 2011, when they even outperformed case studies. Naturally, reviews and surveys are the least common types as they summarize existing research results.

2.4.4 Conclusion

We analyzed the trending phrases in bibliographies to see the rising and falling ones. The most rising trends are feature (and concept) location and the study of open source systems. Program slicing and the study of legacy systems are the most falling trends.

Chapter 3

Buildability of Software Systems

Being able to build a software system from its source code is a prerequisite for almost all comprehension activities. In the cases when we aim to label parts of source code with metadata using runtime information, this requirements is especially important: It is necessary to have both a runnable software system and its source code, while this runnable system was produced exactly from the version of the source code we possess. Consider the following three situations:

- A researcher wants to use an open source system in his next program comprehension experiment. He downloads the software project of interest in a form of a source code archive, opens it in an IDE (integrated development environment) and tries to build it.
- A student is willing to contribute to her favorite open source program with a new feature. She pulls the source from the project's version control system. Before starting the actual work, she checks whether the project can be built using a command-line interface of the build system.
- A practitioner is trying to fix a bug in a third-party library their company extensively uses. Again, the necessary precondition is to make sure the library builds.

All three situations have something in common: The message "Build failed" is likely to appear. Instead of doing useful work, the developers start to inspect cryptic error messages, search them on the internet, modify project configuration files, manually install packages and configure paths. This can take hours to fix – or, at worst, the developer gives up. Such situations significantly hinder the program comprehension efforts of developers.

3.1 Synopsis

In the broadest sense, the purpose of a build system is to generate output data given a set of inputs [230]. Considering a more specific, typical scenario, a build system reads a buildfile – e.g., a Maven POM (Project Object Model) file. Using the supplied information, it downloads necessary third-party components. Next, it executes a compiler to produce binary files from textual source code. Finally, it packages the program in a format suitable for deployment.

As tools like Make have existed since the 70's [69], build systems are often perceived as a solved problem by researchers [175]. Nevertheless, developers often encounter build problems which negatively affect their work [112]. While there exists anecdotal evidence that

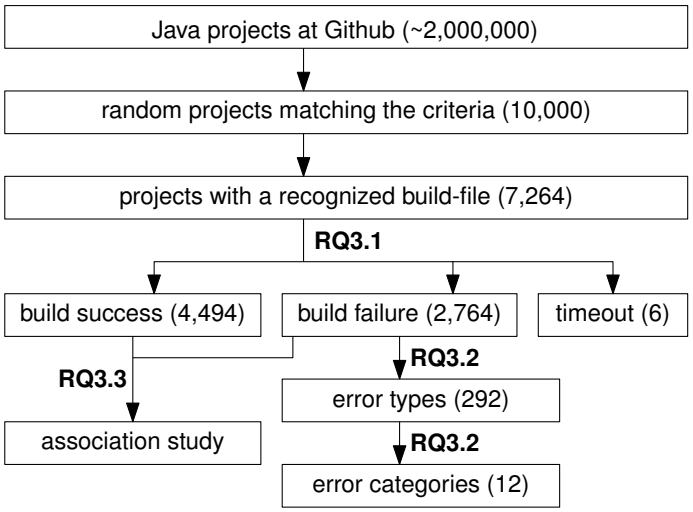


Figure 3.1: Build study overview

open source systems are sometimes difficult to build [175], large-scale studies on a substantial number of projects are rare. In this chapter[1], we aim to provide empirical evidence that build system invocations of many open source projects fail. Furthermore, we will look at the reasons of failures and factors affecting them.

A software system should be buildable from source in one step [232]. In the Java ecosystem, we expect an invocation of a build system like Maven, Gradle, or Ant to represent this step.

We strive to simulate a simple programming environment of a Java developer, download a large number of open source Java projects from the software forge GitHub, fully automatically execute the build command for each of them, and observe the build process outcomes.

We formulate our research questions as follows:

RQ3.1 *What portion of projects fails to build?*

RQ3.2 *What types of build errors do occur most frequently?*

RQ3.3 *Is there an association between the build success/failure and other project's properties (e.g., the used build tool, age)?*

For a high-level overview, see Figure 3.1. In section 3.2, we answer **RQ3.1** by a simulation study. Using the collected data, a semi-automated classification of failed builds (**RQ3.2**) is presented in section 3.3. An association study is also performed on the data (**RQ3.3**), which is described in section 3.4.

3.2 Build Failure Proportion

In this section, we answer the first research question.

3.2.1 Method

The data collection process consisted of two steps: obtaining a list of suitable projects, and the building process itself. The whole process was fully automated, controlled by a script.

[1] The content of this chapter was published in our article [251].

Table 3.1: The studied build systems (auxiliary log-formatting arguments were omitted)

Tool	File name	Build command
Gradle	build.gradle	`gradle clean assemble`
Maven	pom.xml	`mvn clean package -DskipTests`
Ant	build.xml	`ant clean; ant jar || ant war || ant dist || ant`

Project Selection

From a set of all public repositories at a large software forge, GitHub, we selected a random sample of projects corresponding to the following inclusion criteria:

- projects written in Java,
- not using JNI (Java Native Interface), Android or Java Micro Edition (ME) APIs,
- having an open source license,
- and forked at least once.

Now we will describe the rationale behind the criteria and details of the process.

In this study, we focused only on projects written in the Java language. Java seems like an ideal choice, since it is a compiled language and it offers very feature-rich build systems. To determine the project's main language, the GitHub API[2] was used. There were approximately 2,000,000 Java projects on GitHub at the time the study was conducted (see Figure 3.1).

We were interested only in open source projects. To assess whether a particular project is open-source, we utilized the information supplied by the GitHub API. GitHub produces these data by matching the contents of "License" files with a set of well-known licenses. Around 20% of GitHub repositories contain a recognized license.[3]

Only repositories with at least one fork were included. First, this indicates that there is some interest in cooperation on a project. Second, this criterion lowered the number of results by a factor of 10 to keep the number of searched repositories reasonable.[4]

We requested the GitHub API with a combination of queries corresponding to the above criteria. From the list of returned repositories, a random sample was selected. The most recent version (the current "master" branch) of each project was downloaded in a form of a tarball. The total number of downloaded archives was 14,567.

After extraction of the archives, the presence of JNI, Android and Java ME APIs was tested using file name and content patterns. The reason for exclusion of projects using these APIs was to maximize internal validity. Thanks to JNI, Java source code could call methods written in other languages, while we were interested in pure Java. We would be no longer testing just Java build systems, but also other toolchains outside the Java ecosystem, which also applies to Android and Java ME.

The resulting set of random projects, corresponding to the inclusion criteria, consists of 10,000 projects.

Build Process

For each of the 10,000 projects, a primary build tool was recognized by the script. In our study, we focused on three popular build systems: Gradle, Maven and Ant. To assess which

[2] http://developer.github.com/v3/

[3] http://github.com/blog/1964-open-source-license-usage-on-github-com

[4] The GitHub API does not support random selection of repositories, so we needed to retrieve metadata for all projects matching the criteria and sample them locally.

build system a particular project uses, the presence of a build-file was checked in the project's root folder. For a list of file names, see Table 3.1, column "File name". In case multiple build-files are found in the root directory, Gradle takes precedence over Maven, and Maven has higher priority than Ant. The ordering is based on an assumption that newer systems are used as a primary build tool, while preserving the legacy ones for compatibility.

If a project contained build-file(s) only in non-root directories, or did not include a build-file at all, we excluded it from further processing. This leaves us with 7,264 projects.

The builds were executed using the lightweight virtualization platform Docker[5]. The goal was to simulate a simple yet functional software environment of a Java programmer. The Docker image (available at http://quay.io/sulir/builds) contained the following software:

- the Linux distribution Fedora,
- Java SE Development Kit (JDK) 8,
- Gradle, Maven and Ant build tools,
- the dependency manager Ivy,
- the version control system Git,
- a Ruby interpreter (to execute the control script),
- and the "tar" and "unzip" utilities.

The control script executed the build process for each project sequentially. For every project, an appropriate build command was run, corresponding to the project's build tool – see Table 3.1, column "Build command". Their general goal is to produce a runnable Java archive from source files. The underlying idea is that a developer should be able to execute the corresponding command on any project using the given build tool, without reading and following any complicated instructions or modifying the project's files, and the build output should be generated successfully. In cases when the build tool enables exclusion of tests via command-line arguments, we utilized it, since the output archive can be generated and software can be often successfully executed even if tests fail.

During each build process execution, both standard and error output streams of a build process were redirected to a log file. The exit code of the process was recorded: a non-zero exit code signifies failure (or timeout). The execution time of one build was limited to one hour. This was necessary since we encountered infinite builds during pilot runs. The project data – exit codes and auxiliary metrics like the number of files in source archives – were stored in a CSV (comma-separated values) file for further analysis.

3.2.2 Results

In total, 72.64% of 10,000 projects have a recognized buildfile in its root directory. The most popular tool is Maven – see Figure 3.2.

Among other findings, 8.54% of projects contained such buildfile in a non-root directory. Since Ant does not provide dependency resolution capabilities itself, we were interested in how many projects using it also utilize the dependency manager Ivy. The number is relatively low: 10.18%.

In Figure 3.3, we see that 38.05% of builds failed. This means in almost 4 of 10 cases, the programmer would not be able to produce a target archive from the source code without manual intervention.

A negligible part of builds did not meet the time requirement. Some of the reasons are lengthy dependency resolution and downloading, and expecting user input.

[5] http://www.docker.com

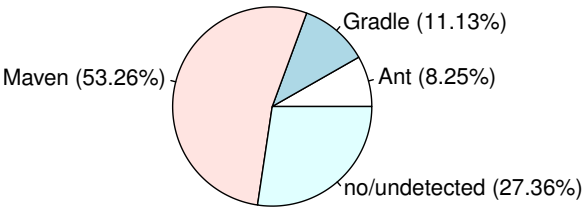


Figure 3.2: Recognized build tools

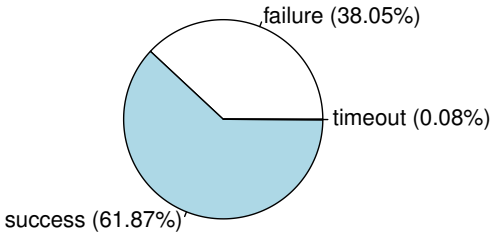


Figure 3.3: Build success vs. failure

3.3 Build Error Types

In this section, we will analyze the reasons of the mentioned failures.

3.3.1 Method

To determine the kind of failure, the logs were searched for the presence of certain patterns. Since each build tool has its own log format, first we associated a buildtool-dependent, machine-readable error *type* with each log (e.g., "MojoFailure:maven-compiler-plugin"). This process was automated. Second, a significant portion of error types was manually grouped into tool-independent, human-readable *categories* (e.g, "Java compilation").

Maven

A failing build is caused by a thrown Java exception, which is present in the log. Therefore, we decided to use exception class names as a Maven error types, with the "Exception" suffix removed for brevity.

However, the exception can be chained. For example, a `LifecycleExecutionException` can be caused by a `CompilationFailureException`, which can be caused by another one, etc. Fortunately, Maven logs contain a URL of a web page describing the relevant exception. We extracted the exception class name from this URL. In cases when multiple URLs were present, we extracted the class name from the last URL since it usually contained the most specific exception.

Some exception class types, e.g. `MojoExecutionException`, are too vague. In such cases, we searched for a presence of the pattern "Failed to execute goal. . . " and extracted the Maven plugin name which caused the failure, for instance, `maven-javadoc-plugin`. We then labeled the project with an error type in the form of "ExceptionClass:plugin-name" (see Table 3.2 for examples).

Gradle

For Gradle, the thrown exception class was considered too. Since Gradle does not include a URL in its logs like Maven, we analyzed the printed Java exception ourselves. The situation was more complicated because Gradle sometimes utilizes multi-cause exceptions, when one exception is caused by more than one other exceptions, forming "exception trees". Assigning multiple error types to one build would unnecessarily complicate further analysis. Therefore, after a manual inspection of a subset of Gradle logs, we decided to use the following method to find the most relevant exception: If it was a simple (unchained) exception, it was selected. Otherwise, the first most direct cause of the root exception was selected. For example, in the exception tree "A caused by B and C (caused by D)", we selected the exception B.

Two exception classes were too vague, so we enriched the error type with the name of the failed task in these cases. Finally, for the `PluginApplicationException`, the failing plugin name was appended.

Ant

Since Ant throws the same exception (`BuildException`) for all possible build failure reasons, it is useless for error classification. However, Ant prints names of individual targets as it executes them. Therefore, we were able to extract the last executed target – i.e., the failing one. The error type is in the form "Target:target-name" in this case. If the build failed before even one target was started, we analyzed the log for a presence of frequently occurring natural language patterns, determined by a manual inspection of the remaining logs.

Categorization of Error Types

In total, there were 292 different error types. Our goal was to categorize them, using human-readable names. Since this process was performed manually and the number of error types was large, we decided to categorize only more frequently occurring error types. While just 84 error types were categorized, they represented more than 86.5% of build failures.

The categorization was performed by one of the authors. He discussed the documentation, inspected sample logs, and searched web forums to seek help with the categorization. When he was unsure, he left the error type uncategorized.

3.3.2 Results

First, we will present error types for individual build tools. Next, a overview of error categories will be presented.

3.3.3 Error Types

The most frequently occurring Maven error types are displayed in Table 3.2. A very large portion of Maven builds ends with the "DependencyResolution" error. This includes temporary and permanent network problems of the servers hosting the dependencies, authorization errors, missing files on servers due to reasons like discontinued dependency hosting or removed older versions, invalid POM files, and many others.

For the most frequent Gradle error types, see Table 3.3. At the top, there is a "CompilationFailed" error which includes traditional Java compilation errors like undefined symbols, missing packages, incompatible types, etc.

Table 3.4 displays Ant error types. The most frequently failing target is "compile".

Table 3.2: The most frequent Maven error types

Error type	**%**
DependencyResolution	39.11
MojoFailure:maven-compiler-plugin	18.16
UnresolvableModel	10.59
MojoExecution:maven-javadoc-plugin	8.17
ProjectBuilding	2.47
PluginResolution	2.41
MojoExecution:git-commit-id-plugin	1.70
PluginManager	1.65
MojoExecution:maven-antrun-plugin	1.37
MojoExecution:exec-maven-plugin	1.10
MojoExecution:maven-enforcer-plugin	1.04
AetherClassNotFound	0.77

Table 3.3: The most frequent Gradle error types

Error type	**%**
CompilationFailed	22.36
MissingProperty	11.59
PluginApplication:ShadowJavaPlugin	7.66
Gradle:javadoc	5.59
InvalidUserData	5.18
ModuleVersionNotFound	3.93
Exec	3.52
ModuleVersionResolve	3.52
Resolve	2.69
IllegalArgument	2.48
CustomMessageMissingMethod	2.07
MultipleCompilationErrors	1.86

Table 3.4: The most frequent Ant error types

Error type	**%**
Target:compile	21.83
CannotImport	17.03
Target:-do-compile	12.23
Target:build	3.28
Target:build-project	2.84
unknown	2.84
Target:compile-import-shared	2.62
Target:-init-check	1.75
Target:jar	1.75
MissingPath	1.53
Target:-do-init	1.53
TaskdefNotFound	1.31

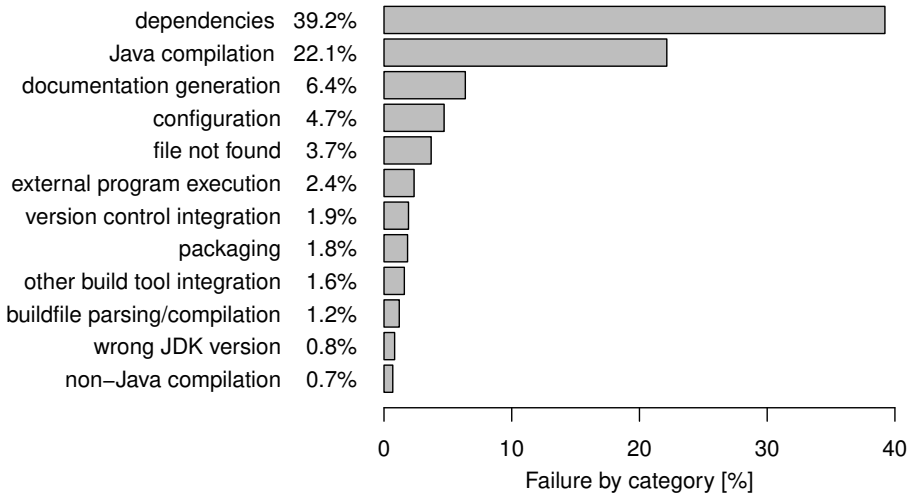


Figure 3.4: Build error categories

3.3.4 Error Categories

Error categories combined for all build tools are shown in Figure 3.4. Notably the largest portion of builds are dependency-related (39.2%). This is caused by Maven builds to a large degree – more than a half of failed Maven builds end with various dependency-related exceptions.

Compilation of Java source code caused 22.1% of failures. Note that some compilation failures might be a consequence of missing dependencies [224].

Surprisingly, 6.4% of failures occurred during documentation generation. Some builds failed because of missing or invalid configuration properties (4.7%) and missing files or directories (3.7%).

3.4 Relation of Build Results to Project Properties

We hypothesize there is some form of association between the build result of a project and its properties. Specifically, we consider the project's build tool, size, popularity, age and last update recency.

The hypotheses are formulated as follows:

H3.1 *The probability that a build fails depends on the build tool which the project uses.*

H3.2 *Builds of larger projects fail more likely than that of smaller ones.* Project size was measured by a file count, as suggested by McIntosh et al. [157].

H3.3 *Failing project are less popular, in terms of "stars" received on GitHub.*

H3.4 *Older projects, considering creation dates, fail more likely.*

H3.5 *More recently updated projects have a higher probability of a passing build.*

3.4.1 Method

The build status is a nominal variable with two possible values: "pass" and "fail". The build tool is a nominal variable, while all other project properties are numeric (interval).

To be usable in statistical tests, dates must be converted to numeric values. We decided to use relative measures for readability reasons. Project age was measured as a number of days

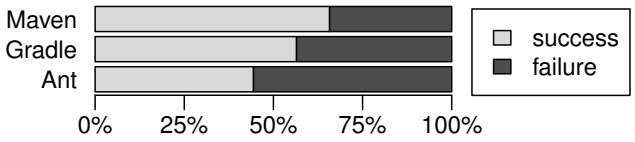


Figure 3.5: Build status for individual build tools

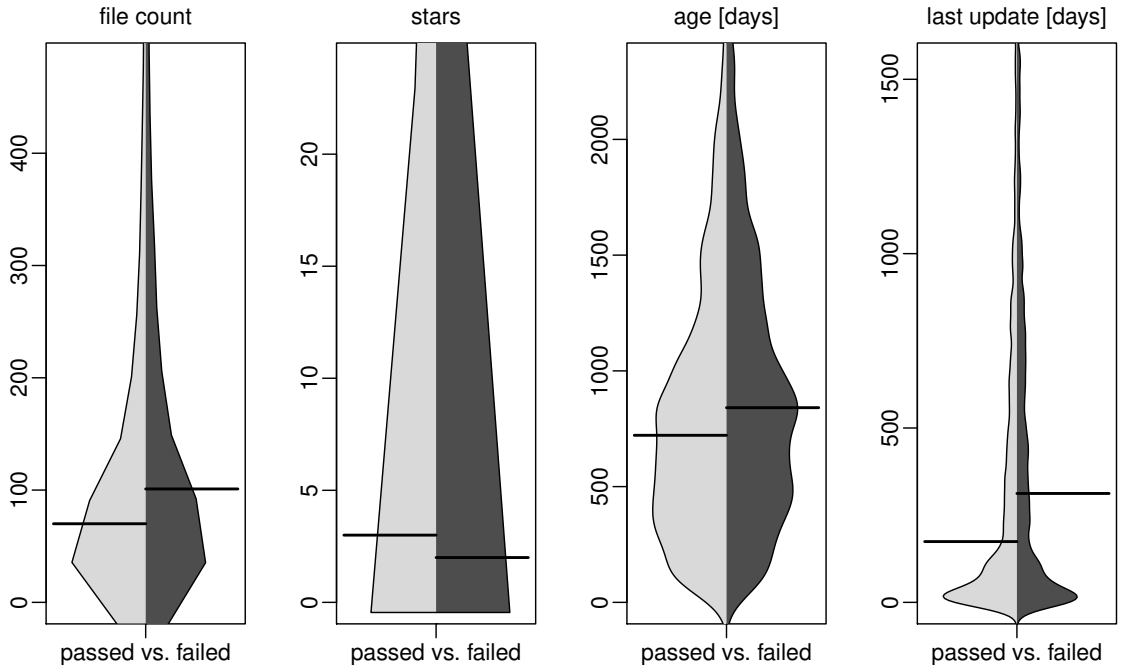


Figure 3.6: Beanplots (distributions+medians) of project characteristics

between the creation date of a particular project and of the newest project. A similar measure was used for recency – using the "last updated" dates.

To determine whether the dependence of the build status and build tool is statistically significant, we used the Pearson's chi-squared test. To assess the dependence of build status and all other properties, a two-sided Mann-Whithey U test was performed. A confidence level of 99% was used (p-value should be < 0.01).

3.4.2 Results

The chart of projects' build status for each build tool is in Figure 3.5. Maven builds tend to be the most successful (65.8%), while Ant fails the most often (44.4% success rate). The differences are statistically significant (p-value < 0.0001), confirming **H3.1**. Cramér's V is 0.146, which means small to medium effect size.

In Figure 3.6, there are four beanplots depicting distributions and medians (horizontal lines) of file count, star count, age and update recency for projects with passed (left side) vs. failed (right side) builds. Outliers were trimmed from the display, but preserved in calculations.

Projects with a successful build had a median of 70 source files, while this number was 101 for the failing ones (a 44% difference). This is natural – larger projects are usually more complex and thus more error-prone. The result is statistically significant, with a p-value of less than 0.0001, so **H3.2** is accepted.

There is a difference in the number of "stars" for the two groups – a median of 3 vs. 2. The difference is not statistically significant ($p = 0.0226$).

When considering dates of creation (**H3.4**), there is ~17% difference of age between passing (median ~722 days) and failing (841 days) builds. For a date of last update (**H3.5**), the situation is similar, but the difference is more striking: 174 vs. 312, i.e, a difference of 79%. Both results are statistically significant ($p < 0.0001$). A possible explanation is that build scripts require maintenance – even if the project is untouched – to deal with changes in build tools themselves. After inspecting the results for individual build tools, Maven builds showed the largest difference.

Note that this is only an association study – it does not mean that simply selecting Maven will make your project less prone to build failures. There may be complex relationships among multiple factors. For example, Ant projects tend to be older and slightly larger than other.

3.5 Discussion

Some portion of the failed projects could be buildable if we manually read the README file and followed the instructions provided. We argue that automated builds are not fully automated if they require a programmer to manually download dependencies, edit configuration files or perform other steps.

Nevertheless, we tried to manually download and build 3 random failing projects, using instructions provided in the README files (if present). If errors occurred anyway, we tried to correct them.

The first build failed even after we followed the instructions. It was successfully build only after we downgraded Gradle – the build system itself – from version 2.14 to 2.10. This shows that some projects require specific older versions of tools. An interesting fact is that the project was last updated less than a year before the study execution, so the "project rot" can occur quite fast.

The second project, with a name ending with "-core", failed because of a missing JAR file. This file could be produced by manually downloading and building a related "-utils" project. In README, there was only a brief mention that the project depends on the "utils" package. It is questionable why automated dependency management was not used, since both projects were using Maven, which has a built-in support for this.

The third project did not mention any building instructions in its README. Its build failed because a "snapshot" version of a third-party dependency was used. This version was probably available in the Maven repository in the past, but now it is not. After we corrected the version to a final one in the Maven POM file, a compilation error "package does not exist" occurred because of another missing dependency, unrelated to the previous one. Although we were trying to edit the POM file to download this dependency, builds were always failing with various download errors. We even tried downloading necessary JARs manually, but without success.

3.6 Threats to Validity

First, our results are valid only for the presented project selection criteria. Further studies should extend the criteria, particularly to include Android applications/libraries, which are becoming increasingly popular. Furthermore, about 80% of GitHub repositories do not

contain a license file[6] – including them could affect the results. Since analyzing GitHub alone has its disadvantages [106], we could also explore software forges other than GitHub.

The project's primary programming language was determined using the GitHub API, which uses a mapping of file extensions to languages, and returns a language in which the most bytes are written. This means the projects could utilize also other languages. However, we consider the presence of a Java build script in the project root directory a sign that the project (or its Java part) is buildable using Java tools. Furthermore, we excluded projects using JNI (although exclusion patterns for Android and JNI might not be faultless). Finally, only 2.4% of failures were caused by external program execution, and even only some portion of them were because of an absence of a specific tool.

The selection of tools in our virtual environment might not represent a typical developer's setup. Nevertheless, they represent a minimal toolchain to download further tools and libraries as dependencies if necessary.

For Gradle and Maven, tests were excluded from the build process. For Ant, there is no command-line switch available for test exclusion, so they could potentially run. Our goal was that a single, universal command should be executed, which should just produce an output archive, without performing other activities like testing and deployment. The ad-hoc nature of Ant makes it difficult, so we resorted to the nearest behavior possible – similar to what a real developer could do without modifying build scripts. Overall, less than 0.4% of failures are test-related, which makes this threat negligible.

Some build tools cache downloaded dependencies. Our script does not clean the cache directories between individual builds. While this could in theory decrease internal validity, external validity is strengthened – the programmer does not clean them often either.

Not all error types were categorized, so there may exist other categories, and the actual percentages of existing ones can slightly differ. However, 86.5% of failed builds correspond to categorized error types. A similar approach was used by Seo et al. [224] – they covered about 90% of their dataset.

Since Ant target names are generally free-form, they might not be the best categorization criterion. Using a custom logger, like in [156], we could extract task names to obtain more precise results.

Although categorization of error types was performed manually and solely by one researcher, this study is fully replicable and the analytical part is fully reproducible. The Docker image, scripts for analysis, and other materials are available at http://sulir.github.io/build-study.

3.7 Related Work

In this section, we will review some works related to build systems, with a focus on failure analysis.

Kerzazi et al. [112] found that 18% of automated builds of an industrial web application failed during a period of 6 months. In an exploratory study by Neitsch et al. [175], 4 of the 5 selected multilanguage Ubuntu packages could not be built or rebuilt without intervention. While these studies thoroughly investigated a small number of systems, our study takes a quantitative approach: it encompassed over 7,000 various projects.

A study [224] conducted at Google showed that 30% of developers' Java builds executed on a centralized build server failed. In contrast to our study, they used own proprietary

[6] http://github.com/blog/1964-open-source-license-usage-on-github-com

build system, analyzed only compiler messages instead of full build system logs, and studied multiple builds of the same systems over time.

According to Beller et al. [21], 59% of broken builds on a continuous integration server Travis CI were caused by a test failure. Our study complements these results by examining failure reasons other than unsuccessful tests.

Vasilescu et al. [272] described association of various project characteristics with the success of their build. They analyzed existing builds on a continuous integration server, while we created a virtual environment simulating a developer's local system.

McIntosh et al. [157] performed a study of build systems on a large number of projects. Some of their findings are consistent with ours – for example, that Maven requires the largest build maintenance effort. They were not interested in build successes and failures, though.

Among other characteristics, McIntosh et al. [156] studied the build-time length of open source projects. However, success/failure rates and failure reasons were not presented.

In a study by Hochstein and Jiao [92], 11% of regression test failures were due to failed builds. They investigated only one project, though.

3.8 Conclusion

In a virtual environment containing programming tools, we tried to automatically build more than 7,000 Java projects from GitHub, corresponding to the selection criteria. Answering **RQ3.1**, more than 38% of these projects failed to build. Regarding **RQ3.2**, the most frequent errors were dependency-related, followed by Java compilation and documentation generation. To answer **RQ3.3**, the likelihood of a build failure is associated with the build tool used; we also found that larger, older and less recently updated projects fail more.

Regarding the methodology, we presented an example of a simulation study in software engineering. In real life, we observed interesting behavior on a small number of projects. In a virtual environment, we tried to simulate the behavior which would be performed by programmers (i.e., building the projects) on a large sample and observed the outcomes.

While we did not study closed-source industrial systems themselves, they are also affected by this study since they often incorporate open source libraries in various ways.

3.9 Future Work

Especially for novices, error messages are often unreadable [151]. By determining what build error messages are the most common, we can focus on making them more programmer-friendly. For example, we can break a frequently occurring generic message into multiple specific ones.

External dependency management causes both maintenance effort [157] and build failures. Creating a system automatically generating and updating a build configuration by analyzing libraries in the source code would be useful.

Although we manifested the severity of build failures, we did not provide much guidance how to avoid them. It would be very useful to create a list of recommendations for programmers and build maintainers.

Researchers are welcome to perform further empirical studies using our images and scripts (http://sulir.github.io/build-study). Extending the scope to C, C++ and other languages, performing a longitudinal study – analyzing the history of successful and broken builds over time, or measuring build/rebuild time are only a few interesting future research directions.

Chapter 4

Labeling Source Code with Metadata

Often a developer needs to understand the program when looking at its source code, but the critical information he seeks is not present in the code. Consider the following examples.

A small, but tricky piece of code worked a few days ago, but now it does not. The programmer must open a web browser, navigate to the version control system (VCS) website and find the relevant commit. It refers to the issue tracking system, which is a separate website. After reading the whole, particularly long issue description and a multitude of related comments, he finally finds the reason of the malfunction.

Another programmer tries to comprehend a rather complicated algorithm. It is difficult to understand it just by looking at the code, so she decides to provide sample input data and debug the program using a built-in debugger of an IDE (Integrated Development Environment). While it is possible to display a value of any variable at any time, the debugger does not present any overview of values of a particular variable over time. Each time a program stops, the programmer must remember a value of interest and compare it with previous values in mind. As the capacity of short-term memory is very limited, the developer soon starts writing notes in a separate document. This in turn creates a burden of switching between two separate views (a split-attention effect [20]).

These two – at the first glance unrelated – scenarios have something in common: The information a developer needed was available *sometimes* or *somewhere*. But it was not available in the right place at the right time: in the IDE, and associated with the particular piece of code the developer was looking at.

Many researchers have realized this problem and provided various approaches, methods and tools to partially solve some of its aspects. However, as the research in this area is not very mature, authors use a rather large variety of terms to describe them. For this reason, we decide to provide an overview of existing approaches.

In this chapter[1], we will provide a preliminary systematic mapping study of selected existing approaches to label parts of source code with additional metadata with the purpose of program comprehension improvement. The sample of articles is categorized by four criteria, which form a taxonomy.

Our general research questions for this survey are:

RQ4.1 *What approaches (and tools implementing them) do exist to label parts of source*

[1] The content of this chapter was published in our article [254].

code with additional metadata and present them to a programmer in order to improve program comprehension?

RQ4.2 *How can these approaches be categorized?*

4.1 Method

We decided to conduct a systematic mapping study, which is a form of a systematic literature review (SLR). In contrast to an SLR, a mapping study has more general research questions [190] and the main goal is to classify research to categories, rather than provide precise quantitative results [117].

4.1.1 Search Strategy

Since our view of literature through a notion of "source code labeling" is not very common and the terminology is inconsistent, we decided to try multiple different search strategies and combine their results.

Manual Search

First, we performed a manual search among all articles published in 8 journals and 4 conferences, selected by the authors' discretion (partially inspired by a list in [228]). In this first part of the search process, arbitrary two years (2009 and 2012 in our case) were selected, as suggested by [286]. The journals of interest were:

- IEEE Transactions on Software Engineering (TSE),
- ACM Transactions on Software Engineering and Methodology (TOSEM),
- Computer Languages, Systems and Structures (COMLAN),
- Science of Computer Programming (SCP),
- Journal of Systems and Software (JSS),
- Empirical Software Engineering (ESE),
- Information and Software Technology (IST),
- Journal of Software: Evolution and Process (JSEP), formerly known as Journal of Software Maintenance and Evolution (JSME),

and conferences:

- International Conference on Software Engineering (ICSE),
- International Conference on Program Comprehension (ICPC),
- Working Conference on Reverse Engineering (WCRE)
- and International Conference on Software Maintenance (ICSM).

A Scopus[2] query was constructed based on the criteria, the results list was exported as a CSV file and inspected in a spreadsheet processing program. A total of 1546 articles were manually assessed based on titles and abstracts, resulting in a list of 16 relevant articles.

Keyword Search

Continuing the methodology of Zhang et al. [286], we inspected the terminology used in articles obtained during the manual search and based on it, we constructed and tried multiple keyword-based search queries. The final Scopus query is as follows:

[2] http://www.scopus.com

```
TITLE-ABS-KEY(
  ("source code" OR "program comprehension")
  AND ("tagging" OR "enriching" OR "augmenting" OR "labeling")
) AND SUBJAREA(COMP)
AND NOT SUBJAREA(bioc OR medi OR envi OR neur)
```

Basically, it searches the specified terms in titles, abstracts and keywords of computer science literature, excluding interdisciplinary research. The search yielded 85 results, of which 6 were newly found relevant ones.

The methodology by Zhang et al. [286] prescribes trying slightly different queries until one of them returns at least 80% of articles from the first (manual) phase. For the query presented above, this number was far below 20%. Broadening the terms caused the count of results to skyrocket. The number of false positives was high, without a significant positive impact on the relevant result count. For this reason, we decided to leave the methodology and continue with other techniques.

References Search

We searched for all forward and backward references of 22 articles collected so far. Again, we used Scopus.

Backward references mean all articles cited in the "References" section of particular papers. They are generally older than the article citing them. From 305 results, we considered 14 unique and relevant.

Forward references are articles for which a search engine knows they cite a particular paper. This is useful to find newer articles. Of 137 results, 3 were relevant and not yet found in previous searches.

Other Sources

Five more relevant articles were found recursively in the references of articles found during the phase of references search. Finally, we added three more papers present in the authors' personal bibliography.

4.1.2 Inclusion Criteria

During the selection process, a paper was considered relevant if:

- it presented a new approach or tool to associate metadata with pieces of source code,
- the purpose of these metadata was to improve program comprehension
- and a form of presentation of these data to a programmer was described.

Examples of excluded articles are papers describing a labeling algorithm without discussing how to present results to developers, and purely empirical studies comparing existing approaches.

4.1.3 Final Article List

From the 47 selected articles, 17 were just descriptions of the same or similar idea in another research phase. Five were considered irrelevant after skimming or reading the full text.

The final article list thus contains 25 articles. For an overview, see Table 4.2. Further details will be provided in section 4.3.

Table 4.1: Source code labeling taxonomy

Dimension	Attribute	Description
Source	*human*	Manually entered information, previously present only in human mind.
	code	Results of static source code analysis.
	runtime	Results of the program execution; dynamic program analysis.
	interaction	Interaction patterns of a single developer in the IDE.
	collaboration	Collaboration artifacts of multiple developers like VCS commits or e-mails.
Target	*folder*	A directory or a package.
	file	A file or a class.
	multi-line	A multi-line part of a file, e.g., a method.
	line	One line in a file, such as a variable declaration.
	line part	A character range, e.g., a method call.
Presentation	*code view*	The editable source code view is augmented with metadata.
	existing view	Other existing views in an IDE (e.g., a package explorer) are augmented.
	separate view	A separate view is created just to present the information of interest.
Persistence	*internal*	The metadata is stored directly in the source code file (e.g., using a comment).
	external	A separate file, database or server is used to store the labels.
	none/unknown	The metadata are only presented to the user, but not stored; or a method of persistence was not mentioned in the article.

4.1.4 Data Extraction

The full text of 25 relevant articles was read, carefully watching for similar and distinguishing signs regarding source code labeling. Succinct notes about each article were written in a tabular form, gradually forming a taxonomy.

4.2 Taxonomy

First, we will introduce our taxonomy, answering **RQ4.2**. Similar to Dit et al. [59], articles (approaches) were evaluated according to multiple criteria, called dimensions. For each dimension, an article can belong to one or more attributes.

Our taxonomy has four dimensions: source, target, presentation and persistence. For an overview, see Table 4.1. Now we will describe the dimensions and attributes in detail.

4.2.1 Source

A "source" dimension denotes where the metadata were originally available before they were assigned to a part of source code. The most problematic source is human mind. In order to obtain information present only in the memory of the programmer, he must manually enter these data into a system for each artifact which should be labeled. The most primitive kind of a label with a "source" of type *human* is a traditional source code comment. The developer writes the label – a natural language text easing program comprehension – above a piece

of code. The assignment of the comment to a piece of code is therefore performed by its positioning.

Approaches categorized as *code* analyze the source code of a system without executing it, i.e., using static analysis.

Useful metadata can be collected by execution of the program, using some form of dynamic analysis. These approaches are marked as *runtime*. The analyzed program must be buildable (which is often a problem [251]) and automated tests should be available (or the program must be executed manually).

For tools utilizing the *human* source, a programmer must purposefully enter the metadata with a sole intention that they will improve program comprehension. This is expensive on human resources. On the other hand, *interaction* data are collected automatically, possibly without the developer even knowing it (although that would be unethical). Using heuristics, these tools can infer relationships between artifacts from captured keystrokes, mouse actions, or even eye gazes [275].

As software engineering is not an individual activity, collaboration artifacts are formed naturally as the team communicates and collaborates. These artifacts include e-mails, instant messages, forum posts and VCS commit messages. These artifacts are often poorly or nowise connected with the relevant source code. The purpose of *collaboration* approaches to code labeling is to fill this gap.

Many tools use a combination of multiple methods. For example, a static analysis may be used to assign source code artifacts their documentation; then a user must manually confirm or reject the suggested links [16].

4.2.2 Target

The purpose of source code labeling is to assign metadata to a particular piece of code. Subject of the "target" dimension is what that "piece of code" means.

This is quite a problematic question, as there are two separate views: file-based and element-based. Some tools assign metadata to files, lines and character ranges. Other assign them to classes, methods, variables, method calls, etc. These views are often mixed in one tool. Furthermore, it is not clear whether a code bookmark, labeling a line containing just a variable declaration, relates to the line or to the declaration. Therefore, we decided to mix the views in our taxonomy.

The attributes are sorted according to granularity. A *folder* represents a package in some object-oriented languages. A *file* often corresponds to a whole class. Method and function definitions are *multi-line* elements. Elements considered *line* include variable declarations. We decided to consider method calls and variable usages *line parts*.

4.2.3 Presentation

Once the metadata were retrieved and associated with a proper source code segment, it should be presented to the developer in order to be useful.

One of the best places to show code-related data is obviously the main, editable source *code view* of an IDE. This is the place which draws the most attention of a programmer, occupies a large screen portion and offers many existing features (e.g., code completion). The code editor can be augmented by various coloring, visual overlays (like a box surrounding a piece of code) or images. Harward et al. [88] call them "in situ" visualizations. Syntax highlighting may be considered a common visual augmentation [262]. We decided to regard

also gutter/ruler annotations (icons in the left or right code editor margin) as a *code view* presentation.

Except for the source code editor, modern IDEs offer various supplemental views like Package Explorer, Favorites or Class Hierarchy. Plugins can augment *existing views* by additional information. For instance, packages and classes in a tree view may be augmented by colored squares according to a metric [212].

Some tools, despite associating a part of code with additional information, display the association in a *separate* view, or even window. In a better case, clicking a particular widget in this view automatically opens and focuses the code of interest in the source code editor. Otherwise, the user must manually open and find the code according to the displayed element name.

4.2.4 Persistence

The association between the code and the label, and sometimes also the label data themselves, can be saved to a permanent storage.

Rationale

There are three possible reasons for persistence.

First, in the case of fully automated static *code* and *collaboration*-sourced techniques, the process of retrieving and associating metadata can be resource-intensive. The storage acts as a cache[3].

Second, the *runtime* and *interaction* data are partially a product of human work. Each program execution and IDE interaction can be unique. It is therefore desirable to save at least the data like traces or interaction logs. Once they are persisted, the analysis and association process can be performed every time when necessary.

Third, the persistence is an absolute requirement in the case of *human*-sourced metadata. A tool must not repeatedly ask a programmer to describe code, if exactly the same information had already been entered for the same piece of code, using the same tool.

Persistence Methods

The labels can be stored in a source code file itself. The association with the target element is thus implicit through the position of the label in the code. A typical example of *internal* persistence method is a specially formatted comment.

External persistence means labels are stored separately from the source code file. They can be saved in an XML file, or a database management system (DBMS), accessed either locally or through a server.

Finally, the persistence method *none/unknown* denotes the labels are either not stored permanently, or persistence was not mentioned in a given article at all. If a tool produces only reports or exports which cannot be subsequently loaded, it was also incorporated into this category.

[3] Suppose collaboration artifacts are already persisted in external systems.

Table 4.2: Labeling approaches and tools

Article	Source					Target					Presentation			Persistence		
	human	code	runtime	interaction	collaboration	folder	file	multi-line	line	line part	code view	existing view	separate view	internal	external	none/unknown
ConcernMapper [208]	■	□	□	□	□	□	□	■	■	□	□	■	■	□	■	□
eMoose [56]	■	□	□	□	□	□	□	■	□	□	■	□	□	■	■	□
Pollicino [85]	■	□	□	□	□	□	□	□	■	□	■	□	■	□	■	□
SE-Editor [222]	■	□	□	□	□	■	■	■	■	□	■	□	□	■	□	□
Spotlight [205]	■	□	□	□	□	□	■	■	■	■	■	□	□	□	■	□
TagSEA [241]	■	□	□	□	□	□	■	■	■	□	■	□	■	■	□	□
DepDigger [23]	□	■	□	□	□	□	□	□	□	■	■	□	■	□	□	■
RegViz [18]	□	■	□	□	□	□	□	□	□	■	■	□	■	□	□	■
Stacksplorer [109]	□	■	□	□	□	□	□	■	□	■	■	□	□	□	□	■
Traceclipse [118]	■	■	□	□	□	■	■	■	□	□	□	□	■	□	■	□
TraceME [16]	■	■	□	□	□	□	■	□	□	□	□	□	■	□	■	□
GUITA [220]	□	□	■	□	□	□	□	□	□	■	■	□	■	□	□	■
Impromptu HUD [262]	□	□	■	□	□	□	□	■	■	□	■	□	□	□	□	■
in situ profiler [20]	□	□	■	□	□	□	□	■	□	■	■	□	□	□	□	■
Senseo [212]	□	□	■	□	□	■	■	■	□	■	■	■	■	□	■	□
sparklines [19]	□	□	■	□	□	□	□	□	■	□	■	□	□	□	□	■
CnP [97]	■	■	□	■	□	□	■	■	■	■	■	□	□	□	■	□
HeatMaps [213]	□	□	□	■	■	□	■	■	□	□	□	■	□	□	■	□
iTrace [275]	■	□	□	■	□	□	■	■	■	□	□	□	■	□	■	□
Deep Intellisense [94]	□	□	□	□	■	□	□	■	□	□	□	□	■	□	■	□
Miler [9]	□	■	□	□	■	□	■	□	□	□	■	■	■	□	■	□
Rationalizer [31]	□	□	□	□	■	□	□	□	■	□	■	□	□	□	■	□
101companies [67]	■	■	□	□	□	□	■	■	■	■	□	□	■	□	■	□
Code Bubbles [32]	■	■	■	□	□	□	■	■	■	■	■	■	■	□	■	□
CoderChrome [88]	□	■	□	■	■	□	□	■	■	■	■	□	□	□	□	■

4.3 Approaches and Tools

In Table 4.2, we can see an overview of all reviewed approaches and tools, which answers **RQ4.1**. Now we will briefly describe each of them. The approaches will be grouped by their primary "source".

4.3.1 Human

A concern is a piece of information about a code element, such as a feature it implements or a design decision [247]. ConcernMapper [208] is both an Eclipse plugin and a framework to associate parts of programs with concerns, both through a GUI (graphical user interface) and an API (application programming interface).

The eMoose approach [56] allows tagging of API usage directives, like state restrictions

or locking, in JavaDoc comments. Subsequent highlighting of calls to such methods in an IDE improves awareness of programmers, who then spot errors quickly.

Pollicino [85] is a plugin for collective code bookmarks, providing a more feature-rich version of classical source code bookmarks found in the majority of IDEs.

SE-Editor [222] makes it possible to embed web pages in the source code editor of the Eclipse IDE. The web page is embedded using a comment beginning with `/***`, followed by an URL. This might be useful to display code-related diagrams, tutorial videos, etc.

Spotlight [205] is an IDE plugin to tag source code with concerns. Each concern can be assigned a color, which is then displayed in the left gutter (margin) of the source code editor. Thanks to this, a programmer can more easily identify where the given concern is located in the program.

TagSEA [241] integrates waypoints and social tagging into the Eclipse IDE. Programmers note waypoints – places worth marking and sharing – into the source code as comments in the form `//@tag tagname : message`. Hierarchical tags and metadata (author, date) are also supported. Collections of waypoints can be connected into routes to create source code guides.

4.3.2 Code

DepDigger [23] visualizes the measure (metric) dep-degree by changing the background color of source code elements – on a scale ranging from white to red – in a code view.

RegViz [18] visually augments a regular expression in-place, without a need for a separate view. For example, groups are underlined and labeled with a group number.

Stacksplorer [109] visualizes method's callers in a column on the left of the code editor view, callees in the right one. Graphical overlays may be shown to visually connect the current method definition with the left column and method calls in the source code view to items in the right column.

Traceclipse [118] and TraceME [16] are traceability management recovery tools. They link source artifacts (like documentation) to the target artifacts (usually source code). In the mentioned tools, the linking is performed based on the textual similarity, using IR (information retrieval) methods.

4.3.3 Runtime

GUITA [220] annotates GUI-related method calls with GUI snapshots of a selected widget at the time when this method is called. This facilitates the navigation between the dynamic user interface world and the static source code world.

The next three approaches belong to a group of "in situ visualizations" – small graphical elements displayed directly in the source code editor. Impromptu HUD [262] displays a realtime clock-like visual near each scheduled function, informing the programmer when the timing event will fire. An in-situ profiler [20] shows small diagrams with runtime performance information next to method declarations and calls. Code sparklines [19] are small charts depicting values of a particular numeric variable over time.

Senseo [212] gathers and displays dynamic information in static views of an IDE. Information like methods' callers, callees, dynamic (overridden) argument types and return values are shown in a tooltip of a method in the code view. Selected metrics like execution frequencies, object allocation counts or memory consumption can be viewed in gutters/rulers (using heatmaps) and the package explorer (numbers). Dynamic collaborators of packages, classes and methods, and a Calling Context Ring Chart are displayed in a separate view. A

controlled experiment demonstrated an improvement of maintenance correctness and speed when using Senseo [212].

4.3.4 Interaction

The CnP tool [97] proactively tracks, visualizes and supports editing of code clones in an IDE. Instead of a batch input of source files, the tool captures copy and paste operations in Eclipse.

HeatMaps [213] display artifacts with a background color on a scale from blue to red, according to various numeric values assigned to them. The values are, for example, the recency and frequency of browsing and modification (obtained by instrumenting an IDE), artifact age, and a version count. The blue color means "cold" (e.g, least recently browsed), red "hot". The approach is general and can be potentially applied in many tools and to various views in an IDE.

Interaction-sourced approaches are not limited to traditional mouse and keyboard operations. iTrace [275] analyzes eye gazes (using an eye tracker) in an IDE to infer traceability links between artifacts.

4.3.5 Collaboration

Deep Intellisense [94] displays an interleaved list of relevant bugs, commits, e-mails, specifications and other documents for a given code element. Furthermore, a list of related people is shown. The lists are updated each time a user clicks on a code element, but they are displayed in a separate view. Rationalizer [31] integrates similar information directly into the code editor. On the right side of each source code line, it shows three columns: when this line was last modified, who, and why changed it. On the other hand, it processes only data from a VCS and an issue tracker.

Miler [9] is a toolset to retrieve, process and associate e-mail data to source code artifacts. E-mails are assigned to a source code based on textual analysis. Information about e-mails relevant to a class is displayed in the IDE's package explorer and rulers.

4.3.6 Mixed Approaches

The following approaches use multiple sources of information, without any of them being dominant.

101companies [67] is a software chrestomathy – a collection of many implementations of one system using various technologies, stored in a repository and linked with metadata. Metadata like an implementation language, dependence on a technology, features, and a highlighting renderer are assigned to files or file fragments using a rule-based DSL (domain specific language). The assignment can be performed according to criteria like a file extension, a regular expression for file content, or even by a separate script [67]. For example, all classes in files called "*.java", ending with "Listener", could be assigned the "observer design pattern" label.

Code Bubbles [32] is a working-set based IDE using fragments called bubbles instead of traditional file-based views. It supports various forms of labeling, ranging from arrows between method calls and definitions, to small images (e.g., a literal "bug") attachable to code bubbles.

CoderChrome [88] provides a generic framework for mapping between a metric and an in-situ augmentation. Examples of such visual augmentations are background colors, glyphs at the beginning or end of lines, and underlining. Metrics can range from categorical to

numeric ones, e.g.: a start or end of a block, the last author, the property of having a primitive type, the code age, and a line length.

4.4 Threats to Validity

The set of articles included in this study is by no means complete. Even during data extraction, we found multiple possibly relevant papers which were not included in the study. Nevertheless, due to a large number of papers pertaining to a research area, collecting all related articles is often unrealistic and it is just important for the selected subset to be representative with respect to the research field [190].

One could argue that the whole search process relies on one search resource – Scopus. However, during backward references search, also papers not indexed by Scopus were returned (so-called secondary documents in their terminology).

Article selection and data extraction was performed by a sole researcher, which could produce biased results based on subjective decisions. Brereton et al. [34] suggest either independent extraction by at least two researchers and then a comparison of results, or checking the data afterwards. In case of the lack of resources, a random sample of data may be cross-checked [117].

4.5 Related Work

Now we will mention works related to our review.

4.5.1 Other Labeling Forms

Artifacts other than source code can be labeled, too. One of such artifacts are execution traces. An approach called UI traces [259] splits large lists of method calls into multiple smaller ones, using UI (user interface) events. Each such fragment is labeled with an UI snapshot to facilitate feature location.

4.5.2 Location Referencing

Tools using an external persistence must use some form of location referencing. Reiss [203] compared various methods, e.g., storing an exact line content, context of a few lines around the line, AST matching, a diff-based approach, and their combinations. An experiment was performed, when automatic methods were compared to human judgment about which line was changed to which one (or deleted). The performance (space to store the reference and time) was also considered. A recommended method to track lines through changed versions of code is based on an approximate string matching of the line and exact matching of the lines around it.

A similar approach, using advanced string similarity algorithms, was described in [197].

4.5.3 Related Research Fields

The research area of feature location partially overlaps with source code labeling. Features can be considered one of possible source code labels. Dit et al. [59] reviewed 89 feature location techniques and classified them into a taxonomy. Our “source” dimension is similar to

their “type of analysis”; and “target” to “output”. On the other hand, we were more concerned about the presentation and persistence.

The *runtime* source in our taxonomy indicates a use of various dynamic analysis approaches. Cornelissen et al. [47] reviewed 176 articles related to dynamic source code analysis. However, labeling of source code with obtained information was out of scope of the mentioned survey.

Source code labeling with the *collaboration* source seeks to improve workspace awareness [238] – knowledge of the tasks and artifacts of others in a distributed software development team.

The goal of traceability research [280] is to allow following links among various forms of software artifacts. Often the target artifact is source code, just as in the case of source code labeling. Two traceability tools, Traceclipse [118] and TraceME [16], were included in this mapping study.

4.5.4 Terminology

The term “source code labeling” has multiple synonyms and near synonyms in the literature:

- source code tagging,
- augmenting,
- enriching,
- annotation
- and marking.

However, we decided to use the word “labeling” as the other ones are either too specific or ambiguous. The term tagging [241] is used mainly for short, textual metadata. Augmenting [20] and enriching [212] is usually concerned with visual presentation, using short-term graphical markers. The term annotation is ambiguous as it refers both to visual annotations [262] and attributes in a form of Java annotations [246]. Marking [85] presents navigation cues like bookmarks.

4.6 Conclusion

In this systematic mapping study, 2,079 articles were assessed (including duplicates). 25 unique articles were considered relevant to the field of source code labeling. A taxonomy containing four dimensions was created by the analysis of the articles.

The contribution of this work is threefold. Researchers can both get a quick overview of many source code labeling approaches, and find interesting future research directions by analyzing the gaps. IDE developers can use it as an inspiration about which features to implement in their product. Practitioners struggling to analyze existing large codebases may find information about available tools and approaches here.

Interesting observations can be made by looking at patterns in Table 4.2. For example, all approaches using an in-code persistence use purely human source of information.

It is important to note that some of the approaches described here are already implemented in industrial IDEs. Furthermore, there exist some features of commercial IDEs not described here. It would be interesting to compare a set code labeling features of common IDEs and academic tools.

Chapter 5

Manual and Runtime-Based Concern Annotation

When programmers develop a program, they continuously create a mental model, which is a representation of the program in their mind. They try to express the mental model in the programming language constructions. However, not all parts of the mental model are transferred into the source code and some details are lost. Later, during the program maintenance phase, developers make a tremendous effort to recover such types of information from the source code [130].

An important part of the mental model is a mapping from concerns to source code parts. According to one of its most general definitions, a *concern* is "anything a stakeholder may want to consider as a conceptual unit" [207]. In our work, we characterize a concern as a developer's intent of a particular piece of code: What should this code accomplish? How would I tersely characterize it? Is there something special about it? Some concerns may be obvious by looking at the code itself (chiefly from names of the identifiers), but many concerns are hidden.

Since one piece of code can belong to multiple concerns [193], we cannot simply name all classes and methods according to their concerns since one identifier can have only one name. On the other hand, it is possible for a class or method in Java to have more than one annotation (attribute in C#).

In his dissertation, Milan Nosáľ [181] advocates for the use of source code annotations to implement the concern-to-code mapping. Such annotations are called *concern annotations* – the main topic of this chapter[1]. For each distinct concern, he recommends to create one Java *annotation type*. For example, we can create an annotation type `@Persistence` which tells us the code marked with it fulfills the task of persistent storage of objects. Subsequently, in our imaginary program, we could mark the methods like `FileDialog.open()`, `Note.load()`, `Note.save()` and a class `FileFormat` with it. We will call them *annotation occurrences*. At the same time, the first of the mentioned methods could be also annotated with the `@GUI` concern, as it presents a GUI (graphical user interface) dialog to a user.

[1] This chapter contains material from our articles: [246], [247], [248], [252]. Milan Nosáľ contributed to the design and execution of the experiments in sections 5.1 and 5.2, mainly by suggesting potential tasks and questions. In section 5.4, we uses the results by Milan Nosáľ to compute the overlap between non-authors of the source code.

Compared to traditional source code comments, concern annotations are more formal. We can use standard IDE features like navigating to the declaration, usages searching, refactoring and other on them. Concern annotations can have parameters to further specify their properties. They may be also commented by natural language comments if needed.

M. Nosáľ [181] distinguishes three types of concern annotations:

- *Domain annotations* document concepts and features of the application (problem) domain. For example, all source code elements representing the feature of filtering a list of items can be marked with an annotation `@Filtering`. Similarly, all code related to bibliographic citations could be annotated by `@Citing`. Domain annotations roughly correspond to *features*, i.e. realizations of requirements which are visible in the program and can be triggered by its end user [64].
- *Design annotations* document design and implementation decisions like design patterns, e.g., `@Observer`.
- *Maintenance annotations* are intended to replace traditional TODO-like comments. An example is the `Unused` annotation for parts of code not used in a project at all.

The first part of this chapter investigates the viability of annotations as a medium to share parts of developers' mental models by empirically testing the following hypothesis:

H5.1 *The presence of concern annotations in the code improves program comprehension and maintenance correctness, time and confidence.*

Traditionally, concern annotations are inserted into source code manually, which is expensive on human resources. In section 5.3, we will present a prototype tool which annotates the program semi-automatically, using runtime information. This runtime-based tool will be focused on domain annotations (features).

Finally, in section 5.4, we try to investigate how the semi-automatic annotation approach compares to manual annotation. However, since our tool uses an existing (and rather old) algorithm to compute the correspondence between code parts and concerns, it is not on par with state-of-art feature location approaches [59]. Instead of regarding the manually annotated code as a gold standard, we will expand the study of M. Nosáľ [181] who compared the overlap between annotations of multiple people annotating the same code. He asked 7 participants to independently annotate the source code of EasyNotes[2] – a small-scale desktop application for bibliographic note-taking. None of the participants was an author of the code. He found that only 28% of concern annotation occurrences in the code were shared (overlapped) by at least two participants. Therefore, we ask the following research question:

RQ5.1 *How does the overlap between semi-automatic and code author's annotations compare to the overlap among annotations of multiple non-authors?*

5.1 The Effect of Annotations on Program Maintenance

First, we performed a controlled experiment to study the quantitative effect of the annotated source code on program comprehension and maintenance.

The guidelines to perform software engineering experiments on human subjects [119] were used. To present our findings, the experiment reporting guidelines [103] were followed. We customized them to the specific needs of this experiment.

[2] http://github.com/MilanNosal/easy-notes

5.1.1 Hypotheses

We hypothesize that the presence of concern annotations in the source code improves program comprehension and maintenance correctness, time and confidence. Thus we formulate the null and alternative hypotheses:

H5.1$_{\text{corr, null}}$: The correctness of the results of program comprehension and maintenance tasks on an annotated project = the correctness on the same project without concern annotations.

H5.1$_{\text{corr, alt}}$: The correctness of the results of program comprehension and maintenance tasks on an annotated project > the correctness on the same project without concern annotations.

H5.1$_{\text{time, null}}$: Time to complete program comprehension and maintenance tasks on an annotated project = time to complete them on the same project without concern annotations.

H5.1$_{\text{time, alt}}$: Time to complete program comprehension and maintenance tasks on an annotated project < time to complete them the same project without concern annotations.

H5.1$_{\text{conf, null}}$: Participants' confidence of their answers to program comprehension questions on an annotated project = their confidence on the same project without concern annotations.

H5.1$_{\text{conf, alt}}$: Participants' confidence of their answers to program comprehension questions on an annotated project > their confidence on the same project without concern annotations.

We will statistically test the hypotheses with a confidence interval of 95% ($\alpha = 5\%$).

5.1.2 Variables

Now we will define independent variables, i.e., the factors we control, and dependent variables – the outcomes we measure.

Independent Variables

There is only one independent variable – the presence of concern annotations in the project. It has a nominal scale which means there is a finite number of possible values without any meaningful ordering [282]. The levels (possible values) of this variable are: yes ("annotated") and no ("unannotated").

Dependent Variables

The correctness was measured as a number of correct answers (or correctly performed tasks) divided by the total number of tasks (5). The tasks are not weighted, each of them is worth one point. The assessment is subjective – by a researcher.

The second dependent variable is the time to finish the tasks. Its scale is of a ratio type since the ratio between two values is meaningful [282]. Although it was technically measured with millisecond precision, we will use the unit "minutes" rounded to two decimal places in subsequent analysis. We are interested mainly in the total time, i.e., a sum of times for all tasks.

Instead of measuring just time alone, it is possible to define efficiency as a number of correct tasks and questions divided by time. On one hand, efficiency depends on correctness, which already is a dependent variable. On the other hand, efficiency can deal with participants who fill the answers randomly to finish quickly [123]. We decided to use efficiency only as an

auxiliary metric to make sure that time differences are still significant even if the correctness is considered.

For each comprehension question, we also asked a subject how confident (s)he was on a 3-point Likert scale: from Not at all (1) to Absolutely (3). Since we asked a subject about the confidence equally for each task, we consider it meaningful to calculate the mean confidence, which is the third dependent variable.

5.1.3 Experiment Design

Now we will describe the design of the experiment.

Materials

The study was performed on the EasyNotes project[3]. We prepared two different versions:

- with concern annotations created by other people
- and without annotations.

The first version was obtained by asking 7 participants to independently annotate the source code of EasyNotes – a small-scale desktop application for bibliographic note-taking.

As the project was only scarcely commented, we deleted all traditional source code comments from both versions to remove a potential confounding factor. Only comments for the annotation types themselves were left intact, as we regard them as their integral part.

During this experiment, we used the NetBeans IDE.

Participants

We used 18 first-year master's degree Computer Science students as participants. Carver et al. [40] recommend to integrate software engineering experiments performed on students with teaching goals. We decided to execute the experiment as a part of the Modeling and Generation of Software Architectures course, which contained Java annotations in its curricula.

The course was attended by students focused not only on software engineering, but also on other computer science subfields. Inclusion criteria were set to select mainly students with a prospective future career as professional programmers. Additionally, as EasyNotes is a Java project, a sufficient Java language knowledge was required.

The experiment was performed during three lessons in the same week – the first time with four students, then 9 and finally with the remaining 5 students. Each session lasted approximately 1.5 hours. The study was executed in a separate room, so the participants were not disturbed.

Group Assignment

When assigning the subjects to groups, we applied a completely randomized design [282]. This means that each group received only one treatment – either an annotated or an unannotated program – and the assignment was random. Each participant drew a piece of paper with a number on it. Subjects with an odd number were assigned to the "annotated" group, participants with an even number to the "unannotated" one. Our design was thus balanced, with n=9 per group.

[3] http://github.com/MilanNosal/easy-notes

Instruments

To both guide the subjects and collect the data, we designed an interactive web form[4]. All fields in the form were mandatory, so the participants could not skip any task.

We asked the subjects to install a NetBeans plugin, SSCE[5]. Although it is not its primary feature, it provides an option to record programming sessions – time elapsed, a list of open files and NetBeans windows gaining the focus. Just before the start of each task, a subject clicked the button to start a new session, named after the task (e.g., "Filter"). Immediately after the task was finished, (s)he clicked the button again to end the session. The collected data were written to an XML file which the participants uploaded to the web form at the end of the experiment.

5.1.4 Procedure

Now we will describe the experimental procedure.

Training

At the beginning of the experiment, the users were given 5 minutes to familiarize themselves with EasyNotes from an end-user perspective, without looking at the source code. This provided them an overview of the application domain and helped them to better understand their subsequent tasks. Then, the session monitoring plugin was briefly introduced.

A short presentation about concern annotations usage in the NetBeans IDE followed. A researcher presented how to show a list of all available concerns, how to use the Find Usages feature on an annotation type and how to navigate from the annotation occurrence to an annotation type.

Just before each of the two maintenance tasks, the researcher presented the participants a running application with the required feature already implemented. This had two positive effects:

- It significantly lowered the task ambiguity. While a natural-language description was available in the web form during the tasks, seeing the finished application gave the participants greater confidence, so almost nobody asked unnecessary questions.
- We consider this a replacement for unit tests. As GUI code is notoriously difficult to test [160], we decided not implement the test code. At the same time, the researcher's discretion about the task correctness was not indispensable during the experiment. Students just knew they finished the task when their application did the same thing as we had showed them.

In addition, the participants later uploaded their modified version of the source code to the web form. This way, potential disputes could be resolved without time stress.

Tasks

The experiment comprised of:

- one additive maintenance task (we will name it *Filter*),
- three program comprehension questions (*Test*),
- one corrective maintenance task (*Cite*),

[4] http://www.jotformeu.com/sulir/sharing-annotations

[5] http://github.com/MilanNosal/sieve-source-code-editor

in that order.

The tasks were formulated as follows:

Filter In a running EasyNotes application, load the sample notes and look at the searching feature (the down-right part of the window). Try how searching in the note text works (the option "Search in": "text"). Your task will be to add a filter to the EasyNotes application, which will search in the notes title (the option "title").

Cite In the EasyNotes application, there is a field "Cite as:" (the down-right window part). Currently, it displays information in the form: *somePubID* where somePubID is the value of the "PubID:" field. Your task is to modify the program so the "Cite as:" field will display information in the form: *\cite{somePubID}*.

Both tasks were simple, although the *Filter* task was slightly more complex than the latter. It required the creation of a new class with approximately 15 lines of code, whereas the *Cite* task could be accomplished by modifying just one source code line.

The questions asked in the *Test* about program comprehension were:

Q1 What does the `runDirMenuItemActionPerformed` in the class `easynotes.swingui.EasyNotesFrame` do?

Q2 How is the class `easynotes.model.abstractModel.UTFStringComparator` used in the EasyNotes project?

Q3 What method/s (and in which class) do perform note deleting?

Demographic Data Collection

At the end of the experiment, the form contained three demographic questions about the subjects' abilities:

- a general programming experience,
- their experience with Java, annotations and NetBeans
- and their English level.

Each question had possible answers on a 5-point Likert scale: from Beginner to Expert. We did not perform any specialized tests to measure programming experience, as a subjective opinion is good enough [68].

Debriefing

We also included a question asking to what extent did the subjects use annotations when comprehending the code. Possible answers ranged from Never to Always on a 5-point Likert scale. Finally, the form also contained a free-form question where the participants could describe how the annotations helped them in their own words.

5.1.5 Results

The measured values and their summary is presented in Table 5.1. To analyze the results, we used the R scripting language, auxiliary Ruby scripts and spreadsheets.

Each specific hypothesis considers one independent variable on a nominal scale with two levels (annotated, unannotated) and one dependent variable (either correctness, time or confidence). For each dependent variable, we displayed the values on a histogram and a normal Q-Q plot. None of the variables looked normally distributed, so we used the Mann-Whitney U test as a statistical test for our hypotheses.

Table 5.1: Experiment results for individual subjects

(a) the "annotated" group

ID	Correctness [true/false]						Time [min]				Efficiency [task/min]	Confidence [1-3]				Files	Annot. useful [1-5]
	Filter	Cite	Q1	Q2	Q3	Total	Filter	Cite	Test	Total		Q1	Q2	Q3	Mean		
1	1	1	1	0	1	80%	12.03	3.00	13.96	28.99	0.14	1	3	5	3.00	19	1
3	0	1	0	0	0	20%	5.23	2.81	11.13	19.17	0.05	5	3	5	4.33	10	3
5	1	1	1	1	1	100%	17.43	3.93	6.71	28.07	0.18	5	5	5	5.00	18	4
7	0	1	1	1	1	80%	7.79	1.43	11.85	21.07	0.19	5	5	5	5.00	10	4
9	1	1	1	0	1	80%	6.72	5.86	3.87	16.45	0.24	5	5	3	4.33	20	2
11	1	1	1	0	1	80%	8.41	4.58	4.32	17.31	0.23	5	5	5	5.00	13	4
13	1	1	0	0	1	60%	20.97	3.48	8.80	33.25	0.09	5	3	5	4.33	11	3
15	1	1	1	0	1	80%	4.64	1.91	5.22	11.77	0.34	3	3	5	3.67	11	2
17	1	1	1	0	1	80%	25.08	6.38	6.99	38.45	0.10	3	3	5	3.67	16	4
Median	**1**	**1**	**1**	**0**	**1**	**80%**	**8.41**	**3.48**	**6.99**	**21.07**	**0.18**	**5**	**3**	**5**	**4.33**	**13**	**3**
Std.dev.	**-**	**-**	**-**	**-**	**-**	**22%**	**7.41**	**1.67**	**3.57**	**8.80**	**0.09**	**-**	**-**	**-**	**0.70**	**4**	**-**

(b) the "unannotated" group

ID	Correctness [true/false]						Time [min]				Efficiency [task/min]	Confidence [1-3]				Files	Annot. useful [1-5]
	Filter	Cite	Q1	Q2	Q3	Total	Filter	Cite	Test	Total		Q1	Q2	Q3	Mean		
2	1	1	1	0	1	80%	2.84	4.72	10.66	18.22	0.22	5	3	5	4.33	9	NA
4	0	0	1	0	1	40%	8.76	23.45	8.10	40.31	0.05	5	3	5	4.33	19	NA
6	1	1	1	0	0	60%	18.24	5.23	5.62	29.09	0.10	5	3	1	3.00	17	NA
8	1	1	1	0	1	80%	6.47	5.59	11.23	23.29	0.17	5	3	5	4.33	8	NA
10	1	0	1	0	1	60%	4.82	9.64	17.50	31.96	0.09	5	3	5	4.33	8	NA
12	1	0	1	0	1	60%	11.11	2.09	11.30	24.50	0.12	3	3	3	3.00	13	NA
14	0	1	0	0	1	40%	30.73	7.19	5.50	43.42	0.05	3	3	5	3.67	17	NA
16	1	1	1	0	1	80%	12.56	18.39	16.07	47.02	0.09	5	3	5	4.33	14	NA
18	1	1	1	0	1	80%	25.54	9.59	12.94	48.07	0.08	5	3	5	4.33	16	NA
Median	**1**	**1**	**1**	**0**	**1**	**60%**	**11.11**	**7.19**	**11.23**	**31.96**	**0.09**	**5**	**3**	**5**	**4.33**	**14**	**NA**
Std.dev.	**-**	**-**	**-**	**-**	**-**	**17%**	**9.56**	**6.98**	**4.18**	**11.06**	**0.06**	**-**	**-**	**-**	**0.59**	**4**	**-**

Correctness

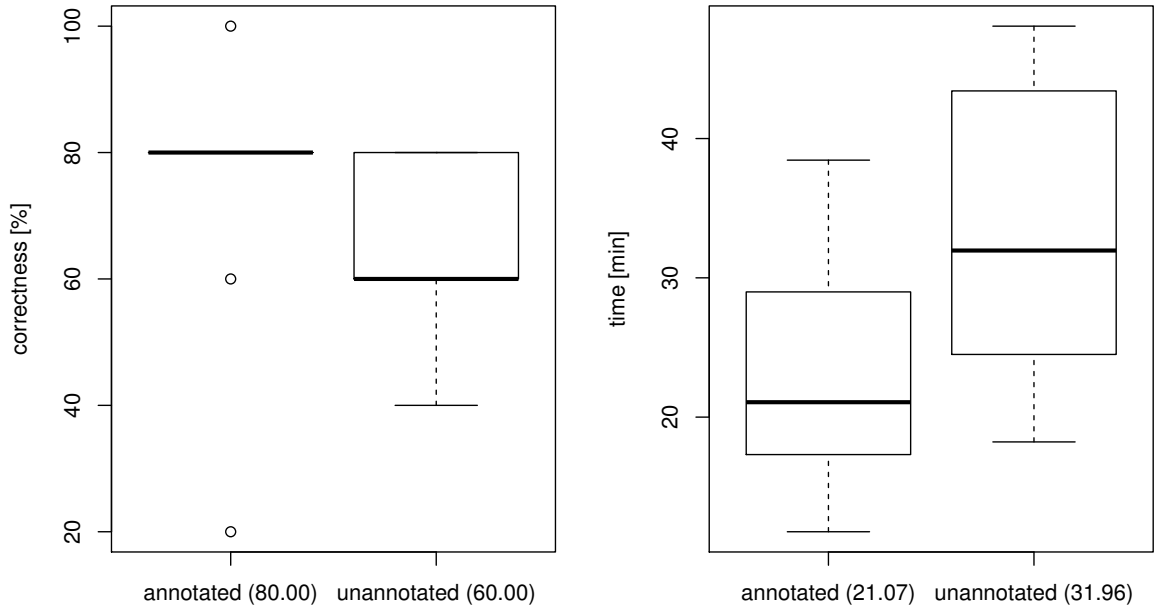
The median of correctness for both the "annotated" group was 80%, while the "unannotated" one achieved 60%. See Fig. 5.1a for a plot.

The computed p-value (roughly speaking, the probability that we obtained the data by chance) is 0.0938, which is more than 0.05 (our significance level). This means we accept $\mathbf{H5.1_{corr,\,null}}$. The effect of the presence of annotations on program comprehension and maintenance correctness was not confirmed.

As we can see in Table 5.1 (column Correctness), the most difficult question was Q2. Only two participants answered it correctly – both from the "annotated" group. The class of interest was not used in EasyNotes at all. This fact was noticeable by looking at the `Unused` annotation.

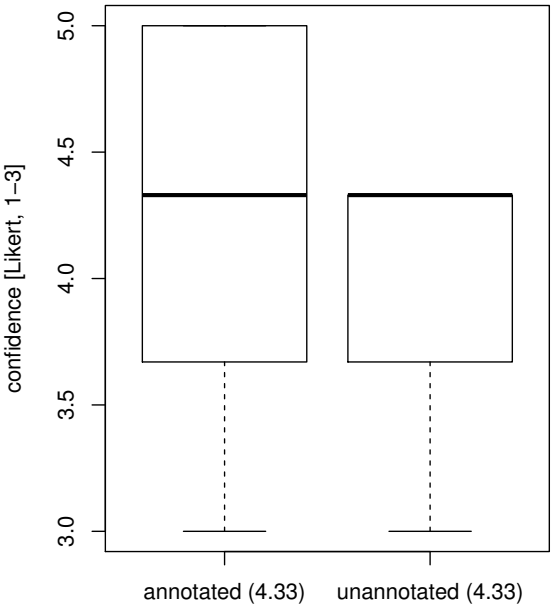
Time

The differences in the total time for all tasks between two groups are graphically depicted in the box plot in Fig. 5.1b. The median time changed from 31.96 minutes for the "unannotated" group to 21.07 minutes for the "annotated" one, which is a decrease by 34.07%.



(a) correctness

(b) time



(c) confidence

Figure 5.1: Results of the controlled experiment

The p-value of time is 0.0252, that is less than 0.05. The difference is statistically significant, therefore we reject $\mathbf{H5.1}_{\mathbf{time,\ null}}$ and accept $\mathbf{H5.1}_{\mathbf{time,\ alt}}$. The presence of concern annotations improves the program comprehension and maintenance time.

It is possible to see from Table 5.1 (column Time) that the median time for each individual task was better for the "annotated" group. The most prominent difference was for the task *Cite*. This can be due to the fact that the project contained the concern annotation `Citing` which helped the participants find the relevant code quickly.

The median of efficiency, which we defined as the number of correctly performed tasks (and answers) divided by total time, raised by 89.76% (p=0.0312). This means the time improvement is significant even if we take correctness into account.

Confidence

The median of mean confidence is the same for both groups (2.67), as obvious from Table 5.1, column Confidence / Mean and Fig.5.1c. The p-value is 0.1710 (> 0.05) and therefore we accept $\mathbf{H5.1_{conf,\ null}}$. The effect of concern annotations on confidence was not demonstrated.

Looking at individual questions, Q2 was clearly perceived the most difficult. This corresponds with our previous finding about the correctness. An interesting fact is that no participant in the "unannotated" group was confident enough to select the level 3 (Absolutely), while in the "annotated" group, there were 4 such subjects and two of them really answered correctly.

Other Findings

Although not included in our hypotheses, we also measured how many unique Java source files did the subjects open in the IDE during the whole experiment. We excluded the annotation type files in these numbers. The results are in Table 5.1, column Files. There was only a marginal and statistically insignificant improvement in medians (from 14 to 13).

As seen from Table 5.1, column "Annotations useful?", concern annotations were perceived relatively helpful by the participants (median 3 from 5-point Likert scale).

Answers to a free-form question asking how specifically were the annotations useful included:

- faster orientation in a new project,
- faster searching (mainly through Find Usages on annotations),
- less scrolling,
- "they helped me to understand some methods",
- "annotations could be perfect, but I would have to get used to them".

5.1.6 Threats to Validity

To analyze threats to validity of this experiment, we used [176] as guidelines.

Construct Validity

Similar to Kosar et al. [123], we compensated the students with points for the participation in the experiment, which increased their enthusiasm. Unlike them, we did not reward the participants with bonus points for good results because our experiment spanned several days with the same tasks and this would motivate the students to discuss the task details with classmates, which could negatively affect the construct validity (interaction between subjects). Furthermore, the participants were explicitly told not to share experiment details with anyone. Therefore, we do not consider executing the experiment in three separate sessions an important validity threat.

To measure confidence, we used only a 3-point Likert scale. This decision was not optimal. Because subjects rarely select the value of 1 (which can be interpreted as guessing an answer), there were only two values left. This could be one of the reasons we did not found a statistically significant difference in confidence.

Internal Validity

There could be a selection bias in the experiment because we selected the participants using subjective criteria. As we already mentioned, we concentrated on subjects with a potential future career as developers.

We divided the subjects into groups randomly. Another possibility was a quasi-experiment (nonrandom assignment), i.e., to divide the subjects evenly according to the most important co-factor affecting the results, like their programming experience. However, random assignments tend to have larger effect sizes than quasi-experiments [107].

We did not perform a full pilot testing with third-party participants, only tested the comprehension questions on one of the researchers. We rejected 3 out of the 6 prepared questions because we considered them too difficult and ambiguous. Despite this, all tasks were either completed by almost all participants (Filter, Q1, Q3, Cite) or by almost none (Q2) during the experiment. This negatively affected the results for the "correctness" variable.

During the actual experiment, we used a native language (Slovak) version of the web form to eliminate the effect of English knowledge on the results. While the source code was in English, this did not present a validity threat since all participants had at least a medium level (3 on a 5-point Likert scale) of English knowledge.

External Validity

We invited only students to our experiment, no professional developers. We can take this fact positively – concern annotation consumers are expected to be mainly junior developers, whereas potential annotation creators are mostly senior developers. Furthermore, some students may already work in companies during their study.

EasyNotes is a small-scale Java project – around 3 kLOC (thousands of lines of code), including annotations. The effect of concern annotations on larger programs should be investigated.

Reliability

The concern annotations training (tutorial) was presented manually by a researcher. However, there were two independent researchers which took turns.

The experiment is replicable, as we published the data collection form[6] which contains both the guidelines and links to the materials (two versions of the EasyNotes project).

Conclusion Validity

A small number of subjects (n=9 per group) is the most important conclusion validity threat. If we used a paired design (to assign both treatments to every subject), we could easily reach n=18. However, the participants would quickly become familiar with EasyNotes and the second set of tasks would be affected by their knowledge.

5.1.7 Conclusion

We successfully confirmed the hypothesis that concern annotations have a positive effect on program comprehension and maintenance time. The group which had concern annotations available in their project reached a time more than 34% shorter than the group without them ($p < 0.05$).

[6] http://www.jotformeu.com/sulir/sharing-annotations

On the other hand, we did not discover a significant change in correctness and confidence. However, the correctness results were positively included towards the "annotated" group.

5.2 Replication of the Controlled Experiment

While replication is a crucial scientific method, less than 18% of controlled experiments in software engineering are replications [228]. We decided to repeat the experiment with slightly modified conditions, i.e., to perform a differentiated replication [282].

The hypotheses, independent and dependent variables, comprehension questions and maintenance tasks were exactly the same as in the original experiment. Demographic characteristics of the participants, training and execution details were different.

5.2.1 Method

This time, we concentrated on developers with some industrial experience. We contacted the developers we known directly by e-mail. Additionaly, we distributed an announcement using a social network and an Internet forum. The participation was voluntary and no compensation was provided.

20 subjects decided to participate in the experiment. However, one participant completed only the first task, so we excluded him completely. During the analysis of time results for 19 subjects, we found a strong outlier. After contacting the person, we found out that he was interrupted during the experiment. We decided to exclude this subject, too. Therefore, we take only the 18 subjects who participated in the whole experiment into account.

The average (mean) age of the subjects was 26.8 years. The mean full-time programming experience of the participants was 2.7 years. However, some subjects had only part-time work experience. Each participant worked in one of at least 6 different companies, thus assuring high diversity.

Instead of providing a presentation in person, the training consisted of a written tutorial contained in the form.

5.2.2 Results

In Table 5.2, we can see detailed results of the experiment replication. Now we will describe the results of correctness, time and confidence.

Correctness

In Figure 5.2a, there is a box plot of the results of tasks correctness. The "annotated" group was better: its median was 80%, compared to only 60% for the "unannotated" group. Therefore, the correctness is 33% higher for the "annotated" group. The computed p-value was 0.0198 – the result is statistically significant. We accept $\mathbf{H5.1_{corr,\,alt}}$.

Time

The time for "annotated" group was now higher: the median was 31.50 minutes, compared to 24.52 minutes for the control group. However, the result is statistically insignificant (p=0.1487). The results for time are depicted in Figure 5.2b.

Table 5.2: Detailed results of experiment replication

(a) the "annotated" group

ID	Correctness [true/false]						Time [min]				Efficiency [task/min]	Confidence [1-3]				Files	Annot. useful [1-5]
	Filter	Cite	Q1	Q2	Q3	Total	Filter	Cite	Test	Total		Q1	Q2	Q3	Mean		
19	1	1	1	0	1	80%	11.94	5.48	35.95	53.37	0.07	5	4	4	4.33	18	2
21	1	1	1	1	1	100%	19.87	14.49	9.30	43.66	0.11	4	5	5	4.67	14	3
23	1	1	1	0	1	80%	5.97	23.20	6.69	35.86	0.11	5	4	5	4.67	17	4
27	1	1	1	1	1	100%	10.67	5.01	28.90	44.58	0.11	4	3	5	4.00	14	3
29	1	1	1	0	1	80%	19.80	3.22	7.12	30.14	0.13	4	4	4	4.00	13	3
31	1	1	0	0	1	60%	12.17	3.80	6.08	22.05	0.14	4	5	5	4.67	11	3
51	0	1	1	0	1	60%	9.12	3.59	6.29	19.00	0.16	4	5	5	4.67	12	4
53	1	1	1	1	1	100%	9.07	13.43	9.00	31.50	0.16	5	5	5	5.00	22	3
55	1	1	1	0	1	80%	2.83	3.82	5.75	12.40	0.32	4	4	4	4.00	9	1
Median	**1**	**1**	**1**	**0**	**1**	**60%**	**10.67**	**5.01**	**7.12**	**31.50**	**0.13**	**4**	**4**	**5**	**4.67**	**14**	**3**
Std.dev.	**-**	**-**	**-**	**-**	**-**	**16%**	**5.67**	**7.01**	**11.34**	**13.32**	**0.07**	**-**	**-**	**-**	**0.37**	**4**	**-**

(b) the "unannotated" group

ID	Correctness [true/false]						Time [min]				Efficiency [task/min]	Confidence [1-3]				Files	Annot. useful [1-5]
	Filter	Cite	Q1	Q2	Q3	Total	Filter	Cite	Test	Total		Q1	Q2	Q3	Mean		
20	1	1	0	0	0	40%	7.88	15.09	49.71	72.68	0.03	5	3	4	4.00	19	1
22	1	1	1	0	1	80%	4.71	3.19	11.93	19.83	0.20	5	2	5	4.00	11	1
24	1	1	1	1	1	100%	6.78	6.12	16.62	29.52	0.17	4	3	5	4.00	10	1
26	1	0	0	0	1	40%	4.68	5.61	14.23	24.52	0.08	4	4	5	4.33	9	3
30	1	0	1	0	0	40%	2.53	9.78	5.00	17.31	0.12	5	4	4	4.33	NA	1
32	1	1	1	0	1	80%	9.27	2.29	17.87	29.43	0.14	4	5	5	4.67	9	1
50	1	1	0	0	0	40%	5.37	1.71	11.58	18.66	0.11	5	4	3	4.00	11	1
52	1	1	1	0	0	60%	7.02	8.22	17.39	32.63	0.09	5	4	4	4.33	11	1
54	1	1	0	0	1	60%	4.21	8.79	4.85	17.85	0.17	4	5	5	4.67	16	1
Median	**1**	**1**	**1**	**0**	**1**	**40%**	**5.37**	**6.12**	**14.23**	**24.52**	**0.12**	**5**	**4**	**5**	**4.33**	**NA**	**1**
Std.dev.	**-**	**-**	**-**	**-**	**-**	**22%**	**2.08**	**4.25**	**13.34**	**17.30**	**0.05**	**-**	**-**	**-**	**0.28**	**NA**	**-**

Confidence

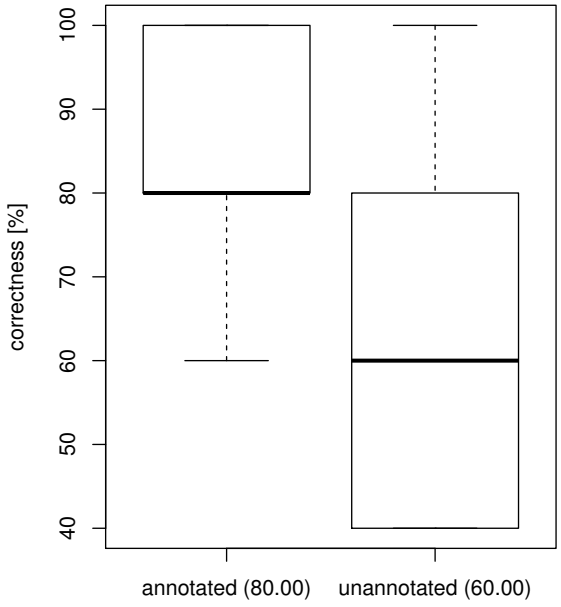
In Figure 5.2c, there are the results for the confidence according to the answers in the form. The median of mean confidence for three comprehension questions was higher for the "annotated" group: 4.67, compared to 4.33. Nevertheless, the p-value is 0.1417, so the result is insignificant.

5.2.3 Threats to Validity

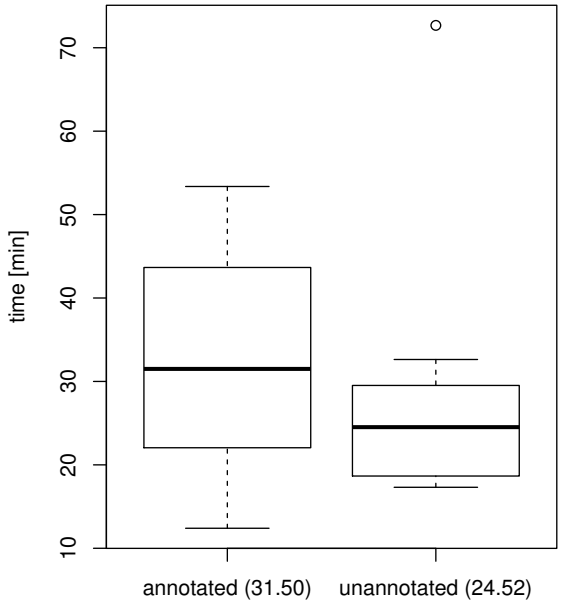
The threats to validity are similar to the original experiment. The main difference is the increased external validity thanks to the inclusion of industrial developers from multiple companies.

5.2.4 Conclusion

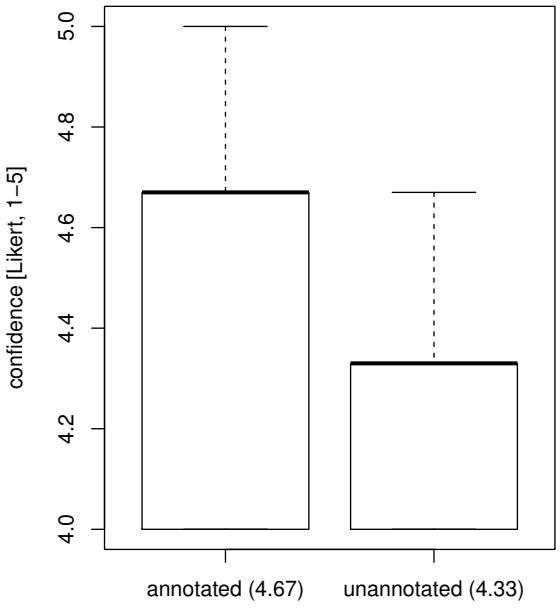
We performed a replication of the experiment studying the effect of concern annotations presence in the source code. In this replication, we measured a statistically significant 33% increase of correctness for comprehension tasks and questions.



(a) correctness



(b) time



(c) confidence

Figure 5.2: Results of the experiment replication

5.3 Automation of Concern Annotation

Although the presence of concern annotations improves comprehension time and correctness, manually labeling the source code is a laborious process. For this reason, we will introduce a simple automation method in this section. To compute the correspondence of source code parts to concerns, we will use an existing technique of differential code coverage [226]. We are interested only in domain annotations, representing program features visible to the end user.

An overview of the whole process for one specific concern is in Fig. 5.3. The process can be also run with a list of multiple concern names supplied. In this case, the AutoAnnot part is

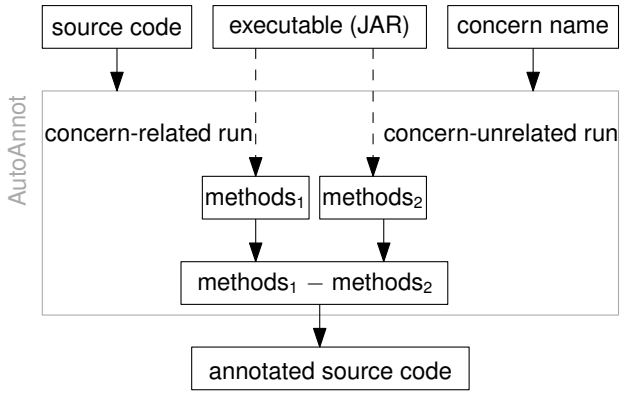


Figure 5.3: An annotation process for a specific concern. Solid arrows represent automated processes, dashed partially manual ones. © 2015 IEEE

repeated for each concern.

5.3.1 Creating Annotation Types

First, the programmer must create an annotation type for each concern they recognize in a program. This process is not automated and remains the same as before.

5.3.2 Differential Code Coverage

Differential code coverage [226] is based on an idea of running an application two times – with and then without a particular feature utilized – and subtracting the sets of method executed. In this section, we will apply this idea to annotate the resulting set of methods.

Instrumentation

The program is augmented so that execution traces are collected when it is run. We used BTrace[7]. It was run as a Java agent – a program which can instrument the bytecode at runtime. BTrace uses a file with Java-like syntax, specifying which actions should be performed e.g. on each method call.

We configured BTrace to log a fully-qualified name of each called method to a file. A fully-qualified name unambiguously identifies the method – it includes a package and class name, a method name and parameters types. Each method is included only once in a log file, even if it is called multiple times. We also included constructors in the logs.

Execution

For each concern, our tool (AutoAnnot) executes the program twice. During these runs, a programmer performs user interface actions in the following way:

- first with the particular concern utilized – a *concern-related run*
- and the second time in a manner when the concern is not utilized – a *concern-unrelated run*.

For example, if the concern of interest is “file saving”, the user first creates a simple drawing in the application and saves it to a file; in the second run (s)he creates the drawing again but discards the changes.

[7] http://kenai.com/projects/btrace

During these executions, AutoAnnot records the execution traces. Now we have two log files for a particular concern – one containing the methods from the first run, one from the second.

Subtraction

Let us call the set of methods executed for concern c during the first run $M_1(c)$, during the second one $M_2(c)$. Then the set of methods most specific to the concern c is:

$$M(c) = M_1(c) - M_2(c)$$

This way, only methods specific to a particular concern are included in the result, leaving out utility methods and methods common to multiple concerns.

5.3.3 Writing Annotations

Once a set of methods corresponding to a particular concern is computed, each of these methods is annotated. The original source code is parsed, modified and written back using the Roaster library[8].

There was a problem with Java's anonymous classes – mapping from the class during runtime to a class in the source code was ambiguous. For this reason, we chose not to annotate methods inside anonymous classes.

This could be a example of an excerpt from the resulting annotated source code of a diagramming application:

```
@Drawing
@Erasing
public void changeTool(Tool tool) {
  ...
}

@Erasing
public void clearRegion(Shape region) {
  ...
}
```

5.4 Comparing Manual and Automated Annotation

A traditional form of the evaluation of approaches annotating code parts with features or concerns (feature location or concern location) is to compare the automatically annotated source code with the code annotated manually by the author of the code (or a maintainer of the project). The manually annotated project is considered "the gold standard" and the correctness of this code-concern mapping is usually not questioned. However, multiple human annotators also have mutual discrepancies in their views of concerns and their mapping to code parts. We will compare:

- an overlap (agreement) between the annotation occurrences of multiple human annotators of the same project (who are not its authors)
- and an overlap between the annotation occurrences marked by the author of the project and the semi-automated annotator.

[8] http://github.com/forge/roaster

5.4.1 Method

We performed this study on EasyNotes[9] – a 2500 LOC (lines of code) desktop Java application for bibliographic note-taking.

Annotations of Non-authors

We used manually-annotated source code from the previous study [246], where the procedure was as follows: Seven participants, who were not authors of the code, were asked to cretate an annotation type for any concern recognized in the application. Then they marked elements with concern annotations they thought are relevant.

For this study, we selected 8 domain concerns (features) which were recognized by at least two participants. We we considering only concerns which were clearly distinguishable from other ones (e.g., we excluded "filtering" because there existed a very related concern "filter management").

Annotations of the Author

The author of the EasyNotes application was asked to mark methods in the source code with the 8 given features according to his own discretion, thus producing the author-annotated source code.

Automatic Annotation

Again, we used the same 8 domain-related concerns recognized by participants of the previous study. For every concern, a researcher executed the program two times as described in section 5.3.2. The decision what a particular concern comprises was subjective – left on the researcher. Annotations of methods resulting from the differential code coverage were written into the source code.

Comparison

First, we calculated the overlap between non-authors. The *overlap* (agreement) among n persons is defined as the number of annotation occurrences shared by n people divided by the number of all annotation occurrences. For example, if six persons annotate only the method `saveLink` with the concern "Links", and one person annotates only the method `loadLink` with this concern, the overlap among 6 persons is 50% for this particular concern. Analogously, we can also say there was a 50% overlap among "at least 6 persons", which is a sum of the overlap by exactly 6 and 7 persons.

Next, the overlap between the author-annotated code and the semi-automated (AutoAnnot) annotation was computed. This is a special case of the above metric for only two annotators. Formally, we can express it as:

$$\frac{|M_{author} \cap M_{AutoAnnot}|}{|M_{author} \cup M_{AutoAnnot}|}$$

where M_{author} is a set of methods annotated with a particular concern by the author, $M_{AutoAnnot}$ a set of semi-automatically annotated ones.

[9] http://github.com/MilanNosal/easy-notes

Table 5.3: Features and their overlaps

Concern	Overlap between			
	$\geq$ 2 non-authors	$\geq$ 3 non-authors	$\geq$ 4 non-authors	author & AutoAnnot
Citing	0.00%	0.00%	0.00%	16.67%
Links	38.46%	7.69%	0.00%	22.37%
NoteAdding	38.10%	23.81%	4.76%	25.00%
NoteDeleting	18.75%	12.50%	6.25%	66.67%
NoteEditing	60.00%	12.00%	0.00%	11.76%
NotesLoading	35.00%	25.00%	10.00%	38.46%
NotesSaving	30.77%	15.38%	3.85%	41.67%
Tagging	64.29%	28.57%	0.00%	8.33%
Total	**38.19**%	**16.67**%	**3.47**%	**24.73**%

5.4.2 Results

Table 5.3 contains the results. First, there are computed overlaps between at least 2, 3 and 4 non-authors. The last column represents the overlap between the author an the semi-automated annotator. The "Total" row represents a weighted average, i.e., the overlap calculated for all concerns.

We can see that the overlap (agreement) between the code author and AutoAnnot is higher than the agreement among at least 3 of the 7 persons, but lower than the agreement among at least 2 persons.

5.4.3 Threats to Validity

Now we will present threats to validity according to guidelines [215].

Construct Validity

Instead of a custom-defined overlap metric, we could use standard metrics like Cohen's kappa or Fleiss' kappa. However, since we used the results of a previous study, we decided to use the same (analogous) metric for a simpler direct comparison.

Internal Validity

Participants of our previous study (manual annotation by non-authors) were asked to annotate methods, types (classes, enums, etc.) and member variables. During the code author's and semi-automatic annotation, only methods were allowed to be annotated. It is not certain what annotating a class in respect to its methods means. Do absolutely all methods of the class pertain to the given concern? Or does it hold for a majority of methods? Probably the most elegant solution is to refrain from labeling classes and annotate only methods instead. However, we did not compare the code where all types of elements could be annotated directly with the code where only methods were annotable. Therefore, we do not consider this threat significant.

External Validity

We performed the comparison only on one particular project. The source code is small: 2.5 kLOC in 33 classes, excluding annotations. The study should be repeated on a larger project, preferably from another domain and using other technologies, to confirm the generality of results.

Reliability

The execution step of differential code coverage for all concerns was performed entirely by one researcher. The results are therefore affected by his subjective decisions, mainly during the execution phase of differential code coverage.

5.5 Related Work

The works related to concern annotations and the performed studies will be presented now.

5.5.1 Concerns

Reinikainen et al. [201] present concern-base queries to reason about the program. However, their approach is based on UML (Unified Modeling Language) models, while we use the source code of a system.

Niu et al. [180] propose the application of HFC (hierarchical faceted categories) on source code. Their approach requires specialized tools whereas we use source code annotations which have a standard IDE support.

Revelle et al. [205] performed case studies regarding the identification of concerns in source code. However, the tagging was performed only manually, and their study included only two concern sets. They also present a list of concern tagging guidelines. Sadly, these guidelines are not fully applicable for us, since they created an Eclipse plug-in allowing to tag and color-mark the source code with character granularity. Standard Java does not allow annotating arbitrary statements below the method level.

ConcernMapper [208] allows a programmatical assignment of concerns to code. However, it stores the metadata in an external file. In [208], the authors also asked a similar question than us: What does it mean to label a class? They finally decided to allow labeling only fields and methods.

Developers often write "TODO comments" like `// TODO: fix this` to mark mark parts of source code which need their attention [242] (maintenance concerns). IDEs can then try to parse and display these notes in a task list window. Our approach is more formal, as annotations are a part of the standard Java grammar and can be parsed unambiguously. Furthermore, it is possible to distinguish between multiple maintenance note types through individual annotation types.

5.5.2 Annotations

Today, one of the most common applications of annotations is configuration. A programmer marks selected source code elements with appropriate annotations. Later, they are processed either by an annotation processor (during compilation) or by reflection (at runtime). For example, annotations can be used to define concrete syntax in parser generators [192] and to declare references between elements in a domain-specific language [127]. This way,

annotations can indirectly modify the annotated program semantics. In contrast, our approach utilizes annotations just as clues for a programmer which are processed by an IDE when needed.

Sabo and Porubän [216] use source code annotations to preserve design patterns. Therefore, their approach does not include recording and sharing domain and maintenance annotations.

In Java, annotations can be only applied to packages, classes, member variables, methods and parameters. @Java [41], an extension to the standard Java language, brings annotations below the method level. It allows to mark individual source code statements like assignments, method calls, conditions and loops with annotations. This could be useful to capture e.g., algorithmic design decisions with concern annotations.

5.5.3 Differential Code Coverage

The idea of subtracting code coverage results was originally described by Wilde et al. [279]. In contrast to our approach, the traced application was not executed by a user interactively. They used test cases written in a form of a source code instead. The evaluation was performed by comparing the program authors' feature-code mapping to the results obtained by an automated analysis. In [278], Wilde and Casey called their method "software reconnaissance". Sherwood and Murphy [226] observed programmers using an IDE plugin based on this approach and gave it a more expressive name "differential code coverage".

5.5.4 Feature Mapping

A commonly used technique to assign features to source code fragments are `#ifdef` preprocessor macros in C and C++. They enable variability [141]: when a particular feature is configured to be included in the produced software, the implementing code is compiled, otherwise not. Although the C preprocessor is language-independent [141] and can be in theory used also for Java and C#, it rarely is in practice. Besides, our goal was not to support feature variability via feature-oriented software product lines [263], but rather to ease program comprehension.

The CIDE tool [110] enables annotation of source code elements with features, visualizing them with colors, hiding irrelevant features, and exporting parts into modules. In contrast to our approach, it maps only features, not various run-time hints, to parts of source code. Furthermore, they save the mappings to external files, while we overwrite the source code itself.

Ji et al. [104] assessed the cost of manually annotating features in source code via specially formatted comments. According to the study they performed, the cost of adding and maintaining such annotations is small compared to the cost of the whole development process.

5.6 Conclusion

In this chapter, we reported the results of experiments studying concern annotations – source code annotations aiming to capture parts of developers' mental model, namely their concerns (intents). Each concern (e.g., "searching", "editing", "GUI code", "unused code") is implemented as an annotation type. Subsequently, all classes and methods relevant to that concern are marked with the given annotation.

First, in the controlled experiment, we showed the presence of concern annotations in the code causes a statistically significant improvement of development time when performing program comprehension and maintenance tasks on a small-scale Java project. The group which had an annotated version of the same program available, consumed 34% less time than the group which did not have concern annotations present in the source code. In the replication with industrial developers, we measured a 33% increase of correctness. We can conclude that concern annotations are useful for program comprehension and maintenance, confirming the "correctness" and "time" parts of our hypothesis (**H5.1**). Although the results regarding confidence were not statistically significant, they were positively inclined towards the "annotated" group in both experiment executions.

Second, we presented a simple an approach for semi-automatic concern annotation. The input of our demo tool, AutoAnnot, is an unannotated source code, the executable file of an application to be annotated and a concern name (or a list thereof). The user runs the program two times – once using the features of interest (a concern-related run) and then purposely not using them (a concern-unrelated run). AutoAnnot traces the sets of methods executed and subtracts the second set from the first one. The resulting set of methods is marked with Java annotations, overwriting the original source code. AutoAnnot acts as an example how a runtime-based approach can be viewed as an alternative to manual annotation.

To answer **RQ5.1**, we found that the overlap (agreement) between semi-automatic and code author's annotations is worse than the agreement among at least 2 of 7 independent annotators, but better than the agreement among at least 3 of 7 such annotators.

In our controlled experiments, the source code was commented scarcely or not at all. An interesting future experiment would consider an annotated program vs. a program without annotations, but with high-quality traditional source code comments instead.

For now, the annotation types must be supplied manually. However, user interfaces (UIs) contain domain terms which could be used for automated annotation type creation.

Chapter 6

Persisting Runtime Metadata

In chapter 4, we reviewed source code labeling approaches, including tools which assign runtime metadata (data obtained by dynamic analysis) to parts of source code. The mentioned tools usually fall into one of two categories:

- The information is displayed while the program is running, and then discarded.
- The collected data are stored in a file separate from the source code.

In the first case, the program must be re-run with the same input each time a developer needs to access the run-time information.

In the second case, each piece of information must link to the corresponding source code element. As the source code changes through time, the references may be no longer valid if the tool managing them is not notified about the modification. For example, if we refer to a source code element using its file name and line number, and a few lines are inserted above the element, the reference becomes invalid. Several approaches exist to cope with the problem of source location tracking [203]. Nevertheless, none of them works flawlessly: Even some of the most precise methods correctly refer to only roughly 90% of identifiers after source code changes [203].

Furthermore, in both cases, the tool authors have two main implementation possibilities:

- an IDE (integrated development environment) plugin
- or a standalone tool.

Each of these two possibilities has its advantages and shortcomings.

IDE plugins allow for a utilization of a tight integration between the source code, the current IDE features and the new feature of the tool. For example, we can represent a comprehensibility metric with a light or dark red background [23] directly in the editable source code view of the IDE. The background color supplements existing source code highlighting and the programmer can view this augmentation and edit the source code at the same time, without switching between two separate tools. However, plugins have also disadvantages. Since individual IDEs have very different plugin development interfaces, large parts of plugins must be developed individually for each supported IDE. For example, to support only some of the most popular Java IDEs, we must create a separate plugin for Eclipse, IntelliJ IDEA, NetBeans and JDeveloper. The situation becomes even worse when we have to consider also text editors like jEdit, Vim and Emacs. Finally, there are slight differences between individual versions of the same IDE, so a plugin developed for Eclipse

4.7 may be incompatible with Eclipse 4.3 and vice versa.

Standalone tools display the source code and metadata separately, without any connection with an IDE. This poses cognitive burden on the developer – he must switch back and forth between two tools, navigate to correct parts in both of them and mentally connect the corresponding parts. This is called a split-attention effect [20]. On the other hand, the advantage of separate tools is lower creation and maintenance cost for the tool author.

If a tool was writing the produced data directly into source code, next to the related code parts, it could be IDE-independent, while the developers could use their favorite IDE to view the data. Furthermore, the problem of code location tracking would be eliminated. Therefore, in this chapter[1], we ask the following research question:

RQ6.1 *How can runtime metadata be persisted directly in source code files?*

We will discuss two aspects of this question. First, it is necessary to select the format in which the data should be written. Second, we will be interested how such an approach could be integrated with an existing software development workflow.

6.1 Format

We suggest two possible persistence formats: annotation-based and comment-based. The former is suitable for metadata pertaining to a specific class, member variable, method or parameter. The latter can be used for metadata with line-level granularity.

6.1.1 Annotation-Based Metadata

Java annotations (or attributes in C#) are a form of meta-programming, marking program elements with additional metadata. Traditionally, they are written manually by programmers. Then, these annotations are programatically read during compilation by annotation processors, or at runtime – using reflection. Consider this example:

```
@Min(3)
@Max(120)
private int age;
```

The programmer is giving "hints" to a system that the age has valid values from 3 to 120. The system can then use these manually supplied information at runtime – to perform data validation.

We will show how an opposite of the traditional process can be advantageous. Annotations will be written into the source code automatically (using runtime information), and read later by programmers to aid program comprehension.

Let us consider a situation when a programmer wants to analyze the program using a tool which outputs results for each class, method, member variable or other annotable element in the source code. He runs a tool, independent of a particular IDE. The tool overwrites the existing source code files, writing an annotation above each method. The resulting source code has this form (Java syntax):

```
@Metadata{value}
source_code_element
```

[1] This chapter contains material from our articles: [252], [258].

The `@Metadata` annotation must be unique and not yet present in the code (before analysis).

For example, if a tool produces a numeric metric for each method, a hypothetical source code snippet looks like this:

```
@SomeMetric{1.7}
public void method() {
  ...
}
```

We will demonstrate the idea on various smaller examples.

Profiling

Output of a profiler is often used by a single programmer, not shared among developers. Hotspots, i.e., the methods which are executed for the longest time according to the profiler, could be automatically marked with the `HotSpot` annotation. The parameters might include the time percentage and an optional programmer-supplied use case name:

```
@HotSpot(selfTimePercent=32, useCase="Order")
public BigDecimal getPrice() {
  ...
}
```

The method can be then optimized by another member of the team. Re-running a tool would automatically remove the annotation if it was no longer a hotspot. Alternatively, it could be removed manually when fixing a performance-related bug.

Run-time Call Graph

A call graph consists of methods as nodes, and all possible calls between them as directed edges. While a static call graph can be generated from the source code itself, it does not distinguish whether a particular call was really executed. A dynamic call graph contains just actual method calls for a particular execution. We could annotate methods with their direct callers:

```
@Caller(clazz=Order.class, method="getTotal")
@Caller(clazz=Cart.class, method="add")
public BigDecimal getPrice() {
  ...
}
```

Call graph exploration tools were shown to improve maintenance efficiency [109]. In our case, the exploration itself would require no additional tool except a standard text editor.

Dynamic Type Information

Due to polymorphism, a method with a parameter (or a return value) of a given abstract type can accept (or return) an object of any inherited concrete type. However, while reading the source code in an IDE, only the abstract type is apparent to the developer [212]. Suppose we have a system with a complicated shape hierarchy, containing many abstract and concrete classes. Consider the following example:

```
public void clearRegion(Shape region) {
  ...
}
```

By looking at the method, the developer has no idea what types of regions were cleared during the testing of an application. Using a debugger, it is possible to find the concrete types. However, such process is laborious and the data are discarded as soon as the debugging session ends. The list of concrete classes could be determined by a tool, which would annotate the method (using Java 8 parameter annotations):

```
public void clearRegion(
 @Types({Square.class, Dot.class}) Shape region
) {
  ...
}
```

Faults and Flaws

There exist many tools for automatic or semi-automatic detection of potential faults, resource leaks or security flaws. However, the results of their analysis, which could include human work, are often thrown away after the programming session ends.

We propose that such tools should annotate the source code with the found flaws. For example, if a monitoring tool determines that in the moment of program termination, a particular file (represented by a member variable) was not yet closed, it could annotate it:

```
@ResourceLeak(Leak.FileNotClosed)
FileReader inputFile;
```

Thanks to this, the flaw remains permanently noted until it is fixed – when it is removed either by a programmer or by the automated tool. Furthermore, all team members working with this piece of code can become aware of it.

6.1.2 Comment-Based Metadata

Sometimes, class-, method- or member-level granularity is not enough. In case we need to assign line-level metadata, we can use one-line source code comments in instead of annotations. To ensure the comments can be unambiguously removed by a tool after they are no longer necessary, they must have a specific format, not present in the clean, unannotated code. In general, lines labeled with metadata have this form:

```
code_line //special_character(s) metadata
```

For instance, suppose we want to display code coverage results. We need to assign a piece of metadata to each executable line of code: whether a line was executed or not. Then a possible source code snippet can look like this:

```
public void method() {
  int i = 1; //$ executed
  if (i == 2) { //$ executed
    doSomething(); //$ not executed
  }
}
```

A similar example is the assignment of execution counts to individual lines:

```
public void method() {
  int i = 0; //+1
  while (i < 2) { //+3
    doSomething(); //+2
    i++; //+2
  }
}
```

6.2 Workflow

The annotated source code can be checked into a version control system, and thus shared with the whole development team. This improves awareness about non-functional software properties like performance and security, which are otherwise not obvious by looking at the clean code.

However, not all dynamic data are suitable to be shared across the whole team. They could cause unnecessary pollution and problems when merging the code. For this reason, we devised two types of scenarios: a local-only and shared workflow.

6.2.1 Local-Only Workflow

Before performing an enhancement or a bug-fix requiring extensive comprehension of the source code, the developer executes a tool, which annotates the source code according to collected runtime information.

After the developer finishes the work, the tool will delete all annotations previously inserted into the source code. This way, unnecessary code pollution and merging problems are prevented.

Suppose the developer inspected the source code along with the metrics and modified it with respect to some task. Now he no longer needs the metrics to be displayed – for example, he starts to work on other task, or he wants to commit his changes into a version control system (VCS). There are two possibilities:

- He manually executes the tool, instructing it to remove the annotations from the source code (e.g., by command-line arguments).
- The tool is running in the background, waiting for the VCS command to start. When it starts, the tool pauses the VCS process, removes the annotations automatically and then resumes the process.

Either way, the source code submitted to a VCS will be clean and co-workers will not be aware of the programmer using the tool.

An synopsis of this approach is that the tool is completely independent of a specific IDE. At the same time, the developer can utilize an IDE of his choice for comprehension, without constantly switching between it and a separate comprehension tool.

There is one necessary property which must be fulfilled: If the metadata are written into the source code and subsequently removed from it, all files should remain the same before performing the analysis (suppose the developer did not modify them meanwhile). If we call the labeling process *label* and the removal process *remove*, the following should hold regarding the source code (*code*):

$$remove(label(code)) = code$$

6.2.2 Shared Workflow

When using a shared workflow, the tool is executed and the IDE is then used in the same way as in the local-only workflow. However, the annotations are not deleted after the session ends. The annotated source code is checked into a version control system, where also other team members can see it.

Although the annotations can be helpful to the team, they can cause merging problems and make a version control log hard to read. Furthermore, once the data are written, they can easily become obsolete as the code changes. Since a change in some part of a program can affect the behavior of other parts, detecting such discrepancies is difficult, particularly for runtime-based information.

6.2.3 Workflow Selection

For a specific kind of annotations, it is necessary to decide which workflow to use. To avoid excessive cognitive burden on the developers, unnecessary metadata should not be kept in the source code.

Profiling data can be easily trimmed – for example, we can limit the annotation process to 3 most resource-hungry methods. In this case, they are suitable for sharing. After performance is improved, the annotations can be removed by the programmer committing the fix.

A number of methods callers tends to be large: even in a small-scale application like EasyNotes, there are more than 160 method calls within the project itself (determined by Eclipse's references search using static analysis). Although dynamic analysis reduces the amount of annotations, it may be restricted to a specific use case the programmer is trying to debug. Therefore, we suggest to use call graph annotations only in the local workflow. A similar recommendation holds for dynamic type information annotations.

Faults and flaws should be shared in a version control system to inform all developers about them. If a specific flaw is fixed, the annotation should be removed in the same commit. It is, however, questionable whether this is not redundant withe respect to issue tracking systems.

6.3 Limitations

Selected limitations of our approach will be discussed in this section.

6.3.1 Input of the Analysis

One of the advantages of our approach is its IDE-independence. However, except the source code, metadata-producing tools certainly require more input: the program execution scenarios, file paths, names of classes to be analyzed, types of the analysis and its parameters, etc. This input needs to be provided to a standalone tool – using command-line arguments, configuration files or GUIs. Nevertheless, we still eliminate the need to switch between two separate views (the metadata viewer and the editor/IDE) during the phase when the developer inspects the results.

6.3.2 Automatic Reloading

It is convenient to trigger an analysis while the project of interest is open in an IDE. Therefore, a file which is currently displayed in the IDE is modified by another process. This can possibly

Table 6.1: File-reloading capability of selected IDEs and text editors. © 2017 IEEE

IDE or text editor	Manual file reloading	Automatic file reloading	
		supported	default
Eclipse	supported	yes	off
IntelliJ IDEA	supported	yes	off
NetBeans	supported	yes	off
jEdit	supported	yes	on
SciTE	supported	yes	on
Vim	supported	yes	off

lead to confusion if the file is not synchronized. The user will not see the results of the analysis before closing and reopening the given file, which would limit the usability of the tool.

Fortunately, many IDEs and text editors offer a possibility to automatically reload a file if it is modified by other process. We summarized the possibilities and behavior of selected popular Java IDEs and text editors in Table 6.1.

We can see that both manual (on-request) and automatic (in background) file reloading is available in all surveyed applications. Furthermore, in some text editors, automatic file reloading is turned on by default. When the default option is off, it can be easily adjusting in the application's settings. Therefore, we can conclude that file reloading is not an issue at all.

6.3.3 Presentation Limitations

The most prominent limitation of our approach is its usability only for purely textual metadata. Insertion of graphics, such as graphs, diagrams, etc., is inherently not possible in its current version. However, Schugerl et al. [222] present an Eclipse plugin SE-Editor which displays images directly in the IDE code editor – in places where specially-formatted comments are inserted in the source code (e.g., `/*** http://image.url`). If these two approaches were combined and SE-Editor was implemented also for other IDEs and text editors, if would effectively mean our approach could support also graphical elements.

Another practical limitation is the non-interactivity of our approach. Standard IDE plugins offer rich interaction possibilities. In contrast to them, the inserted comments comments are generally static and non-navigable. However, annotations offer some interaction possibilities in many IDEs. Since they can be programatically queried, by annotating the source code of a program, we could also ease querying of run-time information associated with source code elements. All standard tools which can be used on annotations are applicable. For example, the Find Usages capability of an IDE may be used to find all hotspots or potential flaws in the code. It is possible to use standard mouse and keyboard navigation (e.g., Ctrl+click) on the class names in the caller lists or concrete type annotations. Each flaw can have its documentation, displayed in a tooltip above the flaw type.

6.3.4 Applicability to Other Languages

Annotation-based labeling require the given language to support attribute-oriented programming. Examples of such languages are Java and C#.

Comment-based labeling requires the given language to have one-line comments which can be appended at the end of lines. The range of languages supporting one-line comments is very broad.

6.4 Related Work

Now we will present related works.

6.4.1 Code Annotation

Concern annotations [246] are annotations acting as hints for developers, informing them about intentions behind a code element. Originally, they were intended to be inserted manually into the source code, preferably by programmers writing the code. In contrast to them, we discuss the automated insertion of metadata to source code using dynamic analysis, both in the form of annotations and comments, and the options of their removal.

Joy et al. [105] presented an approach to automatic C source code back-annotation using information obtained at runtime. However, they collected only timing and power information, which is later used in embedded software simulation. Therefore, their annotations are not intended for human consumption and program comprehension, as in our approach.

6.4.2 Runtime Information

Senseo [212] displays various information like a number of invoked methods or created objects, callers, callees and concrete argument types, in an IDE window. However, since Senseo is an Eclipse plugin, it is IDE-dependent.

An in-situ profiler [20] displays small visuals next to method declarations and calls, presenting performance data obtained by profiling. The results are not persisted in any way, though.

6.4.3 IDE Extensions

Lee et al. [133] designed a generic infrastructure for framework-specific IDE extensions. They try to reduce authors' work when creating an IDE extension for a new object-oriented framework. However, providing an infrastructure for multiple different IDEs is out of the scope of their work.

Asenov et al. [7] present an IDE which enables programmers to query and modify programs using a combination of source code and other resources. Custom queries can be written using scripts, and the queries are composable using pipes. While they try to clear the line between plugin authors and application developers, their approach is currently limited to a single IDE which they created from scratch. In contrast to them, we try to reuse existing IDEs as much as possible.

6.4.4 Workspace Sharing

Code Bubbles [32] is an unconventional development environment. Among other features, it supports sharing parts of source code annotated with values of variables obtained by debugging, notes, image flags, etc. While the presentation is visually attractive, a specialized tool needs to be used by all developers to utilize such features. In the case of Java annotations, any standard IDE can be used.

TagSEA [241] is an IDE extension enabling "social tagging". The developers can insert specially-formatted comments into source code. Such marks are then shared in the development team. There are two shortcomings: First, the tags in comments are not a part of Java

syntax, thus requiring a special tool support to be processed. Second, they are inserted into source code manually by programmers.

Source code bookmarks are one of the standard features present in today IDEs. They allow a developer to mark specific source code lines with notes and later list the bookmarks and jump to each of them. Collective code bookmarks [85] provide a way to share parts of developers' mental models. However, this approach is IDE-dependent and the bookmarks are not saved to the original source code files which complicates standard procedures like versioning and merging.

6.5 Conclusion

In this chapter, we discussed the forms of persistence of metadata obtained during executions of programs – data from dynamic analysis – directly in the pertaining source code files.

Two types of in-code storage formats were suggested: annotations and one-line comments. Although the latter format can be used in any language and allows line-level granularity, the former one offers richer interaction possibilities in IDEs.

The metadata can be either used only locally or shared with colleagues. The purpose of the local-only workflow is to enable independence of a tool on a particular IDE. Therefore, the programmer can use an environment or text editor of his or her choice. When the metadata are no longer necessary, they are removed to prevent VCS pollution. On the other hand, the shared workflow is suitable for metadata which could be useful for the whole team. Their potential obsolescence and information overload are its disadvantages.

Although we were focused on runtime information, the approach can be extended to map practically any machine-produced information to a piece of code – output from static analysis, build log messages, etc.

Chapter 7

Location of GUI Concepts in Source Code

Developers understand a program only when they are able to mentally connect structures in the program with real-world concepts [25]. Naturally, this connection can be established much more easily if the vocabulary used in the source code resembles the domain terms displayed in the GUI of a running program.

One of the most frequent activity performed by a programmer is feature location – finding an initial source code location implementing a given functionality [59]. To perform it, developers rarely use complicated feature location tools and plugins [147], and rely on simple textual search instead [53].

Consider a developer trying to fix a bug in a program he does not know. He will probably start with an exploration of a running UI (user interface) relevant to the bug. He will start to concentrate on particular GUI items, like buttons and menu items causing the bug to manifest. Then, he will try to search for the labels of these GUI items (button captions, menu names) in the source code of the program, using standard search functionality of an IDE (integrated development environment). The IDE will search the (static) source code of the program, producing a list of all results – potentially relevant locations in the code.

Sometimes this tactic can be successful, and the programmer can quickly and almost effortlessly find the code of interest. Another time, the list of results can be empty or contain too many results. The programmer must use another, more complicated techniques, to find the relevant code. In this chapter[1], we describe a simple simulation study trying to quantify the potential successfulness of static source code search for GUI terms. We scraped the GUI of four Java desktop applications and performed an automated search of all strings (and words contained in them) in the corresponding source code.

We formulate our research questions for this chapter as follows:

RQ7.1 *What portion of strings and words displayed in the GUIs of running desktop Java applications are located in their static source code too?*

RQ7.2 *Mainly in what types of files are these terms located?*

RQ7.3 *If some strings are not located in the source code, what are common reasons?*

[1] This chapter uses material from our article [250].

7.1 Method

We automatically extract strings from a running GUI of a few applications and try to search for these strings (and their parts) in the source code of the corresponding program.

In Table 7.1, there is a summary of the studied objects. We selected three desktop Java applications from the SF110 [74] corpus of open source projects: FreeMind[2] is mind-mapping software, PDFsam[3] splits and merges PDF files, and Weka[4] is machine learning software. Additionally, ArgoUML[5] – a UML modeling tool – was selected as a popular, medium-sized project. The "Java LOC" column in Table 7.1 denotes the number of source code lines in Java files, measured by the the CLOC[6] program.

Table 7.1: The applications used in the study

Application	Java LOC	GUI strings	GUI words
ArgoUML 0.34	195,363	307	2,391
FreeMind 1.0.1	67,357	353	1,050
PDFsam 2.0.0	23,774	65	168
Weka 3.6.13	275,036	592	904

7.1.1 GUI Scraping

Before running the experiment, we ensured English localization was set in all applications, since the source code is written in English and a mismatch between the code and GUI language would produce skewed results. In the case of the FreeMind application, language adjustment in the settings was necessary, all other programs had the language already set correctly.

Every application was fed to a GUI ripper [160] which is a part of the project GUITAR [177]. The GUI ripper fully automatically opens all available windows in the program, checks all check-boxes, clicks the menus, etc., in a systematic way. The properties of all widgets are written in a form of an XML file.

From the XML file, a text and title was extracted for all recorded widgets. The number of unique strings for each application is in Table 7.1, column "GUI strings". Examples of these strings include button labels and tooltips, text-area contents, items in combo-boxes and many more. We excluded strings shorter than two characters, as they do not represent realistic search queries for further analysis.

7.1.2 Analysis

Some of the strings contain multiple words, or even lines of text. For this reason, we broke them into individual words. We define a word as a sequence of three or more letters. The number of unique words for each application is in the column "GUI words" in Table 7.1.

For each string contained in the GUI, we searched it in the source code files of the corresponding project. The same process was repeated for individual words.

[2] http://sourceforge.net/projects/freemind/
[3] http://sourceforge.net/projects/pdfsam/
[4] http://sourceforge.net/projects/weka/
[5] http://argouml.tigris.org/
[6] http://github.com/AlDanial/cloc

Table 7.2: The occurrence counts of whole strings from GUIs in the source code

Application	Occurrences of GUI strings in code				
	0	**1**	$[\mathbf{2},\mathbf{10})$	$[\mathbf{10},\mathbf{100})$	$[\mathbf{100},\infty)$
ArgoUML	20.5%	10.4%	42.3%	12.1%	14.7%
FreeMind	7.9%	2.8%	60.6%	13.0%	15.6%
PDFsam	13.8%	13.8%	50.8%	4.6%	16.9%
Weka	8.1%	12.0%	14.0%	30.2%	35.6%
Total	**11.2%**	**9.3%**	**34.9%**	**20.1%**	**24.4%**

Table 7.3: The occurrence counts of individual words from GUIs in the source code

Application	Occurrences of GUI words in code				
	0	**1**	$[\mathbf{2},\mathbf{10})$	$[\mathbf{10},\mathbf{100})$	$[\mathbf{100},\infty)$
ArgoUML	4.8%	3.1%	13.6%	29.4%	49.2%
FreeMind	0.7%	0.1%	26.0%	32.9%	40.4%
PDFsam	4.8%	7.7%	4.2%	18.5%	64.9%
Weka	6.6%	0.7%	5.9%	32.5%	54.3%
Total	**4.2%**	**2.1%**	**14.6%**	**30.4%**	**48.7%**

Regarding the source code, we used tarballs of the same versions as the binaries. The PDFsam tarball contained also automatically generated JavaDoc API documentation, which we removed, since such files should not be included in source archives.

The searching was performed fully automatically, via a script. We decided to perform a case-sensitive search, which should be more precise, especially to locate whole GUI strings.

7.2 Quantitative Results

In this section, we will answer the first two research questions.

7.2.1 Occurrence Counts

First, we would like to simulate a situation when a programmer tries to find a whole GUI string in the source code. For each string, we determine a number of occurrences in the project – essentially the number of search results he would get in an IDE. For example, the string "Generate Data" (a button label in the Weka application) has 2 occurrences in the source code of the Weka project. Only text files were searched – this behavior is consistent with the majority of common IDEs which ignore binary files when searching.

In Table 7.2, we can see what portion of all GUI strings has no occurrence in the source code, exactly one occurrence, from 2 to 10 occurrences, etc. For example, 20.5% of strings from the ArgoUML GUI were not found in the source code at all. The "Total" row represents a weighted average for all four projects.

Similar statistics, but for individual words, are located in Table 7.3. This represents the situation when the programmer is unhappy about the results and starts searching for smaller parts of the given string – usually words.

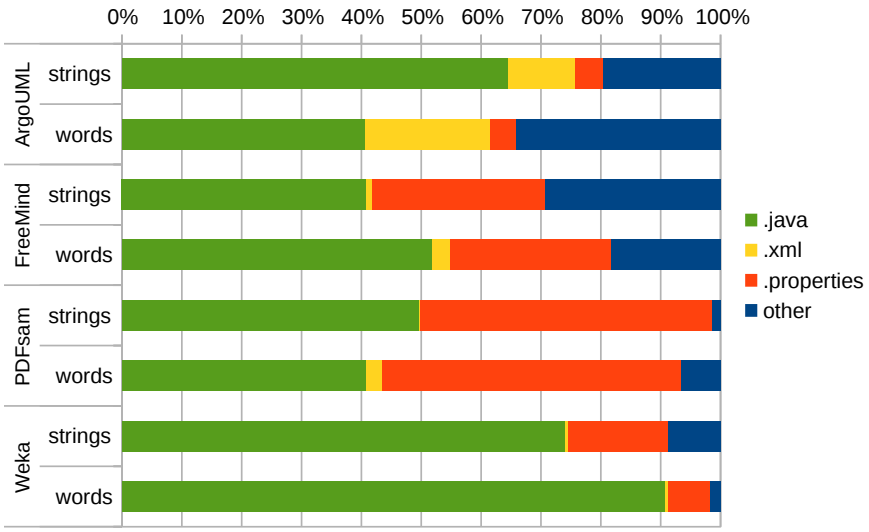


Figure 7.1: Occurrence counts of strings/words divided by file types

An ideal situation arises when a search gives exactly one result. It can (in theory) mean that the programmer found the sole piece of code relevant to the GUI widget. The higher is the number of occurrences, the longer he must sift through search results to find the relevant code. However, a situation when a searched string is not found in the source code is unfavorable, as the developer would have no idea where to start searching for an implementation of the given feature.

7.2.2 File Types

Ideally, the search results point to Java source files (`*.java`). This way, it is possible to directly find executable code related to some program feature. However, the source code of a project usually contains many kinds of files – not just Java source files.

For each project, we took all GUI string (and word) occurrences in all files and divided them by an extension of a file they are located in. In Figure 7.1, there is a graphical representation of the results.

In ArgoUML, 65% of all occurrences of GUI strings are contained directly in Java source code files. Although the project uses `*.properties` files for internationalization, GUI concepts are often used also as identifiers in Java code. For example, a GUI label “Notation” is present many times in the source code in the form of the class name `Notation`.

The FreeMind project uses “properties” localization files, too.

PDFsam uses a system where each key in a `*.properties` file is the original English string, and the value is the translated one. Therefore, the GUI strings are located both in `*.java` and `*.properties` files.

While Weka also uses `*.properties` files for localization, many GUI strings are also contained in special DSL [124] (domain specific language) files with an extension `.ref`.

7.3 Qualitative Results

First, we will mention examples of strings not found at all, to answer the last research question. Then we will select a few strings which were found in the code and show that in this case, the search can be successful or impractical.

7.3.1 Strings Not Present in Code

Many GUI strings were not found in the source code of the application because they were a part of a universal dialog supplied by the GUI toolkit. For example, color-related terms like "Saturation" were not found in the FreeMind source tree because they are a component of the standard Java Swing color chooser dialog. Examples of such strings in ArgoUML and Weka are "File Name:" – a part of a file selection dialog and "One Side" – a term from the printing dialog.

The string "http://simplyhtml.sf.net/" displayed in FreeMind was located only in a class file inside a third-party JAR archive. Therefore, it is invisible for a standard textual search.

The string "Mode changed to MindMap Mode" was not found in the FreeMind source code as a whole because it was instantiated at runtime from the template "Mode changed to {0}". This forces the developer to find smaller portions of a string until a match appears, as we mentioned earlier.

The label "Show Icons Hierarchically" was not found in the source code of FreeMind because it was written as "Show Icons &Hierarchically" to specify the keyboard shortcut Alt+H.

Examples of strings which are not present in the code as a whole, but their parts can be found there, are help texts, logs and exception stack traces.

Regarding PDFsam, the label "Thumbnail creator:" was not found in the code because the colon was programatically concatenated.

7.3.2 Strings Present in Code

First, we tried to search for a string which is present exactly once in the FreeMind code: "Change Root Node". It was located in a localization file, as a value of a key named "accessories/plugins/ChangeRootNode.properties_name". Opening the file "accessories/plugins/ChangeRootNode.java" revealed that this Java class is really relevant to the feature.

Next, we searched for a string present 35 times in the code – "Bubble". It represents a node format in the mind-mapping software. This time, the exploration of the results took more time and we required multiple iterations using different search terms, even with some dead ends, until we finally found the relevant source code.

Finally, we searched the string "Export" (a menu item), present 1,988 times in the source code. Just skimming through such a long list is a lengthy activity. Therefore, other strategies are necessary to efficiently find the feature of interest in the code. For example, the programmers can reformulate their search queries, use structured navigation (tools like Call Hierarchy) or debugging techniques [53].

We conclude that in some cases, simple textual searching is a feasible way to find code relevant to a GUI element. Ideally, a GUI string should be located exactly once (or just a few times) in the source code, to allow easy finding of source code relevant to a GUI feature. Furthermore, finding an occurrence in a non-Java file makes it more difficult to find relevant source code than finding it directly in a Java file.

7.4 Threats to Validity

We will now look at the threats to validity of our study. Construct validity is concerned with the correctness of the measures. External validity discusses whether the findings can be generalized. Reliability denotes whether similar results would be obtained by another researcher replicating the study [215].

7.4.1 Construct Validity

While the GUI ripper in the GUITAR suite gives good results when scraping the GUI, it is not perfect and it could miss some of the strings visible in the user interface.

During an automated search for whole GUI strings in the code, also long texts like exception stack traces were included. It is not probable that a programmer will actually try to search for a whole stack trace in the code textually, as-is. Instead, he will directly look at some of the methods mentioned in the trace.

We performed a case-sensitive search, which is more precise. However, in practice, case-insensitive search is probably the preferred way, as it is often default in IDEs. In the future, a case-insensitive search should be also performed to better reflect the manual searching behavior of programmers.

7.4.2 External Validity

All four applications in our study were desktop Java programs, using the Swing GUI widget toolkit. However, common contemporary applications have Web and mobile front-ends. Scraping Web applications could have produced much different results. For example, they often display texts downloaded from external databases. This could be one of the reasons for non-presence of GUI strings in the code of these applications.

Even in the world of Java Swing applications, the selected ones represent just a small sample. However, they are representative of common desktop Java projects, as three of them were included in the standard SF110 [74] benchmark.

7.4.3 Reliability

The quantitative results were produced chiefly by an automated script. Therefore, the subjectivity of a researcher is eliminated. The strings presented in the qualitative part were selected manually, but the purpose of this part was mainly illustrative.

7.5 Related Work

Now we will present related work.

7.5.1 GUI Ripping

To rip GUIs, we used the tool GUI ripper [160] which is a part of the GUITAR [177] suite. Swing UIs are one of the best supported technologies, however, there is a partial support for SWT, Web, and Android. To crawl highly dynamic Web applications, Crawljax [161] could be used.

The DEAL method [15] creates a DSL from a GUI. However, the process is not automated and a user must manually traverse the user interface.

7.5.2 Feature Location Using GUIs

Of course, finding a string from a UI using IDE's textual search is not a sole option to perform feature location. GUITA [220] allows to take a snapshot of a running GUI widget. The snapshot is associated with a method which was last called on the given widget. However, this approach is dependent on a particular GUI framework.

Another approach, UI traces [259], splits a long method trace into smaller ones, each associated with a graphical snapshot of the GUI in the given state.

7.5.3 Feature Location in General

There exists a large number of feature location techniques – see [59] for a survey. An example of a method utilizing source code comments and identifiers is presented by Marcus et al. [152]. Carvalho et al. [39] use a combination of static and dynamic analysis, specifications and ontologies to map problem domain concepts to source code elements. However, none of these approaches use labels directly from GUIs of running programs.

7.5.4 Other Studies

Václavík et al. [271] analyzed words used in names of identifiers in the source code of Java EE application servers and web frameworks. They tried to determine what portion of these words are meaningful according to the WordNet database. The more words from the source code of a project are meaningful, the more understandable it should be. We could perform a similar experiment, but use a dictionary built from the GUI of an application instead of WordNet.

7.6 Conclusion

A simple study was performed: We scraped all strings contained in a GUI of four open-source Java desktop applications and tried to automatically find them (either as a whole or single words contained in them) in the static source code.

Regarding question **RQ7.1**: 88.8% of strings and 95.8% of words from GUIs were found in the source files. However, 24.4% of displayed strings and 48.7% of words were found at least 100 times, which is a number of results we consider impractical to inspect during maintenance tasks.

Answering **RQ7.2**, the GUI terms are often located in Java source code files (67.0% of strings, 51.4% of words), `*.properties` localization files, XML and custom DSL files.

To answer **RQ7.3**, common reasons of a non-presence of a GUI string in the source code were: a string was a part of a standard dialog, a third-party library, or the string was dynamically generated at runtime.

In further research, we can replicate the study on Web applications, or study logs in addition to GUIs. Another idea is to consider a set of GUI words the application's problem domain dictionary. The percentage of GUI words present in the source code can be regarded as a measure of code understandability. For example, if the code contains too few concepts from the GUI, it can be considered obfuscated.

Chapter 8

Searching in Runtime Values

Suppose a developer needs to answer a common question [227]: Where in the source code is the label displayed in the UI (user interface) of a running program located? Traditional source code search is performed on static source code, and it does not utilize dynamic information. In the previous chapter, we shown that such a search often does not produce desired results, since only a portion of the strings is located in the source code in a form of string constants. Dynamically generated and localized strings, user input, data obtained from other systems, and many other strings may not present in the static source code of the application. Instead, the developer should ask: Which expression contains the given string in its value at runtime?

A question like this could be possibly answered by writing custom scripts and queries in automated and scriptable debuggers such as Coca [62], FrTime [150] or EXPOSITOR [113]. However, we should take the reality of developers [202] into account: The cost of learning to use a tool must be lower than the expected benefit of its application. Ideally, the developer should use a tool immediately or only after minimal training. Even when using existing tools like a textual search, developers prefer simple queries compared to complicated, sophisticated ones [53].

Considering the developer already found the source code location related to the UI element, she will probably want to examine it from the dynamic viewpoint – inspect the concrete values of variables at a specific moment, view the stack frame, etc. Currently, searching and debugging are considered separate actions accomplished with different tools. However, according to multiple empirical studies, these two activities are often interleaved. For example, based on the results of a field study with professional developers, Damevski et al. [53] argue there should be better integration between code search, navigation and debugging. Wang et al. [276] studied developers locating relevant parts of source code; two of three search patterns the developers used encompassed running and debugging the program.

After finding and inspecting an initial point of investigation, developers often aim to find other occurrences relevant to this point. For instance, they try to determine where the data from a certain variable flow [120]. In theory, this is easy to accomplish using approaches such as dynamic program slicing [1]. In practice, the data can flow through multiple third-party systems and return back to the inspected application. For example, a user input can be saved to a database, then retrieved and displayed in another part of the application. Classical program slicing approaches could fail to find such a connection. Although cross-system slicing approaches emerge, they suffer from performance issues [26] or are technology-dependent

[178].

With the mentioned considerations in mind, we designed RuntimeSearch, described in this chapter[1]. It is a variation of a traditional textual search in source code, but in this case, we are searching in the values of expressions at runtime. A programmer enters a string which she wants to locate. It can be, for instance, a UI label visible in a running program for which she wants to find the source code part displaying it; or a string which she hypothesizes is a value of some unknown variable or expression. If the program is not already running, it is launched. During the runtime, all evaluated string expression values are being compared with the searched term. When a match is found, the program execution is paused and a traditional IDE debugger is opened. The programmer can explore runtime properties of the program like the call stack and current variable values. Then, she can continue with debugging, running, or search some string again.

In essence, RuntimeSearch is an extension of a traditional debugging process. In addition to standard debugging operations like Step In or Continue, a Find In Runtime action is available. This runtime-search action provides a user interface resembling a simple textual search dialog, which practically all developers are familiar with. Therefore, the tool should require almost no training.

The source code of our RuntimeSearch implementation is available online[2]. A case study was performed to show the feasibility and usefulness of the technique. It provides us an answer to our first question:

RQ8.1 *What are possible use cases of RuntimeSearch?*

As a debugging extension, RuntimeSearch incurs some runtime overhead. Therefore, we performed a preliminary performance evaluation, asking:

RQ8.2 *What is the performance overhead of searching strings in the runtime?*

Finally, a controlled experiment on human participants was performed to validate our approach, trying to confirm the hypothesis:

H8.1 *RuntimeSearch improves the efficiency of search-focused maintenance tasks.*

8.1 Runtime-Searching Approach

First, we will describe RuntimeSearch from the user's (programmer's) point of view. Next, we will look at its design and implementation.

8.1.1 User's View

The interaction and user interface of RuntimeSearch was modeled with two principles in mind:

1. To look similar to a traditional textual search in source code whenever it is possible. Since code searching is a familiar operation for practically every programmer, this should make learning the tool easy.
2. To integrate the approach with the debugging infrastructure already present in IDEs and used by developers.

Each runtime searching begins by triggering the action "Find in Runtime" in an IDE, which shows a query prompt, where the developer enters the searched text. Subsequently, the searching process is started in one of three ways:

[1] This chapter contains material from our article [255].

[2] https://github.com/sulir/runtimesearch

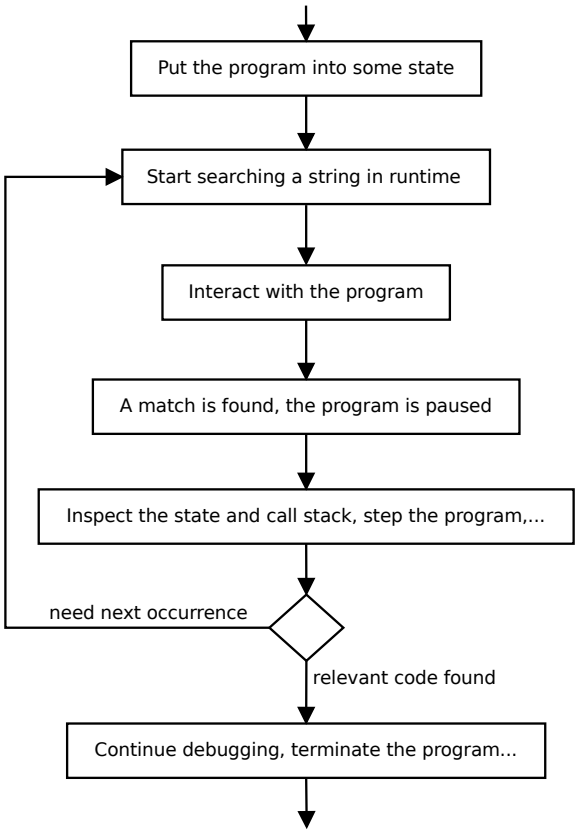


Figure 8.1: A possible example of RuntimeSeach utilization. © 2017 IEEE

- If no program is currently being debugged, a new instance is launched.
- If a debugged program is paused, it is resumed.
- If a debugged program is running, it is left running.

The string is searched from the moment it was entered into a prompt until a match is found. When this happens, the program is immediately paused using a programmatically invoked breakpoint. This causes the IDE to highlight the currently executed line and show current variable values. Furthermore, all debugging features of the given IDE are available – including the stack frames view, expression evaluation, advanced object state inspections, etc.

When the programmer considers the current search result irrelevant or wants to find the next occurrence, she triggers the action “Find Next in Runtime”. In case she wants to change the search string, she chooses the action “Find in Runtime”. The process continues as already described.

If the developer does not want to search a string anymore, she can continue debugging with traditional operations like Step In and Step Over or resume the program. At any moment, it is possible to search the string in the runtime if desirable.

One of many possible examples of the developer’s interaction with RuntimeSearch is displayed in Fig. 8.1. However, the process is not prescriptive and can be adapted to developer’s specific needs. In section 8.2, we will describe multiple usage scenarios.

8.1.2 Principle

We look at a running program as a series of expression evaluations. Consider the following Java source code excerpt:

```
String var = "text";
var = var.toUpperCase();
```

It produces the following expression values, in the given order: "text" (a string constant), "text" (the value of the variable `var` read on the second line), and "TEXT" (the return value of the method `toUpperCase`).

The program is instrumented, so the result of every string expression is captured and compared to the searched text. Currently, only expressions of type `String` are captured – it makes the most sense to compare the searched text only to strings. However, the approach is not limited to this behavior and in the future, we could convert all objects to their string representations using a `toString`-like method.

Namely, we capture the following `String` expressions: constants, local variables, member variables, constructor calls, and method calls returning values.

In our early implementation, the evaluated strings are matched against the query using a simple string containment: if the evaluated string contains the searched text, a match is found. Again, this is not an inherent limitation of the approach. It can be easily extended with standard text-search options like "match case", "whole words only", regular expression matching, or advanced features like approximate (fuzzy) string matching.

8.1.3 Implementation Details

Our RuntimeSearch implementation consists of a Java agent (a piece of instrumentation code which can be attached to any Java program) and a plugin for the IntelliJ IDEA IDE.

The agent is configurable through an argument – a pattern specifying what packages or classes should be instrumented. This way, it is possible to specify, e.g., whether to include only application or also system classes.

Instrumentation is performed at the bytecode level. In stack-based virtual machines like the Java Virtual Machine, each expression evaluation is represented by an instruction pushing a value on the operand stack, which can be used with advantage.

The IDE plugin is very lightweight and consists mainly of a simple form and a module for communication with the agent.

8.2 Case Study

Now we will show how searching in the runtime can be useful to perform navigation and debugging tasks. The demonstration will be performed on Weka[3] – an open source Java machine learning software. It is a Swing GUI (graphical user interface) application with approximately 350,000 lines of code. A short video with parts of the case study is available online[4].

8.2.1 Finding an Initial Point

One of the questions programmers ask during program maintenance tasks is "Where in the code is the text in this error message or UI element?" [227] Weka includes a package manager which displays a list of additional packages. Suppose we want to find the code which retrieves

[3] https://sourceforge.net/projects/weka/

[4] https://sulir.github.io/runtimesearch

package names, such as “AffectiveTweets”. Searching the static source code for the term “AffectiveTweets” does not give any results, since this string was generated at runtime.

Therefore, we run the application in a debug mode and wait until the main window appears. Then we trigger the “Find in Runtime” menu in the IDE, enter the query “AffectiveTweets” and press Find. After choosing the menu “Package manager” in Weka, the program automatically pauses and the IDE shows us the currently executed line, containing the searched string expression. We see the first runtime occurrence of the string “AffectiveTweets”, i.e., its origin. Not only can we notice it was read from a file input stream (it is obvious by reading the source code), but thanks to the IDE showing a string representation of the stream object at runtime, we also see its path in the file system – the name of the “package list” file from which the string “AffectiveTweets” was read.

8.2.2 Searching for Occurrences

After quickly finding the initial point of investigation, it is up to us how to continue. We can explore the source code, find static references, or debug the program. Another interesting possibility is to search for all following runtime occurrences of the string “AffectiveTweets”. This can be achieved by repeatedly triggering the action “Find Next in Runtime” in the IDE (e.g., using a shortcut) – each time, a new occurrence is found.

This way, we find many precise locations in source code relevant to package name retrieval and displaying. This includes reading a package list file, manipulation with an object representing the given package, reading a version list file, GUI code displaying the package name, and various helper methods. After each step, we are free to explore the current runtime properties of the program to improve our understanding, or step into a method of interest. For example, if we are interested how the version file URL is determined, we can step into the `getConnection` method while we are in the method `getRepositoryPackageVersions(String packageName)`.

8.2.3 The Fabricated Text Technique

Instead of just “passively” searching for a string which the application itself displayed, we can utilize an interesting technique: Enter a made-up string into a text field and search for its runtime occurrences. This allows us to track the data flow of the string across program layers: from presentation through model to persistence.

For instance, we open the Weka Bayes Network Editor, start searching for a fabricated string like “Node987” in the runtime using RuntimeSearch, and create a new node named “Node987” in the Bayes Net editor. The debugger will immediately pause at the GUI code processing the node name. By searching for next occurrences, we are being navigated through more or less specific node-processing and data structure classes. After pressing the Save button in Weka and continuing the search, we are navigated through methods converting Bayes network nodes to XML representation and, finally, to the file-saving method.

8.2.4 Non-GUI Strings

Until now, we were searching only for strings present in the GUI in some way. However, RuntimeSearch is not limited to such texts.

We created a simple layout in the Weka KnowledgeFlow Environment, containing a DataGrid connected to a Database Server, all with default settings. Weka KnowledgeFlow contains an option to save the layout file in the KFML format (an XML dialect). We tried

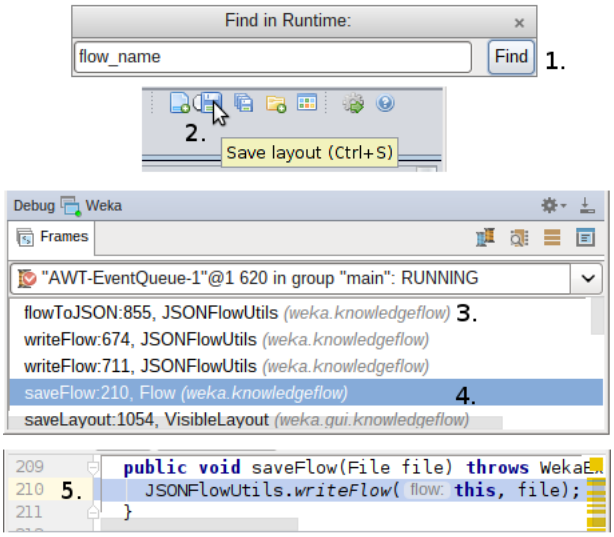


Figure 8.2: RuntimeSearch in action: The IDE plugin (1.), an action in Weka (2.), IntelliJ IDEA debug window and source code view (3.–5.). See the main text for full description. © 2017 IEEE

it, but it does not behave as expected – after selecting this option, a file is saved as JSON (JavaScript Object Notation) instead.

We view the content of the file in a text editor and choose an excerpt, such as the string "flow_name". We search for this excerpt in the runtime using RuntimeSearch (Fig. 8.2, part 1). After triggering the Save action in Weka (part 2 of Fig. 8.2), the program is automatically paused. We inspect the call stack: three methods at the top are located in a class starting with "JSON" (Fig. 8.2, label 3). This means during the execution of these methods, the wrong output format is already selected. Therefore, we click on the fourth item (label 4 in the figure). Immediately, we see the cause of the bug: The method `saveFlow` contains a hard-coded call to `JSONFlowUtils.writeFlow` (see part 5 of Fig. 8.2).

This is a demonstration how RuntimeSearch facilitates finding where in the project is the layout-saving code located, while immediately providing a context for debugging (the call stack inspection and subsequent root cause identification).

8.2.5 Hypothesis Confirmation

The strings searched so far were present either in the GUI or in external resources like files. RuntimeSearch can go even further. During debugging, developers often form hypotheses about program behavior [132]. If the programmer has a hypothesis about a presence of a certain string in a specific member variable, she can utilize field watchpoints in common IDEs. However, if she does not know which variable is concerned, or even whether it is a variable and not only a temporary expression, runtime searching is very convenient.

If we want to open the Weka package manager from the main menu while there is no network connection available, nothing visible happens – even no error message is displayed in the GUI, which is certainly not user-friendly. We hypothesize Weka is trying to establish an HTTP connection to load the package list, but it fails. We search for the string "http://" using RuntimeSearch and click the Package Manager menu item in Weka. The debugger pauses at the URL creation code. We find out the first part of our hypothesis is confirmed: An HTTP connection is created and tried to be opened. After a few "Step Over" actions in the IDE, we see a thrown `UnknownHostException` is caught, the stack trace is printed to the standard error stream, but no GUI message window is shown.

Similar hypotheses could be formed about SQL queries in information systems, regular expressions in parsers, etc.

8.3 Performance

As a form of a primitive benchmark, we measured the execution time of all unit tests from the package "weka.core" under three circumstances: without instrumentation, with instrumentation but without searching, and while searching for a short string during the whole execution. The execution was performed in debug mode, and system classes were not instrumented. A mean of three measurements was computed for each condition.

The difference between the plain and instrumented execution was negligible (14.653 vs. 14.908 s). Searching incurred reasonable overhead (38%), the measured time was 20.223 s.

The most prominent slowdown was noticed during the "test instantiation" phase, not included in the above times. During the instrumented run, it lasted 8-9x more than in the plain one. This can be attributed to high overhead of the class instrumentation process itself. Note that this slowdown predominantly affects startup time, further interaction with the application is swift. Furthermore, this is only an implementation issue, and using an alternative instrumentation library should improve the performance significantly.

8.4 Quantitative Evaluation

To measure the effect of using RuntimeSearch, we decided to perform a controlled experiment with human participants. They performed maintenance tasks involving the search of strings in a large code base. One group of subjects could use RuntimeSearch, while the other group had only a standard IDE available. Our null and alternative hypotheses are:

H8.1$_{\text{null}}$ The efficiency of search-focused maintenance tasks with RuntimeSearch = the efficiency using only a standard IDE.

H8.1$_{\text{alt}}$ The efficiency of search-focused maintenance tasks with RuntimeSearch > the efficiency using only a standard IDE.

8.4.1 Method

Now we will describe the subjects, objects, tasks and measurement procedures of the study.

Participants

The experiment was performed during three Metaprogramming classes. All participants were students, although many of them reported they already had work experience.

The total number of participants was 41. However, we excluded one participant since he did not follow the tutorial correctly and subsequently decided not to use RuntimeSearch during the tasks despite he was in the control group. Therefore, we processed the data of 40 subjects.

The assignment to groups was random: Upon arrival, each student drew a number with a number of the computer which (s)he should use. On each computer, there was one version of the materials available – with or without RuntimeSearch. 21 of the analyzed subjects were assigned to the treatment (RuntimeSearch) group, 19 to the control group.

Objects and Tasks

The study was performed on jEdit[5], a desktop Java text editor with approximately 125,000 lines of code. The IDE used during the experiment was IntelliJ IDEA 2018.1 Community Edition.

First, the participants read a tutorial which included small practical tasks that should they perform in the IDE. The treatment group used RuntimeSearch in the tutorial, while the tutorial of the control group contained a review of standard static search and navigation techniques such as a textual search or "find usages".

The main part of the experiment consisted of two tasks. The synopsis of the tasks was:

- **Task 1:** In the Category column of the Plugin Manager installation tab, display all category names in lowercase.
- **Task 2:** In the jEdit title bar, show a colon (":") instead of a dash ("-") after a custom title is set, e.g. "jEdit: Untitled".

The participants were instructed to try to complete the tasks as soon as possible. If they could not complete a task, they were asked to the next task after about 30 minutes, but this limit was not enforced in any way.

One session lasted 1 hour and 30 minutes in total. We performed the experiment in 3 sessions, each with a part of the participants.

Measurement

We used a web form as a primary data collection mechanism. For each completed task, a subject pasted the textual patch of the changes made in the source code into the form. After the experiment, we manually inspected the patches and decided whether they are correct or not.

Task completion time was measured by a time-tracking IntelliJ IDEA plugin. While the participants were asked to start and stop the timer manually, many of them did not do this precisely of forgot to use it at all. Fortunately, we also recorded the screens during the experiment. We inspected the screen recordings and noted the exact time completion tasks for all participants. One participant opted not to record the screen – we used the information given in the form in this case.

The independent variable is the usage or non-usage of RuntimeSearch. The dependent variable is efficiency, defined as the number of successfully completed tasks divided by the total time consumed on both tasks (in hours). The null hypothesis will be rejected if the p-value is less than 0.05.

8.4.2 Results

In Table 8.1, there are the complete results of the experiment. Figure 8.3 contains a graphical comparison of the efficiency of two groups. The median efficiency for the RuntimeSearch group is 2.89 tasks/hour, while the control group had a median of only 1.80 tasks/hour. Therefore the RuntimeSearch group achieved 60% higher median efficiency.

We inspected the distribution of the efficiency using histograms. Since the data were not normally distributed, we performed a one-sided Mann-Whithey U test. The resulting p-value is 0.0291, which means the result is statistically significant.

[5] http://www.jedit.org

Table 8.1: Detailed results of the RuntimeSearch experiment

(a) the “RuntimeSearch” group

ID	Task 1		Task 2		Efficiency [tasks/hour]
	Correct	Time [m:s]	Correct	Time [m:s]	
01a	yes	14:10	yes	2:27	2.89
01b	yes	12:11	yes	1:21	3.55
01c	yes	19:49	yes	2:04	2.19
03b	no	17:54	yes	3:45	1.11
03c	yes	9:45	yes	2:52	3.80
08a	yes	10:12	yes	2:55	3.66
08c	no	2:47	yes	1:13	6.00
09a	yes	26:26	yes	4:40	1.54
09b	yes	3:00	yes	1:32	10.59
11a	yes	14:08	yes	4:10	2.62
11b	yes	12:17	yes	7:20	2.45
14c	yes	11:35	yes	3:33	3.17
16a	yes	22:32	yes	9:35	1.49
16b	yes	6:56	yes	4:02	4.38
16c	yes	25:25	yes	2:33	1.72
17a	yes	8:18	yes	2:01	4.65
17b	yes	12:13	yes	1:29	3.50
17c	yes	27:40	yes	1:50	1.63
19a	yes	25:14	yes	14:32	1.21
19b	no	30:32	no	21:18	0.00
19c	yes	3:22	yes	1:46	9.35
Median	**-**	**12:17**	**-**	**02:52**	**2.89**
Std.dev.	**-**	**08:33**	**-**	**04:59**	**2.59**

(b) the control group

ID	Task 1		Task 2		Efficiency [tasks/hour]
	Correct	Time [m:s]	Correct	Time [m:s]	
02c	yes	6:34	yes	8:00	3.30
04a	yes	8:37	yes	16:28	1.91
04b	yes	14:40	yes	8:27	2.08
04c	yes	10:50	yes	4:51	3.06
05a	yes	15:46	yes	4:29	2.37
05b	yes	26:02	yes	2:14	1.70
05c	yes	6:02	yes	2:46	5.45
07b	no	41:23	-	0:00	0.00
07c	no	28:41	no	23:45	0.00
10c	yes	8:48	yes	2:18	4.32
12a	yes	29:15	yes	9:55	1.23
12c	yes	22:35	yes	4:03	1.80
13a	no	32:15	no	21:22	0.00
13b	yes	42:00	yes	4:36	1.03
13c	yes	9:37	yes	2:24	3.99
15a	no	33:37	yes	6:51	0.59
15b	yes	26:15	yes	10:45	1.30
15c	yes	11:33	yes	3:56	3.10
20c	yes	9:27	no	18:55	0.85
Median	**-**	**15:46**	**-**	**04:51**	**1.80**
Std.dev.	**-**	**11:57**	**-**	**07:00**	**1.54**

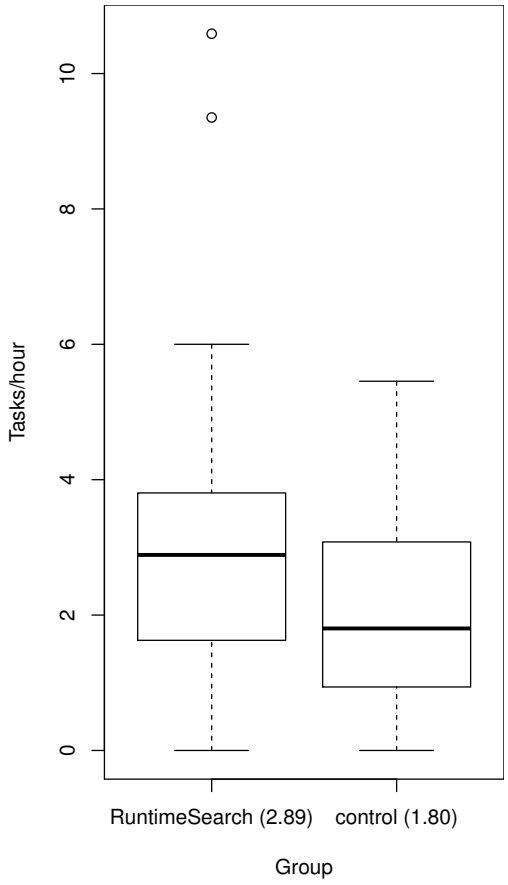


Figure 8.3: Efficiency of tasks for a group using RuntimeSearch vs. standard IDE features

8.4.3 Threats to Validity

Now we will present internal and external validity threats of the controlled experiment.

Internal Validity

During the skimming of the screen-recorded videos, we found a participant which did not follow the instructions in the tutorial correctly. Although all participants were explicitly told to ask the instructor in case of problems, he continued and after a few attempts to use it, he abandoned RuntimeSearch and used only standard search capabilities of the IDE, thus effectively re-assigning himself into the control group. Therefore, we decided to exclude this subject from further processing.

External Validity

Although all participants were students, according to their answers to a question in the form, 75% of them already had work experience in programming. They were all Masters students who voluntarily selected to attend the Metaprogramming course.

The experiment contained only simple tasks, focused on searching the text displayed in a running application. However, this fulfills its purpose – since RuntimeSearch is a utility focused on certain kinds of tasks, there would be no point in selecting problems unrelated to text search. However, the tasks were selected by the RuntimeSearch author, who might be biased. In the future, an experiment with other tasks can be performed to strengthen the external validity.

8.4.4 Conclusion

In this experiment, we tried to determine how a group of participants using RuntimeSearch compares to a group using only standard IDE capabilities when performing search-related maintenance tasks. The median efficiency of the RuntimeSearch group was 60% higher, showing a great potential of this approach.

8.5 Related Work

Related work includes unconventional code search techniques, advanced or automated debuggers, and concept location approaches.

8.5.1 Searching

A tool by Michail [162] builds a database containing the associations between messages shown in GUI widgets and their callbacks (along with other related functions). Then a programmer can search for a message, and associated functions are shown – or vice versa. In contrast to RuntimeSearch, their approach is static and requires separate support for each GUI framework. Furthermore, our tool is not limited to GUI messages.

Holmes and Notkin [95] describe an approach when the "find references" capability of an IDE is filtered using information from dynamic analysis – only methods executed in the given scenario are returned. Compared to RuntimeSearch, they do not capture nor search in the values of expressions.

SPOTTER [44] is a framework for the creation of custom search processors in the Pharo environment. Although such search processors have access also to runtime data, and in theory, a tool similar to RuntimeSearch could be built, no such search processor was described in the article.

GUITA [220] allows a programmer to associate a snapshot of GUI widget in a running application to a method which was last called on the given widget. The disadvantage is its dependence on a particular GUI framework. Furthermore, we offer searching also in non-GUI strings.

8.5.2 Debugging

Automated debuggers like Coca [62] or a scriptable debugger by Marceau et al. [150] perceive an executing program as a set of events. They allow developers to write predicates describing when the program should pause and scripts automatically performing given actions. EXPOSITOR [113] adds time-traveling capabilities to such scripts. Although these and similar systems are more feature-rich than RuntimeSearch, we believe the simplicity of our approach will allow developers to learn it quickly, which should accelerate its possible industrial adoption. Instead of writing scripts and queries, RuntimeSearch offers a familiar user interface of a text search dialog and applies it to runtime.

Whyline [121] allows developers to ask Why and Why Not questions about the observed program behavior. For example, we can click on a drawn square and ask why it has a specific color. The relevant piece of code is shown, along with a sequence of steps how the color was computed. Compared to RuntimeSearch, Whyline operates not only with strings but also with graphical elements in a program. However, it has multiple limitations. First, it is offline, i.e., trace-based, which seriously limits its practical adoption because of a massive amount of data collected [121]. Second, it substitutes, not complements the built-in IDE debugger

which many developers are already used to. Third, it uses identifier names to suggest possible questions, so it relies heavily on good naming.

With object-centric debugging [204], we can place breakpoints on specific object instances at runtime. The breakpoints are then triggered when this instance is manipulated in a given way. For example, we could track a specific string instance in a running program. However, to use object-centric debugging, the program must be already paused and a specific object instance must be manually selected. RuntimeSearch automatically finds such an instance.

8.5.3 Concept Location

In general, concept location (or feature location) is the process of finding where the given concept or feature is implemented in the source code [195]. Feature location approaches take a high-level description of a feature as an input, and they produce a list of code elements contributing to this functionality. In this sense, RuntimeSearch is not a feature location approach. On the other hand, one of the possible uses of our tool is to locate UI terms in the values of expressions in the source code. Therefore, we can consider RuntimeSearch an auxiliary tool useful in the concept location process.

Chen and Rajlich [42] present feature location as a guided search across the program's abstract system dependence graph. Starting from a selected program element, the developer regulates the feature location process by choosing relevant nodes, while the computer controls the traversal. Similar to RuntimeSearch, the technique is interactive and alternates the control between a user and a computer. However, it is purely static and it does not utilize information from dynamic analysis.

Bohnet and Döllner [28] describe a feature location approach where execution traces are collected and visualized in a form of a call graph. Then, the developer inspects the call graph and selects specific points of interest. In a subsequent run, values of parameters or variables are collected at the specific points, and the graph and source code is annotated with them. Compared to RuntimeSearch, their approach requires two separate instrumented executions. In their method, only a small, manually selected portion of variable values is collected. Finally, their tool is not integrated with an IDE and its ready-to-use debugging capabilities.

Some feature location approaches (e.g., SITIR [142]) combine dynamic analysis with information retrieval applied on the source code – this relies on the programmers using proper identifiers. In contrast to them, we utilize the values of string variables and expressions at runtime.

A feature location approach by Anwikar et al. [5] utilizes concrete variable values in the analysis, but only in a very limited way. First, the approach uses partial evaluation, and no real program execution is performed. Second, the approach is limited to "function variables" in legacy software – variables which determine the type of an action to be performed, containing one of a predefined set of values.

FLAT^3 [221] and I3 [17] are advanced user interfaces for feature location. Instead of providing a custom GUI, RuntimeSearch tries to utilize existing IDE capabilities as much as possible.

One of the open problems of dynamic feature location is the selection of appropriate program inputs. Hayashi et al. [91] propose a technique to guide the identification of unexplored scenarios. Since our technique does not aim for a complete feature-code mapping, only for temporary queries, scenario selection is much less of a concern. A developer executes a program according to a scenario (s)he is interested in, and the results are specific to this execution.

8.6 Conclusion and Future Work

We presented RuntimeSearch – a simple search engine which, instead of searching in the static source code, searches in the values of all string expressions of a running program. It applies a well-known "find text" metaphor in the runtime. Our tool integrates into the debugging infrastructure of an IDE and extends its capabilities.

In a case study on a 350 kLOC open source system, we have shown how it can be used to locate an initial investigation point in the program by finding the location displayed in the GUI. Next, we continued to find other occurrences of the same string to find more (possibly) related pieces of code. Our tool is helpful also during debugging, when the programmer hypothesizes about the presence of a certain string in an unknown variable.

During a small benchmark, the overhead of running instrumented code without an active search was negligible after the start-up (class-loading) period. Searching for a string incurred and overhead of about 38%. Optimization and a thorough performance evaluation are planned.

In a controlled experiment with human participants, a group using RuntimeSearch achieved a 60% improvement of efficiency during search-related tasks, compared to a group using standard IDE features.

The currently implemented version of RuntimeSearch is very limited. The first future research area is an improvement of the matching options, e.g. regular expression or fuzzy string matching. Ideally, a programmer would be able to enter a high-level description of a feature and the program would pause when the feature is executed, which could be regarded as "feature breakpoints".

Sometimes we encountered a situation when multiple "Find Next in Runtime" operations in a row pointed to the same statement. This happened mainly in loops which contained the searched string expressions in their body. Adding an option to automatically skip such occurrences is an interesting future work idea. Similarly, we could limit the search granularity by pausing only at one occurrence during a method call.

Storing previous occurrences and thus enabling the "Find Previous" operation could be useful too. Of course, the previous occurrences would be only static source code locations, since storing runtime state would effectively mean building a time-traveling debugger.

Searching only for `String` objects is limiting. Extension to numeric values is a viable option. In many languages, including Java, it is possible to convert an object to its string representation using a method like `toString()`. This could be used to perform a text search in various non-string objects.

Chapter 9

Generating Documentation from Runtime Information

In the previous chapter, we showed how "searching in the runtime" allows for better integration between runtime values and the static source code, which in turn facilitates certain comprehension and maintenance tasks. However, after the relevant piece of code is found, many tasks still remains unsolved. One of them is the comprehension of the behavior of individual methods.

When developers try to comprehend what a particular method or procedure in the source code does, or what are its inputs and outputs, they often turn to documentation. For instance, in Java, the methods can be documented by comments specially formatted according to the Javadoc specification [125]. Similar standards and possibilities exist in other languages.

However, the API (application programming interface) documentation is often incomplete, ambiguous, obsolete, lacking good examples, inconsistent or incorrect [268]. In the worst case, it is not present at all.

Consider the following simplified excerpt from Java 8 API[1] – a method `getAuhority()` in the class `URL` and its documentation:

```
public class URL {
  ...
  /**
    * Gets the authority part of this URL.
    * @return the authority part of this URL
    */
  public String getAuthority() {...}
}
```

For a person who is not a domain expert in the field of Internet protocols, this documentation in certainly not as useful as it should be. To get at least partial understanding of what the method does, he must study the corresponding RFC (Request for Comments) documents, various web tutorials and resources, or browse the rest of the lengthy API documentation.

Now, consider the same method, but with a documentation containing concrete examples of arguments and return values:

[1] https://docs.oracle.com/javase/8/docs/api/java/net/URL.html#getAuthority--

```
public class URL {
  ...
  /**
    * Gets the authority part of this URL.
    * @return the authority part of this URL
    * @examples When called on http://example.com/path?query,
    *     the method returned "example.com".<br>
    *     When called on http://user:password@example.com:80/path,
    *     the method returned "user:password@example.com:80".
    */
  public String getAuthority() {...}
}
```

This kind of documentation should give a developer instant "feeling" of what the method does and help him to understand the source code.

Creating and maintaining documentation manually is time-consuming and error-prone. There exist multiple approaches for automated documentation generation [174]. Many of these approaches work by combining static analysis with natural language processing (NLP) [234, 154] or repository mining [114]. NLP-based and static analysis approaches have an inherent disadvantage: they only present information already available in the static source code in another way. Mining-based methods require large repositories of code using concerned APIs. Furthermore, none of the mentioned approaches provide concrete, literate string representations of arguments, return values and states from runtime: at best, they provide code examples which use the API. While there exist dynamic analysis approaches collecting run-time values of variables [144, 135], they are not oriented toward textual documentation generation.

In this chapter[2], we describe a method documentation approach based on dynamic analysis. The program of interest is executed either manually or using automated tests. During these executions, a tracer saves string representations (obtained by calling a `toString()`-like method) of the methods' arguments, return value, and object states before and after executing the method. For each method, a few sample executions are chosen, and documentation sentences similar to the exhibit shown above are generated. The generated Javadoc documentation is then written into the source files.

To show the feasibility of our approach, named DynamiDoc, a prototype implementation was constructed. It is available at https://github.com/sulir/dynamidoc. We applied DynamiDoc on multiple open-source projects and performed qualitative evaluation: we described its current benefits and drawbacks.

We also performed a small-scale quantitative evaluation of the representation-based approach. The lengths of the generated documentation sentences are analyzed and compared to the lengths of the methods they describe. Furthermore, we look at the proportion of objects in the generated documentation which have a custom (overridden) string representation, which is a prerequisite for this approach to be useful.

9.1 Documentation Approach

Now we will describe the documentation approach in more detail. Our method consists of three consecutive phases: tracing, selection of examples, and documentation generation.

[2]This chapter contains material from our articles: [253], [256].

Listing 9.1: The tracing algorithm executed around each method

```
1  function around(method)
2    // save string representations of arguments and object state
3    arguments ← []
4    for arg in method.args
5      arguments.add(to_string(arg))
6    end for
7    before ← to_string(method.this)
8
9    // run the original method, save return value or thrown exception
10   result ← method(method.args)
11   if result is return value
12     returned ← to_string(result)
13   else if result is thrown exception
14     exception ← to_string(result)
15   end if
16
17   // save object representation again and write record to trace file
18   after ← to_string(method.this)
19   write_record(method, arguments, before, returned, exception, after)
20
21   // proceed as usual (not affecting the program's semantics)
22   return/throw result
23 end function
```

9.1.1 Tracing

First, all methods in the project we want to document are instrumented to enable tracing. Each method's definition is essentially replaced by the code presented in Listing 9.1. In our implementation, we used bytecode-based AspectJ instrumentation; however, the approach is not limited to it.

The target project is then executed as usual. This can range from manual clicking in a graphical user interface of an application to running fully automatized unit tests of a library. Thanks to the mentioned instrumentation, selected information about each method execution is recorded into a trace file. A detailed explanation of the tracing process for one method execution follows.

First, all parameter values are converted to their string representations (lines 3–6 in Listing 9.1). By a string representation, we mean a result of calling the `toString()` method[3] on an argument. For simple numeric and string types, it is straightforward (e.g., the number 7.1 is represented as `7.1`). For more complicated objects, it is possible to override the `toString()` method of a given class to meaningfully represent the object's state. For instance, `Map` objects are represented as `{key1=value1, key2=value2}`.

Each non-static method is called on a specific object (called `this` in Java), which we will call a "target object". We save the target object's string representation before actual method execution (line 7). This should represent the original object state. In our example from from

[3]There are some exceptions – for example, on arrays, we call `Arrays.deepToString(arr)`. Similar methods exist in other languages, such as C# or Ruby.

the introduction of this chapter, a string representation of a URL object constructed using `new URL("http://example.com/path?query");` looks like `http://example.com/path?query`.

We execute the given method and convert the result to a string (lines 10–15). For non-void methods, this is the return value. In case a method throws an exception, we record it and convert to string – even exceptions has their `toString()` method. In the example we are describing, the method returned `example.com`.

After the method completion, we again save the string representation of the target object (`this`, line 18) if the method is non-static. This time, it should represent the state affected by the method execution. Since the `getAuthority()` method does not mutate the state of a URL object, it is the same as before calling the method.

Finally, we write the method location (the file and line number) along with the collected string representations to the trace file. We return the stored return value or throw the captured exception, so the program execution continues as it would do without our tracing code.

To sum up, a trace is a collection of stored executions, where each method execution is a tuple consisting of:

- *method* – the method identifier (file, line number),
- *arguments* – an array of string representations of all argument values,
- *before* – a string representation of the target object state before method execution,
- *return* – a string representation of the return value, if the method is non-void and did not throw and exception,
- *exception* – a thrown exception converted to a string (if it was thrown),
- *after* – a string representation of the target object state after method execution.

9.1.2 Selection of Examples

After tracing finishes, the documentation generator reads the written trace file which contains a list of all method executions. Since one method may have thousands of executions, we need to select a few most suitable ones – executions which will be used as examples for documentation generation. Each execution is assigned a metric representing its suitability to be presented as an example to a programmer. They are then sorted in descending order according to this metric and the first few of them are selected. In the current implementation, we limit the number of examples for each method to 5.

In the current version of DynamiDoc, we use a very simple metric: execution frequency. It is a number of times which the method was executed in the same state with the same arguments and return value (or thrown exception) and resulted in the same final state – considering the stored string representations. This means we consider the most frequent executions the most representative and use them for documentation generation. For instance, if the `getAuthority()` method was called three times on `http://example.com/path?query` producing `"example.com"`, and two times on `http://user:password@example.com:80/path` producing `"user:password@example.com:80"`, these two examples are selected, in the given order.

9.1.3 Documentation Generation

After obtaining a list of a few example executions for each method, a documentation sentence is generated for each such execution. The generation process is template-based.

First, an appropriate sentence template is selected, based on the properties of the method (static vs. non-static, parameter count and return type) and the execution (whether an exception

Table 9.1: A decision table for the documentation sentence templates

Method kind	Parameters	Return type	Exception	State changed	Sentence template
static	0	void	no	no	-
		non-void			The method returned {return}.
		any	yes		The method threw {exception}.
	≥ 1	void	no		The method was called with {arguments}.
		non-void			When {arguments}, the method returned {return}.
		any	yes		When {arguments}, the method threw {exception}.
instance	0	void	no	no	The method was called on {before}.
				yes	When called on {before}, the object changed to {after}.
		non-void		no	When called on {before}, the method returned {return}.
				yes	When called on {before}, the object changed to {after} and the method returned {return}.
		any	yes	no	When called on {before}, the method threw {exception}.
				yes	When called on {before}, the object changed to {after} and the method threw {exception}.
	≥ 1	void	no	no	The method was called on {before} with {arguments}.
				yes	When called on {before} with {arguments}, the object changed to {after}.
		non-void		no	When called on {before} with {arguments}, the method returned {return}.
				yes	When called on {before} with {arguments}, the object changed to {after} and the method returned {return}.
		any	yes	no	When called on {before} with {arguments}, the method threw {exception}.
				yes	When called on {before} with {arguments}, the object changed to {after} and the method threw {exception}.

was thrown, or a string representation of the target object changed by calling the method). The selection is performed using a decision table displayed in Table 9.1. For instance, the `getAuthority()` method is non-static (instance), it has 0 parameters, does not have a void type, its execution did not throw an exception and the URL's string representation is the same before and after calling it (the state did not change from our point of view). Therefore, the sentence template "When called on {before}, the method returned {return}." is selected.

Next, the placeholders (enclosed in braces) in the sentence template are replaced by actual values. The meaning of individual values was described at the end of section 9.1.1. An example of a generated sentence is: "When called on http://example.com/path?query, the

method returned "example.com".” Note that we use past tense since in general, we are not sure the method always behaves the same way – the sentences represent some concrete recorded executions.

Finally, we write the sentences into Javadoc documentation comments of affected methods. Javadoc documentation is structured – it usually contains tags such as `@param` for a description of a parameter and `@see` for a link to a related class or method. We append our new, custom `@examples` tag with the generated documentation sentences to existing documentation. If the method is not yet documented at all, we create a new Javadoc comment for it. When existing examples are present, they are replaced by the new ones. The original source code files are overwritten to include the modified documentation. Using a custom “doclet” [125], the `@examples` tag can be later rendered as the text “Examples:” in the HTML version of the documentation.

9.2 Qualitative Evaluation

First, a qualitative evaluation was performed. We applied our documentation approach on three real-world open source projects and inspected the generated documentation. Our research question is:

RQ9.1 *What are the strengths and weaknesses of DynamiDoc?*

The mentioned open source projects are:

- Apache Commons Lang[4],
- Google Guava[5],
- and Apache FOP[6].

The first two projects are utility libraries aiming to provide core functionality missing in the standard Java API. To obtain data for dynamic analysis, we executed selected unit tests of the libraries. The last project is a Java application reading XML files containing formatting objects (FO) and writing files suitable for printing, such as PDFs. In this case, we executed the application using a sample FO file as its input.

A description of selected kinds of situations we encountered and observations we made follows.

9.2.1 Utility Methods

For simple static methods accepting and returning primitive or string values, our approach generally produces satisfactory results. As one of many examples, we can mention the method `static String unicodeEscaped(char ch)` in the “utility class” `CharUtils` of Apache Commons Lang. The generated sentences are in the form:

```
When ch = 'A', the method returned "\u0041".
```

For many methods, the Commons Lang API documentation already contains source code or pseudo-code examples. Here is an excerpt from the documentation of the aforementioned method:

```
CharUtils.unicodeEscaped('A') = "\u0041"
```

[4] https://commons.apache.org/lang/
[5] https://github.com/google/guava
[6] https://xmlgraphics.apache.org/fop/

Even in cases when a library already contains manually written code examples, DynamiDoc is useful for utility methods on simple types:

- to save time spent writing examples,
- to ensure the documentation is correct and up-to-date.

The latter point is fulfilled when the tool is run automatically, e.g., as a part of a build process. Sufficient unit test coverage is a precondition for both points.

9.2.2 Data Structures

Consider the data structure `HashBasedTable` from Google Guava and its method `size()` implemented in the superclass `StandardTable`. The DynamiDoc-generated documentation includes this sentence:

```
When called on {foo={1=a, 3=c}, bar={1=b}}, the method returned 3.
```

Compare it with a hypothetical manually constructed source-code based example:

```
Table<String, Integer, Character> table = HashBasedTable.create();
table.put("foo", 1, 'a');
table.put("foo", 3, 'c');
table.put("bar", 1, 'b');
System.out.println(table.size()); // prints 3
```

Instead of showing the whole process how we got to the given state, our approach displays only a string representation of the object state (`{foo={1=a, 3=c}, bar={1=b}}`). Such a form is very compact and still contains sufficient information necessary to comprehend the gist of a particular method.

9.2.3 Changing Target Object State

When the class of interest has the method `toString()` meaningfully overwritten, DynamiDoc works properly. For instance, see one of the generated documentation sentences of the method `void FontFamilyProperty.addProperty(Property prop)` in Apache FOP:

```
When called on [sans-serif] with prop = Symbol, the object changed to
[sans-serif, Symbol].
```

Now, let us describe an opposite extreme. In the case of Java, the default implementation of the `toString()` method is not very useful: it displays just the class name and the object's hash code. When the class of interest does not have the `toString()` method overridden, DynamiDoc does not produce documentation of sufficient quality. Take, for example, the generated documentation for the method `void LayoutManagerMapping.initialize()` in the same project:

```
The method was called on
org.apache.fop.layoutmgr.LayoutManagerMapping@260a3a5e.
```

While the method probably changed the state of the object, we cannot see the state before and after calling it. The string representation of the object stayed the same – and not very meaningful.

Although this behavior is a result of an inherent property of our approach, there exists a way how this situation can be improved: to override `toString()` methods for all classes when it can be at least partially useful. Fortunately, many contemporary IDEs support automated

generation of `toString()` source code. Such generated implementations are not always perfect, but certainly better than nothing.

We plan to perform an empirical study assessing what portion of existing classes in open source projects meaningfully override the `toString()` method. This will help us to quantitatively assess the usefulness of DynamiDoc.

9.2.4 Changing Argument State

The current version of DynamiDoc does not track changes of the passed parameter values. For example, the method `static void ArrayUtils.reverse(int[] array)` in Apache Commons Lang modifies the given array in-place, which is not visible in the generated documentation:

```
The method was called with array = [1, 2, 3].
```

Of course, it is possible to compare string representations of all mutable objects passed as arguments before and after execution. We can add such a feature to DynamiDoc in the future.

9.2.5 Operations Affecting External World

Our approach does not recognize the effects of input and output operations. When such an operation is not essential for the method, i.e., it is just a cross-cutting concern like logging, it does not affect the usefulness of DynamiDoc too much. This is, for instance, the case of the method `static int FixedLength.convert(double dvalue, String unit, float res)` in Apache FOP. It converts the given length to millipoints, but also contains code which logs an error when it occurs (e.g., to a console). A sample generated sentence follows:

```
When dvalue = 20.0, unit = "pt" and res = 1.0, the method returned
20000.
```

On the other hand, DynamiDoc is not able to generate any documentation sentence for the method `static void CommandLineOptions.printVersion()`, which prints the version of Apache FOP to standard output.

9.2.6 Methods Doing Too Much

Our approach describes methods in terms of their overall effect. It does not analyze individual actions performed during method execution. Therefore, it is difficult to generate meaningful documentation for methods such as application initializers, event broadcasters or processors. An example is the method `void FObj.processNode(String elementName, Locator locator, Attributes attlist, PropertyList pList)`. An abridged excerpt from the generated documentation follows.

```
The method was called on ...RegionAfter@206be60b[@id=null] with
elementName = "region-after", locator = ...LocatorProxy@292158f8,
attlist = ...AttributesProxy@4674d90
and pList = ...StaticPropertyList@6354dd57.
```

9.2.7 Example Selection

The documentation of some methods is not the best possible one. For instance, the examples generated for the method `BoundType Range.lowerBoundType()` in Google Guava are:

```
When called on (5..+∞), the method returned OPEN.
When called on [4..4], the method returned CLOSED.
When called on [4..4), the method returned CLOSED.
When called on [5..7], the method returned CLOSED.
When called on [5..8), the method returned CLOSED.
```

This selection is not optimal. First, there is only one example of the OPEN bound type – but this is only a cosmetic issue. The second, worse flaw is the absence of a case when the method throws an exception (`IllegalStateException` when the lower bound is $-\infty$).

The method `Range encloseAll(Iterable values)` in the same class has much better documentation, which shows the variety of inputs and outputs (although ordering could be slightly better):

```
When values = [0], the method returned [0..0].
When values = [5, -3], the method returned [-3..5].
When values = [0, null], the method threw
    java.lang.NullPointerException.
When values = [1, 2, 2, 2, 5, -3, 0, -1], the method returned [-3..5].
When values = [], the method threw java.util.NoSuchElementException.
```

Improvement of the example selection metric will be necessary in the future. A possible option is to include the most diverse examples: some short values, some long, plus a few exceptions.

Furthermore, like in any dynamic analysis approach, care must be taken not to include sensitive information like passwords in the generated documentation.

9.3 Quantitative Evaluation

In this section, we will present two small-scale studies of the selected aspects which could affect the usability of the representation-based documentation approach in practice. First, we will look at the length of the generated documentation. Second, a study of the overridden vs. default string representation in the generated documentation is performed.

9.3.1 Documentation Length

In order to be useful, a summary of the method should be as short as possible, while retaining necessary information. Therefore, we decided to perform a quantitative evaluation regarding the length of generated summaries, asking the following research question:

RQ9.2 *What is a typical length of documentation sentences generated by DynamiDoc?*

Method

First, we ran all tests from the package `org.apache.commons.lang3.text` from Apache Commons, while recording execution data. Next, we generated the documentation sentences – at most five for each method. Finally, we computed statistics regarding the length of the generated documentation sentences and the length of the methods' source code.

The following metrics were computed:

- the length of each sentence in characters,

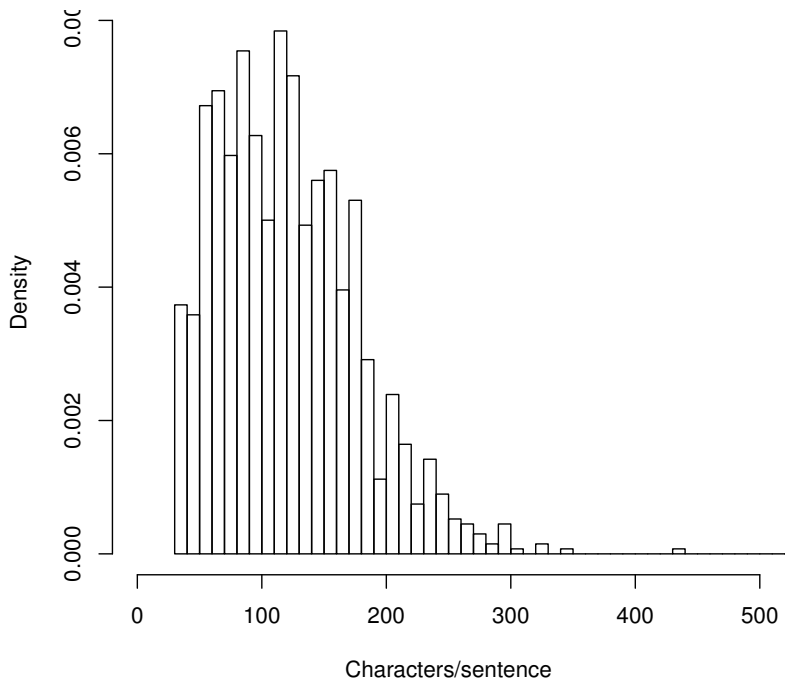


Figure 9.1: Length of a documentation sentence (histogram)

- the "proportional length" – the length of each sentence divided by the length of the corresponding method's source code (in characters).[7]

Results

In Figure 9.1, there is a histogram of lengths of individual documentation sentences. The average (mean) length of a sentence is 137 characters, the median is 117.

In the histogram, we can see the vast majority of documentation sentences is 30-300 characters long.

Figure 9.2 displays a histogram of proportional lengths. On average, one documentation sentence has 0.104 of the length of the method it describes. The median value is 0.096.

It is visible that a vast majority of documentation sentences is shorter than 20-30% of the method it describes.

Threats to Validity

It is disputable whether a length of a sentence is a good measure of comprehensibility. However, it is clear that too long sentences would require more time to read. By ensuring the sentences are short enough, we filled the first requirement of a good summary.

While documentation sentences are written in natural language, the methods' source code is written in a programming language. Therefore, comparing these two lengths might not be fair. However, our aim was only to roughly estimate the reading effort, rather than perform a comprehension study.

[7] Note that the proportional length was chosen only as an arbitrary relative measure to compare the length of the sentence to the length of the method. We do not try to say that one sentence is enough to comprehend the whole method – it is difficult to automatically determine what number of sentences is necessary to comprehend a method without performing a study with human subjects. In general, no number of documentation sentences can precisely describe possibly infinite scenarios encompassed in the method's source code.

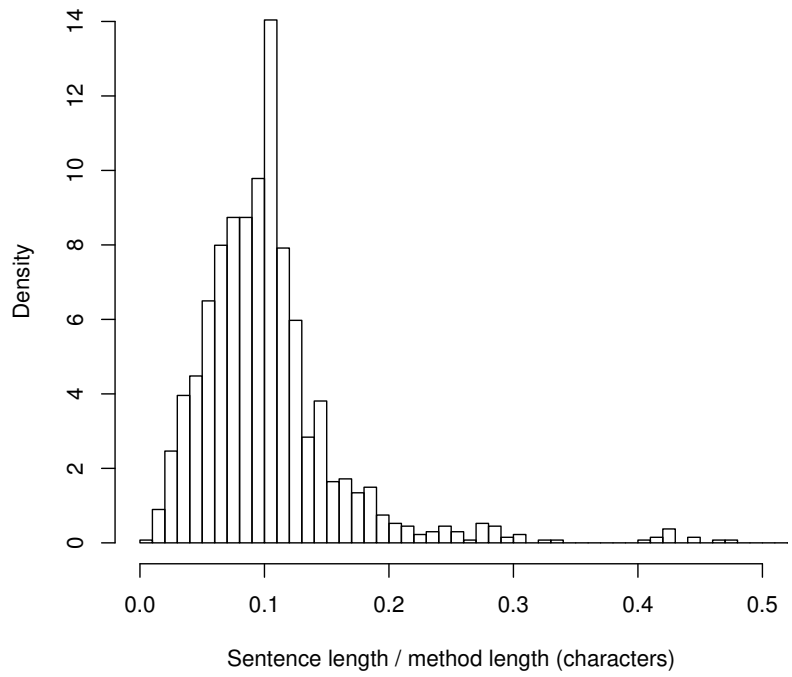


Figure 9.2: Length of a documentation sentence divided by the length of the method it describes (histogram)

Regarding external validity, the presented results were obtained by an analysis of a single package of tests in a specific library. However, we inspected also other packages which provided results not too different from the presented ones.

Conclusions

By measuring the length of sentences and methods, we tried to estimate the reading effort of the documentation and the method source code itself.

According to the results, an average sentence is around 10 times shorter than the method it describes. Of course, reading one example (sentence) is rarely sufficient. Nevertheless, even if the programmer read 3 sentences per method, it would be only 30% of the full method's source code length.

In this study, we did not try to determine whether the generated documentation sentences contain enough information to be useful. This would require performing an experiment with human participants, and it is left for future work.

9.3.2 Overridden String Representation in Documentation

For our documentation to be meaningful, the `toString` method should be overridden in as many classes as possible. In the following small-scale study, we tried to estimate what proportion of objects contained in the generated documentation (target objects, parameters, return values) had this method overridden and what proportion relied on the default implementation. The following research question was asked:

RQ9.3 *What portion of objects contained in DynamiDoc-generated sentences has a custom string representation?*

Table 9.2: The numbers of objects with overridden/default string representations in randomly sampled documentation excerpts

Project	Overridden	Default
Apache Commons Lang	27	3
Google Guava	22	5
Apache FOP	21	2
Total	**70 (87.5%)**	**10 (12.5%)**

Method

We used the documentation of Apache Commons Lang, Google Guava and Apache FOP, generated during the qualitative analysis. From each of these three projects, we randomly selected 10 automatically documented methods. For every method, we selected the first documentation sentence.

Next, we manually inspected these 30 sentences. Each of the sentences contained one or more representations of target objects, parameters or return values. For every string representation, we determined whether it is default (only the class name and the hash code) or overridden (any representation other than default).

Results

The results are presented in Table 9.2. We can conclude that in our study, 87.5% of the documented objects had the `toString` method meaningfully overridden. The other 12.5% relied on the default implementation, which displays only the class name and the object's hash code.

Although the number of objects with an overridden implementation is relatively high, note that not all overridden implementations are actually useful. Some of the string representations we encountered showed too little of the object's properties, so they did not help in the comprehension of a specific method.

9.4 Related Work

In this section, we will present related work with a focus on documentation generation and source code summarization. The related approaches are divided according to the primary analysis type they use – static analysis (sometimes enhanced by repository mining) or dynamic analysis.

9.4.1 Static Analysis and Repository Mining

Sridhara et al. [233, 234] generate natural-language descriptions of methods. The generated sentences are obtained by analyzing the words in the source code of methods; therefore, it does not contain examples of concrete variable values which can be often obtained only at runtime. In [235], they add support for parameter descriptions, again using static analysis only.

McBurney and McMillan [154] summarize also method context – how the method interacts with other ones. While the approach considers source code outside the method being described, they still use only static analysis.

Buse and Weimer [36] construct natural language descriptions of situations when exceptions may be thrown. These descriptions are generated for methods, using static analysis.

Long et al. [145] describe an approach which finds API functions most related to the given C function. If adapted to Java, it could complement DynamiDoc's documentation by adding `@see` tags to Javadoc.

Moreno el al. [166] automatically document classes instead of methods. Their natural language descriptions utilize class stereotypes like "Entity", determined using source code analysis.

The tool eXoaDocs [114] mines large source code repositories to find usages of particular API elements. Source code examples illustrating calls to API methods are then added to generated Javadoc documentation. In section 9.2.2, we described the difference between source code based examples and examples based on string representations of concrete variable values. Furthermore, the main point of source code examples is to show how to use the given API. Therefore, the descriptions of results or outputs are often not present in the examples.

Buse and Weimer [37] synthesize API usage examples. Compared to Kim et al. [114], they do not extract existing examples from code repositories, but use a corpus of existing programs to construct new examples.

The APIMiner platform [164] uses a private source code repository to mine examples. The generated Javadoc documentation contains "Examples" buttons showing short code samples and related elements.

9.4.2 Dynamic Analysis

Hoffman and Strooper [93] present an executable documentation approach. Instead of writing unit tests in separate files, special markers in method comments are used to mark tests. Each test includes the code to be executed and an expected value of the expression, which serves as specification and documentation. Compared to our DynamiDoc, they did not utilize string representations of objects – the expected values are Java source code expressions. Furthermore, in the case of DynamiDoc, the run-time values can be collected from normal program executions, not only from unit tests.

ADABU [52] uses dynamic analysis to mine object behavior models. It constructs state machines describing object states and transitions between them. Compared to our approach which used class-specific string representations, in ADABU, an object state is described using a predicate like "isEmpty()". Furthermore, they do not provide examples of concrete parameter and return values.

Lo and Maoz [144] introduce a combination of scenario-based and value-based specification mining. Using dynamic analysis, they generate live sequence charts in UML (Unified Modeling Language). These charts are enriched with preconditions and postconditions containing string representations of concrete variable values. However, their approach does not focus on documentation of a single method, its inputs and outputs; rather they describe scenarios of interaction of multiple cooperating methods and classes.

Tralfamadore [135] is a system for analysis of large execution traces. It can display the most frequently occurring values of a specific function parameter. However, it does not provide a mapping between parameters and a return value. Furthermore, since it is C-based, it does not present string representations of structured objects.

TestDescriber by Panichella et al. [187] executes automatically generated unit tests and generates natural language sentences describing these tests. First, it differs from DynamiDoc since their approach describes only the unit tests themselves, not the tested program. Second, although it uses dynamic analysis, the only dynamically captured information by TestDescriber

is code coverage – they do not provide any concrete variable values except that present literally in the source code.

FailureDoc [287] observes failing unit test execution. It adds comments above individual lines inside tests, explaining how the line should be changed for the test to pass.

SpyREST [231] generates documentation of REST (representational state transfer) services. The documentation is obtained by running code examples, intercepting the communication, and generating concrete request-response examples. While SpyREST is limited to web applications utilizing the REST architecture, DynamiDoc produces documentation of any Java program.

@tComment [265] is a tool using dynamic analysis to find code-comment inconsistencies. Unlike DynamiDoc, it does not produce new documentation – it only checks existing, manually written documentation for broken rules.

9.5 Conclusion and Future Work

In this chapter, we presented DynamiDoc – a novel approach to automated method documentation and its preliminary implementation. It traces program execution to obtain concrete examples of arguments, return values, and object states before and after calling a method. Thanks to `toString()` methods, the values are converted to strings during the program runtime and only these converted values are stored in a trace. For each method, a few execution examples are selected, and natural-language sentences are generated and integrated into Javadoc comments.

We found DynamiDoc is suitable for utility methods and functions manipulating simple data structures. In complicated methods manipulating many objects, we observed its weaknesses.

We evaluated DynamiDoc using two small-scale studies. We found out that the generated sentences are considerably shorter than the methods they describe. Furthermore, the majority of objects described in the generated sentences have an overridden (non-default) string representation. We perceive this facts as prerequisites for our approach to be useful.

The approach should facilitate program comprehension – but this statement remains to be validated in a controlled experiment with human participants. The documentation generated by DynamiDoc is not intended as a complete replacement for manually written documentation, but it can be a good complement. Compared to natural language processing and simple static analysis, automated documentation approaches utilizing dynamic analysis require more effort, e.g., executing automated tests or running the software manually. An investigation whether this additional effort is worth the benefits provided by the generated documentation should be performed.

The currently presented version has some shortcomings which we would like to mitigate before assessing its effect on code understanding.

First, the string representations of objects are not always ideal. We plan to find out the current prevalence of `toString()` methods in Java (and probably C# and other) classes, determine the "state of the art" in object string representation generation and try to improve it.

Second, we do not have empirical findings what are the attributes of a "useful example". Therefore, we used only a very simple metric of execution frequency to sort the example executions and select the best ones. We would like to investigate, both qualitatively and quantitatively, what examples are considered the best by developers, and modify the selection metric accordingly. Existing knowledge in the area of test prioritization [285] and source code example selection [114] can be adapted and extended.

Finally, we could record also the changes of argument states and interactions with external world (input/output operations) to improve the generated summaries. In cases when summaries of overall effects of methods are insufficient, describing also individual actions inside them using runtime values of variables could help.

Chapter 10

Visual Source Code Augmentation

Although the documentation generated from runtime data, presented in the previous chapter, can provide valuable information to a programmer, one of its disadvantage is its form. It is only a static text attached to a method, not offering any visualization or interaction possibilities. This is understandable, since the majority of programmers write their programs using traditional, textual programming languages [6]. They use a standalone text editor or an integrated development environment (IDE) whose most important component is a textual source code editor. However, upon a closer look at these "plain text" editors, we can discover many visual features: from simple syntax highlighting, through underlining of the code violating code conventions, to information about last version control commits in the left margin. We call such visualizations *source code augmentation*. For a quick illustration of some of them, see Figure 10.1.

There exist surveys about software visualization in general [81] and about its various subfields, such as architecture [225] or awareness visualization [243]. Maletic et al. [149] presented a task-oriented taxonomy of software visualization. Sutherland et al. [260] review ink annotations of digital documents, including program code. In chapter 4, we surveyed the assignment of metadata to different parts of source code, including a discussion about the presentation of source code annotations (inside or outside the code). Nevertheless, according to our knowledge, there is no survey available specifically about visual augmentation of source code editors. In this chapter[1], we present a *first systematic mapping study of source code augmentation approaches and tools*, analyzing research papers published during the last twenty years.

In section 10.1, we define the term source code augmentation and provide a brief overview of historical development in this area. We describe the method used for the systematic review in section 10.2. The result of our survey is a taxonomy with seven dimensions (section 10.3) and a categorized list of the surveyed articles (section 10.4). In chapter 11, we will present an augmentation approach which integrates source code with sample runtime values of variables.

[1]This chapter contains material from our submitted article [245], co-authored with Michaela Bačíková and Sergej Chodarev. They evaluated a part of the primary studies in the selection process (about 25% and 30%, respectively) and helped during the data extraction (about 30% each). They have also minor contributions in the taxonomy construction and reporting of the results (about 5%) and took multiple screenshots according to the instructions of the main author.

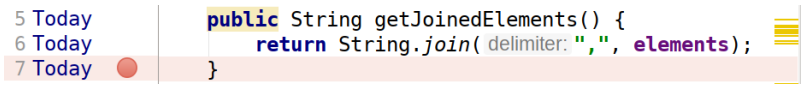


Figure 10.1: A preview of selected augmentation features in an industrial IDE, IntelliJ IDEA. We can see a textual description of the last commit date of each line in the left margin. There is also an icon representing a breakpoint on line 7, which is highlighted also by the red background color. Next, the word "public" is highlighted because of a code inspection warning. The parameter name label "delimiter:" is only a visual augmentation drawn by the IDE in addition to the code itself. Finally, in the right margin, there is a mini-map of warnings (yellow) in the whole file.

10.1 Definition and History

In the research area of virtual reality, an augmented reality system "supplements the real world with virtual objects" [8]. We would like to apply this terminology to the area of source code editing. In our case, the "real world" is represented by the textual source code stored in a file and displayed in a text editor as-is. The "virtual objects" are various line decorations, icons, coloring, images, additional textual labels and other visual overlays.

10.1.1 Definition

Therefore, we define the term *source code augmentation* as an approach which displays additional graphical or textual information directly in plain-text source code. If the code is editable, we can call this also *source code editor augmentation*.

Let us make the definition more precise. First, note that the raw source code displayed in the editor must be the same as the text stored in a file. Projectional editors [273] that use a different editable (visual) and storage representation are, therefore, out of the scope of this article. Furthermore, the augmentation must not remove any part of the displayed source code – this is analogous to the term "diminished reality" in the area of virtual reality[2].

Second, by the expression "directly in the code editor", we mean also the left and right margin of the editor. Note that many IDEs offer many additional views, such as a package explorer or a class navigator. These views are clearly outside the scope of source code augmentation.

Alternative terms to "source code augmentation" are *in situ* software visualization [88] and source code *annotation* [262]. The latter can be easily confused with attribute-oriented programming (e.g., Java annotations). We decided to use the term *augmentation*, particularly for its correspondence with an existing terminology in the field of computer graphics.

10.1.2 Historical View

Probably the most rudimentary source code augmentation feature is *syntax highlighting* (coloring), where each lexical unit is highlighted with a specific color and/or font weight according to its type. One of the first editors supporting real-time source code highlighting was the LEXX editor [50], developed in the eighties.

Another useful augmentation feature is *immediate* syntax error *feedback*, pioneered by the Magpie system [223], which incrementally compiled the program being written. Erroneous code fragments were highlighted directly in the editor.

One of the examples from the 90's is ZStep95 [270]. The expression being currently evaluated was *highlighted* with a border directly in the editor. This augmentation was

[2]While some researchers consider diminished reality a subset of augmented reality [8], we decided to separate these two cases.

interactive: a graphical output produced by this expression was displayed in a floating window located next to such a border.

During the last two decades, the advances in computer performance and incremental analysis algorithms enabled the integration of a multitude of augmentation features into mainstream IDEs.

10.2 Method

Now we will describe the method used for a systematic mapping study we conducted. A systematic mapping study is a form of a systematic literature review with more general research questions, aiming to provide an overview in the given research field.

10.2.1 Research Questions

We were interested in the following two research questions:

RQ10.1 *What source code editor augmentation tools are described in the literature?*

RQ10.2 *How can they be categorized?*

10.2.2 Selection Criteria

Inclusion and exclusion criteria are an important part of every systematic review. In our study, all included articles must fulfill the following criteria:

- It is a journal or a conference article published between years 1998–2017 (inclusive).
- It presents a new tool (or significantly extends an existing one) – e.g., a desktop application, web application or an IDE/editor plugin.
- The tool visually augments the source code editor.
- The article contains a clear textual and graphical description (picture) of the augmentation.

At the same time, articles meeting any of the following criteria were excluded:

- Secondary and tertiary studies, items such as keynotes and editorials, articles for which we could not access a full text.
- Articles describing graphical and semi-graphical languages.
- Projectional editing, hiding and replacing existing code with other information.
- Tools displaying only temporary pop-up windows above code (code completion, tooltips).
- Standard augmentation present in almost all IDEs (syntax highlighting, standard debugging support, syntax error reporting).

10.2.3 Search Strategy

To obtain a list of potentially relevant articles, we used a combination of keyword search in digital libraries, forward and backward references search. For an overview of the article search and selection process, see Figure 10.2.

Keyword Search

First, we performed a keyword search in digital libraries to produce a list of potentially relevant articles. The search query was constructed according to the principles suggested by

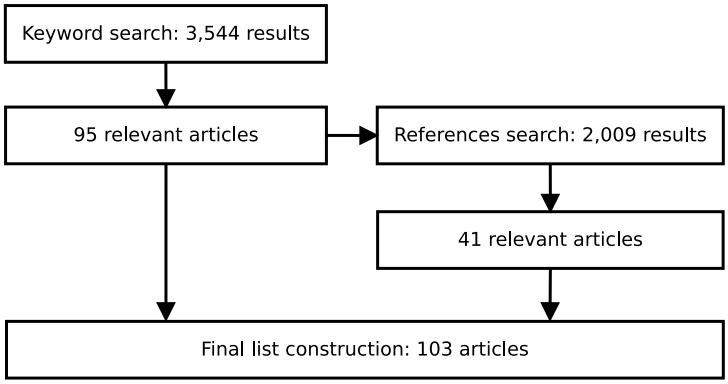


Figure 10.2: The systematic survey search process and article selection overview

Brereton et al. [34]: For each element of the systematic review topic (software maintenance, IDE, source code editor, visualization, augmentation), we found synonyms and related words, connected them by the logical "OR" operator and then connected the subqueries by the "AND" operator. The resulting query is:

```
(software OR program) AND (maintenance OR comprehension OR understanding)
AND (tool OR "development environment")
AND ("source code" OR "source file") AND editor
AND (visualize OR visualization OR display)
AND (augment OR augmentation OR annotate OR annotation)
```

This query was entered into advanced search boxes of four digital libraries: IEEE Xplore[3], ACM Digital Library (DL)[4], ScienceDirect[5] and Springer Link[6]. These four libraries were chosen because of their popularity in software engineering research [286]. In the case of ACM DL, we had to accommodate the query to use a different syntax (without changing its semantics). The search was performed on metadata and full texts. We limited the search to the field of computer science, using the possibilities available in individual libraries, e.g., in IEEE Xplore, we excluded articles not contained in the Computer Society DL. We also filtered the entry types to journal/conference articles with a full text available and the publication year to 1998–2017, written in English.

For all found entries we exported metadata such as titles and abstracts. If the given library did not offer the export of abstracts, we downloaded them from Scopus[7]. After removing duplicate items, this phase produced a list of 3,544 articles.

Then, three of the authors each were given a subset of the articles to evaluate. The researchers manually decided which of these articles fulfill the selection criteria – first based on the title and abstract review and then, if the article seemed potentially relevant, by skimming the full text. The result was a list of 95 selected articles.

References Search

Next, we tried to find relevant articles by examining the references of the 95 articles identified in the previous phase (keyword search). We were interested both in backward references

[3] http://ieeexplore.ieee.org
[4] http://dl.acm.org
[5] http://www.sciencedirect.com
[6] http://link.springer.com
[7] http://www.scopus.com

(literature cited in these articles) and forward references (newer papers citing these articles). To export the lists of backward and forward references along with all necessary metadata, we used Scopus. Similarly to the previous phase, we limited the search to English computer science journal/conference articles published between 1998 and 2017, whenever these options were available.

Two of the articles were not indexed in Scopus and for these, we extracted the backward references manually from the full texts; forward references were exported from the digital libraries of the articles' origin (ACM DL and ScienceDirect).

We merged duplicate entries between forward and backward reference lists. Next, we removed papers already present in the keyword search list. This resulted in 2,009 unique articles.

This set of articles was again manually evaluated by the authors, similarly to the previous phase – first according to the metadata and then a subset of them considering the full texts. The result was a list of 41 more papers. The search was non-recursive since we did not extract the reference lists of these new articles again.

Final List Construction

By combining the results from keyword search and references extraction, we obtained 136 relevant articles. However, 31 of them were describing the same tools as the other ones, only in a different research stage (idea papers, evaluations, etc.). Two articles were classified as "not relevant" after a more careful inspection.

Therefore, the final set consists of 103 relevant articles.

10.2.4 Data Extraction

For each article, we tried to find supplementary material on the web, such as manuals, screenshots, videos and executable tool downloads. We also extracted a tool name from every article; if the tool was unnamed, we devised a suitable pseudonym based on the keywords often used in the paper. In rare cases when one article presented more tools, we selected the most relevant one.

Three of the authors were each given a portion of the tools to classify. Using the full texts of the articles and available supplementary information, they tried to find similar and distinguishing characteristics of the tools. In this first round, each researcher gradually formed his/her own taxonomy (which could be freely inspired by others). A taxonomy consisted of a number of dimensions (e.g., "visualization type") with their attributes (e.g., "icon", "text"). One tool pertained to one or more attributes of every dimension.

Next, two of the classifying authors and the fourth author merged the taxonomies on a personal meeting. The three researchers then re-classified their tools accordingly. Finally, after a discussion, we renamed, split and merged some of the attributes.

The final result is: 1.) a taxonomy with 7 dimensions, each consisting of 2–6 attributes, and 2.) a classification of all analyzed tools according to this taxonomy.

10.2.5 Threats to Validity

Now we will discuss the validity threats of individual systematic review parts.

Since the article selection process was performed by three authors, the inclusion and exclusion criteria could be comprehended differently by individual researchers. However,

we intensively discussed the criteria in the team, along with the examples of individual tools matching and not matching the criteria to lower the variability to a minimum.

The terminology in the area (e.g., the expression "source code augmentation") is not yet standardized, so we could miss multiple relevant articles. Although the keyword search results were not compared to any quasi-gold standard, we constructed it also by considering the results of our previous survey [254] that included multiple articles present in the current review.

During the backward references search in Scopus, we did not include so-called secondary documents, i.e. articles not indexed by Scopus but found in the reference lists. However, this option resulted in many incomplete entries, which would complicate the selection process.

Both the article selection and data extraction was performed by three researchers, each evaluating a non-overlapping portion of articles. Nevertheless, questionable items were discussed until a consensus was reached. Furthermore, after each part, one of the authors checked the list of all relevant articles to ensure the overall quality and to exclude inappropriate items.

Tools are often a neglected aspect of research, so some papers might not describe all available features of the given tool. Note that we do not draw any precise quantitative conclusions about the analyzed articles, such as exact percentages. The main purpose of this chapter is to offer an overview of the field, where each of the analyzed tools acts as an example of particular augmentations and a reference for further information.

10.3 Taxonomy

To answer **RQ2**, we present the resulting taxonomy of source code editor augmentation in Table 10.1. In this section, we will describe the individual dimensions and their attributes, while mentioning representative examples of tools.

10.3.1 Source

The **source** dimension denotes where the data representing the augmentation were originally available before they were visually assigned to a part of source code. This dimension is similar to the one presented in our previous work [254].

Approaches categorized as *code* analyze the source code of a system without executing it, i.e., using static analysis. This can range from simple markers about the presence of an MPI (Message Passing Interface) function call on the given line [277] to a sophisticated calculation of potential deployment costs using static analysis [136]. A very common topic is clone detection (e.g., CloneTracker [61], SimEclipse [269], SourcererCC-I [218]) and warnings about bad smells (JDeodorant [267], Stench Blossom [171]).

Data useful for augmentation can be collected by an execution of the program, using some form of dynamic analysis. These approaches are marked as *runtime*.

The majority of *runtime*-sourced tools collect the data during an execution and display them afterward, when the program has stopped. For instance, IDE sparklines [19] are small graphics displaying the progress of numeric variable values over time. A common application is performance profiling (e.g., In situ profiler [20]). An important problem of this kind of tools is data invalidation: if the source code is modified after the data were collected, the runtime information may not be valid anymore and the program must be run again. This issue is rarely discussed in the reviewed articles. A notable exception is Senseo [212], where the

Table 10.1: The taxonomy of source code editor augmentation

Dimension *and its attributes*	**Description**
Source	Where does the data representing the augmentation come from?
code	Results of static source code analysis.
runtime	Results of the program execution; dynamic program analysis.
human	Manually entered information, previously present only in the human mind.
interaction	Interaction patterns of a single developer in the IDE or a similar tool.
collaboration	Behavior of multiple developers/users and their artifacts (e.g, VCS).
Type	Data of what type does the augmentation directly represent?
boolean	The augmentation can be only present or non-present.
fixed enumeration	One of a set of possible categorical values, the categories are pre-defined by the tool itself.
variable enumeration	One of a set of possible categorical values, the category count is not fixed.
number	Numeric values (numbers, time, etc.).
string	A text string.
object	Another data type (an image, a complex structure, etc.).
Visualization	What does the augmentation look like?
color	A simple area filled with a background color, a color bar or text foreground color.
decoration	Decoration of the code, such as underline, overline or border.
icon	A small rectangular graphical object.
graphics	More complicated graphical representations – charts, arrows, diagrams, photos, etc.
text	An inserted text string (including a number written as a text).
Location	Where is the visual augmentation displayed?
left	To the left of the source code - usually in the left margin.
in code	Directly in the code.
right	It is right-aligned.
Target	To what is the augmentation visually assigned?
line	One line.
line range	Multiple lines, but without distinguishing characters in individual lines.
character range	A number of characters (on one line or multiple lines).
file	The whole file.
Interaction	How can we interact with the augmentation?
popover	A tooltip or a popup with additional information can appear at the place where the augmentation is displayed.
navigate	Performing an action directly on the augmentation navigates us to another code location, window, website, etc.
change	We can edit the displayed information directly by manipulating the augmentation (drag&drop, write, expand, collapse, highlight).
none/unknown	The augmentation does not offer interaction possibilities or they were not mentioned by the authors.
IDE	For what IDE/editor is the augmentation implemented?
existing	An existing IDE/editor is extended.
custom	A tool created by the authors specifically for the purposes described in the article (or prior works).

authors explicitly state the data pertaining to the modified method and its dependencies are always invalidated.

Another type of tools utilizing the *runtime* source, live programming environments such as Impromptu HUD [262] and Gibber [206], display the augmentation in real time as the program is executing.

For tools utilizing the *human* source, a programmer must purposefully enter the data with a sole intention that they will be used by the augmentation. Typical examples are social bookmarks in Pollicino [85], tags in TagSEA [240] or manual concern-to-code mappings in Spotlight [205].

On the other hand, *interaction* data are collected automatically, possibly without the developer even knowing it (although that would be unethical). A typical example is local code change tracking for the purpose of clone detection (CnP [100], CSeR [101]). Some tools also collect data from external applications, for example, HyperSource [87] tracks the user's web browsing activity while editing a code part and then augments that part of the code with potentially relevant browsing history.

As software engineering is not an individual activity, collaboration data are formed naturally as the team communicates and cooperates. These data are utilized by *collaboration* approaches. Some of these tools offer an analysis of existing artifacts. For instance, Rationalizer [31] displays last VCS (version control system) commit times, authors, commit messages and issues related to each source code. The second common category is represented by real-time collaborative source code editors, such as Saros [219], Collabode [79] and IDEOL [54].

Many tools use a combination of multiple methods. For example, Historef [90] automatically collects *interaction* data – local source code edits – in the background and then allows the developer to manually merge, split or reorder these edits (the *human* source) while highlighting the changes with different colors.

10.3.2 Type

The type of data represented directly by the visual augmentation is denoted by the **type** dimension. This can be regarded as an extension of the classical information visualization theory [38], which recognizes nominal, ordered and quantitative data.

The *boolean* type means that the augmentation is either present or not, without distinguishing any visual variations. Two most common boolean augmentations are one-type marker icons and one-color code highlights. For example, Remail [9] displays a marker icon in the left margin on each line which has a related e-mail available. The iXj plugin [30] highlights all code matching the given transformation pattern with a green background. Note that the boolean type is often insufficient to display all necessary information, so it is commonly combined with interaction possibilities (e.g., tooltips) or displayed only for a limited amount of time – after a specific action.

Some approaches use a list of predefined values for displaying different augmentation notations. They can either be firmly stored as a finite list in the tool's data bank, i.e. it's a *fixed enumeration*; or the number of categories can be changed any time by the user or by the tool itself, i.e. *variable enumeration*.

An example of *fixed enumeration* augmentation can be seen in Syde [89] where a red left-margin highlight means a severe collaboration conflict and the yellow color represents moderate conflicts.

A very common application of *variable enumeration* augmentation is an assignment of a different color for every concern or feature in the source code. This approach is used with

variations in at least seven tools: Spotlight [205], CIDE [110], ArchEvol [179], IVCon [217], FeatureIDE [158], FLOrIDA [4] and xLineMapper [288]. Another common approach is to assign a specific color for each developer working on a piece of code, used e.g., in ATCoPE [66]. However, there is a problem with such assignments: The color has only a limited number of easily distinguishable levels [165]. Without additional visualizations and interactions, the utility of such approaches would be questionable if the number of elements in the enumeration (features, people, etc.) raised beyond a reasonable limit.

If the augmentation is represented by a numeric value, e.g. a number of function calls or time, then we use the type *number*. For example, Clepsydra [86] calculates and displays worst-case numbers of processor cycles for individual lines and Theseus [140] displays the number of function calls in a JavaScript file while the application is running in a browser.

The *string* type represents a text string which is not enumerated, i.e., we cannot name all its possible values. For instance, the Live coding IDE [126] shows live values of variables in the source code editor.

Any other displayed information, such as an image or a complex data structure belongs to the *object* type. One such example is SE-Editor [222], which embeds web pages and images directly into the editor.

10.3.3 Visualization

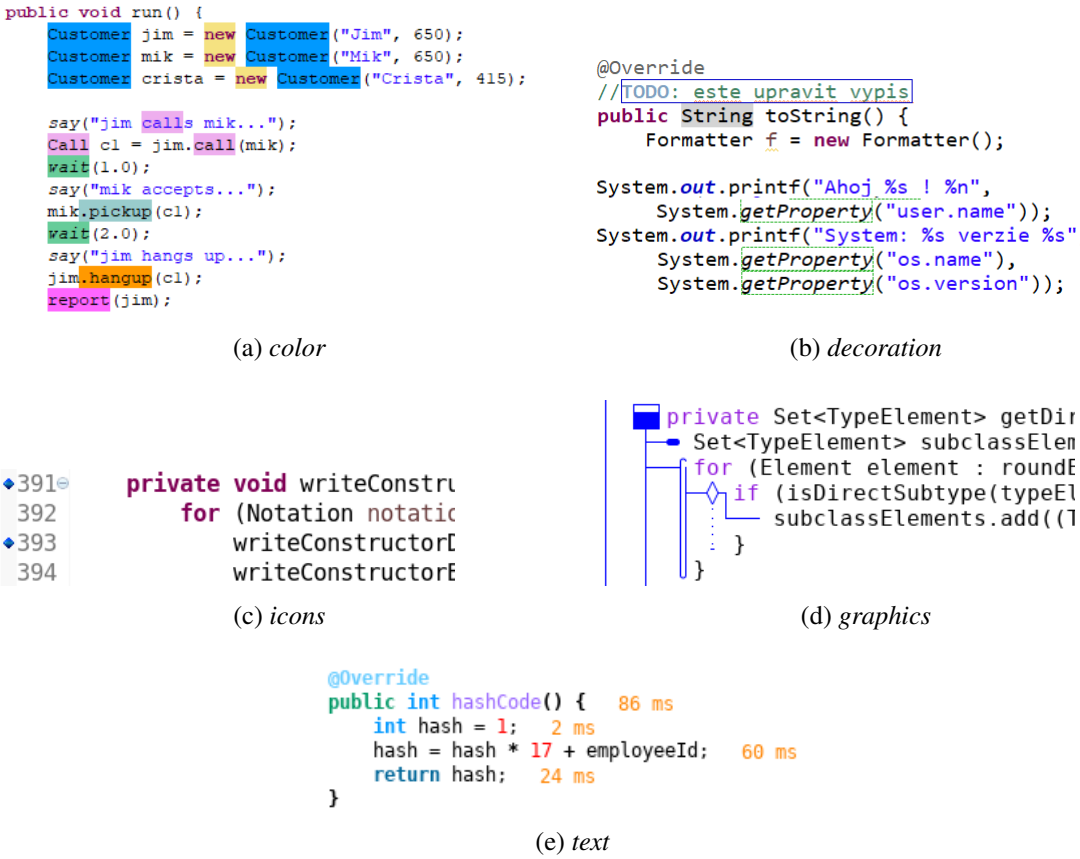
The **visualization** dimension relates to the visual appearance of the augmentation. For an illustration of various visualization kinds, see Figure 10.3.

Color is a very perceivable visual sign and at the same time the most simple one, applicable to both left or right editor bar and the code text itself. Color bars in the margins, text background and foreground color in the editor belong to the *color* category. A tool can utilize one color (Traces view [3]), multiple distinct colors (Jigsaw [49]) or a color spectrum. When utilizing the color spectrum, tools can change the hue (vLens [138]), saturation (CodeMend [210]) or brightness (CnP [100]) according to a numeric value. Note that the code foreground color is often reserved for syntax highlighting. Therefore its use is very limited – we encountered only one such tool, IVCon [217], which does not use syntax highlighting at all.

A less notable highlight is usually represented by a text underline, overline, or border, belonging to the *decoration* category. A dashed border surrounding a code part is used in multiple tools, for example, in eMoose [56] it represents code with associated usage directives, such as threading limitations or reverse TODO references. The XSS marker plugin [14] utilizes a standard Eclipse visualization form, red squiggle underline, but with a different meaning – instead of denoting compilation errors, it warns about cross-site scripting vulnerabilities.

Icons are usually used in editor left bars as small, mostly rectangular and simple graphics. They can visually represent the metaphor of the given tool, e.g., in CodeBasket [24], an egg icon in the left margin represents a “code basket egg”, i.e. a part of the programmers mental model. In other cases they are only attention-catching symbols, e.g., in DSketch [48], red square icons represent lines containing matches of dependency analysis. Occasionally, icons are located directly in the code editor (instead of the left margin) as in the ALVIS Live! tool [98].

More complicated graphical representations such as graphs, charts, arrows, diagrams, photos, etc. belong to the *graphics* group. For example, FluidEdt [185] displays heap graphs in the left margin. I3 [17] offers search similarity and change history views in form of small charts directly in the editor. The Fractional ownership tool [167] displays code authorship history as multi-colored stripes for each code line in the right editor part. A prototype “Error



(a) *color* (b) *decoration*

(c) *icons* (d) *graphics*

(e) *text*

Figure 10.3: Tools with various visualization kinds: (a) Aspect Browser [83] uses text background *colors* to identify different aspects of code according to user-defined patterns; (b) eMoose [56] displays a solid border around TODO comments and adds dashed border *decorations* to code representing associated usage directives or reverse TODO references; (c) Pollicino [85] uses *icons* to represent bookmarks; (d) jGRASP [51] displays control structure diagram *graphics* directly in code; and (e) Clepsydra [86] calculates worst-case numbers of processor cycles and shows them as *textual labels* next to individual code lines.

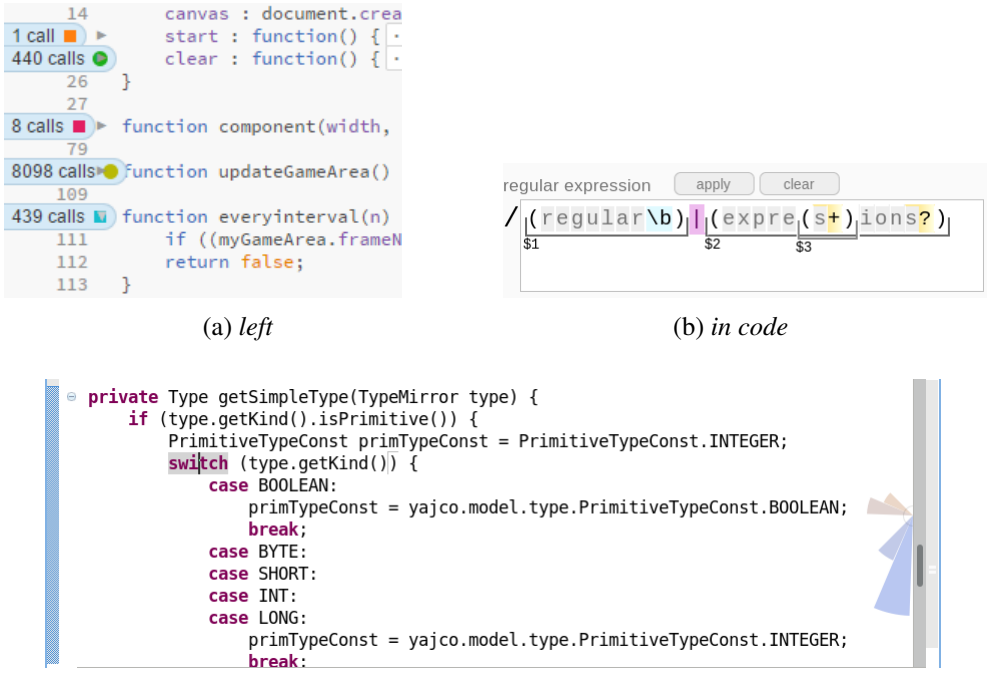
notifications" IDE [13] shows visual descriptions of compiler errors – e.g., in case of a name clash, two relevant code elements are visually connected with a line. DrScheme [71] connects individual uses of a selected language construct in the code by arrows. Finally, the Cares plugin [84] displays photos of code-related people in the editor.

Any piece text added to the editor is considered to be a *text* augmentation. It can be subtle, as in CeDAR [264], where a clone region is labeled with a number in the corner: e.g., "Clone 1". Another example is the "ghost comments" approach of Moonstone [116] non-editable comment-like labels at the end of each line, or the or the Clepsydra tool's [86] line labeling with numbers of processor cycles. On the other hand, Code portals [33] embed significant portions of complementary texts into the code editor, such as relevant source code parts or visualizations of line differences.

10.3.4 Location

The **location** dimension denotes the position of the visual augmentation in the editor, which can either be to the *left* of the source code, displayed directly *in code*, or placed in the *right* editor part, aligned to the right editor margin. See Figure 10.4 for an illustration.

The *left* margin, alternatively called gutter or ruler, traditionally displays line numbers and simple icons (e.g., iMaus [111]). However, it can show also thin color bars (Diver [172]),



(a) *left* (b) *in code*

(c) *right*

Figure 10.4: Tools with various location kinds: (a) Theseus [140] displays the number of function calls in a JavaScript code in the *left* editor part; (b) RegViz [18] highlights regular expressions structure directly *in code*; and (c) Stench Blossom's [171] ambient smell detector that lives on the *right* edge of the program editor.

thicker color fills (HyperSource [87]), "pills" containing text (Theseus [140]) or even a list of callers (Stacksplorer [109]).

Among tools displaying augmentation *in code*, we can distinguish multiple cases. The jGRASP tool [51] displays a control structure diagram in the indentation area of source code containing only tabs or spaces so it does not visually overlap with the text. DrScheme [71] displays arrows directly above code, covering small parts of it; however, this augmentation is only temporary displayed on mouse hover, which makes it acceptable. ARCC [182] draws augmentation behind the code in the form of a background color. SketchLink [10] displays icons directly in the code, but only at the end of lines. Finally, RegViz [18] slightly reflows the text so that the visual annotations do not overlap it.

Right augmentation components are pinned to the editor edges, which means that if the IDE window resizes (changes its width), the components move along with its edges. "Color chips" in CoderChrome [88], displayed in the right part of the editor, are an example of such a visualization. Their meaning can be configured, e.g., to mark starts and ends of code blocks. Because the right margin is usually not the central programmer's focus point, it is an ideal place to implement ambient, non-disturbing augmentations – such as the Stench Blossom [171] code smell visualization. Since this location is near the scrollbar, it is also useful for general overviews: heatmap-colored icons in Senseo [212] referring to available runtime information in the whole file.

10.3.5 Target

Target denotes the code construct to which the augmentation is visually assigned.

Line augmentations are bound to a single line of code, *line range* to multiple consecutive lines. Augmentations displayed in the left margin are practically always assigned to a *line* or a *line range*. For instance, each left-margin crash analysis icon in CSIclipse [183] is related to a

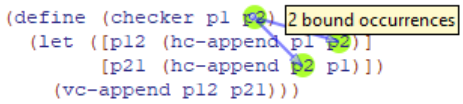


Figure 10.5: DrScheme [71], currently named DrRacket – a tool utilizing the *popover* interaction in form of a tooltip denoting the number of bound occurrences of particular constructs in code.

particular *line* (e.g., "line may have been executed"). *Line range* augmentations are typically denoted as color bars in the left margin (Featureous [184]) or code background color spanning whole lines, up to the right margin (EnergyDebugger [11]).

On the other hand, a *character range* augmentation can mark an arbitrary selection of individual characters. The augmentation can either stay on one line (e.g., a border decoration in ChangeCommander [73]) or potentially span multiple lines (background color highlighting in CodeGraffiti [139]).

Line and *character range* augmentations are often used in tandem: A left margin icon marks the line of interest while the decoration (TexMo [191]) or background color (Gilligan [96]) highlights a more specific source code range in the given line.

If the augmentation relates to the whole code in the currently opened file, then it belongs to the *file* category. Such a visualization can spatially correspond to source code parts (jGRASP [51]), or its placement can be solely a matter of style (Gibber [206]).

10.3.6 Interaction

An icon, text or any other graphics or representation is usually not sufficient to give the user enough information about the augmentation. Tools usually provide some way of displaying more data about the particular case, which is usually initiated by the user interacting with a graphical component representing the augmentation. The **interaction** dimension describes the ways of how it is possible to interact with the displayed augmentation.

The most common interaction (if there is any) is a case when a *popover* (a tooltip or a popup) with additional information displays after a mouse click or a mouse cursor hovering over the augmentation. This can be a simple textual tooltip, as depicted in Figure 10.5 (DrScheme [71]), or a multi-line formatted text with a picture and multiple clickable actions (Cares [84]).

If there is a lot of additional data to display for the augmentation and they would not fit into a popup, clicking or double-clicking on the augmentation component usually fires a *navigation* event, which selects an item in another IDE view/window (traceability links in Morpheus [65]) or displays the information in some external program, e.g. a web browser (a bug report page in the case of Rationalizer [31]). If the augmentation is somehow related to other parts of code, the user is navigated directly to those code parts in the same or another file – e.g., control-flow and data-flow hyperlinks in Flower [229].

Some tools enable direct manipulation of the augmentation itself using mouse or keyboard interaction, which *changes* the displayed information or its appearance. Code portals [33] allow the developer to edit the displayed inline texts. The FixBugs tool [12] offers drag&drop refactoring capabilities directly in the source code editor. In Fluid views [58], the interaction happens in two steps: First, an unobtrusive visual cue changes to an underline decoration upon a mouse hover. After clicking it, the augmentation fully expands and the related code is displayed inside the editor.

If the augmentation component is not interactive in any way or the authors of the tool did not mention any possibilities of interaction in their paper or any supplementary materials, it is

classified as *none/unknown*.

10.3.7 IDE

The last dimension denotes the **IDE** or editor, for which the augmentation is implemented. The solutions are either implemented as plugins into an *existing* IDE or code editor, or they are completely new *custom* environments created by the authors specifically for the purposes described in their article.

Most of the tools we identified were implemented as Eclipse plugins, which is obvious given the popularity and open architecture of this IDE. Some of them were created for other more or less popular IDEs such as Visual Studio (GhostFactor [77]), IntelliJ IDEA (wIDE [169]) or Brackets IDE (Theseus [140]). Examples of custom tools created from scratch are Omnicode [108], Shared-code editor [134], Code portals [33] and Code Bubbles[32].

10.4 Approaches and Tools

As an answer to **RQ1**, we provide a complete list of the surveyed tools in Tables 10.2 and 10.3. The rows are individual tools, sorted alphabetically by the tool name. Each column represents an attribute of the corresponding dimension.

If the tool contains an augmentation fulfilling the given attribute, this is denoted by a filled square (■). Otherwise, an empty square (□) is displayed.

Note that the "IDE" dimension has only one attribute displayed since it has two mutually exclusive attributes. Also note that one tool can contain multiple related or unrelated augmentations (e.g. an icon and color highlighting) or one augmentation fulfilling multiple attributes (e.g., an icon offering both popover and navigation possibilities).

10.5 Conclusion

We presented a systematic mapping study about visual augmentation of source code editors. Using keyword and references search combined with manual filtering, we constructed a list containing 103 relevant tools described in research articles. A taxonomy representing distinct and similar characteristics of these tools was constructed. It contains seven dimensions: source, type, visualization, location, target, interaction and IDE. Each tool was categorized according to this taxonomy, producing a table with article references.

Our main contributions are:

- the definition of the term "source code augmentation",
- a taxonomy of source code editor augmentation features and
- a categorized list of augmentation tools with references.

This article can be a useful resource for researchers aiming to gain an overview of this area or finding a particular example of given augmentations. Furthermore, we provide multiple directions for future work, which we will now describe.

Since we found more than 100 tools augmenting the source code editor area, there is a need to think about filtering possibilities. Even though some augmentations have only a limited lifetime, the IDE may easily become visually cluttered. An interesting question is also how to resolve conflicts when displaying visually overlapping augmentations.

Table 10.2: Augmentation tools

Tool	Source					Type						Visualization					Location			Target				Interaction				IDE
	code	runtime	human	interaction	collaboration	boolean	fixed enum.	variable enum.	number	string	object	color	decoration	icon	graphics	text	left	in code	right	line	line range	character range	file	popover	navigate	change	none/unknown	existing
ALVIS Live! [98]	■	□	□	□	□	■	□	□	□	□	□	□	□	■	□	□	□	■	□	■	□	□	□	□	■	□	□	□
ARCC [182]	■	□	□	□	□	■	□	■	□	□	□	■	■	□	□	□	■	■	□	■	□	■	□	■	□	□	□	■
ArchEvol [179]	□	□	■	□	□	□	□	■	□	□	□	■	■	□	□	□	■	■	□	□	■	□	□	■	□	□	□	■
ASIDE [283]	■	□	■	□	□	■	■	□	□	□	□	■	■	■	□	□	■	■	□	■	□	■	□	■	□	■	□	■
Aspect Browser [83]	■	□	■	□	□	□	□	■	□	□	□	■	□	□	□	□	□	■	□	□	□	■	□	□	■	□	□	■
ATCoPE [66]	□	□	□	□	■	□	□	■	□	□	□	■	□	□	□	□	■	■	□	□	■	□	□	□	□	□	■	■
BeneFactor [76]	■	□	□	■	□	■	□	□	□	□	□	□	□	■	□	□	■	□	□	■	□	□	□	■	□	■	□	■
Cares [84]	□	□	□	□	■	□	■	□	□	□	■	□	□	□	■	□	□	□	■	□	□	□	■	■	■	□	□	■
CeDAR [264]	■	□	□	■	□	■	□	□	□	■	□	■	□	□	□	■	□	■	□	□	■	□	□	□	■	□	□	■
ChangeCommander [73]	■	□	□	□	■	■	□	□	□	□	□	□	■	■	□	□	■	■	□	□	□	■	□	■	□	□	□	■
Cheetah [60]	■	□	□	□	□	□	■	□	□	□	□	■	□	■	□	□	■	■	□	■	□	□	□	■	□	□	□	■
CIDE [110]	□	□	■	□	□	□	□	■	□	□	□	■	□	□	□	□	□	■	□	□	□	■	□	■	□	□	□	■
Clepsydra [86]	■	□	□	□	□	□	□	□	■	□	□	□	□	□	□	■	□	■	□	■	□	□	□	□	□	□	■	■
CloneTracker [61]	■	□	□	□	□	■	□	□	□	□	□	□	□	■	□	□	■	□	□	■	□	□	□	■	□	□	□	■
CnP [100]	□	□	□	■	□	■	□	■	■	□	□	■	□	■	□	□	■	□	□	■	■	□	□	■	□	□	□	■
Code Bubbles [32]	□	□	□	■	□	■	□	□	□	□	□	■	□	□	■	□	■	■	■	■	□	□	□	□	□	■	□	□
Code Orb [146]	□	□	□	□	■	□	□	□	■	□	□	■	□	□	□	□	■	□	□	■	□	□	□	□	■	□	□	■
Code portals [33]	■	■	□	□	■	□	□	■	□	■	□	■	■	□	□	■	□	■	□	■	□	■	□	□	□	■	□	□
CodeBasket [24]	□	□	■	□	□	■	□	□	□	□	□	□	□	■	□	□	■	□	□	■	□	□	□	□	■	□	□	■
CodeGraffiti [139]	□	□	■	□	□	□	□	■	□	□	□	■	□	□	□	□	■	■	□	■	□	■	□	□	■	□	□	■
Codelink [266]	■	□	■	■	□	■	■	□	□	□	□	■	□	□	□	□	□	■	□	□	□	■	□	□	□	■	□	■
CodeMend [210]	■	□	□	□	■	□	■	□	□	□	□	■	□	□	□	□	□	■	□	■	□	■	□	□	■	□	□	□
CoderChrome [88]	■	□	□	□	■	■	■	■	■	□	□	■	■	■	□	□	■	■	■	■	□	■	□	■	■	□	□	■
CodeTalk [237]	□	□	■	□	□	□	■	□	□	□	□	■	□	□	□	□	□	■	□	□	□	□	□	□	□	□	□	■
Collabode [79]	□	□	□	□	■	□	■	□	□	□	□	■	□	■	□	□	■	■	□	□	□	■	□	□	□	□	■	□
CoRED [131]	□	□	□	■	■	■	□	■	□	□	□	■	□	□	□	□	□	■	□	□	□	■	□	■	□	■	□	■
CostHat [136]	■	□	□	□	□	■	□	□	□	□	□	■	□	□	□	□	□	■	□	□	□	■	□	■	□	□	□	■
CSeR [101]	□	□	□	■	□	□	■	□	□	□	□	■	□	□	□	□	□	■	□	□	□	■	□	■	□	□	□	■
CSIclipse [183]	■	■	□	□	□	□	■	□	□	□	□	■	□	■	□	□	■	■	□	■	■	□	□	■	■	□	□	■
CssCoco [80]	■	□	□	□	□	■	□	□	□	□	□	□	■	■	□	□	■	■	□	■	□	□	□	□	□	□	■	■
Debugger Canvas [57]	□	■	□	□	□	□	□	■	□	□	■	■	■	■	■	□	■	□	■	■	□	□	□	■	□	□	□	■
Diver [172]	□	■	□	□	□	■	□	□	□	□	□	■	□	□	□	□	■	□	□	□	■	□	□	□	□	□	■	■
DrScheme [71]	■	□	□	□	□	■	□	□	□	□	■	□	□	□	■	□	□	■	□	□	□	■	□	■	□	■	□	□
DSketch [48]	■	□	□	□	□	□	□	■	□	□	□	■	□	■	□	□	■	■	□	■	□	■	□	□	■	□	□	■
Dynamic text [43]	□	□	■	■	□	□	■	□	□	□	□	■	□	□	□	□	□	■	□	□	□	■	□	□	□	□	■	■
Eclipse PTP [277]	■	□	□	□	□	■	□	□	□	□	□	□	□	■	□	□	■	□	□	■	□	□	□	□	□	□	■	■
EG [63]	□	■	□	□	□	□	■	□	□	□	□	■	□	□	□	□	□	■	□	□	□	■	□	■	□	□	□	■
eMoose [56]	□	□	■	□	□	■	■	□	□	□	□	□	■	■	□	□	■	■	□	■	□	■	□	■	□	□	□	■
EnergyDebugger [11]	□	■	□	□	■	■	□	□	□	□	□	■	□	□	□	□	□	■	□	□	■	□	□	□	□	□	■	■
Error notifications [13]	■	□	□	□	□	□	□	□	□	□	■	□	□	□	■	□	□	■	□	□	□	■	□	□	□	□	■	□
FeatureIDE [158]	■	□	□	□	□	□	□	■	□	□	□	■	□	□	□	□	■	■	■	□	■	□	□	□	□	□	■	■
Featureous [184]	□	■	■	□	□	□	□	□	■	□	□	■	□	□	□	□	■	□	□	□	■	□	□	■	□	□	□	■
FixBugs [12]	■	□	□	□	□	□	■	■	□	□	□	■	■	■	□	□	■	■	□	□	□	■	□	□	□	■	□	■
FLOrIDA [4]	■	□	□	□	□	□	□	■	□	□	□	■	□	□	□	□	□	■	□	■	□	□	□	□	□	□	■	□
Flower [229]	■	□	□	□	□	■	□	□	□	□	□	□	■	□	□	□	□	■	□	□	□	■	□	□	■	□	□	■
Fluid views [58]	■	□	□	□	□	■	□	□	□	■	□	■	■	□	□	■	□	■	□	□	□	■	□	□	□	■	□	■
FluidEdt [185]	□	■	□	□	□	□	□	□	□	□	■	□	□	□	■	□	■	□	□	■	□	□	□	□	□	■	□	□
Fractional owner. [167]	□	□	□	□	■	□	□	□	□	□	■	□	□	□	■	□	□	□	■	■	□	□	□	■	□	□	□	■
GhostFactor [77]	■	□	□	■	□	■	□	□	□	□	□	□	■	□	□	□	□	■	□	□	□	■	□	■	□	□	□	■
Gibber [206]	□	■	□	□	□	□	□	□	□	□	■	□	□	□	■	□	□	■	□	□	□	□	■	□	□	□	■	□
Gilligan [96]	■	□	■	□	□	□	■	□	□	□	□	■	□	■	□	□	■	■	□	■	□	■	□	□	□	□	■	■
Hermion [211]	□	■	□	□	□	■	■	□	□	□	□	□	□	■	□	□	□	■	□	■	□	■	□	■	■	□	□	■

Table 10.3: Augmentation tools (continued)

Tool	Source					Type						Visualization					Location			Target				Interaction				IDE
	code	runtime	human	interaction	collaboration	boolean	fixed enum.	variable enum.	number	string	object	color	decoration	icon	graphics	text	left	in code	right	line	line range	character range	file	popover	navigate	change	none/unknown	existing
Historef [90]	□	□	■	■	□	□	■	□	□	□	□	■	□	□	□	□	□	■	□	□	□	■	□	□	□	□	■	■
HyperSource [87]	■	□	□	■	□	□	■	□	□	□	□	□	■	■	□	□	■	□	□	■	□	□	□	■	□	□	□	■
I3 [17]	■	□	□	□	■	□	■	□	□	□	■	■	□	□	■	□	□	■	□	□	□	■	□	■	□	□	□	■
IDE sparklines [19]	□	■	□	□	□	□	□	□	■	□	□	□	□	□	■	□	□	■	□	■	□	□	□	■	□	□	□	■
IDEOL [54]	□	□	□	□	■	□	□	■	□	□	□	■	□	□	□	■	□	■	■	□	□	■	□	□	□	□	■	□
iMaus [111]	■	□	□	□	□	■	□	□	□	□	□	□	■	■	□	□	■	■	□	■	□	■	□	□	■	□	□	■
Impromptu HUD [262]	□	■	□	□	□	□	□	□	■	□	□	□	□	□	■	■	□	■	□	■	□	□	■	□	□	□	■	■
In situ profiler [20]	□	■	□	□	□	□	■	■	■	□	□	□	□	□	■	□	□	■	□	■	□	□	□	■	□	□	□	■
inCode [75]	■	□	□	□	□	■	□	□	□	□	□	□	□	■	□	□	■	□	□	■	□	□	□	■	□	□	□	■
IVCon [217]	□	□	■	□	□	□	□	■	□	□	□	■	□	□	□	□	□	■	□	□	□	■	□	■	□	□	□	□
iXj [30]	■	□	□	□	□	■	□	□	□	□	□	■	□	□	□	□	■	■	□	□	□	■	□	□	■	□	□	■
Jazz Band [99]	□	□	□	□	■	□	■	□	□	□	□	■	□	■	□	□	■	□	□	□	■	□	□	■	■	□	□	■
JDeodorant [267]	■	□	□	□	□	■	□	□	□	□	□	■	□	□	□	□	□	■	□	□	□	■	□	□	□	□	■	■
jGRASP [51]	■	□	□	□	□	□	□	□	□	□	■	□	□	□	■	□	□	■	□	□	□	□	■	□	□	■	□	□
Jigsaw [49]	■	□	□	■	□	□	■	□	□	□	□	■	□	□	□	□	□	■	□	□	□	■	□	□	□	□	■	■
JScoper [70]	■	□	□	□	□	■	□	□	□	□	□	□	□	■	□	□	■	□	□	■	□	□	□	□	■	□	□	■
Live coding IDE [126]	□	■	□	□	□	□	□	□	■	■	□	□	□	□	□	■	□	□	■	■	□	□	□	□	□	■	□	■
Moonstone [116]	■	□	□	□	□	■	■	□	□	■	□	■	■	■	□	■	■	■	□	■	■	■	□	■	□	□	□	■
Morpheus [65]	■	□	□	■	□	□	■	□	□	■	□	■	■	■	□	■	■	■	□	■	■	■	□	■	■	■	□	■
NLP Eclipse [281]	■	□	□	□	□	■	□	□	□	□	□	□	□	■	□	□	■	□	□	□	□	■	□	■	□	□	□	■
Omnicode [108]	■	□	■	■	□	■	□	□	□	□	□	■	□	□	□	□	■	■	□	■	□	■	□	■	■	□	□	□
PerformanceHat [46]	□	■	□	□	□	■	□	□	□	□	□	■	□	■	□	□	■	■	□	□	□	■	□	■	□	□	□	■
Pollicino [85]	□	□	■	□	□	■	□	□	□	□	□	■	□	■	□	□	■	■	□	■	□	□	□	□	■	□	□	■
PyLighter [29]	□	■	□	□	□	□	□	□	■	□	□	■	□	□	□	□	□	■	□	■	□	□	□	□	□	□	■	□
Rationalizer [31]	□	□	□	□	■	□	■	■	■	■	□	■	□	■	□	■	□	□	■	■	□	□	□	■	■	□	□	■
Refactoring Ann. [170]	■	□	□	□	□	□	□	□	□	□	■	□	□	□	■	□	□	■	□	□	□	■	□	□	□	□	■	■
RegViz [18]	■	□	□	□	□	□	■	□	■	□	□	■	■	□	□	■	□	■	□	□	□	■	□	□	□	■	□	□
Remail [9]	□	□	□	□	■	■	□	□	□	□	□	□	□	■	□	□	■	□	□	■	□	□	□	■	□	□	□	■
Saros [219]	□	□	□	□	■	□	□	■	□	□	□	■	□	□	□	□	■	■	□	□	■	■	□	□	■	□	□	■
SE-Editor [222]	□	□	■	□	□	□	□	□	□	□	■	□	□	□	■	□	□	■	□	□	■	□	□	□	□	□	■	■
Senseo [212]	□	■	□	□	□	□	■	□	□	□	□	■	□	□	■	□	■	□	■	■	□	□	□	□	□	□	■	■
Shared code [134]	■	□	■	■	■	■	□	□	□	■	□	□	■	□	□	■	□	□	□	□	■	□	□	□	□	□	■	□
SimEclipse [269]	■	□	□	□	□	■	□	□	□	□	□	■	□	■	□	□	■	■	□	■	■	□	□	■	■	■	□	■
SketchLink [10]	■	□	□	□	□	■	□	□	□	□	□	□	□	■	□	□	□	■	□	■	□	□	□	■	■	■	□	■
Soot-Eclipse [137]	■	□	□	□	□	■	□	□	□	□	□	■	□	■	□	□	■	■	□	■	□	■	□	■	■	□	□	■
Soul [55]	■	□	□	□	□	■	□	□	□	□	□	□	□	■	□	□	■	□	□	■	□	□	□	■	□	□	□	■
SourcererCC-I [218]	■	□	□	□	□	□	■	□	□	□	□	□	□	■	□	□	■	□	□	■	□	□	□	□	□	□	■	■
Spotlight [205]	□	□	■	□	□	□	□	■	□	□	□	■	■	□	□	□	■	■	□	□	■	□	□	□	□	□	■	■
Stacksplorer [109]	■	□	□	□	□	■	■	■	□	■	□	■	■	■	□	■	■	■	■	□	■	■	□	□	■	■	□	■
Stench Blossom [171]	■	□	□	□	□	□	□	□	■	□	□	□	□	□	■	□	□	□	■	□	□	□	■	■	□	□	□	■
Syde [89]	□	□	□	□	■	□	■	□	□	□	□	■	□	□	□	□	■	□	□	□	■	□	□	■	□	□	□	■
TagSEA [240]	□	□	■	□	□	■	□	□	□	□	□	□	□	■	□	□	■	□	□	■	□	□	□	□	■	□	□	■
TexMo [191]	■	□	□	□	□	□	■	□	□	□	□	■	□	■	□	□	■	■	□	■	□	■	□	■	□	□	□	■
Theseus [140]	□	■	□	□	□	□	■	□	■	■	□	□	□	■	□	■	■	□	□	■	□	■	□	□	■	■	□	■
Traces view [3]	□	■	□	□	□	■	□	□	□	□	□	■	□	□	□	□	□	■	□	■	□	□	□	□	□	□	■	■
vLens [138]	■	■	□	□	□	□	□	□	■	□	□	■	□	□	□	□	□	□	□	□	■	□	□	□	□	□	■	■
wIDE [169]	■	□	□	□	□	□	■	□	□	□	□	■	□	□	□	□	□	■	□	□	□	■	□	□	□	□	■	■
xLineMapper [288]	□	□	■	□	□	■	□	■	□	□	□	■	□	■	□	□	■	■	□	■	□	■	□	□	□	□	■	■
XSS marker [14]	■	□	□	□	□	■	□	□	□	□	□	□	■	■	□	□	■	■	□	■	□	■	□	■	□	□	□	■
YinYang [155]	□	■	□	□	□	□	□	□	□	■	□	□	□	□	□	■	□	■	□	■	□	□	□	□	■	□	□	□
Zelda [198]	■	□	■	□	□	■	□	□	□	□	□	□	□	■	□	□	■	□	□	■	□	□	□	□	□	□	■	■

The underlying data or source code often change after the augmentation is initially displayed. Invalidation and possible recalculation of augmentations is another important issue, seldom discussed in the reviewed articles.

Although many articles include empirical evaluation, the visual code augmentation itself is rarely the main object of the studies. Comparisons of the usability of separate views, in-code augmentations and their variations are definitely welcome.

Chapter 11

Augmenting Code Lines with Runtime Values

In chapter 9, we presented an automated documentation generation approach using runtime values of variables. However, this documentation was limited to the description of whole method definitions, and the result was only a static document. In this chapter[1], we will discuss how to document individual source code lines in symbiosis with an IDE, by designing an augmentation approach similar to the ones described in chapter 10.

Consider the following excerpt from the method createNumber in class NumberUtils of Apache Commons Lang[2]:

```
exp = str.substring(expPos + 1, str.length() - 1);
```

While the variable names give the programmer a hint about their meaning, it requires non-negligible mental effort to construct a mental model of the presented line. Now, consider the same line augmented with concrete values of variables:

```
exp = str.substring(expPos + 1, str.length() - 1);
str: "1.1E-700F"  expPos: 3  exp: "-700"
```

The programmer can now intuitively understand that the line extracts the exponent from a string containing a decimal number in the scientific notation. Furthermore, we can see that the removal of the last character is necessary because it contains the suffix "F" (float).

To perform an investigation of dynamic program properties, such as concrete values of variables, developers often use debuggers [129]. For instance, newer versions of IntelliJ IDEA already display an augmentation similar to the one presented above. However, the use of a debugger requires additional effort from the programmer. First, we must manually choose appropriate breakpoint locations (although an approach by Steinert et al. [236] can simplify this). Next, the program must be executed and the debugger must be guided using the stepping operations to progressively reveal variable values. At one moment, only one state of the program is visible and the programmer cannot get an overview of multiple states. Finally, after these laborious tasks are performed, the obtained dynamic information disappears as soon as the given lines become out of scope or the debugging session is closed.

[1] This chapter contains material from our accepted article [257].

[2] https://commons.apache.org/proper/commons-lang/

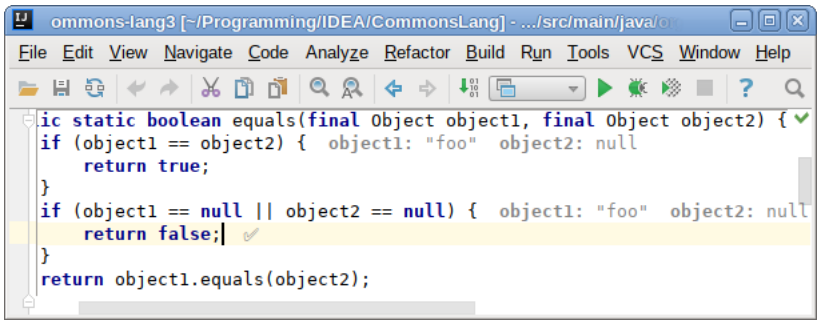


Figure 11.1: Sample values augmentation in RuntimeSamp

To tackle this problem, tracing tools and time-traveling debuggers record the program execution and then allow the programmer to explore any part of the program in any recorded execution state. Nevertheless, these tools still require the user to manually navigate a large trace to select the relevant part. The runtime information and the static source code view are separated, requiring the user to switch between two disconnected views.

There exist several approaches trying to integrate runtime information directly into the source code editor, but the kinds of information they display is often limited. Senseo [212] displays information such as lists of callers, callees and dynamic argument types. Tralfamadore [135] is limited to argument and return values, IDE sparklines [19] to numeric variable types.

Other approaches – Tangible code [159] and Pathfinder [189] – display variable values inline, but they are trace-focused: the programmer browses a trace and manually selects a point in the history which should be displayed.

Although DynamiDoc [253] collects sample values at runtime, the results are only generated documentation sentences, not an interactive approach. Krämer et al. [126] integrate variable values with source code in a live programming environment. Such environments suffer from performance and scalability problems though.

In this article, we introduce an approach to present sample values of variables to a developer in an ambient and unobtrusive manner. Executions of tests or the application itself by a programmer are partially recorded. Then, at the end of each line, a noneditable comment-like label is displayed in the IDE. Each such label shows the values of the local and instance variables read or written on the given line, during one sample execution of this line.

A tool embodying this idea, RuntimeSamp, is implemented as a plugin for IntelliJ IDEA combined with a load-time Java bytecode instrumentation agent. For an illustration, see Figure 11.1. The source code of RuntimeSamp is available online[3].

While the approach itself seems simple, a number of interesting questions arose during the process of designing it. In this chapter, we devise solutions to the encountered problems – some naive and some more elaborate – and try to foster future research in this area.

11.1 Object Representation

Since the variable values are to be displayed in the text editor, they should be ideally represented on one line, using only a small amount of space. For primitive values, such as integers or floats, this is simple – we use their traditional mathematical notation. The situation is

[3] https://github.com/sulir/runtimesamp

more complicated for objects composed of many properties. Therefore, we formulate our first question:

RQ11.1 *How to present complicated objects succinctly on a small space?*

If we consider only purely textual representations, the most straightforward solution is to call the "to string" method (e.g., toString in Java) on a particular object. This is also the solution we used in RuntimeSamp (with a few exceptions, such as manually generating the array representation or shortening too long descriptions). This approach has its disadvantages, though. In many languages, generation of the string representation must be manually programmed for each class. Because it is useful almost exclusively for debugging purposes, developers often omit it. This leaves us with a default implementation similar to "`MyClass@4a54c0de`", which is obviously not useful.

It is trivial to recursively traverse all member variables of the given object down to primitive values or to the given depth and automatically produce a string in the form "{member$_1$: {sub-member$_1$: value$_1$, sub-member$_2$: value$_2$, ...}, ...}." Nevertheless, such a string can be too long for the majority of objects. An interesting future research idea is to filter the list of member variables of each object to include only that relevant for comprehension. For example, we can exclude members representing purely implementation details; or include only frequently read or recently changed fields. Finding what constitutes a given member relevant and useful for comprehension remains an open question.

Regarding graphical and semi-graphical representations, a universal way would be to display a fully collapsed tree of all properties, represented by a clickable plus-sign (⊞) directly in the editor. The developer could expand it on demand and show the values of interest. On the other hand, this approach could affect the ambient nature of RuntimeSamp since it would require manual actions from the developer.

While there exist methods of graphical representation generation (e.g., the Moldable Inspector [45]), they suffer from similar shortcomings as the classical toString() – the implementation is manual, optional and not universal.

11.2 Recording Moment Selection

One variable can be displayed multiple times on each line, every time holding a different value. Thus the following question arises:

RQ11.2 *When exactly should we capture the values of variables?*

The following options are available:

- after reading/writing a variable for the first time on this line,
- after the last reading/writing of it,
- at the end of the line.

Note the value will be displayed at the end of the line. To match its visual position with its meaning, we selected the third option.

Our approach works with physical lines, so even if multiple statements are present on the same line, each variable is displayed only once in the augmentation.

An alternative would be to display a value directly next to each variable in the code (as in Tangible Code [159]). While this would improve spatial immediacy by lowering the distance between the variable occurrence and its value, the source code editor could easily become cluttered, especially in the case of more verbose object representations.

11.3 Iteration Selection

The most important problem regarding our approach stems from the fact that program execution is not sequential. Program constructs such as loops cause one source code line to be executed multiple times, each time in a possibly different context. Therefore, displaying an arbitrary sample value can cause confusion. Consider the following augmented source code excerpt, separately counting the sums of even and odd numbers:

```
1 for (int n : nums)  nums: {1, 2}  n: 1
2   if (n % 2 == 0)  n: 1
3     even += n;  n: 2  even: 2
4   else
5     odd += n;  n: 1  odd: 1
```

Here, each line is augmented with the values recorded during the first time the particular line was executed. Up to the second line, everything seems to be consistent. However, at line 3, the value of `n` suddenly becomes 2, even though on the previous line it was 1. Clearly, the sample values at lines 1, 2 and 5 were recorded during the first loop iteration, while the augmentation at line 3 comes from the second one. This happened because the third line was not executed at all in the first iteration.

It is obvious that we can display only one iteration at a time. This leaves us with a question:

RQ11.3 *How to decide which iteration to display?*

The most elementary solution would be to insert a combo box at each line containing a looping construct (in our case, `for` at line 1) to enable a selection of a particular loop iteration from a list. A similar solution was already used, e.g., in compacted sequence diagrams [173]. However, this would require a nontrivial manual action from the programmer. Imagine selecting a relevant iteration from a list containing hundreds of items, without knowing any information about them besides the iteration numbers.

First of all, we need to provide a convenient way to select an iteration in which a particular line of interest was executed (an iteration which covers it). In RuntimeSamp, we set the "line of interest" to the line where the text cursor (caret) is currently located. For example, if the caret is on line 3, we show the second iteration:

```
1 for (int n : nums)  nums: {1, 2}  n: 2
2   if (n % 2 == 0)  n: 2
3     even += n;  n: 2  even: 2
4   else
5     odd += n;
```

On the other hand, if the cursor is on line 5, the first iteration is displayed:

```
1 for (int n : nums)  nums: {1, 2}  n: 1
2   if (n % 2 == 0)  n: 1
3     even += n;
4   else
5     odd += n;  n: 1  odd: 1
```

This way, we utilize the text cursor as an implicit pointer to the programmer's current focus point.

Of course, many lines can be covered by more than one iteration. The choice of the most relevant one is most probably task-dependent and remains an open research question.

Note that in Figure 11.1, the line with the `return false;` statement is augmented with a check mark (✓). This is used for lines which were executed during the given iteration but do not contain any variables.

11.4 Iteration Detection

Although in the previous text, we intuitively worked with the notion of an "iteration", the situation is not always that simple. For instance, a loop can contain a call to a method defined in another file. An iteration statement is thus spatially disconnected from the place of the manifestation of its effect. Furthermore, such a loop can be located in a library or a system file which is not instrumented and recorded. Finally, many languages contain a variety of looping constructs which are difficult to recognize in the compiled bytecode. This leaves us with a question:

RQ11.4 *How to define an iteration in a way which is easy to detect and present?*

In RuntimeSamp, we used the notion of forward executions. As long as the program progresses forward without jumping back, we consider this one iteration, which we call a "pass". Since in RuntimeSamp, the recording and augmentation are line-oriented, we consider only jumps to another line, not the same one. Execution of another method starts a new pass; however, as soon as this method returns control to the caller, the original pass continues. Our approach is simple to implement using a local variable called `passId`, inserted into each method by instrumentation. This variable is assigned a new value on every method start and backward jump – from a global variable `passCounter`, incremented on each read.

For an illustration, see the following code:

```
1 void caller() {
2   int i = 1;
3   callee();
4   i = 2;
5 }
6 void callee() { doSomething(); }
```

Lines 2, 3, and 4 are a part of the pass with ID 1; line 6 pertains to pass ID 2.

11.5 Collection Efficiency

While the collection of all variable values in the whole program during the whole execution gives precise results, it is clearly not practical because of high time overhead and storage requirements. The following question arises:

RQ11.5 *How to collect enough data for sample values presentation while keeping the overhead reasonable?*

In RuntimeSamp, we aim to present sample values at the end of each line. Therefore, it is convenient to collect values at least once for every line being executed. In the current implementation, we use a simple logic. In a global array called `hits`, we keep the number of times each particular line of the program (identified by its `lineId`) was executed. As soon as a hard-coded "hits per line" limit is reached, we stop data collection for the given line.

When combined with the iteration detection approach described in the previous section, the result is depicted in Algorithm 11.1. Note that to prevent the likely overflow of `passCounter`, we first reset the `passId` to 0 and obtain a new value from `passCounter` only when the data will be actually collected.

Clearly, the presented algorithm, which records the first n executions of each line, is not ideal. Consider the source code from section 11.3 was executed with `nums` containing a thousand even items and one odd item as the last element. We would need to record the first iteration, then skip recording of 999 iterations and record the last one. In case the array was dynamically generated, a nontrivial prediction using dynamic analysis would need to be applied. Otherwise, we need to either capture all executions and sacrifice performance, or end up with incompletely recorded passes. In the example from section 11.3, with *HITS_PER_LINE* in our algorithm set to 1, two passes are recorded: the first pass containing lines 1, 2, 3; and the second one containing only line 5.

We performed a preliminary performance evaluation of our approach. In Table 11.1, we present time overhead of the instrumented runs when compared to plain, non-instrumented ones (zero means no overhead). The benchmarks come from DaCapo [27], version 9.12-MR1. Auxiliary libraries and system classes were not instrumented. Beware our implementation is naive and can clearly be improved.

We may also consider collecting data only for a limited part of the program or an execution period. For example, the values of relevant variables could augment the source code when the program crashed due to an uncaught exception.

11.6 Filtering of Variables

A lot of data can visually clutter the editor and cause information overload. Thus our next question is:

RQ11.6 *Which variable values should be displayed and which not?*

Currently, we show all variables except same-named variables on the same line – e.g., in the expression `this.var = var`, we show only one `var`. In the future, it is possible to devise other filtering patterns to achieve the optimal ratio between readability and completeness.

Algorithm 11.1: Variable-recording instrumentation

On each method start do:
passId ← 0;

Each time when the executing line is about to change from *currentLine* to *nextLine* do:
if *hits[lineId]* < *HITS_PER_LINE* **then**
  hits[*lineId*]++;
  if *passId* = *0* **then**
    passId ← *passCounter*++;
  end
  `SaveToDb`(*fileName, currentLine, passId*);
  for *each variable read/written on currentLine* **do**
    `SaveToDb`(*passId, name, value.toString()*)
  end
end
if *nextLine* < *currentLine* **then**
  passId ← 0;
end

Table 11.1: Time overhead of instrumented runs

Hits per line	Overhead for benchmark			
	avrora	**fop**	**h2**	**xalan**
1	0.78	2.13	1.84	0.88
2	0.89	2.37	2.07	0.93
3	0.98	2.63	2.23	1.19

11.7 Data Invalidation

Our final question is:

RQ11.7 *When and how to invalidate the collected variable values?*

Currently, as a preliminary form of invalidation, we delete all data on any document edit. Of course, a more reasonable solution is to delete only data pertinent to statements dependent on the made changes. However, we can go even further: The augmentation of values which could be recomputed in a reasonable time (and without requiring additional input) should be updated in the editor, which would enable partial liveness.

11.8 Conclusion

We regard the source code view as the primary view of the programmer's interest. Therefore, we aim to deliver a "feeling of runtime" to the static source code by the means of augmenting source code editor lines with examples of concrete variable values. Our final goal is to make runtime information in the editor:

- ambient – using implicit interaction information instead of requiring manual actions,
- unobtrusive – not requiring a change of the developer's tools or workflow,
- relevant – thus aiming to leverage program comprehension.

Throughout the chapter, we discussed several questions related to these properties. Still, many questions remain to be answered in future experiments. Particularly, an evaluation with human participants (industrial developers, students) should be performed, both to quantitatively confirm the utility of RuntimeSamp and to provide qualitative insights about the decisions made when designing it.

Chapter 12

Conclusion

To conclude this dissertation, we will summarize the main results, present a list of the most important contributions and outline future work directions.

12.1 Summary

Our general goal is to improve program comprehension. Therefore, we first examined the research area and its trends. According to the study performed on the bibliographic data of research articles, feature location and open source systems were trending. On the other hand, program slicing and the study of legacy systems were gradually falling.

Being able to build a software system is a prerequisite for program comprehension activities, particularly for approaches connecting runtime information with the source code. In our study, an attempt to build about 7,000 Java projects from GitHub ended with a failure in 38% of cases. The most frequent errors were dependency-related. The probability of a build failure is associated with the tool used, project size and age.

Before focusing specifically on runtime information, we reviewed multiple approaches which label parts of the source code with various kinds of metadata. We categorized 25 of them into a taxonomy with four dimensions: source, target, presentation and persistence.

Then we studied the effect of a certain code labeling approach, concern annotations, on program comprehension. Thanks to the controlled experiments, we found that the presence of concern annotations in the source code improves comprehension and maintenance correctness and time. The measured difference was 33% and 34%, respectively.

Traditionally, concern annotations are inserted into the source code manually. We presented a simple tool, AutoAnnot, which uses runtime information to write them into the source code, shifting the source of the metadata (with respect to our taxonomy) from human to runtime. By integrating the existing results in this area with our findings, we stated that the overlap between the annotations produced by this tool and the code author's annotations is higher than the overlap among at least 3 of 7 non-authors, but lower than the overlap among two of them.

Next, we studied the possibilities of an internal persistence of runtime metadata. Two suitable formats were described: annotations for certain element-level data and comments for line-level metadata. For certain kinds of data, such as critical bottlenecks determined by

profiling, we suggest a shared workflow where the data are checked into a version control system. For other kinds, we recommend a local-only workflow, which can be perceived as an IDE-independent form of integration of tools with source code editors.

Programmers often try to find strings displayed in the GUI of a running program in the source code. In our simulation study performed by GUI-scraping four desktop Java applications, about 11% of strings and 4% of words were not found in the code at all; 24% of strings and 49% of words had at least 100 occurrences. About 33% of the found strings and 49% of words did not occur in Java source files. XML, ".properties" and other file types require the programmer to perform multiple steps until the feature implementation is found. Many strings did not have any occurrence because they were dynamically generated at runtime. To sum up, the strategy of finding the displayed strings using traditional, static source code search available in the current IDEs is often insufficient or impractical.

The GUI study represents one of the motivational factors for our newly designed approach, RutimeSearch. Instead of searching in the static source code, it searches in all strings being evaluated in a running program. When a match is found, the program is paused and all debugging features of the IDE are available. RuntimeSearch is useful to find an initial investigation point in the source code, search for multiple occurrences throughout the application layers, find also non-GUI strings (e.g., the contents of a file buffer) and confirm hypotheses about strings in the program. We also introduced a "fabricated string technique" when the programmer searches for a made-up string which he entered into the component of interest. In a preliminary performance evaluation, the time overhead was near-zero without an active search, about 38% otherwise. A controlled experiment with human participants shown a 60% improvement of efficiency during maintenance tasks focused on searching displayed strings.

After programmers find the relevant portion of the code, they often read the documentation of the methods, which should ideally contain useful examples. We introduced DynamiDoc, a documentation generator utilizing runtime information. It collects string representations of arguments, return values, thrown exceptions and current object changes. Then, it generates documentation sentences containing samples of these collected values. In a qualitative evaluation, we found it suitable particularly for utility methods and methods manipulating simple data structures. On the other hand, in complicated methods manipulating many objects, we observed its weaknesses. As a form of preliminary quantitative evaluation, two findings were presented. First, an average length of a documentation sentence is approximately 10% of the associated method length. Second, 88% of objects taken from a small sample had a custom toString method, which is a prerequisite for the usefulness of DynamiDoc.

As an alternative to the documentation of whole methods using plain-text descriptions, we presented a source code augmentation approach, RuntimeSamp. The main idea is to display sample values of variables on each source code line, using an IDE plugin. We discussed questions which should be answered for our approach to be useful and provided partial solutions to the outlined problems. Currently, the objects are displayed as strings (using the toString method) and recorded at the end of lines. To select an iteration which should be displayed, we utilize the text caret as a pointer to the programmer's attention. Time overhead of RuntimeSamp is currently about 140%, so optimization is necessary in the future.

Since RuntimeSamp is a visual source code augmentation approach, we complemented it with a systematic mapping study of similar tools. By combining keyword search, references inspection and manual filtering, we obtained a list of 103 academic tools displaying visual overlays directly in the source code editor. They were categorized in a taxonomy with 7 dimensions: source, type, visualization, location, target, interaction and IDE.

12.2 Contributions

First, we will summarize the major contributions to the research field of software engineering, mainly to the subfield of program comprehension using dynamic analysis.

- We performed a large-scale study of Java build failures, providing empirical data about build failure probability and error types (chapter 3).
- A taxonomy of source code labeling was devised in chapter 4.
- Two controlled experiments showed the causality between the presence of concern annotations in the source code and comprehension/maintenance correctness and time. These experiments were conducted in collaboration with Milan Nosál', see chapter 5.
- We provided empirical evidence about the presence, occurrence counts and location types of strings from the GUIs of running desktop programs in the source files (chapter 7).
- An approach to "search in the runtime" was designed in chapter 8. It was validated using a controlled experiment on human subjects, showing a positive effect on program maintenance efficiency during a simple search-focused task.
- A novel documentation approach was presented in chapter 9. It is based on concrete examples of objects obtained by dynamic analysis.
- The definition of the term "source code augmentation" was provided in chapter 10. We designed a taxonomy of visual augmentation of source code editors. The systematic literature review, resulting in a list of 103 relevant articles, was performed in collaboration with Michaela Bačíková and Sergej Chodarev.
- An approach to augment source code lines with sample variable values was designed in chapter 11. A list of important questions for future research was also mentioned.

We also provided multiple minor contributions in this dissertation. Some of these minor research contributions are:

- a trend analysis of topics in program comprehension (section 2.4),
- a comparison between human-human and human-machine annotation overlap (section 5.4)
- and a discussion about storage possibilities of runtime metadata directly in source code files (chapter 6).

12.3 Future Work

Many of the topics discussed in this thesis suggest future research possibilities. Now we will mention some of them.

The study about build systems describes the current state, which is certainly not optimal, but it does not provide any solutions. However, motivated also by our study, Hassan et al. [148] examined the details behind build failures, and Macho et al. designed an approach to automatically repair dependency-related build issues [148].

Our study of GUI term occurrences in the code was limited to four desktop Java applications. To increase its external validity, we should examine more programs, written in other languages and using different technologies. Since many desktop applications are gradually becoming legacy, a similar study performed on web applications would be more representative.

RuntimeSearch can be extended with more options resembling the traditional textual search (e.g., "Whole words" or "Find previous") to further strengthen the presented metaphor.

Among the ideas presented in the thesis, we consider RuntimeSearch to be the readiest for a successful transfer to the industry. Then we could perform field studies with industrial programmers on real projects.

One of the disadvantages of our documentation generator, DynamiDoc, is the inability to produce meaningful sentences when the toString methods are not overridden. Automated generation of string representations in such cases can be a fruitful future research idea. The mentioned shortcoming is shared with the current design of RuntimeSamp. However, in this case, we would like to use a more interactive solution. Although Chis et al. [45] introduced moldable (domain-specific) objects inspectors, their construction is prevalently manual. A completely automated generation of semi-graphical object representations, along with the conduction of supporting empirical studies, is a research idea we would like to fully embrace in the future.

Bibliography

[1] AGRAWAL, H., AND HORGAN, J. R. Dynamic program slicing. *SIGPLAN Not. 25*, 6 (June 1990), 246–256.

[2] ALAM, S., AND DUGERDIL, P. EvoSpaces visualization tool: Exploring software architecture in 3D. In *Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on* (Oct. 2007), pp. 269–270.

[3] ALSALLAKH, B., BODESINSKY, P., GRUBER, A., AND MIKSCH, S. Visual tracing for the Eclipse Java debugger. In *Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering* (Washington, DC, USA, 2012), CSMR '12, IEEE Computer Society, pp. 545–548.

[4] ANDAM, B., BURGER, A., BERGER, T., AND CHAUDRON, M. Florida: Feature LOcation DAshboard for extracting and visualizing feature traces. In *Proceedings of the Eleventh International Workshop on Variability Modelling of Software-intensive Systems* (2017), ACM, pp. 100–107.

[5] ANWIKAR, V., NAIK, R., CONTRACTOR, A., AND MAKKAPATI, H. Domain-driven technique for functionality identification in source code. *ACM SIGSOFT Software Engineering Notes 37*, 3 (May 2012), 1–8.

[6] ASENOV, D., AND MULLER, P. Envision: A fast and flexible visual code editor with fluid interactions (overview). In *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (July 2014), pp. 9–12.

[7] ASENOV, D., MÜLLER, P., AND VOGEL, L. The IDE as a scriptable information system. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (New York, NY, USA, 2016), ASE 2016, ACM, pp. 444–449.

[8] AZUMA, R., BAILLOT, Y., BEHRINGER, R., FEINER, S., JULIER, S., AND MACINTYRE, B. Recent advances in augmented reality. *IEEE Comput. Graph. Appl. 21*, 6 (Nov. 2001), 34–47.

[9] BACCHELLI, A., LANZA, M., AND D'AMBROS, M. Miler: A toolset for exploring email data. In *Proceedings of the 33rd International Conference on Software Engineering* (New York, NY, USA, 2011), ICSE '11, ACM, pp. 1025–1027.

[10] BALTES, S., SCHMITZ, P., AND DIEHL, S. Linking sketches and diagrams to source code artifacts. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering* (2014), vol. 16-21-November-2014, ACM, pp. 743–746.

[11] BANERJEE, A., GUO, H.-F., AND ROYCHOUDHURY, A. Debugging energy-efficiency related field failures in mobile apps. In *Proceedings - International Conference on Mobile Software Engineering and Systems, MOBILESoft 2016* (2016), ACM, pp. 127–138.

[12] BARIK, T., SONG, Y., JOHNSON, B., AND MURPHY-HILL, E. From quick fixes to slow fixes: Reimagining static analysis resolutions to enable design space exploration. In *Proceedings -*

2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016 (2016), IEEE, pp. 211–221.

[13] BARIK, T., WITSCHEY, J., JOHNSON, B., AND MURPHY-HILL, E. Compiler error notifications revisited: An interaction-first approach for helping developers more effectively comprehend and resolve error notifications. In *36th International Conference on Software Engineering, ICSE Companion 2014 - Proceedings* (2014), ACM, pp. 536–539.

[14] BATHIA, P., BEERELLI, B., AND LAVERDIÈRE, M.-A. Assisting programmers resolving vulnerabilities in Java web applications. In *Communications in Computer and Information Science* (2011), vol. 133 CCIS, pp. 268–279.

[15] BAČÍKOVÁ, M., PORUBÄN, J., AND LAKATOŠ, D. Defining domain language of graphical user interfaces. In *2nd Symposium on Languages, Applications and Technologies* (Dagstuhl, Germany, 2013), J. P. Leal, R. Rocha, and A. Simões, Eds., vol. 29 of *OpenAccess Series in Informatics (OASIcs)*, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 187–202.

[16] BAVOTA, G., COLANGELO, L., DE LUCIA, A., FUSCO, S., OLIVETO, R., AND PANICHELLA, A. TraceME: Traceability management in eclipse. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on* (Sept. 2012), pp. 642–645.

[17] BECK, F., DIT, B., VELASCO-MADDEN, J., WEISKOPF, D., AND POSHYVANYK, D. Rethinking user interfaces for feature location. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension* (Piscataway, NJ, USA, 2015), ICPC '15, IEEE Press, pp. 151–162.

[18] BECK, F., GULAN, S., BIEGEL, B., BALTES, S., AND WEISKOPF, D. RegViz: Visual debugging of regular expressions. In *Companion Proceedings of the 36th International Conference on Software Engineering* (New York, NY, USA, 2014), ICSE Companion 2014, ACM, pp. 504–507.

[19] BECK, F., HOLLERICH, F., DIEHL, S., AND WEISKOPF, D. Visual monitoring of numeric variables embedded in source code. In *Software Visualization (VISSOFT), 2013 First IEEE Working Conference on* (Sept. 2013), pp. 1–4.

[20] BECK, F., MOSELER, O., DIEHL, S., AND REY, G. In situ understanding of performance bottlenecks through visually augmented code. In *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on* (May 2013), pp. 63–72.

[21] BELLER, M., GOUSIOS, G., AND ZAIDMAN, A. Oops, my tests broke the build: An analysis of Travis CI builds with GitHub. PeerJ Preprints 4:e1984v1, 2016.

[22] BELMONTE, J., DUGERDIL, P., AND AGRAWAL, A. A three-layer model of source code comprehension. In *Proceedings of the 7th India Software Engineering Conference* (New York, NY, USA, 2014), ISEC '14, ACM, pp. 10:1–10:10.

[23] BEYER, D., AND FARAROOY, A. DepDigger: A tool for detecting complex low-level dependencies. In *Program Comprehension (ICPC), 2010 IEEE 18th International Conference on* (June 2010), pp. 40–41.

[24] BIEGEL, B., BALTES, S., SCARPELLINI, I., AND DIEHL, S. CodeBasket: Making developers' mental model visible and explorable. In *Proceedings - 2nd International Workshop on Context for Software Development, CSD 2015* (2015), IEEE, pp. 20–24.

[25] BIGGERSTAFF, T. J., MITBANDER, B. G., AND WEBSTER, D. The concept assignment problem in program understanding. In *Proceedings of the 15th International Conference on Software Engineering* (Los Alamitos, CA, USA, 1993), ICSE '93, IEEE Computer Society Press, pp. 482–498.

[26] BINKLEY, D., GOLD, N., HARMAN, M., ISLAM, S., KRINKE, J., AND YOO, S. ORBS: Language-independent program slicing. In *Proceedings of the 22Nd ACM SIGSOFT Interna-*

tional Symposium on Foundations of Software Engineering (New York, NY, USA, 2014), FSE 2014, ACM, pp. 109–120.

[27] BLACKBURN, S. M., ET AL. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications* (New York, NY, USA, 2006), ACM, pp. 169–190.

[28] BOHNET, J., AND DÖLLNER, J. Analyzing dynamic call graphs enhanced with program state information for feature location and understanding. In *Companion of the 30th International Conference on Software Engineering* (New York, NY, USA, 2008), ICSE Companion '08, ACM, pp. 915–916.

[29] BOLAND, M., AND CLIFTON, C. Introducing PyLighter: Dynamic code highlighter. *SIGCSE Bulletin 41*, 1 (2009), 489–493.

[30] BOSHERNITSAN, M., GRAHAM, S., AND HEARST, M. Aligning development tools with the way programmers think about code changes. In *Conference on Human Factors in Computing Systems - Proceedings* (2007), pp. 567–576.

[31] BRADLEY, A. W., AND MURPHY, G. C. Supporting software history exploration. In *Proceedings of the 8th Working Conference on Mining Software Repositories* (New York, NY, USA, 2011), MSR '11, ACM, pp. 193–202.

[32] BRAGDON, A., REISS, S. P., ZELEZNIK, R., KARUMURI, S., CHEUNG, W., KAPLAN, J., COLEMAN, C., ADEPUTRA, F., AND LAVIOLA, JR., J. J. Code Bubbles: Rethinking the user interface paradigm of integrated development environments. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1* (New York, NY, USA, 2010), ICSE '10, ACM, pp. 455–464.

[33] BRECKEL, A., AND TICHY, M. Embedding programming context into source code. In *IEEE International Conference on Program Comprehension* (2016), vol. 2016-July, IEEE Computer Society.

[34] BRERETON, P., KITCHENHAM, B. A., BUDGEN, D., TURNER, M., AND KHALIL, M. Lessons from applying the systematic literature review process within the software engineering domain. *Journal of Systems and Software 80*, 4 (2007), 571–583.

[35] BROOKS, R. Using a behavioral theory of program comprehension in software engineering. In *Proceedings of the 3rd International Conference on Software Engineering* (Piscataway, NJ, USA, 1978), ICSE '78, IEEE Press, pp. 196–201.

[36] BUSE, R. P., AND WEIMER, W. R. Automatic documentation inference for exceptions. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis* (New York, NY, USA, 2008), ISSTA '08, ACM, pp. 273–282.

[37] BUSE, R. P., AND WEIMER, W. R. Synthesizing API usage examples. In *2012 34th International Conference on Software Engineering (ICSE)* (June 2012), pp. 782–792.

[38] CARD, S. K., AND MACKINLAY, J. The structure of the information visualization design space. In *Information Visualization, 1997. Proceedings., IEEE Symposium on* (Oct. 1997), pp. 92–99.

[39] CARVALHO, N. R., ALMEIDA, J. J., HENRIQUES, P. R., AND PEREIRA, M. J. V. Conclave: Ontology-driven measurement of semantic relatedness between source code elements and problem domain concepts. In *Computational Science and Its Applications – ICCSA 2014*. Springer International Publishing, 2014, pp. 116–131.

[40] CARVER, J., JACCHERI, L., MORASCA, S., AND SHULL, F. A checklist for integrating student empirical studies with research and teaching goals. *Empirical Software Engineering 15*, 1 (2010), 35–59.

[41] CAZZOLA, W., AND VACCHI, E. @Java: Bringing a richer annotation model to Java. *Computer Languages, Systems & Structures 40*, 1 (2014), 2–18. Special issue on the Programming Languages track at the 28th ACM Symposium on Applied Computing.

[42] CHEN, K., AND RAJLICH, V. Case study of feature location using dependence graph. In *Proceedings of the 8th International Workshop on Program Comprehension* (Washington, DC, USA, 2000), IWPC '00, IEEE Computer Society, pp. 241–247.

[43] CHIBA, S., HORIE, M., KANAZAWA, K., TAKEYAMA, F., AND TERAMOTO, Y. Do we really need to extend syntax for advanced modularity? In *AOSD'12 - Proceedings of the 11th Annual International Conference on Aspect Oriented Software Development* (2012), pp. 95–106.

[44] CHIŞ, A., GÎRBA, T., KUBELKA, J., NIERSTRASZ, O., REICHHART, S., AND SYREL, A. Moldable, context-aware searching with Spotter. In *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (New York, NY, USA, 2016), Onward! 2016, ACM, pp. 128–144.

[45] CHIŞ, A., NIERSTRASZ, O., SYREL, A., AND GÎRBA, T. The moldable inspector. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (New York, NY, USA, 2015), Onward! 2015, ACM, pp. 44–60.

[46] CITO, J., LEITNER, P., GALL, H., DADASHI, A., KELLER, A., AND ROTH, A. Runtime metric meets developer: Building better cloud applications using feedback. In *Onward! 2015 - Proceedings of the 2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Part of SPLASH 2015* (2015), ACM, pp. 14–27.

[47] CORNELISSEN, B., ZAIDMAN, A., VAN DEURSEN, A., MOONEN, L., AND KOSCHKE, R. A systematic survey of program comprehension through dynamic analysis. *Software Engineering, IEEE Transactions on 35*, 5 (Sept. 2009), 684–702.

[48] COSSETTE, B., AND WALKER, R. DSketch: Lightweight, adaptable dependency analysis. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering* (2010), pp. 297–306.

[49] COTTRELL, R., WALKER, R., AND DENZINGER, J. Semi-automating small-scale source code reuse via structural correspondence. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering* (2008), pp. 214–225.

[50] COWLISHAW, M. F. Lexx—a programmable structured editor. *IBM J. Res. Dev. 31*, 1 (Jan. 1987), 73–80.

[51] CROSS II, J., HENDRIX, T., AND BAROWSKI, L. Combining dynamic program viewing and testing in early computing courses. In *Proceedings - International Computer Software and Applications Conference* (2011), pp. 184–192.

[52] DALLMEIER, V., LINDIG, C., WASYLKOWSKI, A., AND ZELLER, A. Mining object behavior with ADABU. In *Proceedings of the 2006 International Workshop on Dynamic Systems Analysis* (New York, NY, USA, 2006), WODA '06, ACM, pp. 17–24.

[53] DAMEVSKI, K., SHEPHERD, D., AND POLLOCK, L. A field study of how developers locate features in source code. *Empirical Software Engineering 21*, 2 (2016), 724–747.

[54] DANG, H., NGUYEN, V., DO, K., AND TRAN, T. EduCo: An integrated social environment for teaching and learning software engineering courses. In *Proceedings of the 16th International Conference on Information Integration and Web-based Applications & Services* (2014), vol. 04-06-December-2014, ACM, pp. 17–26.

[55] DE ROOVER, C., NOGUERA, C., KELLENS, A., AND JONCKERS, V. The SOUL tool suite for querying programs in symbiosis with Eclipse. In *Proceedings of the 9th International Conference on the Principles and Practice of Programming in Java, PPPJ 2011* (2011), pp. 71–80.

[56] DEKEL, U., AND HERBSLEB, J. Improving API documentation usability with knowledge pushing. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on* (May 2009), pp. 320–330.

[57] DELINE, R., BRAGDON, A., ROWAN, K., JACOBSEN, J., AND REISS, S. P. Debugger canvas: Industrial experience with the code bubbles paradigm. In *Proceedings of the 34th International Conference on Software Engineering* (Piscataway, NJ, USA, 2012), ICSE '12, IEEE Press, pp. 1064–1073.

[58] DESMOND, M., STOREY, M.-A., AND EXTON, C. Fluid source code views. In *IEEE International Conference on Program Comprehension* (2006), vol. 2006, pp. 260–263.

[59] DIT, B., REVELLE, M., GETHERS, M., AND POSHYVANYK, D. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process 25*, 1 (2013), 53–95.

[60] DO, L., ALI, K., LIVSHITS, B., BODDEN, E., SMITH, J., AND MURPHY-HILL, E. Cheetah: Just-in-time taint analysis for Android apps. In *Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering Companion, ICSE-C 2017* (2017), IEEE, pp. 39–42.

[61] DUALA-EKOKO, E., AND ROBILLARD, M. Tracking code clones in evolving software. In *Proceedings - International Conference on Software Engineering* (2007), pp. 158–167.

[62] DUCASSE, M. Coca: an automated debugger for C. In *Proceedings of the 1999 International Conference on Software Engineering* (May 1999), pp. 504–513.

[63] EDWARDS, J. Example centric programming. *SIGPLAN Not. 39*, 12 (Dec. 2004), 84–91.

[64] EISENBARTH, T., KOSCHKE, R., AND SIMON, D. Locating features in source code. *IEEE Transactions on Software Engineering 29*, 3 (March 2003), 210–224.

[65] EYL, M., REICHMANN, C., AND MÜLLER-GLASER, K. Traceability in a fine grained software configuration management system. In *Lecture Notes in Business Information Processing* (2017), vol. 269, Springer Verlag, pp. 15–29.

[66] FAN, H., SUN, C., AND SHEN, H. ATCoPE: Any-time collaborative programming environment for seamless integration of real-time and non-real-time teamwork in software development. In *GROUP'12 - Proceedings of the ACM 2012 International Conference on Support Group Work* (2012), pp. 107–116.

[67] FAVRE, J.-M., LÄMMEL, R., LEINBERGER, M., SCHMORLEIZ, T., AND VARANOVICH, A. Linking documentation and source code in a software chrestomathy. In *Reverse Engineering (WCRE), 2012 19th Working Conference on* (Oct. 2012), pp. 335–344.

[68] FEIGENSPAN, J., KASTNER, C., LIEBIG, J., APEL, S., AND HANENBERG, S. Measuring programming experience. In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on* (June 2012), pp. 73–82.

[69] FELDMAN, S. I. Make – a program for maintaining computer programs. *Software: Practice and Experience 9*, 4 (1979), 255–265.

[70] FERRARI, A., GARBERVETSKY, D., BRABERMAN, V., LISTINGART, P., AND YOVINE, S. JScoper: Eclipse support for research on scoping and instrumentation for real time Java applications. In *Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology eXchange, eclipse'05* (2005), pp. 50–54.

[71] FINDLER, R., CLEMENTS, J., FLANAGAN, C., FLATT, M., KRISHNAMURTHI, S., STECKLER, P., AND FELLEISEN, M. DrScheme: A programming environment for scheme. *Journal of Functional Programming 12*, 2 (2002), 159–182.

[72] FITTKAU, F., WALLER, J., WULF, C., AND HASSELBRING, W. Live trace visualization for comprehending large software landscapes: The ExplorViz approach. In *Software Visualization (VISSOFT), 2013 First IEEE Working Conference on* (Sept. 2013), pp. 1–4.

[73] FLURI, B., ZUBERBÜHLER, J., AND GALL, H. Recommending method invocation context changes. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering* (2008), pp. 1–5.

[74] FRASER, G., AND ARCURI, A. A large-scale evaluation of automated unit test generation using EvoSuite. *ACM Trans. Softw. Eng. Methodol. 24*, 2 (Dec. 2014), 8:1–8:42.

[75] GANEA, G., VEREBI, I., AND MARINESCU, R. Continuous quality assessment with inCode. *Science of Computer Programming 134* (2017), 19–36.

[76] GE, X., DUBOSE, Q., AND MURPHY-HILL, E. Reconciling manual and automatic refactoring. In *Proceedings - International Conference on Software Engineering* (2012), pp. 211–221.

[77] GE, X., AND MURPHY-HILL, E. Manual refactoring changes with automated refactoring validation. In *Proceedings - International Conference on Software Engineering* (2014), no. 1, IEEE Computer Society, pp. 1095–1105.

[78] GLASS, R. L. Frequently forgotten fundamental facts about software engineering. *Software, IEEE 18*, 3 (May 2001), 112, 110–111.

[79] GOLDMAN, M., LITTLE, G., AND MILLER, R. Real-time collaborative coding in a web IDE. In *UIST'11 - Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology* (2011), pp. 155–164.

[80] GONCHARENKO, B., AND ZAYTSEV, V. Language design and implementation for the domain of coding conventions. In *SLE 2016 - Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, co-located with SPLASH 2016* (2016), ACM, pp. 90–104.

[81] GRAČANIN, D., MATKOVIĆ, K., AND ELTOWEISSY, M. Software visualization. *Innovations in Systems and Software Engineering 1*, 2 (Sept. 2005), 221–230.

[82] GREENFIELD, J., SHORT, K., COOK, S., AND KENT, S. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. John Wiley & Sons, 2004.

[83] GRISWOLD, W., YUAN, J., AND KATO, Y. Exploiting the map metaphor in a tool for software evolution. In *Proceedings - International Conference on Software Engineering* (2001), pp. 265–274.

[84] GUZZI, A., AND BEGEL, A. Facilitating communication between engineers with CARES. In *Proceedings - International Conference on Software Engineering* (2012), pp. 1367–1370.

[85] GUZZI, A., HATTORI, L., LANZA, M., PINZGER, M., AND VAN DEURSEN, A. Collective code bookmarks for program comprehension. In *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on* (June 2011), pp. 101–110.

[86] HARMON, T., AND KLEFSTAD, R. Interactive back-annotation of worst-case execution time analysis for Java microprocessors. In *Proceedings - 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2007* (2007), pp. 209–216.

[87] HARTMANN, B., DHILLON, M., AND CHAN, M. HyperSource: Bridging the gap between source and code-related web sites. In *Conference on Human Factors in Computing Systems - Proceedings* (2011), pp. 2207–2210.

[88] HARWARD, M., IRWIN, W., AND CHURCHER, N. In situ software visualisation. In *Software Engineering Conference (ASWEC), 2010 21st Australian* (Apr. 2010), pp. 171–180.

[89] HATTORI, L., AND LANZA, M. Syde: A tool for collaborative software development. In *Proceedings - International Conference on Software Engineering* (2010), vol. 2, pp. 235–238.

[90] HAYASHI, S., HOSHINO, D., MATSUDA, J., SAEKI, M., OMORI, T., AND MARUYAMA, K. Historef: A tool for edit history refactoring. In *2015 IEEE 22nd International Conference on*

Software Analysis, Evolution, and Reengineering, SANER 2015 - Proceedings (2015), IEEE, pp. 469–473.

[91] HAYASHI, S., KAZATO, H., KOBAYASHI, T., OSHIMA, T., NATSUKAWA, K., HOSHINO, T., AND SAEKI, M. Guiding identification of missing scenarios for dynamic feature location. In *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)* (Dec. 2016), pp. 393–396.

[92] HOCHSTEIN, L., AND JIAO, Y. The cost of the build tax in scientific software. In *Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on* (Sept. 2011), pp. 384–387.

[93] HOFFMAN, D., AND STROOPER, P. Prose + test cases = specifications. In *Proceedings of the 34th International Conference on Technology of Object-Oriented Languages and Systems* (Washington, DC, USA, 2000), TOOLS '00, IEEE Computer Society, pp. 239–250.

[94] HOLMES, R., AND BEGEL, A. Deep Intellisense: A tool for rehydrating evaporated information. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories* (New York, NY, USA, 2008), MSR '08, ACM, pp. 23–26.

[95] HOLMES, R., AND NOTKIN, D. Enhancing static source code search with dynamic data. In *Proceedings of 2010 ICSE Workshop on Search-driven Development: Users, Infrastructure, Tools and Evaluation* (New York, NY, USA, 2010), SUITE '10, ACM, pp. 13–16.

[96] HOLMES, R., AND WALKER, R. Task-specific source code dependency investigation. In *VISSOFT 2007 - Proceedings of the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis* (2007), pp. 100–107.

[97] HOU, D., JABLONSKI, P., AND JACOB, F. CnP: Towards an environment for the proactive management of copy-and-paste programming. In *Program Comprehension, 2009. ICPC '09. IEEE 17th International Conference on* (May 2009), pp. 238–242.

[98] HUNDHAUSEN, C., AND BROWN, J. What You See Is What You Code: A "live" algorithm development and visualization environment for novice learners. *Journal of Visual Languages and Computing 18*, 1 (2007), 22–47.

[99] HUPFER, S., CHENG, L.-T., ROSS, S., AND PATTERSON, J. Introducing collaboration into an application development environment. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work, CSCW* (2004), pp. 21–24.

[100] JABLONSKI, P., AND HOU, D. Aiding software maintenance with copy-and-paste clone-awareness. In *IEEE International Conference on Program Comprehension* (2010), pp. 170–179.

[101] JACOB, F., HOU, D., AND JABLONSKI, P. Actively comparing clones inside the code editor. In *Proceedings - International Conference on Software Engineering* (2010), pp. 9–16.

[102] JBARA, A., AND FEITELSON, D. G. On the effect of code regularity on comprehension. In *Proceedings of the 22nd International Conference on Program Comprehension* (New York, NY, USA, 2014), ICPC 2014, ACM, pp. 189–200.

[103] JEDLITSCHKA, A., AND PFAHL, D. Reporting guidelines for controlled experiments in software engineering. In *Empirical Software Engineering, 2005. 2005 International Symposium on* (Nov. 2005), pp. 95–104.

[104] JI, W., BERGER, T., ANTKIEWICZ, M., AND CZARNECKI, K. Maintaining feature traceability with embedded annotations. In *Proceedings of the 19th International Conference on Software Product Line* (New York, NY, USA, 2015), SPLC '15, ACM, pp. 61–70.

[105] JOY, M., BECKER, M., MUELLER, W., AND MATHEWS, E. Automated source code annotation for timing analysis of embedded software. In *Advanced Computing and Communications (ADCOM), 2012 18th Annual International Conference on* (Dec. 2012), pp. 12–18.

[106] KALLIAMVAKOU, E., GOUSIOS, G., BLINCOE, K., SINGER, L., GERMAN, D. M., AND DAMIAN, D. The promises and perils of mining GitHub. In *Proceedings of the 11th Working*

Conference on Mining Software Repositories (New York, NY, USA, 2014), MSR 2014, ACM, pp. 92–101.

[107] KAMPENES, V. B., DYBÅ, T., HANNAY, J. E., AND SJØBERG, D. I. K. A systematic review of quasi-experiments in software engineering. *Information and Software Technology 51*, 1 (2009), 71–82. Special Section - Most Cited Articles in 2002 and Regular Research Papers.

[108] KANG, H., AND GUO, P. Omnicode: A novice-oriented live programming environment with always-on run-time value visualizations. In *UIST 2017 - Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology* (2017), ACM, pp. 737–745.

[109] KARRER, T., KRÄMER, J.-P., DIEHL, J., HARTMANN, B., AND BORCHERS, J. Stacksplorer: Call graph navigation helps increasing code maintenance efficiency. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology* (New York, NY, USA, 2011), UIST '11, ACM, pp. 217–224.

[110] KÄSTNER, C., APEL, S., AND KUHLEMANN, M. Granularity in software product lines. In *Proceedings - International Conference on Software Engineering* (2008), pp. 311–320.

[111] KAWRYKOW, D., AND ROBILLARD, M. Improving API usage through automatic detection of redundant code. In *ASE2009 - 24th IEEE/ACM International Conference on Automated Software Engineering* (2009), pp. 111–122.

[112] KERZAZI, N., KHOMH, F., AND ADAMS, B. Why do automated builds break? An empirical study. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on* (Sept. 2014), pp. 41–50.

[113] KHOO, Y. P., FOSTER, J. S., AND HICKS, M. Expositor: Scriptable time-travel debugging with first-class traces. In *Proceedings of the 2013 International Conference on Software Engineering* (Piscataway, NJ, USA, 2013), ICSE '13, IEEE Press, pp. 352–361.

[114] KIM, J., LEE, S., HWANG, S.-W., AND KIM, S. Adding examples into Java documents. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering* (Washington, DC, USA, 2009), ASE '09, IEEE Computer Society, pp. 540–544.

[115] KIRINUKI, H., HIGO, Y., HOTTA, K., AND KUSUMOTO, S. Hey! Are you committing tangled changes? In *Proceedings of the 22nd International Conference on Program Comprehension* (New York, NY, USA, 2014), ICPC 2014, ACM, pp. 262–265.

[116] KISTNER, F., BETH KERY, M., PUSKAS, M., MOORE, S., AND MYERS, B. Moonstone: Support for understanding and writing exception handling code. In *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC* (2017), vol. 2017-October, IEEE Computer Society, pp. 63–71.

[117] KITCHENHAM, B., AND CHARTERS, S. Guidelines for performing systematic literature reviews in software engineering. Technical Report EBSE-2007-01, Keele University and Durham University Joint Report, July 2007.

[118] KLOCK, S., GETHERS, M., DIT, B., AND POSHYVANYK, D. Traceclipse: An Eclipse plug-in for traceability link recovery and management. In *Proceedings of the 6th International Workshop on Traceability in Emerging Forms of Software Engineering* (New York, NY, USA, 2011), TEFSE '11, ACM, pp. 24–30.

[119] KO, A., LATOZA, T., AND BURNETT, M. A practical guide to controlled experiments of software engineering tools with human participants. *Empirical Software Engineering 20*, 1 (2015), 110–141.

[120] KO, A. J., AUNG, H. H., AND MYERS, B. A. Eliciting design requirements for maintenance-oriented IDEs: A detailed study of corrective and perfective maintenance tasks. In *Proceedings of the 27th International Conference on Software Engineering* (New York, NY, USA, 2005), ICSE '05, ACM, pp. 126–135.

[121] KO, A. J., AND MYERS, B. A. Debugging reinvented: Asking and answering why and why not questions about program behavior. In *Proceedings of the 30th International Conference on Software Engineering* (New York, NY, USA, 2008), ICSE '08, ACM, pp. 301–310.

[122] KOLLÁR, J., CHODAREV, S., PIETRIKOVÁ, E., AND WASSERMANN, L'. Identification of patterns through Haskell programs analysis. In *Computer Science and Information Systems (FedCSIS), 2011 Federated Conference on* (Sept. 2011), pp. 891–894.

[123] KOSAR, T., MERNIK, M., AND CARVER, J. C. Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments. *Empirical Software Engineering 17*, 3 (2012), 276–304.

[124] KOSAR, T., OLIVEIRA, N., MERNIK, M., PEREIRA, M. J. V., ČREPINŠEK, M., DA CRUZ, D., AND HENRIQUES, P. R. Comparing general-purpose and domain-specific languages: An empirical study. *Computer Science and Information Systems 7*, 2 (Apr. 2010), 247–264.

[125] KRAMER, D. API documentation from source code comments: A case study of Javadoc. In *Proceedings of the 17th Annual International Conference on Computer Documentation* (New York, NY, USA, 1999), SIGDOC '99, ACM, pp. 147–153.

[126] KRÄMER, J.-P., KURZ, J., KARRER, T., AND BORCHERS, J. How live coding affects developers' coding behavior. In *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (July 2014), pp. 5–8.

[127] LAKATOŠ, D., PORUBÄN, J., AND BAČÍKOVÁ, M. Declarative specification of references in DSLs. In *Computer Science and Information Systems (FedCSIS), 2013 Federated Conference on* (Sept. 2013), pp. 1527–1534.

[128] LANZA, M., DUCASSE, S., GALL, H., AND PINZGER, M. CodeCrawler - an information visualization tool for program comprehension. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on* (May 2005), pp. 672–673.

[129] LATOZA, T. D., AND MYERS, B. A. Developers ask reachability questions. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1* (New York, NY, USA, 2010), ICSE '10, ACM, pp. 185–194.

[130] LATOZA, T. D., VENOLIA, G., AND DELINE, R. Maintaining mental models: A study of developer work habits. In *Proceedings of the 28th International Conference on Software Engineering* (New York, NY, USA, 2006), ICSE '06, ACM, pp. 492–501.

[131] LAUTAMÄKI, J., NIEMINEN, A., KOSKINEN, J., AHO, T., MIKKONEN, T., AND ENGLUND, M. CoRED: Browser-based collaborative real-time editor for Java web applications. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work, CSCW* (2012), pp. 1307–1316.

[132] LAYMAN, L., DIEP, M., NAGAPPAN, M., SINGER, J., DELINE, R., AND VENOLIA, G. Debugging revisited: Toward understanding the debugging needs of contemporary software developers. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement* (Los Alamitos, CA, USA, 2013), IEEE Computer Society, pp. 383–392.

[133] LEE, H., ANTKIEWICZ, M., AND CZARNECKI, K. Towards a generic infrastructure for framework-specific integrated development environment extensions. In *Domain-Specific Program Development 2008* (Nashville, United States, 2008).

[134] LEE, S., CHEN, Y., KLUGMAN, N., GOURAVAJHALA, S., CHEN, A., AND LASECKI, W. Exploring coordination models for ad hoc programming teams. In *Conference on Human Factors in Computing Systems - Proceedings* (2017), vol. Part F127655, ACM, pp. 2738–2745.

[135] LEFEBVRE, G., CULLY, B., HEAD, C., SPEAR, M., HUTCHINSON, N., FEELEY, M., AND WARFIELD, A. Execution mining. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments* (New York, NY, USA, 2012), VEE '12, ACM, pp. 145–158.

[136] LEITNER, P., CITO, J., AND STÖCKLI, E. Modelling and managing deployment costs of microservice-based cloud applications. In *Proceedings - 9th IEEE/ACM International Conference on Utility and Cloud Computing, UCC 2016* (2016), ACM, pp. 165–174.

[137] LHOTÁK, J., LHOTÁK, O., AND HENDREN, L. Integrating the Soot compiler infrastructure into an IDE. In *Lecture Notes in Computer Science* (2004), vol. 2985, pp. 281–297.

[138] LI, D., HAO, S., HALFOND, W., AND GOVINDAN, R. Calculating source line level energy information for Android applications. In *2013 International Symposium on Software Testing and Analysis, ISSTA 2013 - Proceedings* (2013), pp. 78–89.

[139] LICHTSCHLAG, L., SPYCHALSKI, L., AND BOCHERS, J. CodeGraffiti: Using hand-drawn sketches connected to code bases in navigation tasks. In *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC* (2014), IEEE Computer Society, pp. 65–68.

[140] LIEBER, T., BRANDT, J. R., AND MILLER, R. C. Addressing misconceptions about code with always-on programming visualizations. In *Proceedings of the 32nd Annual ACM Conference on Human Factors in Computing Systems* (New York, NY, USA, 2014), CHI '14, ACM, pp. 2481–2490.

[141] LIEBIG, J., KÄSTNER, C., AND APEL, S. Analyzing the discipline of preprocessor annotations in 30 million lines of C code. In *Proceedings of the Tenth International Conference on Aspect-oriented Software Development* (New York, NY, USA, 2011), AOSD '11, ACM, pp. 191–202.

[142] LIU, D., MARCUS, A., POSHYVANYK, D., AND RAJLICH, V. Feature location via information retrieval based filtering of a single scenario execution trace. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering* (New York, NY, USA, 2007), ASE '07, ACM, pp. 234–243.

[143] LIU, D., AND XU, S. MuTT: A multi-threaded tracer for Java programs. In *Computer and Information Science, 2009. ICIS 2009. Eighth IEEE/ACIS International Conference on* (June 2009), pp. 949–954.

[144] LO, D., AND MAOZ, S. Scenario-based and value-based specification mining: Better together. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering* (New York, NY, USA, 2010), ASE '10, ACM, pp. 387–396.

[145] LONG, F., WANG, X., AND CAI, Y. API hyperlinking via structural overlap. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering* (New York, NY, USA, 2009), ESEC/FSE '09, ACM, pp. 203–212.

[146] LOPEZ, N., AND VAN DER HOEK, A. The Code Orb - supporting contextualized coding via at-a-glance views (NIER track). In *Proceedings - International Conference on Software Engineering* (2011), pp. 824–827.

[147] MAALEJ, W., TIARKS, R., ROEHM, T., AND KOSCHKE, R. On the comprehension of program comprehension. *ACM Trans. Softw. Eng. Methodol. 23*, 4 (Sept. 2014), 31:1–31:37.

[148] MACHO, C., MCINTOSH, S., AND PINZGER, M. Automatically repairing dependency-related build breakage. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (Mar. 2018), pp. 106–117.

[149] MALETIC, J. I., MARCUS, A., AND COLLARD, M. L. A task oriented view of software visualization. In *Proceedings First International Workshop on Visualizing Software for Understanding and Analysis* (2002), pp. 32–40.

[150] MARCEAU, G., COOPER, G. H., KRISHNAMURTHI, S., AND REISS, S. P. A dataflow language for scriptable debugging. In *Proceedings. 19th International Conference on Automated Software Engineering, 2004.* (Sept 2004), pp. 218–227.

[151] MARCEAU, G., FISLER, K., AND KRISHNAMURTHI, S. Mind your language: On novices' interactions with error messages. In *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (New York, NY, USA, 2011), Onward! 2011, ACM, pp. 3–18.

[152] MARCUS, A., SERGEYEV, A., RAJLICH, V., AND MALETIC, J. I. An information retrieval approach to concept location in source code. In *Reverse Engineering, 2004. Proceedings. 11th Working Conference on* (Nov. 2004), pp. 214–223.

[153] MATSUMURA, T., ISHIO, T., KASHIMA, Y., AND INOUE, K. Repeatedly-executed-method viewer for efficient visualization of execution paths and states in Java. In *Proceedings of the 22nd International Conference on Program Comprehension* (New York, NY, USA, 2014), ICPC 2014, ACM, pp. 253–257.

[154] MCBURNEY, P. W., AND MCMILLAN, C. Automatic documentation generation via source code summarization of method context. In *Proceedings of the 22Nd International Conference on Program Comprehension* (New York, NY, USA, 2014), ICPC 2014, ACM, pp. 279–290.

[155] MCDIRMID, S. Usable live programming. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (New York, NY, USA, 2013), Onward! 2013, ACM, pp. 53–62.

[156] MCINTOSH, S., ADAMS, B., AND HASSAN, A. The evolution of ANT build systems. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on* (May 2010), pp. 42–51.

[157] MCINTOSH, S., NAGAPPAN, M., ADAMS, B., MOCKUS, A., AND HASSAN, A. E. A large-scale empirical study of the relationship between build technology and build maintenance. *Empirical Software Engineering 20*, 6 (2015), 1587–1633.

[158] MEINICKE, J., THÜM, T., SCHRÖTER, R., KRIETER, S., BENDUHN, F., SAAKE, G., AND LEICH, T. FeatureIDE: Taming the preprocessor wilderness. In *Proceedings - International Conference on Software Engineering* (2016), IEEE Computer Society, pp. 629–632.

[159] MELLIS, D. A. Tangible code. Thesis report, Interaction Design Insitute Ivrea, May 2006.

[160] MEMON, A., BANERJEE, I., AND NAGARAJAN, A. GUI ripping: reverse engineering of graphical user interfaces for testing. In *Reverse Engineering, 2003. WCRE 2003. Proceedings. 10th Working Conference on* (Nov. 2003), pp. 260–269.

[161] MESBAH, A., VAN DEURSEN, A., AND LENSELINK, S. Crawling AJAX-based Web applications through dynamic analysis of user interface state changes. *ACM Trans. Web 6*, 1 (Mar. 2012), 3:1–3:30.

[162] MICHAIL, A. Browsing and searching source code of applications written using a GUI framework. In *Proceedings of the 24th International Conference on Software Engineering* (New York, NY, USA, 2002), ICSE '02, ACM, pp. 327–337.

[163] MICHEL, J.-B., SHEN, Y. K., AIDEN, A. P., VERES, A., GRAY, M. K., TEAM, T. G. B., PICKETT, J. P., HOIBERG, D., CLANCY, D., NORVIG, P., ORWANT, J., PINKER, S., NOWAK, M. A., AND AIDEN, E. L. Quantitative analysis of culture using millions of digitized books. *Science 331*, 6014 (2011), 176–182.

[164] MONTANDON, J. A. E., BORGES, H., FELIX, D., AND VALENTE, M. T. Documenting APIs with examples: Lessons learned with the APIMiner platform. In *2013 20th Working Conference on Reverse Engineering (WCRE)* (Oct 2013), pp. 401–408.

[165] MOODY, D. The "physics" of notations: Toward a scientific basis for constructing visual notations in software engineering. *IEEE Trans. Softw. Eng. 35*, 6 (Nov. 2009), 756–779.

[166] MORENO, L., APONTE, J., SRIDHARA, G., MARCUS, A., POLLOCK, L., AND VIJAY-SHANKER, K. Automatic generation of natural language summaries for Java classes. In *2013 21st International Conference on Program Comprehension (ICPC)* (May 2013), pp. 23–32.

[167] MÜLLER, C., REINA, G., AND ERTL, T. In-situ visualisation of fractional code ownership over time. In *Proceedings of the 8th International Symposium on Visual Information Communication and Interaction* (2015), ACM, pp. 13–20.

[168] MÜLLER, H. A., AND KIENLE, H. M. A small primer on software reverse engineering. Tech. rep., University of Victoria, 2009.

[169] MUROLO, A., STUTZ, F., HUSMANN, M., AND NORRIE, M. Improved developer support for the detection of cross-browser incompatibilities. In *Lecture Notes in Computer Science* (2017), vol. 10360 LNCS, Springer Verlag, pp. 264–281.

[170] MURPHY-HILL, E., AND BLACK, A. Breaking the barriers to successful refactoring: Observations and tools for extract method. In *Proceedings - International Conference on Software Engineering* (2008), pp. 421–430.

[171] MURPHY-HILL, E., AND BLACK, A. An interactive ambient visualization for code smells. In *Proceedings of the ACM Conference on Computer and Communications Security* (2010), pp. 5–14.

[172] MYERS, D., AND STOREY, M.-A. Using dynamic analysis to create trace-focused user interfaces for IDEs. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2010), FSE '10, ACM, pp. 367–368.

[173] MYERS, D., STOREY, M.-A., AND SALOIS, M. Utilizing debug information to compact loops in large program traces. In *Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering* (Washington, DC, USA, 2010), CSMR '10, IEEE Computer Society, pp. 41–50.

[174] NAZAR, N., HU, Y., AND JIANG, H. Summarizing software artifacts: A literature review. *Journal of Computer Science and Technology 31*, 5 (2016), 883–909.

[175] NEITSCH, A., WONG, K., AND GODFREY, M. Build system issues in multilanguage software. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on* (Sept. 2012), pp. 140–149.

[176] NETO, A. A., AND CONTE, T. Threats to validity and their control actions – results of a systematic literature review. Technical Report TR-USES-2014-0002, Universidade Federal do Amazonas, Mar. 2014.

[177] NGUYEN, B. N., ROBBINS, B., BANERJEE, I., AND MEMON, A. GUITAR: an innovative tool for automated testing of GUI-driven software. *Automated Software Engineering 21*, 1 (2013), 65–105.

[178] NGUYEN, H. V., KÄSTNER, C., AND NGUYEN, T. N. Cross-language program slicing for dynamic web applications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (New York, NY, USA, 2015), ESEC/FSE 2015, ACM, pp. 369–380.

[179] NISTOR, E., AND VAN DER HOEK, A. Explicit concern-driven development with ArchEvol. In *ASE2009 - 24th IEEE/ACM International Conference on Automated Software Engineering* (2009), pp. 185–196.

[180] NIU, N., MAHMOUD, A., AND YANG, X. Faceted navigation for software exploration. In *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on* (June 2011), pp. 193–196.

[181] NOSÁĽ, M. *Leveraging Program Comprehension with Concern-oriented Projections*. PhD thesis, Technical University of Košice, Apr. 2015.

[182] NUNEZ, W., MARIN, V., AND RIVERO, C. ARCC: Assistant for repetitive code comprehension. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering* (2017), vol. Part F130154, ACM, pp. 999–1003.

[183] OHMANN, P., AND LIBLIT, B. CSIclipse: Presenting crash analysis data to developers. In *ETX 2015 - Proceedings of the Eclipse Technology eXchange* (2015), ACM, pp. 7–12.

[184] OLSZAK, A., AND NØRREGAARD JØRGENSEN, B. Remodularizing Java programs for improved locality of feature implementations in source code. *Science of Computer Programming 77*, 3 (2012), 131–151.

[185] OU, J., VECHEV, M., AND HILLIGES, O. An interactive system for data structure development. In *Conference on Human Factors in Computing Systems - Proceedings* (2015), vol. 2015-April, ACM, pp. 3053–3062.

[186] PANCHENKO, O., KARSTENS, J., PLATTNER, H., AND ZEIER, A. Precise and scalable querying of syntactical source code patterns using sample code snippets and a database. In *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on* (June 2011), pp. 41–50.

[187] PANICHELLA, S., PANICHELLA, A., BELLER, M., ZAIDMAN, A., AND GALL, H. C. The impact of test case summaries on bug fixing performance: An empirical investigation. In *Proceedings of the 38th International Conference on Software Engineering* (New York, NY, USA, 2016), ICSE '16, ACM, pp. 547–558.

[188] PENNINGTON, N. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology 19*, 3 (1987), 295–341.

[189] PERSCHEID, M., STEINERT, B., HIRSCHFELD, R., GELLER, F., AND HAUPT, M. Immediacy through interactivity: Online analysis of run-time behavior. In *Proceedings of the 2010 17th Working Conference on Reverse Engineering* (Washington, DC, USA, 2010), WCRE '10, IEEE Computer Society, pp. 77–86.

[190] PETERSEN, K., VAKKALANKA, S., AND KUZNIARZ, L. Guidelines for conducting systematic mapping studies in software engineering: An update. *Information and Software Technology 64* (2015), to appear.

[191] PFEIFFER, R.-H., AND WĄSOWSKI, A. TexMo: A multi-language development environment. In *Lecture Notes in Computer Science* (2012), vol. 7349 LNCS, pp. 178–193.

[192] PORUBÄN, J., FORGÁČ, M., SABO, M., AND BĚHÁLEK, M. Annotation based parser generator. *Computer Science and Information Systems 7*, 2 (Apr. 2010), 291–307.

[193] PORUBÄN, J., AND NOSÁĽ, M. Leveraging program comprehension with concern-oriented source code projections. In *3rd Symposium on Languages, Applications and Technologies* (Dagstuhl, Germany, 2014), M. J. V. Pereira, J. P. Leal, and A. Simões, Eds., vol. 38 of *OpenAccess Series in Informatics (OASIcs)*, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 35–50.

[194] RAHMAN, F., BIRD, C., AND DEVANBU, P. Clones: what is that smell? *Empirical Software Engineering 17*, 4-5 (2012), 503–530.

[195] RAJLICH, V., AND WILDE, N. The role of concepts in program comprehension. In *Program Comprehension, 2002. Proceedings. 10th International Workshop on* (2002), pp. 271–278.

[196] RAJLICH, V., AND WILDE, N. A retrospective view on: The role of concepts in program comprehension. In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on* (June 2012), pp. 12–13.

[197] RÁSTOČNÝ, K., AND BIELIKOVÁ, M. Metadata anchoring for source code: Robust location descriptor definition, building and interpreting. In *Database and Expert Systems Applications*, H. Decker, L. Lhotská, S. Link, J. Basl, and A. M. Tjoa, Eds., vol. 8056 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2013, pp. 372–379.

[198] RATANOTAYANON, S., SIM, S., AND GALLARDO-VALENCIA, R. Supporting program comprehension in agile with links to user stories. In *Proceedings - 2009 Agile Conference, AGILE 2009* (2009), pp. 26–32.

[199] RATIU, D., AND DEISSENBOECK, F. Programs are knowledge bases. In *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on* (2006), pp. 79–83.

[200] RATIU, D., AND DEISSENBOECK, F. From reality to programs and (not quite) back again. In *Program Comprehension, 2007. ICPC '07. 15th IEEE International Conference on* (June 2007), pp. 91–102.

[201] REINIKAINEN, T., HAMMOUDA, I., LAIHO, J., KOSKIMIES, K., AND SYSTA, T. Software comprehension through concern-based queries. In *Program Comprehension, 2007. ICPC '07. 15th IEEE International Conference on* (June 2007), pp. 265–270.

[202] REISS, S. P. The paradox of software visualization. In *Proceedings of the 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis* (Washington, DC, USA, 2005), VISSOFT '05, IEEE Computer Society, pp. 59–63.

[203] REISS, S. P. Tracking source locations. In *Proceedings of the 30th International Conference on Software Engineering* (New York, NY, USA, 2008), ICSE '08, ACM, pp. 11–20.

[204] RESSIA, J., BERGEL, A., AND NIERSTRASZ, O. Object-centric debugging. In *Proceedings of the 34th International Conference on Software Engineering* (Piscataway, NJ, USA, 2012), ICSE '12, IEEE Press, pp. 485–495.

[205] REVELLE, M., BROADBENT, T., AND COPPIT, D. Understanding concerns in software: insights gained from two case studies. In *Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on* (May 2005), pp. 23–32.

[206] ROBERTS, C., WRIGHT, M., KUCHERA-MORIN, J., AND HÖLLERER, T. Gibber: Abstractions for creative multimedia programming. In *MM 2014 - Proceedings of the 2014 ACM Conference on Multimedia* (2014), ACM, pp. 67–76.

[207] ROBILLARD, M. P., AND MURPHY, G. C. Representing concerns in source code. *ACM Trans. Softw. Eng. Methodol. 16*, 1 (Feb. 2007), 3:1–3:38.

[208] ROBILLARD, M. P., AND WEIGAND-WARR, F. ConcernMapper: Simple view-based separation of scattered concerns. In *Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology eXchange* (New York, NY, USA, 2005), eclipse '05, ACM, pp. 65–69.

[209] ROEHM, T. Two user perspectives in program comprehension: End users and developer users. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension* (Piscataway, NJ, USA, 2015), ICPC '15, IEEE Press, pp. 129–139.

[210] RONG, X., YAN, S., ONEY, S., DONTCHEVA, M., AND ADAR, E. CodeMend: Assisting interactive programming with bimodal embedding. In *UIST 2016 - Proceedings of the 29th Annual Symposium on User Interface Software and Technology* (2016), ACM, pp. 247–258.

[211] RÖTHLISBERGER, D., GREEVY, O., AND NIERSTRASZ, O. Exploiting runtime information in the ide. In *Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension* (Washington, DC, USA, 2008), ICPC '08, IEEE Computer Society, pp. 63–72.

[212] RÖTHLISBERGER, D., HÄRRY, M., BINDER, W., MORET, P., ANSALONI, D., VILLAZÓN, A., AND NIERSTRASZ, O. Exploiting dynamic information in IDEs improves speed and correctness of software maintenance tasks. *Software Engineering, IEEE Transactions on 38*, 3 (May 2012), 579–591.

[213] RÖTHLISBERGER, D., NIERSTRASZ, O., DUCASSE, S., POLLET, D., AND ROBBES, R. Supporting task-oriented navigation in IDEs with configurable HeatMaps. In *Program Comprehension, 2009. ICPC '09. IEEE 17th International Conference on* (May 2009), pp. 253–257.

[214] ROY, C. K., CORDY, J. R., AND KOSCHKE, R. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming 74*, 7 (2009), 470–495. Special Issue on Program Comprehension (ICPC 2008).

[215] RUNESON, P., AND HÖST, M. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering 14*, 2 (2009), 131–164.

[216] SABO, M., AND PORUBÄN, J. Preserving design patterns using source code annotations. *Journal of Computer Science and Control Systems 2*, 1 (2009), 53–56.

[217] SAIGAL, N., AND LIGATTI, J. Inline visualization of concerns. In *Proceedings - 7th ACIS International Conference on Software Engineering Research, Management and Applications, SERA09* (2009), pp. 95–102.

[218] SAINI, V., SAJNANI, H., KIM, J., AND LOPES, C. SourcererCC and sourcererCC-I: Tools to detect clones in batch mode and during software development. In *Proceedings - International Conference on Software Engineering* (2016), IEEE Computer Society, pp. 597–600.

[219] SALINGER, S., OEZBEK, C., BEECHER, K., AND SCHENK, J. Saros: An Eclipse plug-in for distributed party programming. In *Proceedings - International Conference on Software Engineering* (2010), pp. 48–55.

[220] SANTOS, A. L. GUI-driven code tracing. In *Visual Languages and Human-Centric Computing (VL/HCC), 2012 IEEE Symposium on* (Sept. 2012), pp. 111–118.

[221] SAVAGE, T., REVELLE, M., AND POSHYVANYK, D. FLAT^3: Feature location and textual tracing tool. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2* (New York, NY, USA, 2010), ICSE '10, ACM, pp. 255–258.

[222] SCHUGERL, P., RILLING, J., AND CHARLAND, P. Beyond generated software documentation – a Web 2.0 perspective. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on* (Sept. 2009), pp. 547–550.

[223] SCHWARTZ, M. D., DELISLE, N. M., AND BEGWANI, V. S. Incremental compilation in magpie. *SIGPLAN Not. 19*, 6 (June 1984), 122–131.

[224] SEO, H., SADOWSKI, C., ELBAUM, S., AFTANDILIAN, E., AND BOWDIDGE, R. Programmers' build errors: A case study (at Google). In *Proceedings of the 36th International Conference on Software Engineering* (New York, NY, USA, 2014), ICSE 2014, ACM, pp. 724–734.

[225] SHAHIN, M., LIANG, P., AND BABAR, M. A. A systematic review of software architecture visualization techniques. *Journal of Systems and Software 94*, Supplement C (2014), 161–185.

[226] SHERWOOD, K. D., AND MURPHY, G. C. Reducing code navigation effort with differential code coverage. Tech. rep., Department of Computer Science, University of British Columbia, Sept. 2008.

[227] SILLITO, J., MURPHY, G. C., AND DE VOLDER, K. Questions programmers ask during software evolution tasks. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2006), SIGSOFT '06/FSE-14, ACM, pp. 23–34.

[228] SJØBERG, D., HANNAY, J., HANSEN, O., KAMPENES, V., KARAHASANOVIĆ, A., LIBORG, N.-K., AND REKDAL, A. A survey of controlled experiments in software engineering. *Software Engineering, IEEE Transactions on 31*, 9 (Sept. 2005), 733–753.

[229] SMITH, J., BROWN, C., AND MURPHY-HILL, E. Flower: Navigating program flow in the IDE. In *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC* (2017), vol. 2017-October, IEEE Computer Society, pp. 19–23.

[230] SMITH, P. *Software Build Systems: Principles and Experience*. Addison-Wesley Professional, 2011.

[231] SOHAN, S. M., ANSLOW, C., AND MAURER, F. SpyREST: Automated RESTful API documentation using an HTTP proxy server. In *Proceedings of the 2015 30th IEEE/ACM*

International Conference on Automated Software Engineering (ASE) (Washington, DC, USA, 2015), ASE '15, IEEE Computer Society, pp. 271–276.

[232] SPOLSKY, J. *Joel on Software*. Apress, Berkeley, CA, 2004, ch. The Joel Test: 12 Steps to Better Code, pp. 17–30.

[233] SRIDHARA, G., HILL, E., MUPPANENI, D., POLLOCK, L., AND VIJAY-SHANKER, K. Towards automatically generating summary comments for Java methods. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering* (New York, NY, USA, 2010), ASE '10, ACM, pp. 43–52.

[234] SRIDHARA, G., POLLOCK, L., AND VIJAY-SHANKER, K. Automatically detecting and describing high level actions within methods. In *Proceedings of the 33rd International Conference on Software Engineering* (New York, NY, USA, 2011), ICSE '11, ACM, pp. 101–110.

[235] SRIDHARA, G., POLLOCK, L., AND VIJAY-SHANKER, K. Generating parameter comments and integrating with method summaries. In *Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension* (Washington, DC, USA, 2011), ICPC '11, IEEE Computer Society, pp. 71–80.

[236] STEINERT, B., PERSCHEID, M., BECK, M., LINCKE, J., AND HIRSCHFELD, R. Debugging into examples. In *TESTCOM '09/FATES '09* (2009), Springer-Verlag, pp. 235–240.

[237] STEINERT, B., TAEUMEL, M., LINCKE, J., PAPE, T., AND HIRSCHFELD, R. CodeTalk - conversations about code. In *8th International Conference on Creating, Connecting and Collaborating through Computing, C5 2010* (2010), pp. 11–18.

[238] STEINMACHER, I., CHAVES, A. P., AND GEROSA, M. A. Awareness support in distributed software development: A systematic review and mapping of the literature. *Comput. Supported Coop. Work 22*, 2-3 (Apr. 2013), 113–158.

[239] STOREY, M. Theories, methods and tools in program comprehension: past, present and future. In *Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on* (May 2005), pp. 181–191.

[240] STOREY, M., RYALL, J., SINGER, J., MYERS, D., CHENG, L.-T., AND MULLER, M. How software developers use tagging to support reminding and refinding. *Software Engineering, IEEE Transactions on 35*, 4 (July 2009), 470–483.

[241] STOREY, M.-A., CHENG, L.-T., BULL, I., AND RIGBY, P. Shared waypoints and social tagging to support collaboration in software development. In *Proceedings of the 2006 20th Anniversary Conference on Computer Supported Cooperative Work* (New York, NY, USA, 2006), CSCW '06, ACM, pp. 195–198.

[242] STOREY, M.-A., RYALL, J., BULL, R. I., MYERS, D., AND SINGER, J. TODO or to bug: Exploring how task annotations play a role in the work practices of software developers. In *Proceedings of the 30th International Conference on Software Engineering* (New York, NY, USA, 2008), ICSE '08, ACM, pp. 251–260.

[243] STOREY, M.-A. D., ČUBRANIĆ, D., AND GERMAN, D. M. On the use of visualization to support awareness of human activities in software development: A survey and a framework. In *Proceedings of the 2005 ACM Symposium on Software Visualization* (New York, NY, USA, 2005), SoftVis '05, ACM, pp. 193–202.

[244] SULÍR, M. Program comprehension: A short literature review. In *SCYR 2015: 15th Scientific Conference of Young Researchers* (May 2015), pp. 283–286.

[245] SULÍR, M., BAČÍKOVÁ, M., CHODAREV, S., AND PORUBÄN, J. Visual augmentation of source code editors: A systematic review. *Computer Languages, Systems & Structures* (2018). Submitted.

[246] SULÍR, M., AND NOSÁĽ, M. Sharing developers' mental models through source code annotations. In *2015 Federated Conference on Computer Science and Information Systems (FedCSIS)* (Sept. 2015), pp. 997–1006.

[247] SULÍR, M., NOSÁĽ, M., AND PORUBÄN, J. Recording concerns in source code using annotations. *Computer Languages, Systems & Structures 46* (Nov. 2016), 44–65.

[248] SULÍR, M., AND PORUBÄN, J. Semi-automatic concern annotation using differential code coverage. In *2015 IEEE 13th International Scientific Conference on Informatics* (Nov. 2015), pp. 258–262. © 2015 IEEE.

[249] SULÍR, M., AND PORUBÄN, J. Trend analysis on the metadata of program comprehension papers. In *2015 13th International Conference on Engineering of Modern Electric Systems (EMES)* (June 2015), pp. 153–156. © 2015 IEEE.

[250] SULÍR, M., AND PORUBÄN, J. Locating user interface concepts in source code. In *5th Symposium on Languages, Applications and Technologies (SLATE'16)* (Dagstuhl, Germany, 2016), vol. 51 of *OpenAccess Series in Informatics (OASIcs)*, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 6:1–6:9.

[251] SULÍR, M., AND PORUBÄN, J. A quantitative study of Java software buildability. In *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools* (New York, NY, USA, 2016), PLATEAU 2016, ACM, pp. 17–25. http://doi.org/10.1145/3001878.3001882.

[252] SULÍR, M., AND PORUBÄN, J. Exposing runtime information through source code annotations. *Acta Electrotechnica et Informatica 17*, 1 (Apr. 2017), 3–9.

[253] SULÍR, M., AND PORUBÄN, J. Generating method documentation using concrete values from executions. In *6th Symposium on Languages, Applications and Technologies (SLATE 2017)* (Dagstuhl, Germany, 2017), vol. 56 of *OpenAccess Series in Informatics (OASIcs)*, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 3:1–3:13.

[254] SULÍR, M., AND PORUBÄN, J. Labeling source code with metadata: A survey and taxonomy. In *2017 Federated Conference on Computer Science and Information Systems (FedCSIS)* (Sept. 2017), pp. 721–729.

[255] SULÍR, M., AND PORUBÄN, J. RuntimeSearch: Ctrl+F for a running program. In *Proceedings of the 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2017), pp. 388–393. © 2017 IEEE.

[256] SULÍR, M., AND PORUBÄN, J. Source code documentation generation using program execution. *Information 8*, 4 (2017), 148.

[257] SULÍR, M., AND PORUBÄN, J. Augmenting source code lines with sample variable values. In *Proceedings of the 2018 26th IEEE/ACM International Conference on Program Comprehension (ICPC)* (May 2018). Accepted.

[258] SULÍR, M., PORUBÄN, J., AND ZORIČÁK, O. IDE-independent program comprehension tools via source file overwriting. In *2017 IEEE 14th International Scientific Conference on Informatics* (2017), pp. 372–376. © 2017 IEEE.

[259] SUTHERLAND, A., AND SCHNEIDER, K. UI traces: Supporting the maintenance of interactive software. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on* (Sept. 2009), pp. 563–566.

[260] SUTHERLAND, C. J., LUXTON-REILLY, A., AND PLIMMER, B. Freeform digital ink annotations in electronic documents: A systematic mapping study. *Computers & Graphics 55* (2016), 1–20.

[261] SUZUKI, T., SAKAMOTO, K., ISHIKAWA, F., AND HONIDEN, S. An approach for evaluating and suggesting method names using n-gram models. In *Proceedings of the 22nd International*

Conference on Program Comprehension (New York, NY, USA, 2014), ICPC 2014, ACM, pp. 271–274.

[262] SWIFT, B., SORENSEN, A., GARDNER, H., AND HOSKING, J. Visual code annotations for cyberphysical programming. In *Live Programming (LIVE), 2013 1st International Workshop on* (May 2013), pp. 27–30.

[263] TÁBORSKÝ, R., AND VRANIĆ, V. Feature model driven generation of software artifacts. In *2015 Federated Conference on Computer Science and Information Systems (FedCSIS)* (Sept. 2015), pp. 1007–1018.

[264] TAIRAS, R., AND GRAY, J. Increasing clone maintenance support by unifying clone detection and refactoring activities. *Information and Software Technology 54*, 12 (2012), 1297–1307.

[265] TAN, S. H., MARINOV, D., TAN, L., AND LEAVENS, G. T. @tComment: Testing Javadoc comments to detect comment-code inconsistencies. In *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation* (Washington, DC, USA, 2012), ICST '12, IEEE Computer Society, pp. 260–269.

[266] TOOMIM, M., BEGEL, A., AND GRAHAM, S. Managing duplicated code with linked editing. In *Proceedings - 2004 IEEE Symposium on Visual Languages and Human Centric Computing* (2004), pp. 173–180.

[267] TSANTALIS, N., CHAIKALIS, T., AND CHATZIGEORGIOU, A. JDeodorant: Identification and removal of type-checking bad smells. In *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR* (2008), pp. 329–331.

[268] UDDIN, G., AND ROBILLARD, M. P. How API documentation fails. *IEEE Software 32*, 4 (July 2015), 68–75.

[269] UDDIN, M. S., ROY, C. K., AND SCHNEIDER, K. A. Towards convenient management of software clone codes in practice: An integrated approach. In *Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering* (2015), IBM Corp., pp. 211–220.

[270] UNGAR, D., LIEBERMAN, H., AND FRY, C. Debugging and the experience of immediacy. *Communications of the ACM 40*, 4 (Apr. 1997), 38–43.

[271] VÁCLAVÍK, P., PORUBÄN, J., AND MEZEI, M. Automatic derivation of domain terms and concept location based on the analysis of the identifiers. *Acta Universitatis Sapientiae. Informatica 2*, 1 (2010), 40–50.

[272] VASILESCU, B., SCHUYLENBURG, S. V., WULMS, J., SEREBRENIK, A., AND BRAND, M. G. J. V. D. Continuous integration in a social-coding world: Empirical evidence from GitHub. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution* (Washington, DC, USA, 2014), ICSME '14, IEEE Computer Society, pp. 401–405.

[273] VOELTER, M., SIEGMUND, J., BERGER, T., AND KOLB, B. Towards user-friendly projectional editors. In *Software Language Engineering* (2014), Springer International Publishing, pp. 41–61.

[274] VON MAYRHAUSER, A., AND VANS, A. Program comprehension during software maintenance and evolution. *Computer 28*, 8 (Aug. 1995), 44–55.

[275] WALTERS, B., FALCONE, M., SHIBBLE, A., AND SHARIF, B. Towards an eye-tracking enabled IDE for software traceability tasks. In *Traceability in Emerging Forms of Software Engineering (TEFSE), 2013 International Workshop on* (May 2013), pp. 51–54.

[276] WANG, J., PENG, X., XING, Z., AND ZHAO, W. An exploratory study of feature location process: Distinct phases, recurring patterns, and elementary actions. In *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance* (Washington, DC, USA, 2011), ICSM '11, IEEE Computer Society, pp. 213–222.

[277] WATSON, G., AND DEBARDELEBEN, N. Developing scientific applications using Eclipse. *Computing in Science and Engineering 8*, 4 (2006), 50–61.

[278] WILDE, N., AND CASEY, C. Early field experience with the software reconnaissance technique for program comprehension. In *Software Maintenance 1996, Proceedings., International Conference on* (Nov. 1996), pp. 312–318.

[279] WILDE, N., GOMEZ, J., GUST, T., AND STRASBURG, D. Locating user functionality in old code. In *Software Maintenance, 1992. Proceedings., Conference on* (Nov. 1992), pp. 200–205.

[280] WINKLER, S., AND PILGRIM, J. A survey of traceability in requirements engineering and model-driven development. *Softw. Syst. Model. 9*, 4 (Sept. 2010), 529–565.

[281] WITTE, R., SATELI, B., KHAMIS, N., AND RILLING, J. Intelligent software development environments: Integrating natural language processing with the Eclipse platform. In *Lecture Notes in Computer Science* (2011), vol. 6657 LNAI, pp. 408–419.

[282] WOHLIN, C., RUNESON, P., HÖST, M., OHLSSON, M. C., REGNELL, B., AND WESSLÉN, A. *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated, 2012.

[283] XIE, J., CHU, B., LIPFORD, H., AND MELTON, J. ASIDE: IDE support for web application security. In *Proceedings of the 27th Annual Computer Security Applications Conference* (2011), pp. 267–276.

[284] XU, B., QIAN, J., ZHANG, X., WU, Z., AND CHEN, L. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes 30*, 2 (Mar. 2005), 1–36.

[285] YOO, S., AND HARMAN, M. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability 22*, 2 (2012), 67–120.

[286] ZHANG, H., BABAR, M. A., AND TELL, P. Identifying relevant studies in software engineering. *Information and Software Technology 53*, 6 (2011), 625–637. Special Section: Best papers from the APSECBest papers from the APSEC.

[287] ZHANG, S., ZHANG, C., AND ERNST, M. D. Automated documentation inference to explain failed tests. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering* (Washington, DC, USA, 2011), ASE '11, IEEE Computer Society, pp. 63–72.

[288] ZHENG, Y., CU, C., AND ASUNCION, H. Mapping features to source code through product line architecture: Traceability and conformance. In *Proceedings - 2017 IEEE International Conference on Software Architecture, ICSA 2017* (2017), IEEE, pp. 225–234.

Selected Author's Publications

Important Conferences

1. SULÍR, M., AND PORUBÄN, J. Augmenting source code lines with sample variable values. In *Proceedings of the 2018 26th IEEE/ACM International Conference on Program Comprehension (ICPC)* (May 2018). Accepted.
2. SULÍR, M., AND PORUBÄN, J. RuntimeSearch: Ctrl+F for a running program. In *Proceedings of the 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2017), pp. 388–393.
3. SULÍR, M., AND PORUBÄN, J. Labeling source code with metadata: A survey and taxonomy. In *2017 Federated Conference on Computer Science and Information Systems (FedCSIS)* (Sept. 2017), pp. 721–729.
4. SULÍR, M., AND PORUBÄN, J. Generating method documentation using concrete values from executions. In *6th Symposium on Languages, Applications and Technologies (SLATE 2017)* (Dagstuhl, Germany, 2017), vol. 56 of *OpenAccess Series in Informatics (OASIcs)*, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 3:1–3:13.
5. SULÍR, M., AND PORUBÄN, J. A quantitative study of Java software buildability. In *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools* (New York, NY, USA, 2016), PLATEAU 2016, ACM, pp. 17–25.
6. SULÍR, M., AND PORUBÄN, J. Locating user interface concepts in source code. In *5th Symposium on Languages, Applications and Technologies (SLATE'16)* (Dagstuhl, Germany, 2016), vol. 51 of *OpenAccess Series in Informatics (OASIcs)*, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 6:1–6:9.
7. SULÍR, M., AND NOSÁĽ, M. Sharing developers' mental models through source code annotations. In *2015 Federated Conference on Computer Science and Information Systems (FedCSIS)* (Sept. 2015), pp. 997–1006.

Current Contents Journals

8. SULÍR, M., BAČÍKOVÁ, M., CHODAREV, S., AND PORUBÄN, J. Visual augmentation of source code editors: A systematic review. *Computer Languages, Systems & Structures* (2018). Submitted.
9. NOSÁĽ, M., PORUBÄN, J., AND SULÍR, M. Customizing host IDE for non-programming users of pure embedded DSLs: A case study. *Computer Languages, Systems & Struc-*

tures 49 (2017), 44–65.

10. SULÍR, M., NOSÁĽ, M., AND PORUBÄN, J. Recording concerns in source code using annotations. *Computer Languages, Systems & Structures 46* (2016), 44–65.

Other Journals

11. SULÍR, M., AND PORUBÄN, J. Source code documentation generation using program execution. *Information 8*, 4 (2017), 148.
12. SULÍR, M., AND PORUBÄN, J. Exposing runtime information through source code annotations. *Acta Electrotechnica et Informatica 17*, 1 (2017), 3–9.
13. NOSÁĽ, M., SULÍR, M., AND JUHÁR, J. Language composition using source code annotations. *Computer Science and Information Systems 13*, 3 (2016), 707–729.
14. SULÍR, M., AND ŠIMOŇÁK, S. A terse string-embedded language for tree searching and replacing. *Acta Electrotechnica et Informatica 14*, 2 (2014), 28–35.

Other Conferences

15. SULÍR, M. Facilitating program comprehension with runtime metadata: A Report. In *SCYR 2018: 18th Scientific Conference of Young Researchers* (May 2018). Accepted.
16. SULÍR, M., PORUBÄN, J., AND ZORIČÁK, O. IDE-independent program comprehension tools via source file overwriting. In *2017 IEEE 14th International Scientific Conference on Informatics* (Nov. 2017), pp. 372–376.
17. SULÍR, M. Facilitating program comprehension with source code labeling: A progress report. In *SCYR 2017: 17th Scientific Conference of Young Researchers* (May 2017).
18. SULÍR, M. Improving program comprehension: Preliminary results and research plan. In *SCYR 2016: 16th Scientific Conference of Young Researchers* (May 2016).
19. SULÍR, M., AND PORUBÄN, J. Semi-automatic concern annotation using differential code coverage. In *2015 IEEE 13th International Scientific Conference on Informatics* (Nov. 2015), pp. 258–262.
20. SULÍR, M., AND PORUBÄN, J. Trend analysis on the metadata of program comprehension papers. In *2015 13th International Conference on Engineering of Modern Electric Systems (EMES)* (June 2015), pp. 153–156.
21. SULÍR, M. Program comprehension: A short literature review. In *SCYR 2015: 15th Scientific Conference of Young Researchers* (May 2015), pp. 283–286.