Reverse Engineering Source Code:
Empirical Studies of Limitations and Opportunities

Reverse Engineering Source Code:
Empirical Studies of Limitations and Opportunities

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof. dr. ir. K. I. J. Maex
ten overstaan van een door het College voor Promoties
ingestelde commissie,
in het openbaar te verdedigen in de Agnietenkapel
op donderdag 5 oktober 2017, te 10.00 uur

door

Davy Landman

geboren te Sittard

Promotiecommissie:

Promotores:	prof. dr. P. Klint	Universiteit van Amsterdam
	prof. dr. J. J. Vinju	Technische Universiteit Eindhoven
Overige leden:	prof. dr. J. A. Bergstra	Universiteit van Amsterdam
	dr. C. U. Grelck	Universiteit van Amsterdam
	prof. dr. T. M. van Engers	Universiteit van Amsterdam
	prof. dr. T. Vos	Open Universiteit
	prof. dr. S. Demeyer	Universiteit van Antwerpen
	prof. dr. M. W. Godfrey	University of Waterloo
Faculteit:	Faculteit der Natuurwetenschappen, Wiskunde en Informatica	

Centrum Wiskunde & Informatica

The work in this thesis has been carried out at Centrum Wiskunde & Informatica (CWI) under the auspices of the research school Institute for Programming research and Algorithmics (IPA) and has been supported by the NWO TOPGO grant #612.001.011 "Domain-Specific Languages: A Big Future for Small Programs".

CONTENTS

ACKNOWLEDGMENTS

The main goal of my time as a PhD candidate was learning as much as possible. That is the main reason I accepted a position with Paul and Jurgen. They have taught me many things, of which a few ended up in this thesis.

Paul, thank you for inspiring the main theme in this thesis: "is dat nou wel zo?". We often talked about research and software engineering in a general sense, while leaving me the freedom to explore topics you were sometimes skeptical about. In the end I think we have spent more hours discussing (Rascal) software engineering challenges than we have on research challenges. Thanks for being this beacon of engineering; after too many days of paper-writing, there was always a nice challenge waiting. I will always remember the nice extra curriculum things we undertook: the LEGO Turing machine, session at NEMO, hosting several high school classes, and programming a dancing robot together with a class of 40 young girls.

Jurgen, thank you for supporting my stubbornness and recognizing the stories we wanted to tell. Many of the tackled subjects were as much a learning curve for you as they were for me, I really appreciated your honesty in this. Helping me ignore most of the publication-pressure has been the greatest gift a PhD candidate could wish for. You have shown me how to combine gut feeling and rational thinking into our successful collection of publications. We have had (long) discussions about almost any imaginable subject, and I hope we will continue to do so.

After 6 years I still feel there is much to learn from Paul and Jurgen, so I am very glad we will continue our collaboration in the form of our recently founded company: SWAT.engineering. In this new journey we will undoubtedly learn a lot from each other, while improving the state of software engineering project after project.

Thank you Alexander and Eric for embarking on the journey of publishing a paper together. Alexander, thank you for being more critical than myself, and guiding me through the wilderness of statistical methods. Eric, thank you for showing me how SIG handles the hard questions surrounding metrics.

In my 6 years being part of the SWAT group I have had the fortune of working along side of many colleagues. Properly acknowledging you all in this chapter would run into HUGE printing costs and increase the risk of forgetting something. I therefore take the easy way out and will thank you all for the many discussions and nice outings: Aiko, Alexander, Ali, Anastasia, Angelos, Anya, Arnold, Ashim, Atze, Bas, Bert, Floor, Gauthier, Hans, Jan, Jeroen, Jouke, Kai, Lina, Magiel, Mark, Mauricio, Michael, Mike, Mircea, Oscar, Pablo, Riemer, Robert, Rodin, Sunil, Thomas, Tijs, Tim, Vadim, Yanja, and many master students. Thank you for all the pair programming, teaching me humility, showing me the multi-cultural world we live in, out-geeking me, and leaving me flabbergasted about subjects I know nothing about.

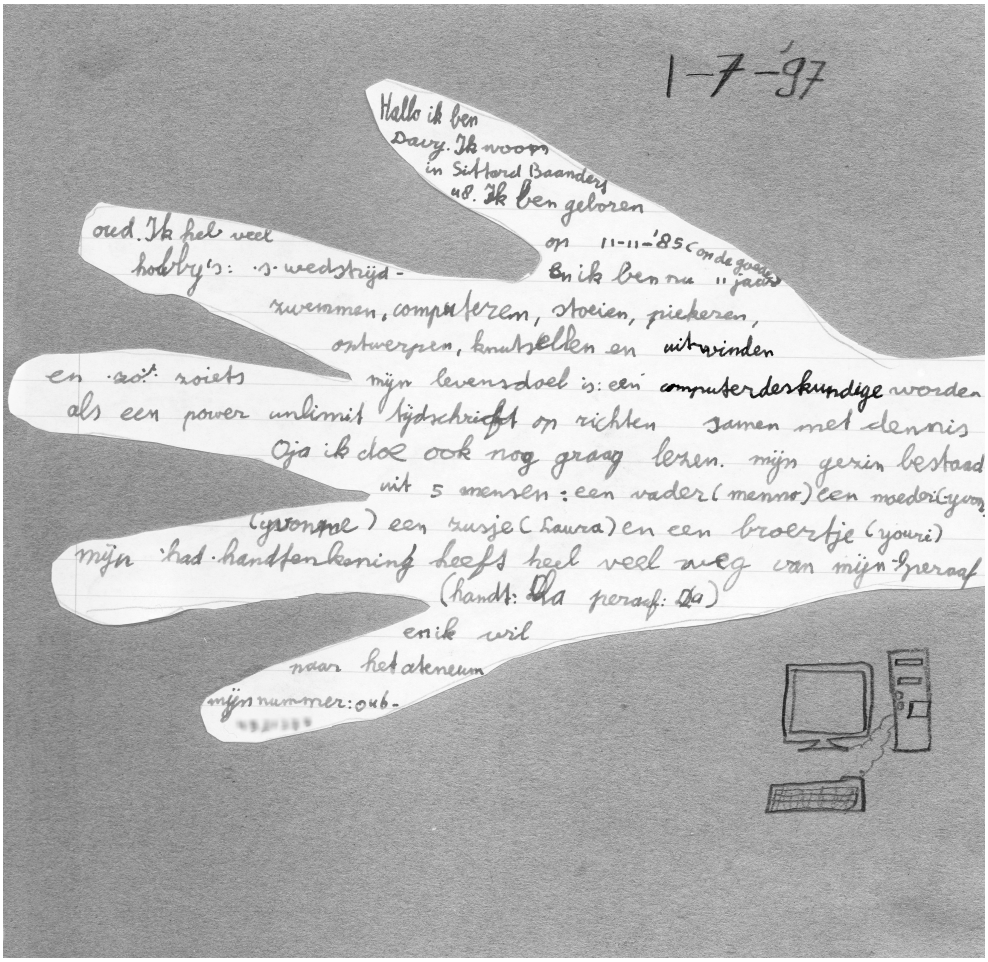
I have asked Jeroen and Wietse to be my paranymphs. One of the reasons is that they most frequently asked me why I was doing a PhD. Whereafter they stayed around for the discussion on where to next. Jeroen, thank you for making sure I was never the biggest nerd at CWI, all the burgers, and joining most engineering quests I proposed. Wietse, thank you for expanding my horizons outside software engineering, providing a much needed mirror to my assumptions, and teaching me how to stop worrying and make a choice.

The research community has been very friendly in accepting me and my strange questions. I want to thank the committee for reviewing and accepting my thesis.

My parents have always accepted my strange or skeptical questions. Thank you for letting me simmer in them, so that at a later point in life I could finally use them. Laura, thank you for teaching me much about how to have discussions. Youri, thank you for the many distractions we managed to squeeze in. I will always remember the beers we shared, the people we met, and our strange journeys back home. I have missed birthdays and dinners due to conferences and deadlines, thank you: Mam, Pap, Anke, Jeffrey, Laura, Linda, Rina, Taco, Theo, and Youri for being very accommodating.

Lastly, my own family. Petra, you have been my rock, it seems your parents were very foreseeing. For every chapter in this thesis, there was a point where our discussion solved a deadlock that neither my supervisors or my coauthors could break. In helping me finish this thesis, you have taken over more of our household chores than I would like to admit. Thank you for this. I will make sure to repay this with great cooking and a Davy-biased chore distribution. The greatest gift has been a new purpose in my life.* You and Tom have shown me how there is more in life than software engineering and have given me a new challenge that will last for the rest of my life, AWESOME!

*Sorry, some cliches are just too true.



Short English summary: 20 years ago I already liked inventing stuff and aspired to become a computer expert.

LIST OF ABBREVIATIONS

API	Application Programming Interface
AST	Abstract Syntax Tree
CC	Cyclomatic Complexity
DSL	Domain-Specific Language
IDE	Integrated Development Environment
IR	Information Retrieval
JDT	Java Development Tools
LLOC	Logical Lines of Code
LOC	Lines of Code
MVC	Model View Controller
NLP	Natural Language Parsing
OO	Object Oriented
ORM	Object-Relational Mapping
PMBOK	Project Management Body of Knowledge
PMI	Project Management Institute
RMR	Repeated Median Regression
SAT	Software Analysis Toolkit
SCM	Source Code Management
SLOC	Source Lines of Code
SLR	Systematic Literature Review
SPS	Software Projects Sampling
UI	User Interface
UML	Unified Modeling Language
WMC	Weighted Methods per Class

1 INTRODUCTION

The goal of *software renovation* is to modernize existing software [CC90; vDKV99]. Modern software tools can be used to refresh aging software to better match its technical and business environment. The overarching motivation for this thesis is providing better methods and tools to software maintenance teams for renovating their software.

There are two general approaches to software renovation [vDKV99]. The first approach is to transform the software system to a new version without raising the level of abstraction. The second approach is to first reverse engineer [CC90] higher level abstractions from the existing system, and transform these to a new improved system. This approach is called *re-engineering*.

We focus on using reverse engineering to support re-engineering by extracting a higher level of abstraction than the current level. We explore the feasibility to recover domain models from source code, explore the relationship between two very common source code metrics, and explore one of the limits of static analysis. This chapter introduces the concepts, research questions, research methods, contributions, and the global structure of this thesis.

1.1 REVERSE ENGINEERING

Reverse engineering is a broad term. Chikofsky and Cross formulated the following most commonly used definition of reverse engineering: "Reverse engineering is the process of analyzing a subject system to: identify the system's components and their interrelationship, and create representations of the system in another form or at a higher level of abstraction" [CC90]. More recently, Tonella et al. broadened this to: "every method aimed at recovering knowledge about an existing software system in support to the execution of a software engineering task" [TTB+07].

Chikofsky and Cross [CC90] identified the following key objectives:

Cope with complexity: automate evolution of software [Leh80] to deal with the growing volume and complexity of a system.

Generate alternate views: automatically create graphical and non-graphical models of the system.

Recover lost information: rediscover knowledge lost in the evolution of a long-lived system.

Detect side effects: automatically detect anomalies and problems.

Synthesize higher abstractions: construct alternate views (or models) that describe the system at a higher level, opens up opportunities for generating code.

Facilitate reuse: detect reusable software components in existing systems.

Reverse engineering research has introduced methods to achieve one or more of these objectives. A method is a description of how to use certain information to support a software engineering task. Tonella et al. [TTB+07] published a non-exhaustive overview of reverse engineering methods. The following list shows examples of popular methods that can be applied to support reverse engineering:

Code visualisation: illustrate the actual source code by adding graphical marks or converting it to a more graphical format [Mye90]. The information illustrated comes from a different method.

Design recovery: recreate design abstractions by combining informal information, existing documentation, and source code [Big89].

Traceability: linking (sections of) source code to other artifacts such as requirements, documentation, models, and visualizations [ACC+02].

Impact analysis: assessing the effect of a change to one or more elements in the system [TM94].

Slicing: extract the part of a program that affect values computed at some point of interest [Tip95; Wei79].

Concept assignment: discover concepts (programming or human) or other concerns and relate them to source code [BMW93]. Feature location [DRG+13] is a popular instance of the second half of the concept assignment problem: where is a given feature located?

There are a whole range of reverse engineering methods, and most are (partially) automated. Automating reverse engineering is necessary to scale to larger software systems. This automation often comes at the cost of over- or underapproximation. The research questions of the following section explore the limits of these approximations.

1.2 RESEARCH QUESTIONS

The research published in this thesis shares a common thread: reverse engineering knowledge from the source code of software systems. The first question explores the limits of domain model recovery (an instance of concept assignment) by manually recovering models. In trying to automate this recovery, we identified challenges that hold for a wider range of reverse engineering methods than just domain model recovery. The second and third questions explore these challenges in the broader context of reverse engineering.

Here we will introduce our three research questions, relevant background knowledge, used the research methods and the obtained results. To answer these research questions we use the same empirical research method, which will be discussed in Section 1.3.

Listing 1.1: This constructed example of a button click handler shows how challenging it can be to recover domain models from source code. There are only 3 lines (💡) which might document a new domain concept or relation. The other lines are related to database access and user interface logic.

```
1  public void handleSaveButtonClick() {
2      try (Transaction trans = transactions.acquire()) {
3          int iterationId = iterationSelection.getIndex();
4          if (iterationId < -1) {
5              throw new SelectIterationException();
6          }
7          Iteration it = database.iterationById(iterationId);
8          User newUser = database.userById(userSelection.getIndex());
9          if (newUser.project == iteration.project) {          💡
10             if (it.assigned != null) {
11                 it.unassign(it.assigned);                    💡
12             }
13             it.assign(newUser);                              💡
14         }
15         labelSuccess.text = newUser.getName() + " was assigned";
16     }
17     catch (Exception e) {
18         trans.revert();
19         labelError.text = "Failure: " + e;
20     }
21 }
```

1.2.1 *RQ1: Exploring the limits of domain model recovery*

Throughout the lifetime of a software system, domain models (defined below) are needed to support the software maintenance team in its work. When domain models are missing or outdated, they might be recoverable from source code. Listing 1.1 contains an example of which parts of the source code contain recoverable domain knowledge. The first research question (defined below) explores how much we can learn about the domain of a software system by analyzing its source code.

Background

Domain Model Software is constructed to automate an activity or support an interest of its stakeholders. The area that the software covers is its domain. Example domains are project planning, human resource management, order management, online booking, accounting, application life cycle management, etc. Software development teams translate their understanding (or knowledge) of this domain into source code, that after compilation, a computer executes. A *domain model* is defined by Evans as

"a rigorously organized and selective abstraction of that knowledge" [Eva03]. These domain models are the explicit representations of domain knowledge.

For every domain, there may be multiple models. Models are a way to solidify knowledge from a specific angle. Sometimes, for a new information requirement, new and different models have to be constructed. For example: a Unified Modeling Language (UML) Class Diagram [RJB99] or an Entity Relation Diagram [Che76] can be used to describe the entities and their interrelations. Likewise, a UML State Machine [RJB99] can be used to model a process.

Recovery During the maintenance of a software system, knowledge of the domain is often required to add new features or fix bugs. Maintenance teams lose domain knowledge, either by the passing of time or staff turnover. *Outdated* models could be available, or the knowledge was never crystallized into models to begin with. Before performing most maintenance tasks, the maintainer needs to understand a subset of the domain to use as a frame of reference. When this subset is unfamiliar, it will have to be recovered somehow.

Domain models can be recovered by conducting interviews (with stakeholders or original developers), reading documentation, or reading the source code. Interviews are often necessary during the recovery of domain models. However, since they involve humans, they are sensitive to inaccuracy, incompleteness, and subjectivity. Other information will be needed to triangulate more objective knowledge. Existing documentation can be a useful source, however, this documentation is often outdated [LSF03]. Source code is yet another suitable source of information.

Recovering knowledge from source code has potential benefits. Since it is the source code of the currently running system, it is more objective and complete. Reading all source code is infeasible, but large parts of the source code can be processed automatically at relatively low costs. However, in the translation from the developer's knowledge to source code, both the intent and the context can get lost. This (possible) loss of domain knowledge in the translation to source code motivates the first research question.

Research question

We know that recovering design information – such as domain models – can be hard since source code lacks relevant information [Big89]. It may be easier to recover the information by other means; especially for legacy applications written in low level languages that lack the opportunity for design clues. However, how about the software written today that soon turns into legacy applications? Tomorrow this software will also require reverse engineering [vDKV99]. The first research question tries to find the *upper limit* of reverse engineering domain models from software written today.

Research Question 1 (RQ1)

How much of a domain model can be recovered from source code under *ideal* circumstances?

The upper limit is important, as it both frames and motivates the future work in automating domain model recovery.

Method

The ideal circumstance for recovering domain models is inspired by the Object Oriented (OO) methodology. A common practice in OO is to model concepts of the world using objects (made even more popular by Evans in Domain Driven Design [Eva03]). Certain popular libraries and patterns – such as Object-Relational Mapping (ORM) libraries and the Model View Controller (MVC) pattern – promote the construction of a domain model inside the source code even more. We therefore selected software systems (from the project planning domain) implemented in an OO language, and further selected those that either used the MVC pattern or used an ORM library. These systems at the very least have some kind of model in their source code. Hereby creating the highest chance of recovering their full domain model.

To measure the quality of the recovered domain models, we need an oracle. An oracle classifies relations or concepts in a domain model as either correct or incorrect. Actually, we need two oracles. The first oracle measures how much of a domain can be learned by reading the source code of the program in that domain. This oracle has to be constructed outside the context of any software. The second oracle is used to measure how much of a domain the program actually covers. To approximate the domain of the program, this oracle has to be constructed from the users perspective. Other views on the program are too closely related to source code.

These oracles have to be manually constructed, otherwise they would reflect the quality – or inaccuracy – of the tools that constructed them. The first oracle is based on the Project Management Body of Knowledge (PMBOK) book [Ins08] by methodically translating key sentences to a domain model of project planning. For the second oracle all screens of the application's user interface were manually traversed and translated to a domain model. Chapter 2 answers RQ1 by manually recovering domain models from the source code of two software systems and comparing them to the manually constructed reference models (oracles) to measure the precision and recall. Precision and recall are two appropriate relevancy measures in case of binary classification.

Result

For the two systems used in the study, most information can be recovered. Reading source code of an application can teach us about its domain, with comparable quality as traversing the user interface of the application.

As already mentioned, manual reverse engineering does not scale to larger software systems. Most reverse engineering methods automatically gather information from source code (or other inputs) and present that to the user for further improvements. What are the challenges for automating this recovery?

1.2.2 RQ2: *Exploring the relationship between Cyclomatic Complexity and Lines of Code*

While trying to automate the manual recovery of Chapter 2 (RQ1) we observed that complex code tended to explain more about the relationship and interpretation of concepts than less complex code fragments. This suggested that code complexity metrics could be used to identify code fragments of interest. Software metrics [FB14] are used in a wide variety of reverse-engineering methods to filter methods or files of interest [DDL99; PSR+05]. Two common complexity metrics are Source Lines of Code (SLOC) and Cyclomatic Complexity (CC) (defined in the following background subsection). Listing 1.2 contains an example method annotated with these two metrics. SLOC and CC appear in every available commercial and open-source source code metrics tool, for example: `http://sonarqube.org`, `http://ndepend.com`, and `http://grammatech.com/codesonar`. They are commonly used next to each other in software assessment [HKV07] and fault/error prediction [FO00].

On the other hand, the general conclusion of experimental studies [BP84; FF79; JMF14; SCM+79] on the relationship between CC and SLOC is that they have a strong linear correlation. This linear correlation is often interpreted as the reason to discard CC for the simpler to calculate SLOC [SCM+79], or to normalize CC for SLOC [EBG+01]. The relevance of our second research question is much wider than recovering domain models. For this study we specifically analyze the linear correlation between SLOC and CC. Given that we are still using them both next to each other, is this correlation present?

Background

The term software metrics can be used for multiple software measurement activities [FB14]. Examples are: effort, quality, security, and complexity measurement. In general, software metrics measure an attribute of interest. What are software metrics and how can they be used?

Listing 1.2: Example of a Java method that approximates the square root. Out of the 10 lines in this listing, the SLOC measure counts 7 of them (</>). The CC of this method is 2 (🔱), in the control flow graph there is a path including the `while` body, and one that does not.

```
1  public static double sqrt(double num, double epsilon) {          </> 🔱
2      double result = num / 2.0;                                    </>
3
4      // repeat newton step until precision is achieved
5      while (abs(result - (num / result)) > (epsilon * result)) { </> 🔱
6          result = 0.5 * (result + (num / result));                 </>
7      }                                                             </>
8
9      return result;                                                </>
10 }                                                                 </>
```

Fenton and Bieman use measurement theory to explain what measurements are:

> Formally, we define *measurement* as the mapping from the empirical world to the formal, relational world. Consequently, a *measure* is the number or symbol assigned to an entity by this mapping in order to characterize an attribute. [FB14, p. 30]

For example in software measurement, observable properties of software, such as the size of source code are mapped to the number of lines measure. However, the relation between the attribute of interest, for example maintainability, and the observable property is not always agreed upon. This is especially the case when the attribute reflects a personal preference.

Measurement theory further describes relations between different measurements of the same property, for example that if A is larger than B, and B is larger than C, is C also larger than A? Further details of measurement theory are outside of the scope of this introduction and we refer to the Software Metrics book by Fenton and Bieman [FB14].

We primarily use software metrics (or just metrics) as a way to measure the same attribute in different ways. The common attribute is complexity, and we look at how the values of SLOC related to the values of CC.

Lines of Code Larger methods or files are harder to understand due to the amount of context the reader has to keep in mind while reading them. One of the most common measures of size is the Lines of Code (LOC) software metric. While in essence a simple software metric, the interpretation of what should count as a line varies.

In general, there are two categories of LOC [Par92]. The *physical* LOC measure describes the physical length of the code for people to read it. The *logical* LOC (LLOC)

measure ignores physical layout and counts instructions. The SEI technical report by Park [Par92] discusses the many factors that influence both kinds of LOC measures. For example how comments, generated code, cloned code, blank lines, and non executable code should be counted.

The LLOC measure ignores formatting of code and counts only certain categories of tokens in the source code. The common argument is to remove the noise caused by different coding styles of developers. It is however harder to compare to other languages, and to other tools that measure LLOC in a slightly different way.

For physical LOC there exist a few common approaches. They differ primarily in how to handle comments, white space, and single curly braces. In general, LOC counts *all* newlines. After LOC the most popular physical measure is SLOC. The SLOC measure ignores comment and blank lines. The definition of SLOC is as follows:

> A line of code is any line of program text that is not a comment or blank line, regardless of the number of statements or fragments of statements on the line. This specifically includes all lines containing program headers, declarations, and executable and non-executable statements [CDS86, p. 35].

Cyclomatic Complexity Control flow is another aspect of complexity. A commonly used measure of control flow is Cyclomatic Complexity (CC) [McC76]. CC counts the independent paths in a control flow graph, and while initially introduced to estimate the amount of test cases needed, it has been widely applied for different measurement goals. McCabe defined CC as follows:

> The cyclomatic complexity of a program* is the maximum number of linearly independent circuits in the control flow graph of said program, where each exit point is connected with an additional edge to the entry point [McC76].

The definition is based on the control flow graph of a program, which is more complicated to calculate than merely parsing the source code. McCabe therefore also suggested counting the statements that cause forks in the control flow graph. This simpler approach is the more popular way to calculate CC.

Simply counting certain statements introduces discussion on which statements to count. This discussion is primarily on how to handle short circuiting boolean operators that cause forks in the control flow graph. This has even caused the proposal of an extended CC measure which explicitly mentions the short circuiting boolean operators [Mye77]. However, the original definition was sufficiently general, any statement that creates a new path in the control flow graph increments the value of CC. The work of Abran [Abr10] contains an in-depth discussion on CC's semantics.

*In this context a "program" means a subroutine of code like a procedure in Pascal, function in C, method in Java, sub-routine in Fortran, program in COBOL.

Research question

The duality between the popularity and the reported redundancy between SLOC and CC – as already mentioned above – motivated the second research question. Before we try to use these metrics for recovering domain models or in any other reverse engineering method, we first have to understand how they are related to each other:

Research Question 2 (RQ2)

Is there a strong linear (Pearson) correlation between CC and SLOC metrics?

How to answer this question in the context of all the related work that does report a linear correlation?

Method

First a Systematic Literature Review (SLR) is performed to collect all related work that contributes data to this discussion. Using the SLR new hypotheses are formulated on why this correlation is reported and which other factors might explain it. The CC and SLOC are then measured on two large corpora of Java and C software (which had to be constructed), and statistically analyzed for the different hypothesis.

Result

Contrary to all related work, we only found a moderate correlation in Chapter 3, and identified several statistical problems with the claimed relation. After identifying possible – statistically incorrect – transformations of the data that could explain the observations in related work, the reported high correlations could be reproduced. We concluded that we did not find evidence of CC being redundant to SLOC, and that they can continue to be used next to each other.

1.2.3 *RQ3: Exploring the limits of static analysis and reflection*

The SLOC and CC metrics can be calculated on just syntactical information of the source code. More complicated reverse engineering queries on the source code require more than purely syntactic information, these queries require an abstraction of the source code's semantics. Static analysis enables these more complicated queries that reason about the code's semantics. The accuracy of a static analysis can be decreased by dynamic behavior; Java's Reflection Application Programming Interface (API) offers this dynamic behavior. Listing 1.3 contains an example Java method that showcases how dynamic reflective methods can get. How much does reflection affect static analysis methods?

Listing 1.3: Constructed example of a Java method that uses reflection in a way that complicates static analysis. If a static analysis wants to understand which methods can be invoked on line 9, it has to model the effects of the control flow and related parts of the Reflection API. The complicating statements are annotated with the ϟ symbol.

```
1  public String applyFilter(Class<?> klass, String prefix, String[] toFilter) {
2      Method[] candidates = klass.getMethods();                                ϟ
3      for (Method m: candidates) {
4          if (m.getName().startsWith(prefix)) {                                ϟ
5              Parameter[] params = m.getParameters();                          ϟ
6              if (params.length > 0
7                  && params[0].getType().isAssignableFrom(String.class)) {     ϟ
8                  try {
9                      return (String) m.invoke(null, toFilter[0], toFilter);
10                 }
11                 catch (ReflectiveOperationException e) {
12                     // try next candidate                                    ϟ
13                 }
14             }
15         }
16     }
17     return toFilter[0];
18 }
```

Background

There are two flavors of analyzing semantics: *dynamic* and *static* analysis. They can be used separately or combined. However, they do differ and have their own weaknesses.

A dynamic analysis executes the source code (or the binary in case of a compiled language) in one or more runs, and gathers information of the behavior of interest during its execution. Dynamic analysis has high precision, all reported facts are correct, since they are based on observations of the actual running program. However, the recall of dynamic analysis is influenced by the offered input to execution of the source code, certain parts of the source code can be completely missed. Increasing this coverage automatically remains challenging.

Static analysis tries to reason about the effect of source code without actually executing it. There are a whole range of static analysis methods and a wide variety of users of static analysis methods. Name binding for example connects identifiers to points on the heap and stack with either data or code. A compiler uses this name binding to generate the application that manipulates data and executes code. Static analysis has multiple trade-offs, an important trade-off is between soundness and performance. A static analysis is sound when all the behaviors that can occur in the runtime are contained in its result, or in other words, no false negatives. However,

to achieve this, static analysis tools make over-approximations, which cause false positives. These over-approximations are often costly in the performance of both the tool (memory and CPU usage) and the user of the tool (many false positives to ignore).

Research question

Static analysis is used in a wide range of research challenges, for example: security analysis [BBC+06; CM04], refactoring [MT04], and finding bugs [AHM+08]. However, even for statically typed languages – which should be easy to analyze – there are limitations. Dynamic language features – such as the Reflection API in Java – represent one of these limitations. When just one instance of them is present in the source code of a system, it can hurt the global recall and precision of a static analysis tool.

Only in the last 10 years research has suggested heuristics to handle Java's dynamic language features in a pragmatic, unsound, way [LWL05]. How much of current – real world – Java source code can be handled? And which challenges remain? Therefore, the third and final research question is:

Research Question 3 (RQ3)

What are the limits of state-of-the-art static analysis tools supporting the Reflection API and how do these limits relate to real world Java code?

Method

First we constructed an overview of how the interesting parts of the Reflection API are used. Similar to the research method of RQ2 we used a SLR to collect all the related work. This SLR identifies common limitations, which are then translated into patterns that match violations of the limitations. After constructing a new representative corpus of Java software, we use the meta-programming language Rascal [KvdSV09] to scan for occurrences of the patterns in this corpus.

Result

The dynamic part of the Reflection API is used in 80% of all projects in the corpus. Certain limitations of static analysis are relatively often breached by normal Java systems. We propose patterns for software engineers to simplify their source code, and propose new assumptions and heuristics for static analysis tools to handle these limitations.

1.3 RESEARCH METHOD

For all of our three research questions, we have applied empirical research methods. What is empirical research? And how do we mitigate threats to the validity of our conclusions?

1.3.1 *Background*

For software engineering research Basili [Bas93] and Glass [Gla94] summarized four research methods:

Scientific observe the world, model it, measure it, analyze it, and validate hypotheses.

Engineering observe existing solutions, propose improvements, develop them, measure and analyze the effect, and evaluate the improvement.

Empirical propose a model, evaluate it using empirical studies.

Analytical propose a formal theory, develop a theory, derive results, and compare with empirical observations.

Depending on the kind of research questions posed, one or more of these methods are suited. The empirical method – popular in social science and psychology – can be a better fit for questions on how software is *engineered*. While the analytical method is better suited for questions on how to *engineer* software. For example, the analytical method is suited for exploring the best implementation of an ORM framework, but it is less suited for exploring how developers actually implement it.

Software engineering, in the end, is human-intensive, based on the intelligence and creativity of people [WRH+12]. Developing a theory on how humans think is infeasible, therefore, it would be hard to apply the analytical method to understand how developers implement something. We know that given a programming language and a set of requirements, there are multiple possibilities to implement them (even using the same set of libraries). Using the empirical method we can investigate which possibilities occur "in the wild".

While an empirical study can take many forms, the validity of the conclusions of these studies depends on design choices of the research method. For empirical research, the common classification of threats to this validity are [WRH+12]:

Conclusion validity the statistical method applied to the data is correct.

Internal validity the observed effect is not caused by unknown or uncontrolled variables; there are no unknown biases.

Construct validity the observed effect can be explained by theory; all inferences are made on valid measurements or observations.

External validity the results can be generalized to other settings.

1.3.2 *Mitigating threats to validity*

In Chapter 2 (RQ1) we performed the task of domain model recovery for two software systems. To control for bias (internal validity) we performed the study by hand and traced every step of the modeling to its origin. To minimize the threats to construct and external validity we explicitly framed our results as a exploration of the limitations of the ideal case, and base all conclusions on directly observable data.

In Chapter 3 (RQ2) we performed a large study on the relationship between SLOC and CC. To improve the representatives of the results (external validity), we used two large corpora. To reduce the threats to conclusion validity, we explored multiple statistical analyses, and discussed statistical assumptions explicitly. Since our large corpora could introduce new threats to internal validity by containing unknown biases, we mitigated this by performing a sensitivity analysis on random subsets of the corpora. Due to the setup of our study and to avoid threats to construct validity, our conclusion is subtle: "We do not conclude that CC is redundant to SLOC".

In Chapter 4 (RQ3) we performed a large study on the presence of reflection in Java software systems. To avoid the *internal* validity threats caused by large corpora while keeping the advantages of large corpora to reduce *external* validity threats, we constructed a new compact yet diverse corpus of Java systems. Again, we mitigate threats to construct validity by linking the conclusions and corresponding hypotheses to included observations and results.

Moreover, the following mitigations for threats to validity were shared for at least two of the chapters:

Take random samples of data in case of large datasets it is hard to avoid unknown biases (internal validity). Random sampling can unearth certain common yet unknown biases. This is especially important in case of unexpected observations.

Clean data with care even after selecting a data source such as Sourcerer [LBN+09], follow a structured process to remove artifacts that could introduce bias (internal validity). This bias is often the result of using data that was meant for a different purpose, for example, projects that contain the source code of their dependencies to simplify the compilation.

Publish all data such that other researchers can use this data to test for new suspected threats to internal or construct validity. A positive side effect of this is that other empirical research can reuse this data.

Automate the analysis and publish it again, other researchers can repeat our analysis, on the same data set to check for internal validity, or on a new set of data, to test external validity.

1.4 CONTRIBUTIONS

This section lists and summarizes the peer-reviewed contributions, and explains how they are translated to the chapters in this thesis. I was the primary author of these four publications.

1.4.1 *Chapter 2 Exploring the limits of Domain Model Recovery*

> P. Klint, D. Landman, and J. J. Vinju. "Exploring the Limits of Domain Model Recovery". In: *2013 IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, September 22-28, 2013*. IEEE Computer Society, Sept. 2013, pp. 120–129. DOI: 10.1109/ICSM.2013.23

Chapter 2 answers RQ1 by manually recovering domain models from source code and comparing them to manually recovered reference domain models. We observe that for modern software, most concepts of the domain can be recovered, while the relationships between concepts remain hard to recover.

1.4.2 *Chapter 3 Exploring the relationship between SLOC and CC*

> D. Landman, A. Serebrenik, and J. J. Vinju. "Empirical Analysis of the Relationship between CC and SLOC in a Large Corpus of Java Methods". In: *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*. IEEE Computer Society, 2014, pp. 221–230. DOI: 10.1109/ICSME.2014.44
>
> D. Landman, A. Serebrenik, E. Bouwers, and J. J. Vinju. "Empirical analysis of the relationship between CC and SLOC in a large corpus of Java methods and C functions". In: *Journal of Software: Evolution and Process* 28.7 (2016), pp. 589–618. DOI: 10.1002/smr.1760

Chapter 3 answers RQ2 by creating an overview of all related work on the relationship, identifying differences, constructing two large corpora, and analyzing the relationship between SLOC and CC in these corpora. Contrary to related work, we did not conclude that CC is redundant with SLOC, except after questionable data transformations.

In our initial publication [LSV14] we only analyzed the relationship for Java software systems, and performed only a simple literature study. After we have presented this work at ICSME2014, new questions from peer researchers emerged. We therefore extended our study of this relationship [LSB+16] with a more extensive literature study, a new large corpus of C software, analysis of the relationship between SLOC and CC for C, new hypotheses for the higher correlation in related work, and an analysis of the effect of corpus size. In Chapter 3 these two publications are

merged since the second is an extension of the first, and we want to avoid unnecessary duplication.

1.4.3 *Chapter 4 Exploring the limits of static analysis and reflection*

> D. Landman, A. Serebrenik, and J. J. Vinju. "Challenges for static analysis of Java reflection: literature review and empirical study". In: *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*. Ed. by S. Uchitel, A. Orso, and M. P. Robillard. IEEE, 2017, pp. 507–518. DOI: 10.1109/ICSE.2017.53
>
> This paper was awarded the Distinguished Paper Award of the Technical Research Papers track.

Chapter 4 answers RQ 3 by creating an overview of all related research on reflection and static analysis, analyzing common assumptions and limitations, building a representative corpus of Java software, analyzing how reflection is used, and analyzing how often common assumptions or limitations are violated. We found that in Java, reflection is used in almost 80% of the projects, and that certain common limitations occur often. We formulated advise for software engineers on how to avoid these scenarios, and new assumptions for static analysis tools to handle them.

1.4.4 *Datasets*

Every research question required and generated new data. The following data has been made available:

- A reference domain model of project planning and two sets of domain models manually extracted from two project planning applications (Chapter 2)
  D. Landman. *cwi-swat/project-planning-domain*. Apr. 2013. DOI: 10.5281/zenodo.208212
- A curated version of the Sourcerer corpus [LBN+09] with 13 K projects and 362 MSLOC Java (Chapter 3)
  D. Landman. *A Curated Corpus of Java Source Code based on Sourcerer (2015)*. Feb. 2015. DOI: 10.5281/zenodo.208213.
- A corpus of C packages based on the Gentoo distribution with 9.8 K packages and 186 MSLOC C (Chapter 3)
  D. Landman. *A Large Corpus of C Source Code based on Gentoo packages*. Feb. 2015. DOI: 10.5281/zenodo.208215.
- A representative corpus of Java projects representing the Ohloh universe, 461 projects with 79.4 MSLOC Java (Chapter 4)

D. Landman. *A corpus of Java projects representing the 2012 Ohloh universe*. Mar. 2016. DOI: 10.5281/zenodo.162926

As discussed in Section 1.3.2 we mitigated the threat to internal validity by also publishing the source code of our automated analysis. The following source code has been published online:

Chapter 2: D. Landman. *cwi-swat/project-planning-domain*. Apr. 2013. DOI: 10.5281/zenodo.208212

Chapter 3: D. Landman. *cwi-swat/jsep-sloc-versus-cc*. Feb. 2015. DOI: 10.5281/zenodo.293795

Chapter 4: D. Landman. *cwi-swat/static-analysis-reflection*. Oct. 2016. DOI: 10.5281/zenodo.163326

These datasets and the scripts that analyzed them have been published on CERN's research data repository Zenodo. They can easily be downloaded and used in other research. The publication that introduced the dataset contains a detailed discussion on its construction.

1.5 THESIS STRUCTURE

As introduced in this chapter, the following three chapters each answer one main research question related to reverse engineering. Chapter 5 summarizes the conclusions of these chapters and discusses future work.

2 EXPLORING THE LIMITS OF DOMAIN MODEL RECOVERY

Abstract

We are interested in re-engineering families of legacy applications towards using Domain-Specific Languages (DSLs). Is it worth to invest in harvesting domain knowledge from the source code of legacy applications?

Reverse engineering domain knowledge from source code is sometimes considered very hard or even impossible. Is it also difficult for "modern legacy systems"? In this chapter we select two open-source applications and answer the following research questions: which parts of the domain are implemented by the application, and how much can we manually recover from the source code? To explore these questions, we compare manually recovered domain models to a reference model extracted from domain literature, and measured precision and recall.

The recovered models are accurate: they cover a significant part of the reference model and they do not contain much junk. We conclude that domain knowledge is recoverable from "modern legacy" code and therefore domain model recovery can be a valuable component of a domain re-engineering process.

2.1 INTRODUCTION

There is ample anecdotal evidence [MHS05] that the use of DSLs can significantly increase the productivity of software development, especially the maintenance part. DSLs model expected variations in both time (versions) and space (product families) such that some types of maintenance can be done on a higher level of abstraction and with higher levels of reuse. However, the initial investment in designing a DSL can be prohibitively high because a complete understanding of a domain is required. Moreover, when unexpected changes need to be made that were not catered for in the design of the DSL the maintenance costs can be relatively high. Both issues indicate how both the quality of domain knowledge and the efficiency of acquiring it are pivotal for the success of a DSL based software maintenance strategy.

In this chapter we investigate the source code of existing applications as valuable sources of domain knowledge. DSLs are practically never developed in green field situations. We know from experience that rather the opposite is the case: several comparable applications by the same or different authors are often developed before

This chapter was previously published as: P. Klint, D. Landman, and J. J. Vinju. "Exploring the Limits of Domain Model Recovery". In: *2013 IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, September 22-28, 2013*. IEEE Computer Society, Sept. 2013, pp. 120–129. DOI: 10.1109/ICSM.2013.23

we start considering a DSL. So, when re-engineering a family of systems towards a DSL, there is opportunity to reuse knowledge directly from people, from the documentation, from the User Interface (UI) and from the source code. For the current chapter we assume the people are no longer available, the documentation is possibly wrong or incomplete and the UI may hide important aspects, so we scope the question to recovering domain knowledge from source code. Is valuable domain knowledge present that can be included in the domain engineering process?

From the field of reverse engineering we know that recovering this kind of design information can be hard [Big89]. Especially for legacy applications written in low level languages, where code is not self-documenting, it may be easier to recover the information by other means. However, if a legacy application was written in a younger object-oriented language, should we not expect to be able to retrieve valuable information about a domain? This sounds good, but we would like to observe precisely how well domain model recovery from source code could work in reality. Note that both the quality of the recovered information and the position of the observed applications in the domain are important factors.

2.1.1 *Positioning domain model recovery*

One of the main goals of reverse engineering is *design recovery* [Big89] which aims to recover design abstractions from any available information source. A part of the recovered design is the domain model.

Design recovery is a very broad area, therefore, most research has focused on sub-areas. The *concept assignment problem* [BMW93] tries to both discover human-oriented concepts and connect them to the location in the source code. Often this is further split into *concept recovery** [CG07; KDG07; LRB+07], and *concept location* [RW02]. Concept location, and to a lesser extent concept recovery, has been a very active field of research in the reverse engineering community.

However, the notion of a concept is still very broad and *features* are an example of narrowed-down concepts and one can identify the sub-areas of *feature location* [EKS03] and *feature recovery*. *Domain model recovery* as we will use in this chapter is a closely related sub-area. We are interested in a pure domain model, without the additional artifacts introduced by software design and implementation. The location of these artifacts is not interesting either. For the purpose of this chapter, a domain model (or model for short) consists of entities and relations between these entities.

Abebe et al.'s [AT10; AT11] *domain concept extraction* is similar to our sub-area. As is Ratiu et al.'s [RFJ08] *domain ontology recovery*. In Section 2.9 we will further discuss these relations.

*Also known as *concept mining*, *topic identification*, or *concept discovery*.

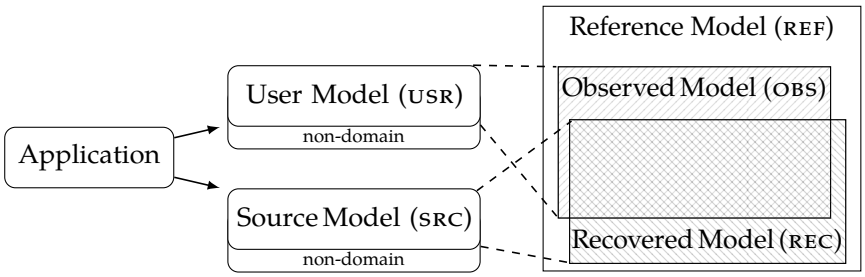


Figure 2.1: Domain model recovery for one application.

2.1.2 *Research questions*

To learn about the possibilities of domain model recovery we pose this question: how much of a domain model can be recovered under *ideal* circumstances? By ideal we mean that the applications under investigation should have well-structured and self-documenting object-oriented source code.

This leads to the following research sub-questions:

SQ1. Which parts of the domain are implemented by the application?
SQ2. Can we manually recover those implemented parts from the object-oriented source code of an application?

Note that we avoid automated recovery here because any inaccuracies introduced by tool support could affect the validity or accuracy of our results.

Figure 2.1 illustrates the various domains that are involved: The *Reference Model* (REF) represents all the knowledge about a specific domain and acts as oracle and upper limit for the domain knowledge that can be recovered from any application in that domain. The *Recovered Model* (REC) is the domain knowledge obtained by inspecting the source code of the application. The *Observed Model* (OBS) represents the part of the reference domain that an application covers, i.e. all the knowledge about a specific application in the domain that a user may obtain by observing its external behavior and its documentation but not its internal structure.

Ideally, both domain models should completely overlap, however, there could be entities in OBS not present in REC and vice versa. Therefore, figure 2.2 illustrates the final mapping we have to make, between SRC and USR. The *Intra-Application Model* (INT) represents the knowledge recovered from the source code, also present in the user view, without limiting it to the knowledge found in REF.

In Section 2.2 we describe our research method, explaining how we will analyze the mappings between USR and REF (OBS), SRC and REF (REC), and SRC and USR (INT) in order to answer SQ1 and SQ2. The results of each step are described in

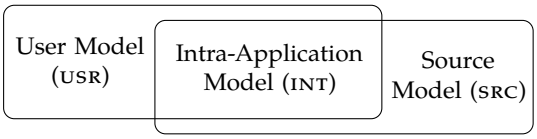


Figure 2.2: INT is the model in the shared vocabolary of the application, unrelated to any reference model. It represents the concepts found in both the USR and SRC model.

detail in Sections 2.3 to 2.8. Related work is discussed in Section 2.9 and Section 2.10 (Conclusions) completes the chapter.

2.2 RESEARCH METHOD

In order to investigate the limits of domain model recovery we study *manually* extracted domain models. The following questions guide this investigation:

1. Which domain is suitable for this study?
2. What is the upper limit of domain knowledge, or what is our reference model (REF)
3. How to select two representative applications?
4. How do we recover domain knowledge that can be observed by the user of the application (SQ1 & OBS)?
5. How do we recover domain knowledge from the source code (SQ2 & REC)?
6. How do we compare models that use different vocabularies (terms) for the same concepts? (SQ1, SQ2)?
7. How do we compare the various domain models to measure the success of domain model recovery? (SQ1,SQ2)?

We will now answer the above questions in turn. Although we are exploring manual domain model recovery, we want to make this manual process as traceable as possible since this enables independent review of our results. Where possible we automate the analysis (calculation of metrics, precision and recall), and further processing (visualization, table generation) of manually extracted information. Both data and automation scripts are available online [Lan13].

2.2.1 *Selecting a target domain*

We have selected the domain of project planning for this study since it is a well-known, well-described, domain of manageable size for which many open source software applications exist. We use the Project Management Body of Knowledge (PMBOK) [Ins08] published by Project Management Institute (PMI) for standard terminology in the project management domain. Note that as such the PMBOK covers a lot more than just project planning.

2.2.2 *Obtaining the Reference Model (REF)*

Validating the results of a reverse engineering process is difficult and requires an oracle, i.e., an *actionable* domain model suitable for comparison and measurement. We have transformed the descriptive knowledge in PMBOK into such a reference model using the following, traceable, process:

1. Read the PMBOK book.
2. Extract project planning facts.
3. Assign a number to each fact and store its source page.
4. Construct a domain model, where each entity, attribute, and relation are linked to one or more of the facts.
5. Assess the resulting model and repeat the previous steps when necessary.

The resulting domain model will act as our Reference Model. and Section 2.3 gives the details.

2.2.3 *Application selection*

In order to avoid bias towards a single application, we need at least two project planning applications to extract domain models from. Section 2.4 describes the selection criteria and the selected applications.

2.2.4 *Observing the application*

A user can observe an application in several ways, ranging from its UI, command-line interface, configuration files, documentation, scripting facilities and other functionality or information exposed to the user of the application. In this study we use the UI and documentation as proxies for what the user can observe. We have followed these steps to obtain the User Model (USR) of the application:

1. Read the documentation.
2. Determine use cases.
3. Run the application.
4. Traverse the UI depth-first for all the use cases.
5. Collect information about the model exposed in the UI.
6. Construct a domain model, where each entity and relation are linked to a UI element of the application.
7. Assess the resulting model and repeat the previous steps when necessary.

We report about the outcome in Section 2.5.

2.2.5 *Inspecting the source code*

We have designed the following traceable process to extract a domain model from each application's source code, the Source Model (SRC):

1. Read the source code as if it is plain text.
2. Extract project planning facts.
3. Store its filename, and line number (source location).
4. Construct a model, where each entity, attribute, and relation is linked to a source location in the application's source code.
5. Assess the model and repeat the previous steps when necessary.

The results appear in Section 2.6.

2.2.6 *Mapping models*

After performing the above steps we have obtained five domain models for the same domain, derived from different sources:

- The Reference Model (REF) derived from PMBOK.
- For each of the two applications:
  - User Model (USR).
  - Source Model (SRC).

While all these model are in the project planning domain, they all use different vocabularies. Therefore, we have to manually map the models to the same vocabulary. Mapping the USR and SRC models onto the REF model, gives the Observed (OBS) and Recovered Model (REC).

The final mapping we have to make, is between the SRC and USR models. We want to understand how much of the User Model (USR) is present in the Source Model (SRC). Therefore, we also map the SRC onto the USR model, giving the Intra-Application Model (INT). The results of all these mappings are given in Section 2.7.

2.2.7 *Comparing models*

To be able to answer Q1 and Q2, we will compare the 11 produced models. Following other research in the field of concept assignment, we use the most common Information Retrieval (IR) approach, *recall* and *precision*, for measuring quality of the recovered data. Recall measures how much of the expected model is present in the found model, and precision measures how much of the found model is part of the expected.

To answer Q1, the recall between REF and USR (OBS) explains how much of the domain is covered by the application. Note that the result is subjective with respect to the size of REF: a bigger domain may require looking at more different applications that play a role in it. By answering Q2 first, analyzing the recall between USR and SRC (INT), we will find out whether source code could provide the same recall as REF

and USR (OBS). The relation between REF and SRC (REC) will confirm this conclusion. Our hypothesis is that since the selected applications are small, we can only recover a small part of the domain knowledge, i.e. a low recall.

The precision of the above mappings is an indication of the quality of the result in terms of how much extra (unnecessary) details we accidentally would recover. This is important for answering Q2. If the recovered information would be overshadowed by junk information[†], the recovery would have failed to produce the domain knowledge as well. We hypothesize that due to the high-level object-oriented designs of the applications we will get a high precision.

Some more validating comparisons, their detailed motivation and the results of all model comparisons are described in Section 2.8.

2.3 PROJECT PLANNING REFERENCE MODEL

Since there is no known domain model or ontology for project planning that we are aware of, we need to construct one ourselves. The aforementioned PMBOK [Ins08] is our point of departure. PMBOK avoids project management style specific terminology, making it well-suited for our information needs.

2.3.1 *Gathering facts*

We have analyzed the whole PMBOK book. This analysis has been focused on the concept of a *project* and everything related to *project planning* therefore we exclude other concepts and processes in the project management domain.

After analyzing 467 pages we have extracted 151 distinct facts related to project planning. A *fact* is either an explicitly defined concept, an implicitly defined concept based on a summarized paragraph, or a relations between concepts. These facts were located on 67 different pages. This illustrates that project planning is a subdomain and that project management as a whole covers many topics that fall outside the scope of the current chapter. Each fact was assigned a unique number and the source page number where it was found in PMBOK. Two example facts are: "A milestone is a significant point or event in the project."`(id: 108, page: 136)` and "A milestone may be mandatory or optional." `(id: 109, page: 136)`.

2.3.2 *Creating the Reference Model REF*

In order to turn these extracted facts into a model for project planning, we have translated the facts to entities, attributes of entities, and relations between entities. The two example facts (108 and 109), are translated into a relation between the classes Project and Milestone, and the mandatory attribute for the Milestone class. The

[†]Implementation details or concepts from other domains.

Table 2.1: Number of entities and relations in the created models, and the amount of locations in the PMBOK book, source code, or UI screens used to construct the model.

Source	Model	entities	relations			unique observations
			associations	specializations	total	
PMBOK	REF	74	75	32	107	83
Endeavour	USR	23	30	8	38	19
	SRC	26	51	8	59	80
OpenPM	USR	22	24	3	27	13
	SRC	28	44	6	50	68

meta-model of our domain model is a class diagram. We use a textual representation in the meta-programming language Rascal [KvdSV09] which is also used to perform calculations on these models (precision, recall).

Table 2.1 characterizes the size of the project planning reference domain model REF by number of entities, relations and attributes; it contains of 74 entities and 107 relations. There is also a set of 49 attributes, but this seems incomplete, because in general we expect any entity to have more then one property. The lack of details in PMBOK could be an explanation for this. Therefore, we did not use the attributes of the reference model to calculate similarity.

The model is too large to include in this thesis, however for demonstration purposes, a small subset is shown in Figure 2.3.

Not all the facts extracted from PMBOK are used in the Reference Model. Some facts carry only explanations. For example "costs are the monetary resources needed to complete the project". Some facts explain dynamic relations that are not relevant for an entity/relationship model. These two categories explain 55 of the 68 unused facts. The remaining 13 facts were not clear enough to be used or categorized. In total 83 of the 151 observed facts are represented in the Reference Model.

2.3.3 *Discussion*

We have created a Reference Model that can be used as oracle for domain model recovery and other related reverse engineering tasks in the project planning domain. The model was created by hand by the second author, and care was taken to make the whole process traceable. We believe this model can be used for other purposes in this domain as well, such as application comparison and checking feature completeness.

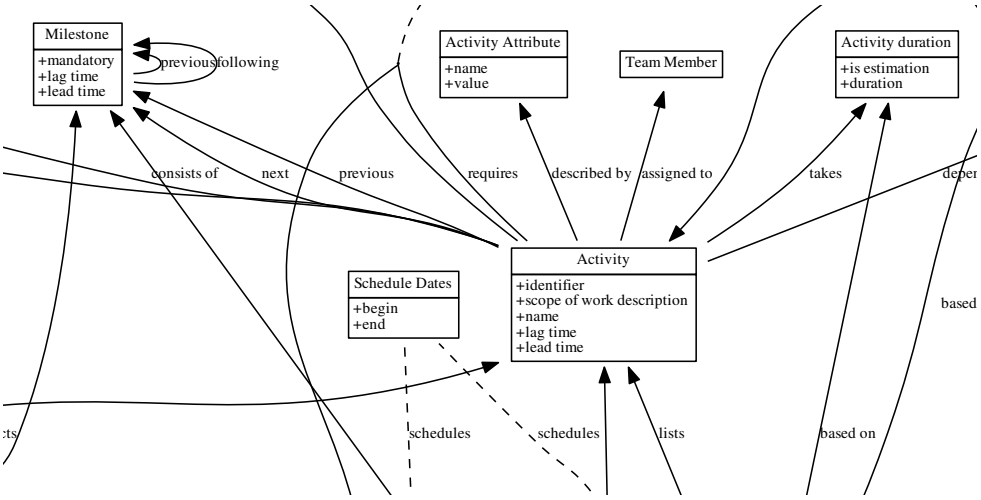


Figure 2.3: Fragment of reference model REF visualized as class diagram. Boxes represent entities, arrows relations, and dashed lines link entities to relations.

Threats to validity

We use PMBOK as main source of information for project planning. There are different approaches to project planning and a potential threat is that some are not covered in this book. Since PMBOK is an industry standard (ANSI and IEEE), we consider this to be a low-risk threat and have not mitigated it.

Another threat is that model recovery by another person could lead to a different model. The traceable extraction of the reference model makes it possible to understand the decisions on which the differences are based. Due to the availability of our analysis scripts, the impact of differences can be easily computed.

2.4 APPLICATION SELECTION

We are interested in finding "ideal" project planning systems to manually read and extract domain models from. The following requirements have guided our search:

- Source code is available: to enable analysis at all.
- No more than 30 KLOC: to keep manual analysis feasible.
- Uses an explicit data model, for example Model View Controller (MVC), or an Object-Relational Mapping (ORM): to ensure that domain elements can be identified in the source code.

Table 2.2: Endeavour: structure and size.

Package	# files	LOC	description
model	29	4474	MVC model.
view	108	10480	MVC view (UI).
controller	49	3404	MVC controller.
Total	186	18358	

Table 2.3: OpenPM: structure and size.

Package	# files	LOC	description
model	29	5591	MVC model.
view	21	1546	MVC controller.
servlets	33	3482	MVC view (UI).
test	75	7137	UI & integration tests.
Total	158	17756	

We have made a shortlist of 10 open source project planning systems[‡]. The list contains applications implemented in different languages (Java, Ruby, and C++) and sizes ranging from 18 KLOC to 473 KLOC.

From this Endeavour and OpenPM satisfy the aforementioned requirements. Endeavour is a Java application that uses a custom MVC design with ThinWire as front-end framework, and Hibernate as ORM. OpenPM uses Java servlets in combination with custom JavaScript. It also uses Hibernate as ORM. Table 2.2 and 2.3 describe the structure and size of the two applications[§]. Note that OpenPM's `view` package contained MVC controller logic, and the `servlets` the MVC views.

Both systems aim at supporting the process of planning by storing the process state but they hardly support process enforcement, except recording dependence between activities.

2.4.1 *Discussion*

A threat to external validity is that both systems are implemented in Java. Looking at systems in multiple modern languages is considered future work.

[‡] ChilliProject, Endeavour, GanttProject, LibrePlan, OpenPM, OpenProj, PLANdora, Project.net, Taskjuggler, Xplanner+.

[§] Number of files and Lines of Code (LOC) are calculated using the cloc tool [Dan13].

2.5 OBTAINING THE USER MODEL

We have used the UI and documentation of the applications to construct the User Model (USR). Use cases were extracted from the documentation when possible.[¶] Following these use cases, a depth-first exploration of the UI is performed. For every entity and relation we have recorded in which UI screen we first observed it. Table 2.1 describes the User Models for both Endeavour and OpenPM.

For example the Task entity in Endeavour's USR Model was based on the sub-window "Task Details" of the "Home" window.

2.5.1 *Discussion*

We have tried to understand the domain knowledge represented by the applications by manually inspecting it from the user's perspective. Both applications used Ajax to provide an interactive experience.

Endeavour uses the Single Page Application style, with a windowing system similar to MS Windows®. The UI is easy to understand, and different concepts are consistently linked across the application. OpenPM uses a more modern interface. However, we experienced more confusion on how to use it. It assumes a specific project management style (SCRUM), and requires more manual work by the user.

We have observed that creating a User Model is simple. For systems of our size, a single person can construct a User Model in one day. This is considerably less than creating a Source Model and suggests that the UI is an effective source for recovering domain models.

Threats to validity

We use the User Model as a proxy for the real domain knowledge exposed by the application. The limit of this knowledge is hard to define, but we believe our approach is an accurate approximation.

We can not be sure about our coverage of the User Model. It could be possible there are other interfaces to the application we are unaware of. Moreover, there could be conditions, triggers, or business rules only observable in very specific scenarios. Some of these issues will be observed in the various model comparisons. We are not aware of other approaches to further increase confidence in our coverage.

[¶] Unfortunately, OpenPM does not provide documentation at the time of writing.

2.6.1 *Domain model recovery*

We have chosen the Eclipse Integrated Development Environment (IDE) to read the source code of the selected applications. Our goal was to maximize the amount of information we could recover. Therefore, we have first read the source code and then used Rascal to analyze relations in the source code. Rascal uses Eclipse's Java Development Tools (JDT) to analyze Java code, and provides a visualization library that can be used to quickly verify hypothesis formed during the first read-through.

For the actual creation of the model, we have designed and followed these rules:

- Read only the source code, not the database scheme/data.
- Do not run the application.
- Use the terms of the applications, do not translate them to terms used in the Reference Model.
- Include the whole model as seen by the application, do not filter out obvious implementation entities.
- Do read comments and string literals.

We have used the same meta-model as used for describing the Reference Model. We replaced the fact's identifiers with source locations (filename and character range), which are a native construct in Rascal. To support the process of collecting facts from the source code we added a menu-item to the context-menu of the Java editor to write the cursor's source location to the clipboard.

The domain model for each application was created in a similar fashion as we did when creating the reference model. All the elements in the domain model are based on one or more specific observations in the source code (see table 2.1).

For example the relation between Task and Dependency in Endeavour's SRC model is based on the `List<Dependency> dependencies` field found on line 35 in file `Endeavour-Mgmt/model/org/endeavour/mgmt/model/Task.java`.

2.6.2 *Results*

Table 2.1 shows the sizes of the extracted models for both applications expressed in number of entities, relations and attributes and the number of unique source code locations where they were found.

Endeavour In Endeavour 26 files contributed to the domain model. 22 of those files were in the model package, the other 4 were from the controller package. The controller classes each contributed one fact. 155 of the source locations were from the model package.

OpenPM In OpenPM 22 files contributed to the domain model. These files were all located in the model package.

2.6.3 *Discussion*

We have performed domain model recovery on two open source software applications for project planning.

Both applications use the same ORM system, but a different version of the Application Programming Interface (API). Endeavour also contains a separate view model, which is used in the MVC user interface. However, it has been implemented as a pass-through layer for the real model.

Threats to validity

A first threat (to internal validity) is that manual analysis is always subject to bias from the performer and that this was performed by the same author who created the other models. We have mitigated this by maximizing the traceability of our analysis: we have followed a fixed analysis process and have performed multiple analysis passes over the source code and published the data.

A second threat (to external validity) is the limited size of the analyzed applications, both contain less than 20 KLOC Java. Larger applications would make our conclusions more interesting and general, but they would also make the manual analysis less feasible.

2.7 MAPPING MODELS

We now have five domain models of project planning: one reference model (REF) to be used as oracle, and four domain models (SRC, USR) obtained from the two selected project planning applications. These models use different vocabulary, we have to map them onto the same vocabulary to be able to compare them.

2.7.1 *Lightweight domain model mapping*

We manually map the entities between different comparable models. The question is how to decide whether to entities are the same. Strict string equality is too limited and should be relaxed to some extent.

Table 2.4 and 2.5 show the mapping categories we have identified for the (un)successful mapping of model entities.

Table 2.4: Categories for successfully mapped entities

Mapping name	Description
Equal Name	Entity has the same name as an entity in the other model. Note that this is the only category which can also be a failure when the same name is used for semantically different entities
Synonym	Entity is a direct synonym for an entity in the other model, and is it not a homonym.
Extension	Entity captures a wider concept than the same entity in the other model.
Specialization	Entity is a specific or concrete instance of the same entity in the other model.
Implementation specialization	Comparable to specialization but the specialization is related to an implementation choice.

Table 2.5: Categories for unsuccessfully mapped entities

Mapping name	Description
Missing	The domain entity is missing in the other model, i.e. a false positive. This is the default mapping failure when an entity cannot be mapped via any of the other categories.
Implementation	The entity is an implementation detail and is not a real domain model entity.
Too detailed	An entity is a domain entity but is too detailed in comparison with the other model.
Domain detail	The entity is a detail of a sub domain, this category is a subclass of "too detailed".

2.7.2 *Mapping results*

We have manually mapped all the entities in the User Model (USR) and the Source Model (SRC) to the Reference Model (REF), and SRC to USR. For each mapping we have explicitly documented the reason for choosing this mapping. For example, in Endeavour's SRC model the entity *Iteration* is mapped to *Milestone* in the Reference Model using specialization, with documented reason: "*Iterations split the project into chunks of work, Milestones do the same but are not necessarily iterative.*"

Table 2.6: Endeavour: Entities in the mapped models, per mapping category

Category	USR	REF	SRC	REF	SRC	USR
Equal Name	7	7	7	7	21	21
Synonym	2	3	2	3	3	2
Extension	0	0	0	0	0	0
Specialization	5	3	5	3	0	0
Implementation specialization	1	1	1	1	0	0
Total	15	14	15	14	24	23
Equal Name†	1	-	1	-	0	-
Missing	1	-	2	-	0	-
Implementation	1	-	2	-	2	-
Domain Detail	5	-	6	-	0	-
Too Detailed	0	-	0	-	0	-
Total	8	-	11	-	2	-

† A false positive, in Endeavour the term Document means something different then the term Documentation in the Reference Model.

Table 2.6 and 2.7 contain the number of mapping categories used for both applications, per mapping. For some mapping categories, it is possible for one entity to map to multiple, or multiple entities to one. For example the *Task* and *WorkProduct* entities in Endeavour's SRC model are mapped on the *Activity* entity in the Reference Model. Therefore, we report the numbers of the entities in both the models, the source and the target.

The relatively large number of identically named entities (7/15) between Endeavour and the reference model is due to the presence of a similar structure of five entities, describing all the possible activity dependencies.

An example of a failed mapping is the *ObjectVersion* entity in the Source Model of OpenPM. This entity is an implementation detail. It is a variant of the Temporal Object pattern[‖] where every change of an entity is stored to explicitly model the history of all the objects in the application.

Table 2.8 contains all the entities per domain model, and highlights the mapped entities.

[‖] See `http://martinfowler.com/eaaDev/TemporalObject.html`.

Table 2.7: OpenPM: Entities in mapped models, per mapping category

Category	USR	REF	SRC	REF	SRC	USR
Equal Name	1	1	1	1	18	18
Synonym	3	3	4	4	4	4
Extension	1	1	1	1	0	0
Specialization	0	0	0	0	0	0
Implementation specialization	1	1	1	1	0	0
Total	6	6	7	7	22	22
Missing	2	-	2	-	1	-
Implementation	12	-	17	-	5	-
Domain Detail	0	-	0	-	0	-
Too Detailed	2	-	2	-	0	-
Total	16	-	21	-	6	-

2.7.3 *Discussion*

We have used a lightweight approach for mapping domain models. Our mapping categories may be relevant for other projects and can be further extended and evaluated.

For future work, we can investigate if whether more automated natural language processing can help, however, remember our motivations for excluding automatic approaches in our current research method.

At most half of the domain models recovered from the applications could be mapped to the reference model. The other half of the extracted models regarded details of the domain or the implementation.

Threats to validity

A threat to external validity is that we have used an informal approach to map the domain models of the two applications to the reference model. The mapping categories presented above, turned out to be sufficient for these two applications, however we have no guarantees for other application of these categories. The categories have evolved during the process and each time a category was added or modified all previous classifications have been reconsidered.

Table 2.8: Entities found in the various domain models.

Source	Model	Entities†
PMBOK	REF	Action, **Activity**, **Activity Attribute**, **Activity Dependency**, **Activity duration**, Activity list, Activity resource, Activity sequence, Activity template, Approver, Budget, Change Control Board, **Change request**, Closing, Communications plan, Composite resource calendar, Composite resource calendar availability, Constrain, Corrective action, **Defect**, Defect repair, **Deliverable**, **Documentation**, Environment, Equipment, External, **FinishFinish**, **FinishStart**, Human Resource Plan, Information, Internal, Life cycle, Main, Material, **Milestone**, Objective, Organisation, Organizing, People, Person, Phase, Planned work, Portfolio, Preparing, Preventive action, Process, Product, **Project**, Project management, Project plan, **Project schedule**, Project schedule network diagram, Quality, **Requirement**, Resource, Resource calendar, Resource calendar availability, Result, Risk, Risk management plan, Schedule, Schedule Dates, Schedule baseline, Schedule data, Scope, Service, Stakeholder, **StartFinish**, **StartStart**, Supplies, **Team Member**, Work Breakdown Structure, Work Breakdown Structure Component, Work Package
Endeavour	USR	**Actor**, **Attachment**, **Change Request**, Comment, **Defect**, **Document**, Event, **FinishFinish**, **FinishStart**, Glossary, **Iteration**, **Project**, **ProjectMember/Stakeholder**, Security Group, **StartFinish**, **StartStart**, **Task**, **Task Dependency**, Test Case, Test Folder, Test Plan, **Use Case**, **X**
Endeavour	SRC	**Actor**, **Attachment**, **ChangeRequest**, Comment, **Defect**, **Dependency**, **Document**, Event, **FinishFinish**, **FinishStart**, GlossaryTerm, **Iteration**, Privilege, **Project**, **ProjectMember**, SecurityGroup, **StartFinish**, **StartStart**, **Task**, TestCase, TestFolder, TestPlan, TestRun, **UseCase**, Version, **WorkProduct**
OpenPM	USR	Access Right, **Attachment**, Button, Comment, Create, Delete, **Effort**, Email Notification, FieldHistory, HistoryEvent, **Iteration**, Label, Link, ObjectHistory, **Product**, Splitter, State, Tab, **Task**, Type, Update, **User**
OpenPM	SRC	Access, Add, **Attachment**, Comment, Create, Delete, **Effort**, EmailSubscription, EmailSubscriptionType, Event, FieldType, FieldVersion, Label, Link, **Milestone**, ObjectType, ObjectVersion, **Product**, Remove, Splitter, **Sprint**, Tab, **Task**, TaskButton, TaskState, TaskType, Update, **User**

† Bold entity in Reference Model is used in application models. Bold entity in application model could be mapped to entity in Reference Model.

Table 2.9: Recall and precision explained per model combination.

Retrieved	Expected	Recall	Precision
USR	REF	Which part of the domain is covered by an application. This is subjective to the size of REF.	How many of the concepts in USR are actually domain concepts, e.g., how much implementation details are in the *application*?
SRC	REF	How much of REF can be recovered from SRC. If high then this should confirm high recall for both USR ◇ REF and SRC ◇ USR.	How much of SRC are actually domain concepts, e.g., how much implementation junk is accidentally recovered from *source*?
SRC	USR	How much of USR can be recovered by analyzing the source code (SRC). This gives no measure of the amount of actual domain concepts found.	How many details are in SRC, but not in USR? If USR were a perfect representation of the application knowledge, this category would only contain dead-code and unexposed domain knowledge.

2.8 COMPARING THE MODELS

We now have five manually constructed and six derived domain models for project planning:

- One reference model (REF) to be used as oracle.
- Four domain models (SRC, USR) obtained from each of the two selected project planning applications.
- Six derived domain models (OBS, REC, INT) resulting from the mapping of the previous four (SRC, USR).

How can we compare these models in a meaningful way?

2.8.1 *Recall and Precision*

The most common measures to compare the results of an IR technique are *recall* and *precision*. Often it is not possible to get the 100% in both, and we have to discuss which measure is more important in the case of our model comparisons.

We have more than two datasets, and depending on the combination of datasets, recall or precision is more important. Table 2.9 explains in detail how recall and

Table 2.10: Endeavour: Recall and precision.

Comparison	Recall		Precision	
	entities	relations	entities	relations
USR ◇ REF	19%	6%	64%	15%
SRC ◇ REF	19%	6%	56%	13%
SRC ◇ USR	100%	92%	92%	74%

Table 2.11: OpenPM: Recall and precision.

Comparison	Recall		Precision	
	entities	relations	entities	relations
USR ◇ REF	7%	3%	23%	16%
SRC ◇ REF	9%	6%	25%	18%
SRC ◇ USR	100%	80%	79%	44%

Table 2.12: Combined: Recall and precision.

Comparison	Recall		Precision	
	entities	relations	entities	relations
USR ◇ REF	22%	7%	40%	14%
SRC ◇ REF	23%	9%	36%	13%

precision will be used and explains for the relevant model combinations which measure is useful and what will be measured.

Given two models M_1 and M_2, we use the following notation. The comparison of two models is denoted by $M_1 \diamond M_2$ and results in recall and precision for the two models. If needed, M_1 is first mapped to M_2 as described in Tables 2.6 and 2.7.

2.8.2 *Results*

Tables 2.10 and 2.11 shows the results for, respectively, Endeavour and OpenPM. Which measures are calculated is based on the analysis in Table 2.9.

2.8.3 *Relation Similarity*

Since recall and precision for sets of entities provides no insight into similarity of the relations between entities, we need an additional measure. Our domain models contain entities and their relations. Entities represent the concepts of the domain, and relations their structure. If we consider the relations as a set of edges, we can directly calculate recall and precision in a similar fashion as described above.

We also considered some more fine grained metrics for structural similarity. Our domain model is equivalent to a subset of Unified Modeling Language (UML) class diagrams and several approaches exist for calculating the minimal difference between such diagrams [KWN05; OWK03]. Such "edit distance" methods give precise indications of how big the difference is. Similarly we might use general graph distance metrics [BS98]. We tried this latter method and found that the results, however more sophisticated, were harder to interpret. For example, USR and REF were 11% similar for Endeavor. This seems to be in line with the recall numbers, 6% for relations and 19% for entities, but the interesting precision results (64% and 15%) are lost in this metric. So we decided not to report these results and stay with the standard accuracy analysis.

2.8.4 *Discussion*

Low precision and recall for relations

On the whole the results for the precision and recall of the relation part of the models are lower than the quality of the entity mappings. We investigated this by taking a number of samples. The reason is that the Reference model is more detailed, introducing intermediate entities with associated relations. For every intermediate entity, two or more relations are introduced which can not be found in the recovered models.

These results indicate that the recall and precision metrics for sets of relations underestimate the structural similarity of the models.

Precision of OBS: USR ◇ REF

We found the precision of OBS to be 64% (Endeavour) and 23% (OpenPM), indicating that both applications contain a significant amount of entities that are unrelated to project planning as delimited by the Reference Model. For Endeavour, out of the 8 unmappable entities (see Table 2.6 in section 2.7), only 2 were actual implementation details. The other 6 are sub-domain details not globally shared within the domain. If we recalculate to correct for this, Endeavour's Observed Model even has a precision of 91%. For OpenPM there are only 2 out of the 16 for which this correction can be applied, leaving the precision at 36%. For the best scenario, in this case represented by Endeavour, 90% of the User Model (USR) is part of the Reference Model (REF).

OpenPM's relatively low precision (36%) can be explained by table 2.7, which show the USR model has a lot of implementation detail (related to version control operations).

Recall of OBS: USR ◇ REF The recall for the Observed Model (OBS) is for Endeavour 19% and for OpenPM 7%. Which means both applications cover less then 20% of the project planning domain.

Precision of REC: SRC ◇ REF The precision of the Recovered Model (REC) is for Endeavour 56% (corrected 88%), and for OpenPM 25% (corrected 39%). This shows that for the best scenario, represented again by Endeavour, the Source Model only contains 12% implementation details.

Recall of REC: SRC ◇ REF The recall for the Recovered Model (REC) is for Endeavour 19% and for OpenPM 9%. The higher recall for OpenPM, compared to OBS, for both entities and relations is an example where the Source Model contained more information then the User Model, which we will discuss in the next paragraph.

Precision and recall for INT: SRC ◇ USR How much of the User Model can be recovered by analyzing only the Source Model? For both Endeavour and OpenPM, recall is 100%. This means that every entity in the USR model was found in the source code. Endeavour's precision was 92% and OpenPM's 79%. OpenPM contains an example where information in the Source Model is not observable in the User Model: comments in the source code explain the *Milestones* and their relation to *Iterations*.

The 100% recall and high precision mean that these applications were indeed amenable for reverse engineering (as we hypothesized when selecting these applications). We could extract most of the information from the source code.

For this comparison, even the relations score quite high. This indicates that User Model and Source Model are structurally similar. Manual inspection of the models confirms this.

Recall for Endeavour and OpenPM combined Endeavour's and OpenPM's recall of USR ◇ REF and SRC ◇ REF measure the coverage of the domain a re-engineer can achieve. How much will the recall improve if we combine the recovered models of the two systems?

We only have two small systems, however, Table 2.12 contains the recall and precision for Endeavour and OpenPM combined. A small increase in recall, from 19% to 23%, indicates that there is a possibility for increasing the recall by observing more systems. However, as expected, at the cost of precision.

Interpretation Since our models are relatively small, our results cannot be statistically significant but are only indicative. Therefore we should not report exact percentages, but characterizing our recall and precision as *high* seems valid. Further research based on more applications is needed to confirm our results.

There are many connections between ontologies and domain models. The model mappings that we need are more specific than the ones provided by general ontology mapping [CSH06].

Abebe and Tonella [AT10] introduced a Natural Language Parsing (NLP) method for extracting an ontology from source code. They came to the same conclusion as we do: this extracted ontology contains a lot of implementation details. Therefore, they introduced an IR filtering method [AT11] but it was not as effective as the authors expected. Manual filtering of the IR keyword database was shown to improve effectiveness. Their work is in the same line as ours, but we have a larger reference domain model, and we focus on finding the limits of domain model recovery, not on an automatic approach. It would be interesting to apply their IR filtering to our extracted models.

Ratiu et al. [RFJ08] proposed an approach for domain ontology extraction. Using a set of translation rules they extract domain knowledge from the API of a set of related software libraries. Again, our focus is on finding the limits of model recovery, not on automating the extraction.

Hsi et al. [HPM03] introduced ontology excavation. Their methodology consists of a manual depth-first modeling of all UI interactions, and then manually creating an ontology, filtering out non-domain concepts. They use five graph metrics to identify interesting concepts and clusters in this domain ontology. We are interested in finding the domain model inside the user-interface model, Hsi et al. perform this filtering manually, and then look at the remaining model. Automatic feature extraction of user interfaces is described in [BP12].

Carey and Gannod [CG07] introduced a method for concept identification. Classes are considered the lowest level of information of an object-oriented system and Machine Learning is used in combination with a set of class metrics. This determines interesting classes, which should relate to domain concepts. Our work is similar, but we focus on *all* the information in the source code, and are interested in the maximum that can be recovered from the source. It could be interesting to use our reference model to measure how accurately their approach removes implementation concerns.

UML class diagram recovery [SM05; WS07] is also related to our work but has a different focus. Research focuses on the precision of the recovered class diagrams, for example the difference between a composition and aggregation relation. We are interested in less precise UML class diagrams.

Work on recovering the concepts, or topics, of a software system [KDG07; LRB+07] has a similar goal as ours. IR techniques are used to analyze all the terms in the source code of a software system, and find relations or clusters. Kuhn et al. [KDG07] use identifiers in source code to extract semantic meaning and report on the difficulty of

evaluating their results. Our work focuses less on structure and grouping of concepts and we evaluate our results using a constructed reference model.

Reverse engineering the relation between concepts or features [EKS03; SLB+11], assumes that there is a set of known features or concepts and tries to recover the relations between them. These approaches are related to our work since the second half of our problem is similar: after we have recovered domain entities, we need to understand their relations.

DeBaud et al. [DMR94] report on a domain model recovery case study on a COBOL program. By manual inspection of the source code, a developer reconstructed the data constructs of the program. They also report that implementation details make extraction difficult, and remark that systems often implement multiple domains, and that the implementation language plays an important role in the discovery of meaning in source code.

We do not further discuss other related work on knowledge recovery that aims at extracting facts about architecture or implementation. One general observation in all the cited work is that it is hard to separate domain knowledge from implementation knowledge.

2.10 CONCLUSIONS

We have explored the limits of domain model recovery via a case study in the project planning domain. Here are our results and conclusions.

2.10.1 *Reference model*

Starting with PMBOK as authoritative domain reference we have manually constructed an actionable domain model for project planning. This model is openly available and may be used for other reverse engineering research projects.

2.10.2 *Lightweight model mapping*

Before we can understand the differences between models, we have to make them comparable by mapping them to a common model. We have created a manual mapping method that determines for each entity if and how it maps onto the target model. The mapping categories evolved while creating the mappings. We have used this approach to describe six useful mappings, four to the Reference Model and two to the User Model.

2.10.3 *What are the limits of domain model recovery?*

We have formulated two research questions to get insight in the limits of domain model recovery. Here are the answers we have found (also see Table 2.9 and remember our earlier comments on the interpretation of the percentages given below).

SQ1: Which parts of the domain are implemented by the application? Using the user view (USR) as a representation of the part of the domain that is implemented by an application, we have created two domain models for each of the two selected applications. These domain models represent the domain as exposed by the application. Using our Reference Model (REF) we were able to determine which part of USR was related to project planning. For our two cases 91% and 36% of the User Model (USR) can be mapped to the Reference Model (REF). This means 9% and 64% of the UI is about topics not related to the domain. From the user perspective we could determine that the applications implement 19% and 7% of the domain.

The tight relation between the USR and the SRC model (100% recall) shows us that this information is indeed explicit and recoverable from the source code. Interestingly, some domain concepts were found in the source code that were hidden by the UI and the documentation, since for OpenPM the recall between USR and REF was 7% where it was 9% between SRC and REF.

So, the answer for SQ1 is: the recovered models from source code are useful, and only a small part of the domain is implemented by these tools (only 7-19%).

SQ2: Can we recover those implemented parts from the source of the application? Yes, see the answer to SQ1. The high recall between USR and SRC shows that the source code of these two applications explicitly models parts of the domain. The high precisions (92% and 79%) also show that it was feasible to filter implementation junk manually from these applications from the domain model.

2.10.4 *Perspective*

For this research we manually recovered domain models from source code to understand how much valuable domain knowledge is present in source code. We have identified several follow-up questions:

- How does the quality of extracted models grow with the size and number of applications studied? (Table 2.12)
- How can differences and commonalities between applications in the same domain be mined to understand the domain better?
- How does the quality of extracted models differ between different domains, different architecture/designs, different domain engineers?
- How can the extraction of a User Model help domain model recovery in general. Although we have not formally measured the effort for model extraction, we have

noticed that extracting a User Model requires much less effort than extracting a Source Model.
- How do our manually extracted models compare with automatically inferred models?
- What tool support is possible for (semi-)automatic model extraction?
- How can domain models guide the design of a DSL?

Our results of manually extracting domain models are encouraging. They suggest that when re-engineering a family of object-oriented applications to a DSL their source code is a valuable and trustworthy source of domain knowledge, even if they only implement a small part of the domain.

3 EXPLORING THE RELATIONSHIP BETWEEN SLOC AND CC

Abstract

Measuring the internal quality of source code is one of the traditional goals of making software development into an engineering discipline. Cyclomatic Complexity (CC) is an often used source code quality metric, next to Source Lines of Code (SLOC). However, the use of the CC metric is challenged by the repeated claim that CC is redundant with respect to SLOC due to strong linear correlation.

We conducted an extensive literature study of the CC/SLOC correlation results. Next, we tested correlation on large Java (17.6 M methods) and C (6.3 M functions) corpora. Our results show that linear correlation between SLOC and CC is only moderate as caused by increasingly high variance. We further observe that aggregating CC and SLOC as well as performing a power transform improves the correlation.

Our conclusion is that the observed linear correlation between CC and SLOC of Java methods or C functions is not strong enough to conclude that CC is redundant with SLOC. This conclusion contradicts earlier claims from literature, but concurs with the widely accepted practice of measuring of CC next to SLOC.

3.1 INTRODUCTION

In previous work [VG12] one of the authors analyzed the potential problems of using the CC metric to indicate or even measure source code complexity per Java method. Still, since understanding code is known to be a major factor in providing effective and efficient software maintenance [vMV95], measuring the complexity aspect of internal source code quality remains an elusive goal of the software engineering community. In practice the CC metric is used on a daily basis for this purpose precisely, next to another metric, namely SLOC [BCS+12; HKV07].

There exists a large body of literature on the relation between the CC metric and SLOC. The general conclusion from experimental studies [BP84; FF79; JMF14; SCM+79] is that there exists a strong linear correlation between these two metrics

This chapter was first published at the ICSME2014 conference, and later extended to a JSEP journal publication. This chapter is the result of merging these two publications: D. Landman, A. Serebrenik, and J. J. Vinju. "Empirical Analysis of the Relationship between CC and SLOC in a Large Corpus of Java Methods". In: *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*. IEEE Computer Society, 2014, pp. 221–230. DOI: 10.1109/ICSME.2014.44 and D. Landman, A. Serebrenik, E. Bouwers, and J. J. Vinju. "Empirical analysis of the relationship between CC and SLOC in a large corpus of Java methods and C functions". In: *Journal of Software: Evolution and Process* 28.7 (2016), pp. 589–618. DOI: 10.1002/smr.1760

for arbitrary software systems. The results are often interpreted as an incentive to discard the CC metric for any purpose that SLOC could be used for as well, or as an incentive to normalize the CC metric for SLOC.

At the same time, the CC metric appears in every available commercial and open-source source code metrics tool, for example `http://www.sonarqube.org/`, and is used in the daily practice of software assessment [HKV07] and fault/effort prediction [FO00]. This avid use of the metric directly contradicts the evidence of strong linear correlation. Why go through the trouble of measuring CC?

Based on the related work on the correlation between CC and SLOC we have the following working hypothesis:

Hypothesis 1 *There is strong linear (Pearson) correlation between the CC and SLOC metrics for Java methods and C functions.*

We studied a C language corpus since it is most representative of the languages analyzed in literature and we could construct a large corpus based on open-source code. Java is an interesting case next to C as it represents a popular modern object-oriented language, for which we could also construct a large corpus. A modern language with a comparable but significantly more complex programming paradigm than C, such as Java, is expected to provide a different perspective on the correlation between SLOC and CC.

Both for Java and C, our results of investigating the strong correlation between CC and SLOC are negative, challenging the external validity of the experimental results in literature as well as their interpretation. The results of analyzing a linear correlation are not the same for our (much larger) corpora of modern Java code that we derived from Sourcerer [LBN+09] and C code derived from the packages of Gentoo Linux. Similarly we observe that higher correlations can only be observed after aggregation to the file level or when we arbitrarily remove the larger elements from the corpus. Based on analyzing these new results we will conclude that CC cannot be discarded based on experimental evidence of a linear correlation. We therefore support the continued use of CC in industry next to SLOC to gain insight in the internal quality of software systems for both the C and the Java language.

The interpretation of experimental results of the past is hampered by confusing differences in definitions of the concepts and metrics. In the following, Section 3.2, we therefore focus on definitions and discuss the interpretation in related work of the evidence of correlation between SLOC and CC. We also identify six more hypotheses. In Section 3.3 we explain our experimental setup. After this, in Section 3.4, we report our results and in Section 3.5 we interpret them before concluding in Section 3.6.

3.2 BACKGROUND THEORY

In this section we carefully describe how we interpret the CC and SLOC metrics, we identify related work, and introduce the hypotheses based on differences observed in related work.

3.2.1 *Defining SLOC and CC*

Although defining the actual metrics for lines of code and cyclomatic complexity used in this chapter can be easily done, it is hard to define the concepts that they actually measure. This lack of precisely defined dimensions is an often lamented, classical problem in software metrics [CC94; She88]. The current chapter does not solve this problem, but we do need to discuss it in order to position our contributions in the context of related work.

First we define the two metrics used in this chapter.

Definition 1 (Source Lines of Code (SLOC)) *A line of code is any line of program text that is not a comment or blank line, regardless of the number of statements or fragments of statements on the line. This specifically includes all lines containing program headers, declarations, and executable and non-executable statements [CDS86, p. 35].*

Definition 2 (Cyclomatic Complexity (CC)) *The cyclomatic complexity of a program* is the maximum number of linearly independent circuits in the control flow graph of said program, where each exit point is connected with an additional edge to the entry point [McC76].*

As explained by McCabe [McC76], the CC number can be computed by counting forks in a control flow graph and adding 1, or equivalently counting the number of language constructs used in the Abstract Syntax Tree (AST) which generate forks ("if", "while", etc.) and adding 1.

This last method is the easiest and therefore preferred method of computing CC. Unfortunately, which AST nodes generate decision points in control flow for a specific programming language is not so clear since this depends on the intrinsic details of programming language semantics. The unclarity leads to metric tools generating different values for the CC metric, because they count different kinds of AST nodes [LLL08]. Also, derived definitions of the metric exist, such as "extended cyclomatic complexity" [Mye77] to account for a different way of computing cyclomatic complexity. Still, the original definition by McCabe is sufficiently general. If we interpret it based on a control flow graph it is applicable to any programming language

*In this context a "program" means a subroutine of code like a procedure in Pascal, function in C, method in Java, sub-routine in Fortran, program in COBOL. From here on we use the term "subroutine" to denote either a Java method or a C function.

which has subroutines to encapsulate a list of imperative control flow statements. Section 3.3 describes how we compute CC for C and Java.

Note that we include the Boolean && and || operators as conditional forks because they have short-circuit semantics in both Java and C, rendering the execution of their right-hand sides conditional. Still, this is not the case for all related work. For completeness sake we therefore put the following hypothesis up for testing as well:

Hypothesis 2 *The strength of linear correlation between CC and SLOC of neither Java methods nor C functions is significantly influenced by including or excluding the Boolean operators && and ||.*

We expect that exclusion of && and || does not meaningfully affect correlations between CC and SLOC, because we expect Boolean operators not to be used often enough and not in enough quantities within a single subroutine to make a difference.

3.2.2 *Literature on the correlation between CC and SLOC*

We have searched methodically for related work that experimentally investigates a correlation between CC and SLOC. This results, to the best of our knowledge, in the most complete overview of published correlation figures between CC and SLOC to date. To increase our coverage we have combined a restricted form of snowballing [Woh14] with a Systematic Literature Review (SLR). We used snowballing to get an initial set of papers to compare the strength of the SLR. Using Google Scholar, we identified 15 relevant papers from both the 600 papers that cite Shepperd's paper from 1988 [She88] and the 200 most relevant results of the search query "empirical" for papers citing McCabe's original paper [McC76].

After this rough exploration of related work, we use an SLR to correct for the limitations of this approach and increase our coverage of the literature. We formulated the PICO criteria inspired by the SLR guidelines of Kitchenham and Charters [KC07]:

Population Software
Intervention CC or Cyclomatic or McCabe
Comparison SLOC or LOC or Lines of Code
Outcomes Correlation or Regression or Linear or R^2

Ideally, following the Kitchenham and Charters' guidelines [KC07] we should have constructed a query using the PICO criteria: "Software and (CC or Cyclomatic or McCabe) and (SLOC or LOC or Lines of Code) and (Correlation or Regression or Linear or R^2)". Unfortunately, Google Scholar does not supported nested conditional expressions. Therefore, we have used the PICO criteria to create $1 \times 3 \times 3 \times 4 = 36$ different queries producing 24 K results. Since Google scholar sorts the results on relevancy, we chose to read only the first two pages of every query, leaving 720 results. After noise filtering and duplication removal 326 papers remained, containing 11 of the

15 papers identified in the previous limited exploration. Together, we systematically scanned the full-text of these papers, using the following inclusion criteria:

1. Is the publication peer-reviewed?
2. Is SLOC or Lines of Code (LOC) measured?
3. Is CC measured (possibly as weight in Weighted Methods per Class (WMC) [CK94])?
4. Is Pearson correlation or any other statistical relation between SLOC and CC reported?
5. Are the measurements performed on method, function, class, module, or file level (higher levels are ignored)?

Using this process we identified 18 new papers. The resulting 33 papers are summarized in Table 3.1.

The SLR guidelines require the inclusion and the search queries to be based on the title, abstract and keywords. We deviated from this because for the current study we are interested in a reported relation between SLOC and CC, whether the paper focuses on this relation or not. This required us to scan the full text of each paper which the Kitchenham and Charter process does not cater for. Note that Google Scholar does index the body of papers.

The result of the above process is summarized by the multi-page Table 3.1. All levels and corpus descriptions in the table are as reported in the original papers: the interpretation of these might have subtle differences, e.g. Module and Program in Fortran could mean the same. Since the original data is no longer available, it is not possible to clarify these differences. The variables mentioned in the Correlation column are normalized as follows. If all lines in a unit (file, module, function, or method) were counted, LOC was reported. If comments and blank lines were ignored, SLOC was reported. If the line count was normalized on statements, we reported Logical Lines of Code (LLOC). We normalized R to R^2 by squaring it whenever R was originally reported.

Figure 3.1 visualizes the R^2 from the related work in Table 3.1 grouped by language and aggregation level. Most related work reports R^2 higher than 0.5, and there is not a clear upwards or downwards trend over the years. The only observable trends are that newer work (after 2000) predominantly performed aggregation on a file level (with the notable exception of four papers [CF07; HGH08; JMF14; MS11]) and that while the early studies have been mostly conducted on Fortran, the most common languages analyzed after 2000 are Java and C.

In the rest of this section we will formulate hypotheses based on observations in the related work: different aggregation methods (Section 3.2.3), data transformations (Section 3.2.4), and the influence of outliers and other biases in the used corpora (Section 3.2.5).

Table 3.1: Overview of related work on CC and SLOC up to 2014, this extends Shepperd's table [She88]. The correlations with a star (*) indicate correlations on the subroutine level. The ∘ denotes that the relation between CC and SLOC was the main focus of the paper. The statistical significance was always high, if reported, and therefore not indicated in this table (except Malhotra [MS11]).

	Publication	Level	Correlation	Language	Corpus	R^2	Comments
∘	[CSM79]	Subroutine	SLOC vs CC	Fortran	27 programs with SLOC ranging from 25 to 225	*0.65 0.81	The first result is for a CC correlation on subroutine level, and the second result is on a program level.
∘	[SCM+79]	Program	SLOC vs CC	Fortran	27 programs with SLOC ranging from 36 to 57	0.41	
∘	[FF79]	Program	log(LLOC) vs log(CC)	PL/1	197 programs with median of 54 statements.	*0.90	
∘	[WHH79]	Subroutine	LOC vs CC	Fortran	26 subroutines	*0.90	
∘	[Pai80]	Module	LOC vs CC	Fortran	10 modules, 339 SLOC	0.90	
∘	[STU+81]	Module	SLOC vs CC	Fortran	25.5 KSLOC over 137 modules	0.65	
∘	[BP84]	Module	SLOC vs CC	Fortran	517 code segments of one system	0.94	No correlation between module SLOC and CC. Grouping modules into 5 buckets (by size) results in a high correlation – for *5 data-points* – between their average CC and SLOC.
∘	[LC87]	Program	SLOC vs CC	Fortran	255 student assignments, range of 10 to 120 SLOC	0.82	Study comparing 31 metrics, showing histogram of the corpus, and scatter-plots of selected correlation.

Table 3.1: (Continued)

	Publication	Level	Correlation	Language	Corpus	R^2	Comments
	[KP87]	Module	SLOC vs CC	S3	Two subsystems with 67 modules	0.83 0.87	After a power transform on the first subsystem the R^2 increased to 0.89.
○	[LV89]	Routine	SLOC vs CC	Pascal & Fortran	1 system, 4.5 K routines, 232 KSLOC Pascal, 112 KSLOC Fortran	*0.72 0.70	The first result was for Pascal, the second Fortran.
	[LH89]	Procedure	SLOC vs CC	Pascal	1 stand-alone commercial system, 7 K procedures	*0.96	
	[GBB90]	Program	LOC vs CC	COBOL	311 student programs	*0.80	
	[HS90]	Module	LOC vs CC	Pascal	981 modules from 27 course projects	0.40	10% outliers were removed.
○	[GK91]	Module	SLOC vs CC	Pascal & COBOL	19 systems, 824 modules, 150 KSLOC	0.90	The paper also compared different variants of CC.
○	[ONe93]	Program	LOC vs CC	COBOL	3 K programs	*0.76	
	[KS97]	File	SLOC vs CC	COBOL	600 modules of a commercial system	*0.79	
	[FO00]	Module	LOC^2 vs CC	Unreported	380 modules of an Ericson system	0.62	Squaring the LOC variable was performed as an argument for the non-linear relationship.
	[GKM+00]	File	LOC vs CC	C & DLSs	1.5 MLOC subsystem of telephony switch, 2.5 K files	0.94	

Table 3.1: (Continued)

	Publication	Level	Correlation	Language	Corpus	R^2	Comments
	[EBG+01]	Class	SLOC vs CC	C++	174 classes	0.77	A study discussing the confounding factor of size for OO metrics, WMC is a sum of CC for the methods of a Class.
	[SBV01]	File	LOC vs CC	RPG	293 programs 200 KLOC	0.86	
	[MPY+05]	Module	LOC vs CC	Pascal	41 small programs	0.59	The programs analysed were written by the authors with the sole purpose of serving as data for the publication.
	[Scho6]	File	LOC vs CC	C	NASA JM1 data set, 22 K files, 11 KLOC	0.71	
	[vdMR07]	File	LOC vs CC	C & C++	77k small programs	0.78	The corpus contains multiple implementation of 59 different challenges. After removing outliers (high CC), the correlation was calculated on the mean per challenge.
	[CF07]	Function	SLOC vs CC	C	XMMS project, 109 KSLOC over 260 files	*0.51	
	[HGR07]	File	log(SLOC) vs log(CC)	C	FreeBSD packages, 694 K Files.	0.87	1 K files suspected of being generated code (large SLOC) were removed.
○	[HGH08]	Diff	LOC vs CC	Java & C & C++ & PHP & Python & Perl	13 M diffs from SourceForge	0.56	The paper contains a lot of different correlations based on the revision diffs from 278 projects. The authors observed lower correlations for C.

Table 3.1: (Continued)

	Publication	Level	Correlation	Language	Corpus	R^2	Comments
	[BKS+09]	File	SLOC vs CC	Java	4813 proprietary Java Modules	0.52	
○	[JHS+09]	File	log(LOC) vs log(CC)	Java & C & C++	2200 Projects from SourceForge	0.78 0.83 0.73	The authors discuss the distribution of both LOC and CC and their wide variance, and calculate a repeated median regression and recalculate R^2: 0.87, 0.93, and 0.97.
○	[HH10]	File	log(SLOC) vs log(CC)	C	ArchLinux packages, 300 K Files, of which 200 K non header files.	0.59 0.69	Observed lower correlation between CC and SLOC, analysis revealed header files are the cause. The second correlation is after removing these. The authors also show the correlation for ranges of SLOC.
○	[MHL+10]	Class	SLOC vs CC	Java & C++	800 KSLOC in 12 hand-picked projects	0.66	
	[MS11]	Class	SLOC vs max CC and mean CC	Java	Arc dataset: 234 classes	0.12 0.08	Correlations were not statistically significant.
	[TAA14]	Module	LOC vs CC	C	NASA CM1 dataset	0.86	
○	[JMF14]	Function	LOC vs CC	C	Linux kernel	*0.77	The authors show the scatter-plot of LOC vs CC, and report on a high correlation. Hereafter they limit to methods with a CC higher than 100, for these 138 functions they find a much lower correlation to SLOC.

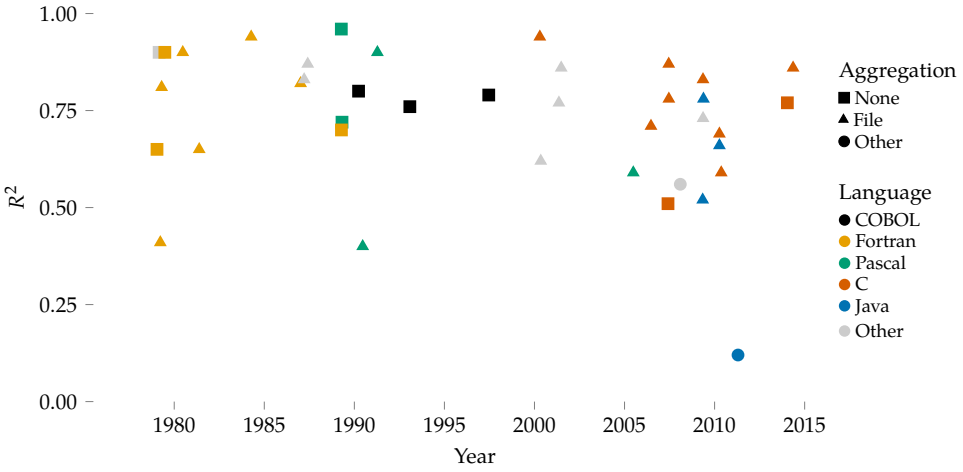


Figure 3.1: Visualization of the R^2 reported in related work (Table 3.1). The colors denote the most common languages, and the shape the kind of aggregation; aggregation "None" means that the correlation has been reported on the level of a subroutine. Note that for languages such as COBOL the lowest level of measurement of CC and SLOC is the File level. Therefore, these are reported as an aggregation of "None" (similar to the * indication in Table 3.1) .

3.2.3 *Aggregating CC over larger units of code*

CC applies to control flow graphs. As such CC is defined when applied to code units which have a control flow graph. This has not stopped researchers and tool vendors to sum the metric over larger units, such as classes, programs, files and even whole systems. We think that the underlying assumption is that indicated "effort of understanding" per subroutine would add up to indicate total effort. However, we do not clearly understand what such sums mean when interpreted back as an attribute of control flow graphs, since the compositions of control flow graphs that these sums should reflect do not actually exist.

Perhaps not surprisingly, in 2013 Yu et al. [YM13] found a Pearson correlation of nearly 1 between whole system SLOC and the sum of all CC. They conclude the evolution of either metric can represent the other. One should keep in mind, however, that choosing the appropriate level of aggregation is vital for validity of an empirical study: failure to do so can lead to an ecological fallacy [PFD11] (interpreting statistical relations found in aggregated data on individual data). Similarly, the choice of an aggregation technique can greatly affect the correlation results [MAL+13; VSvdB11a; VSvdB11b].

Curtis and Carleton [CC94] and Shepherd [She88] were the first to state that without a clear definition of what source code complexity is, it is to be expected that

metrics of complexity are bound to measure (aspects of) code size. Any metric that counts arbitrary elements of source code sentences, actually measures the code's size or a part of it. Both Curtis and Carleton, and Shepherd conclude that this should be the reason for the strong correlation between SLOC and CC. However, even though CC is a size metric; it still measures a different part of the code. SLOC measures all the source code, while CC measures only a part of the statements which govern control flow. Even if the same dimension is measured by two metrics that fact alone does not fully explain a strong correlation between them. We recommend the work of Abran [Abr10], for an in-depth discussion of the semantics of CC.

Table 3.1 lists which studies use which level of aggregation. Note that the method of aggregation is *sum* in all but one of the papers reviewed. A possible explanation for strong correlations could be the higher levels of aggregation. This brings us to our third hypothesis:

Hypothesis 3 *The correlation between aggregated CC for all subroutines and the total SLOC of a file is higher than the correlation between CC and SLOC of individual subroutines.*

If this hypothesis is true it would explain the high correlation coefficients found in literature when aggregated over files: it would be computing the sum over subroutines that causes it rather than the metric itself. Hypothesis 3 is nontrivial because it depends, per file, on the size of the bodies compared to their number what the influence of aggregation may be. This influence needs to be observed experimentally.

A confounding factor when trying to investigate Hypothesis 3 is the size of the code outside of the subroutines; such as import statements and class and field declarations in Java, and macro definitions and function headers, typedefs and structs in C. For the sake of brevity we refer to this part of source code files as the "header", even though this code may be spread over the file. A large variance in header size would negatively influence correlation on the file aggregation level which may hide the effect of summing up the CC of the subroutines. We do not know exactly how the size of the header is distributed in C or Java files and how this size relates to the size of subroutines. To be able to isolate the two identified factors on correlation after aggregation we also introduce the following hypothesis:

Hypothesis 4 *The more subroutines we add up the CC for – the more this aggregated sum correlates with aggregated SLOC of these subroutines.*

This hypothesis isolates the positive effect of merely summing up over the subroutines from the negative effect of having headers of various sizes. Hypothesis 4 is nontrivial for the same reasons as Hypothesis 3 is nontrivial.

3.2.4 *Data Transformations*

Hypothesis 1 is motivated by the earlier results from the literature in Table 3.1. Some newer results of strong correlation are only acquired after a log transform on both variables [FF79; HGR07; HH10; JHS+09]: indeed, log transform can help to normalize distributions that have a positive skew [She07] (which is the case both for SLOC and for CC) and it also compensates for the "distorting" effects of the few but enormous elements in the long tail. A strong correlation which is acquired after log transform does not directly warrant dismissal of one of the metrics, since any minor inaccuracy of the linear regression is amplified by the reverse log transform back to the original data. Nevertheless, the following hypothesis is here to confirm or deny results from literature:

Hypothesis 5 *After a log transform on both the SLOC and CC metrics, the Pearson correlation is higher than the Pearson correlation on the untransformed data.*

We note that the literature suggests that the R^2 values for transformed and untransformed data are not comparable [Kvå85; Loe90]. However, we do not attempt to find the best model for the relation between CC and SLOC, rather to understand the impact of log transformation as used by previous work on the reported R^2 values.

3.2.5 *Corpus Bias*

The aforementioned log transform is motivated in literature after observing skewed long tail distributions of SLOC and CC [HGR07; HH10; JHS+09; TT95]. On the one hand, this puts all related work on smaller data sets which do not interpret the shape of the distributions in a different light. How to interpret these older results? Such distributions make relatively "uninteresting" smaller subroutines dominate any further statistical observations. On the other hand, our current work is based on two large corpora (see Section 3.3). Although this is motivated from the perspective of being as representative as possible for real world code, the size of the corpus itself does emphasize the effects of really big elements in the long tail (the more we look, the more we find) as well as strengthens the skew of the distribution towards the smaller elements (we will find disproportionate amounts of new smallest elements). Therefore we should investigate the effect of different parts of the corpus, ignoring either elements in the tail or ignoring data near the head:

Hypothesis 6 *The strength of the linear correlation between SLOC and CC is improved by ignoring the smallest subroutines (as measured by SLOC).*

Hypothesis 7 *The strength of the linear correlation between SLOC and CC is improved by ignoring the largest subroutines (as measured by SLOC).*

Hypothesis 6 was also inspired by Herraiz and Hassan's observation of an increasing correlation for the higher ranges of SLOC [HH10]. One could argue that the smallest of subroutines are relatively uninteresting, and a correlation which only holds for the more nontrivial subroutines would be satisfactory as well.

Hypothesis 7 investigates the effect of focusing on the smaller elements of the data, ignoring (parts of) the tail. Inspired by related work [HS90; HGR07; vdMR07] that assumes that these larger subroutines can be interpreted as "outliers". It is important for the human interpretation of Hypothesis 1 to find out what their influence is. Although there are not that many tail elements, a linear model which ignores them could still have value.

3.3 EXPERIMENTAL SETUP

In this section we discuss how the study has been set up. To perform empirical evaluation of the relation between SLOC and CC for subroutines we needed a large corpus of such subroutines. To construct such a corpus we have processed Sourcerer [LBN+09] , a collection of 19 K open source Java projects (Section 3.3.1) and Gentoo†, a full Linux distribution containing 9.6 K C packages (Section 3.3.2). Then SLOC and CC have been computed for each method or function (subroutine) in the corpus (Sections 3.3.3 and 3.3.4). Finally, we performed statistical analysis of the data (Section 3.3.5).

3.3.1 *Preparing the Java Corpus*

Sourcerer [LBN+09] is a large corpus of open source Java software. It was constructed by fully downloading the source code of 19 K projects, of which 6 K turned out to be empty. The following process was used to construct our Java corpus based on these projects.

Remove non-Java files While Sourcerer contains a full copy of each project's Source Code Management (SCM), because of our focus on Java, we excluded all non-Java files.

Remove SCM branches When Sourcerer was compiled the whole SCM history was cloned. In particular, this means that multiple versions of the same system are present. However, inclusion of multiple similar versions of the same method would bias statistical analysis. Therefore, we removed all directories named `/tags/`, `/branches/`, and `/nightly/` which are commonly used to indicate snapshot copies of source trees or temporarily forked development.

Remove duplicate projects Sourcerer projects have been collected from multiple sources including Apache, Java.net, Google Code and SourceForge. Based on Sourcerer's meta-data we detected 172 projects which were extracted from

† `https://www.gentoo.org/`

multiple sources – e.g., from both SourceForge and Google Code. Similarly to removal of SCM branches we have kept only one version of each project, in this case we chose the largest version in bytes.

Manually reviewed duplicate files We calculated the MD5 hash per file. The 278 projects containing more than 300 duplicate files (equal hash) were manually reviewed and fixed in case the duplication could be explained. Common reasons were non-standard SCM structure (different labels for tags and branches) and the code of third-party libraries.

Remove out-of-scope code Finally, we have decided to remove code which is either external to the studied project, or is test code. It is *a priori* not clear whether test code exhibits the same relation between SLOC and CC as non-test code. We removed all directories matching the following case-insensitive regular expression: `/[/\-]tests?\/|\/examples?\/|(third|3rd)[\-\_]?party/`.

Performing these steps we have reduced the 390 GB corpus to 14.30 GB containing 13 K projects over 2 M files. The resulting corpus has been made publicly available [Lan15a].

3.3.2 *Preparing the C Corpus*

We are not aware of a C corpus of size, age, and spread of domains comparable to Sourcerer. Therefore we have constructed a new corpus based on Gentoo's Portage packages[‡]. We have chosen Gentoo because its packages cover a wide range of domains. Compared to other Linux distributions, Gentoo distributes the source code instead of pre-compiled binaries, enabling our analysis.

On October 14, 2014 the repository contained 65 K packages. The extensions of 40 K packages indicated an archive (for example `tar.gz`). The following process was used to construct our C corpus based on these packages.

Remove non-code packages We filtered debug-symbols, patch-collections, translations, binary-installers, data-packages, binary packages, auxiliary files, and texlive modules.

Remove multiple versions The Portage repository of Gentoo contains multiple versions of packages. We kept only the newest version of every package. Note that Portage does come with meta-data – "ebuild" – to collect the latest Gentoo packages, selecting a sub-set of the entire repository. We refrained from using this meta-data, because it is based on design decisions which would introduce a selection bias (like hardening for security and library compatibility).

Extract packages The remaining 20 K packages were unpacked, resulting in 8 M files.

Detect C code C and C++ code share file extensions. Both `.c` and `.h` can contain C or C++ code.

[‡] `https://packages.gentoo.org/`

Using heuristics inspired by GitHub's linguist [Pee+15], we developed a tool to detect if a file contained either C or C++ code. The heuristics uses syntactical differences to detect C++ and differences between the often included standard library header files for C and C++.
Of the 1.35 M files with C extensions, 1.02 M contained C code, and 0.33 M contained C++. We removed all the files with C++ code.

Remove out-of-scope code Similarly to the preparation of our Java corpus, we have chosen to remove code which is not part of the application or library studied. We have used the exact same filter, removing the folders: tests, examples, and third-party.

Detect duplicates Similarly to the preparation of our Java corpus, we calculated the MD5 hash of all the files. The 223 packages containing more than 300 duplicate files were manually reviewed and fixed in case the duplication could be explained. Common reasons were failures in detecting multiple versions (90 packages), forks, and included third-party libraries.

Keep only related files For the packages still containing C files, we also kept all files related to the possible compilation of the library. All other files were removed.

Performing these steps resulted in a corpus of 19 GB containing 9.8 K packages with 13 GB of C code in 798 K files. The corpus is publicly available [Lan15b].

3.3.3 *Measuring Java's SLOC and CC*

While numerous tools are available to measure SLOC and CC on a file level[§], to perform our study we require to calculate SLOC and CC per method and to precisely control the definition of both metrics. We use the M3 framework [BHK+15], which is based on the Eclipse Java Development Tools (JDT)[¶], to parse the full Java source code and identify the methods in the corpus. This also generates full ASTs for each method for further analysis. Listing 3.1 depicts the source code of computing the CC from the AST of a method. The code recursively traverses the AST and matches the enumerated nodes, adding 1 for each node that would generate a fork in the Java control flow graph.

For SLOC we decided not to depend on the information in the Eclipse ASTs (ASTs are not designed for precisely recording the lexical syntax of source code). Instead we use the ASTs only to locate the source code of each separate method. To compute its SLOC we defined a grammar in Rascal [KvdSV09] to tokenize Java input into newlines, whitespace, comments and other words. The parser produces a list of these tokens which we filter to find the lines of code that contain anything else but whitespace or

[§] e.g., `http://cloc.sourceforge.net/`, `http://www.sonarqube.org/`

[¶] `http://www.eclipse.org/jdt`

Listing 3.1: Rascal source code to calculate the CC of a given method. The `visit` statement is a combination of a regular `switch` and the visitor pattern. The cases pattern match on elements of the AST.

```
1  int calcCC(Statement impl) {
2      int result = 1;
3      visit (impl) {
4          case \if(_,_) : result += 1;
5          case \if(_,_,_) : result += 1;
6          case \case(_) : result += 1;
7          case \do(_,_) : result += 1;
8          case \while(_,_) : result += 1;
9          case \for(_,_,_) : result += 1;
10         case \for(_,_,_,_) : result += 1;
11         case foreach(_,_,_) : result += 1;
12         case \catch(_,_): result += 1;
13         case \conditional(_,_,_): result += 1;
14         case infix(_,"&&",_) : result += 1;
15         case infix(_,"||",_) : result += 1;
16     }
17     return result;
18 }
```

comments. We tested and compared our SLOC metric with other tools measuring full Java files to validate its correctness.

To be able to compare SLOC of only the subroutines compared to SLOC of the entire file we store the SLOC of each Java method body separately (see Hypothesis 4). For Java, files without method bodies, such as interface definitions, were ignored. Out of the 2 M files, 306 K were ignored since they did not contain any method bodies.

3.3.4 *Measuring C's SLOC and CC*

To perform our analysis on the C code we use the Software Analysis Toolkit (SAT) of the Software Improvement Group[‖] (SIG). This proprietary toolkit uses a robust analysis approach, processes over a billion SLOC per year and forms the basis of the consultancy services of SIG. As part of these services the measurements performed by the toolkit are continuously validated, both by the internal development team as well as externally by the development teams of clients and third-party suppliers.

The measurement process of the SAT consists roughly of four phases: preprocessing, tokenization, scope creation, and measurements. In the first phase, preprocessor directives are removed from the source-code. This step is required to solve issues such as illustrated in Listing 3.2 where only one unit-declaration ends up in the final binary depending on whether `debug` is defined. When both parts are kept two unit

[‖] `http://www.sig.eu`

Listing 3.2: C code example with conditional pre-processor directives.

```
1 #ifdef debug
2     void get_string(char prefix) {
3 #else
4     void get_string() {
5 #endif
6     }
```

headers, but only a single close-bracket would be used as input to the next phase. To prevent problems in the scope creation phase, i.e. not being able to find the correct units, only the first code blocks of conditional preprocessor directives are kept. I.e, in the code in Listing 3.2 only the second and sixth line is passed on to the next phase.

This pragmatic approach is used because running the preprocessor is prone to errors and labour intensive due to projects relaying on specific tools and versions. Moreover, choosing a representative set of system constants is often not possible and adds unnecessary complexity to the assessment process. Processing all sources in the same way reduces overhead and makes the measurement step more objective. In our experience, choosing the first preprocessor block captures most of the code and provides reliable results in assessments where the results are validated with the development teams. Since this validation step is not possible in this experiment all files which after processing contain unbalanced curly braces are removed from the corpus.

In the second phase the code is tokenized using an internally developed tokenizer. The resulting list of tokens is used in the scope creation phase to extract a scope tree containing subroutines, modules, and packages (depending on the language). For C, the token list is inspected for patterns representing the headers of subroutines (for example the second line in the code above) and the body blocks (the brackets on line two and six). These scope blocks are then put into an internal graph structure.

To perform the actual measurements all nodes representing subroutines are processed by a visitor which works on the list of tokens associated with the node. Similar to the approach for Java, SLOC is measured by identifying all lines within a function which contain anything else than comments or whitespace. To calculate the CC all tokens representing the keywords `case`, `if`, `for` and `while` and the operators `||`, `&&` and `?` are counted. Note that since we match on tokens instead of AST nodes the `while` token also captures any `do...while` statements, making this implementation equal to the one defined for Java – Listing 3.1.

C code is split over `.c` and `.h` files. Herraiz and Hassan [HH10] ignored all headers files (`.h`), but we did include them. The reason is that for C, although it is a less common idiom, putting functions in a header file is possible. Our C corpus contains

333 K header files. We chose to ignore all `.c` and `.h` files without any function bodies (similar to Java interfaces). This results in removing 310 K `.h` and 23 K `.c` files.

3.3.5 *Visualization & Statistics Methods*

Before discussing the results (Section 3.4), we will first discuss the chosen visualizations and statistical methods.

Distributions

Before comparing SLOC and CC, we describe the distributions in our data using histograms and descriptive statistics (median, mean, min and max). The shape of distributions does have an impact on the correlation measures used, as explained above. All results (Section 3.4) should be interpreted with these distributions in mind.

Hexagonal Scatter plots

Scatter plots with SLOC on the x-axis and CC on the y-axis represent the data in a raw form. Due to the long tail distributions of both CC and SLOC, the data is concentrated in the lower left quadrant of the plots and many of the dots are placed on top of each other. Therefore, we also use log-log scatter plots. We use hexagonal scatter plots [CLN+87] to address overplotting and Type I errors (false positives). The latter method divides the two-dimensional plane of the plot area in 50 times 50 hexagons. It then counts how many of the data points fall into each individual hexagon and uses a logarithmic 255-step gray scale gradient to color it. Compared to vanilla scatter plots the hexagonal plots are a lot less confusing; the main problem is that a limited resolution on paper can create artifacts such as big black blobs of ink where in fact the raw data does not feature maximum density at all (i.e. overplotting causing Type I errors). Nevertheless, it should be noted that the gradient as well as human perception have a limited resolution and as such hexagonal plots can still hide the full impact of the skewness of the distributions and the variance in the data.

Correlation

Most related work, if reported, uses Pearson product-moment correlation coefficient [Pea95] (hereafter Pearson correlation), measuring the degree of linear relationship between two variables. The square of Pearson correlation is called the coefficient of determination (R^2). R^2 estimates the variance in the power of one variable to predict the other using a simple linear regression. Hereafter we report the R^2 to describe a correlation.

Many researchers have observed that the distributions of SLOC (and CC) are right-skewed. While opinions differ on robustness of the Pearson correlation against

normality violations [EN84; She07], a number of earlier studies attempt to compensate for the skewness of the distribution by applying a log transform and then compute the Pearson correlation [FF79; HGR07; JHS+09]. The important matter of interpreting the results after a log transform back to the original data is discussed in Section 3.5.

Other researchers have transformed the data using more advanced methods in order to improve the chances for linear correlation. For example, using Box-Cox transformation [HH10] or performing the Repeated Median Regression (RMR) method on a random sample [JHS+09]. Box-Cox is a power transform similar to the basic log transform. We have chosen to stick with the simpler method, following the rest of the related work which we are trying to reproduce (Hypothesis 5).

The next method, RMR, may be useful to find some linear model, but it entails a lossy transformation. The median regression method reduces the effect of random measurement errors in the data by computing a running median. We do not have random errors in the CC or SLOC measurements, so a running median would hide interesting data. Therefore, RMR is outside the scope of this chapter.

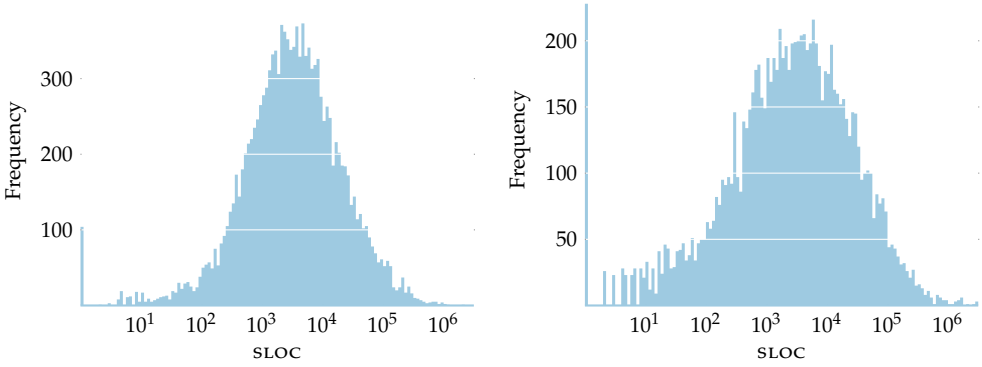
If no linear correlation is to be expected, or is found using Pearson's method, we use Spearman's rank-order correlation coefficient [Spe04] (hereafter Spearman correlation or ρ). Similarly to the Pearson correlation, Spearman's correlation is a bivariate measure of correlation/association between two variables. However, opposed to the Pearson correlation, Spearman's correlation is employed with rank-order data, measuring the degree of monotone relationship between two variables. We apply this method only for completeness sake, since it does not generate a predictive model which we could use to discard one of the metrics.

Regression

The square of Pearson's correlation coefficient is the same as the R^2 in simple linear regression. Hence, if we would find a strong correlation coefficient we would be able to construct a good predictive linear model between the two variables, and one of the metrics would be obsolete. It is therefore important to experimentally validate the reported high correlation coefficients in literature (see Table 3.1). In general for other correlation measures (such as Spearman's method) this relation between regression and correlation is not immediate. In particular, a strong Pearson correlation coefficient after a log transform does not give rise to an accurate linear regression model of the original data. We discuss this in more detail later when interpreting the results in Section 3.5.

3.4 RESULTS

In this section we report the results of our experiments and the statistics we applied to it. We postpone discussion of these results until Section 3.5.



(a) Java Projects (b) C packages

Figure 3.2: Distribution of the non-empty projects/packages over their total SLOC. SLOC is on a $\log_{10}$ scale, bin width is 0.05.

Table 3.2: Statistics of the total SLOC per project in the corpus.

Corpus	Min.	25%	Median	Mean	75%	Max.
Java	0	1009	3219	15 270	10 250	2 207 000
C	1	671	3036	21 200	12 430	3 333 000

3.4.1 *Distributions for Java and C*

Figure 3.2 shows the histogram of SLOC per project and Table 3.2 describes this distribution. The Java corpus contains 17.6 M methods spread out over 1.7 M files and the C corpus has 5.8 M functions spread over 760 K files. The C corpus seems to have a disproportional number of packages with a low SLOC, even on the logarithmic scale. After randomly inspecting a number of packages in the range between 1 and 20 files we concluded that next to naturally small packages these are C files which are part of larger packages written in other languages such as Java, Python or Perl. Lacking any argument to dismiss these files, we assume them to be just as representative of arbitrary C code as the rest.

Figure 3.3 shows the distribution of SLOC per Java method and C function. Table 3.3 describes their distributions. We observe skewed distributions with a long tail. To measure the degree of skewness we calculate the moment coefficient of skewness [JG98], i.e. the third standardized moment of the probability distribution. A positive value indicates that the right-hand tail is longer or fatter than the left-hand one. A negative value indicates the reverse. A value close to zero suggest a symmetric

Table 3.3: Descriptive statistics of the SLOC and CC per Java method and C function.

Corpus	Variable	Min.	25%	Median	Mean	75%	Max.
Java	SLOC	1	3	3	9.38	9	33 850
	CC	1	1	1	2.33	2	4377
C	SLOC	1	6	12	26.49	27	44 880
	CC	1	1	3	5.97	6	18 320

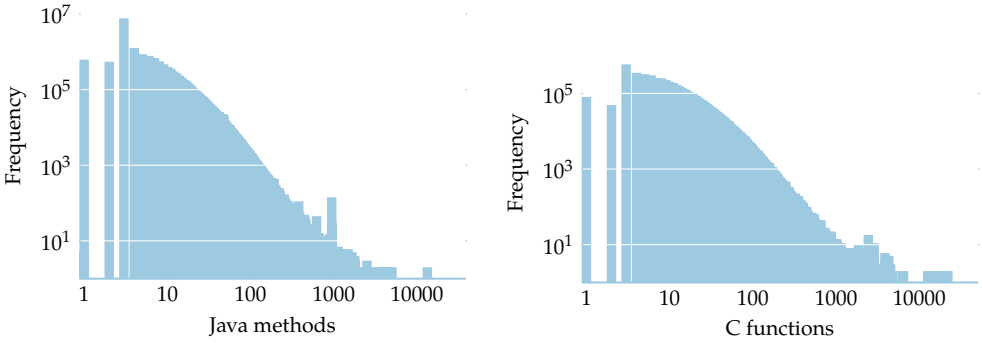


Figure 3.3: Histogram of the SLOC per subroutine in both corpora, in log-log space (bin width is 0.1). Here we see that for both Java and C, small methods and functions are the most common. The bar around 1000 for Java and 3000 for C are two cases where a project contained multiple files of generated code that slightly differed per file. See Figure 3.5 to compare the distribution.

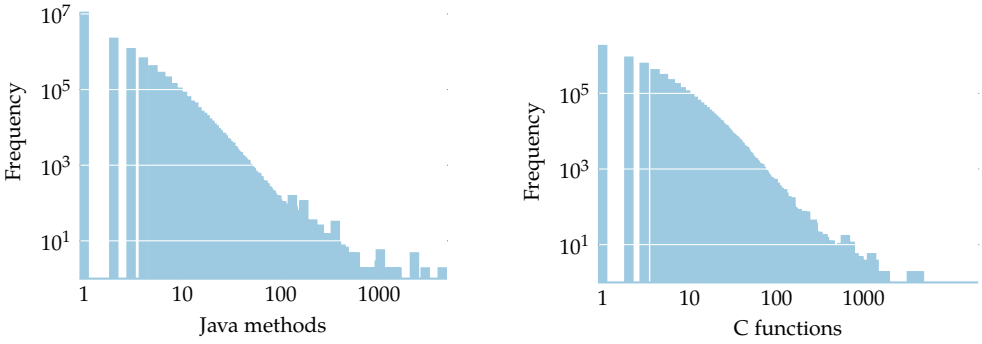


Figure 3.4: Histogram of the CC per subroutine in both corpora, in log-log space (bin width is 0.1). Here we see that for both Java and C, methods and functions with little control flow are the most common. See Figure 3.5 to compare the distribution.

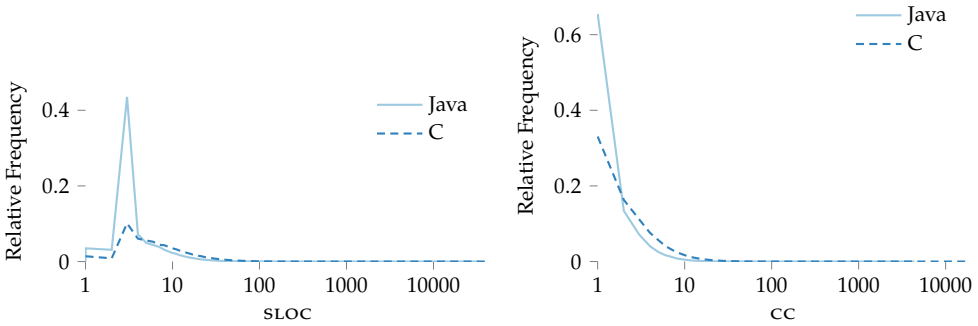


Figure 3.5: Relative frequency polygons for both corpora and both variables. The variables are displayed on a logarithmic scale. Relative frequency polygons are histograms normalized by the amount of data points, the area under the curve is 1. They visualize the relative difference between distributions.

distribution. For our corpora the moment coefficient of skewness equals 234.75 for SLOC in Java and 107.28 for SLOC in C. After the log transform it equals 1.05 for Java and 0.40 for C.

This means that the mean values are not at all representative for the untransformed corpora, and that the smallest subroutines dominate the data. For Java, 8.8 M of the methods have 3 SLOC or fewer. This is 50% of all data points. There are 1.2 M methods with 1 or 2 SLOC, these are the methods with an empty body, in two different formatting styles or (generated) methods without newlines. The other 7.6 M methods of 3 SLOC contain the basic getters, setters, and throwers pattern frequently seen in Java methods – often called one-liners. For C, this is less extreme, only 12% of the functions have a SLOC of 3 or less. The corpora differ in the strength of the skewness here: the C corpus has proportionally fewer of the smallest subroutines than the Java corpus has. Nevertheless both plots have their mode at 3 SLOC.

Figure 3.4 shows the distribution of CC per Java method and C function. For the Java corpus, 15.2 M methods have a CC of 3 or less. This is 86% of all data points. There are 11.6 M methods without any forks in the control flow (1 CC), i.e. 65%. This observation is comparable with the 64% reported by Grechanik et al. for 2 K randomly chosen Java projects from SourceForge [GMD+10]. We observe that the lion's share of Java methods are below the common CC thresholds of 10 (97.00%) [McC76] or 15 (98.60%) [MK93]. The C corpus shows a comparable picture, but again with a more even distribution which puts less emphasis on the smallest subroutines. For C the median is at 3 while for Java it was 1. Still 33% of the C subroutines have a CC of 1 (straight line code). We do see that both corpora have their mode of CC at 1. For C 85.60% functions are below the common CC threshold of 10 and 91.70% below 15.

Comparing the shape of Java's and C's distributions is complicated by the difference in corpus size. To visualize the difference in the distribution, we have used relative frequency polygons (Figure 3.5). These relative frequency polygons are normalized by the size of the corpus and thus the area under the curve is 1. This more clearly shows the difference in distribution between Java and C; for Java there are relatively more methods with a small SLOC and CC than C functions. The shape of the distributions is a controversial matter which we consider outside the scope of this article.

3.4.2 *Scatter plots*

Figure 3.6 shows two zoomed in (CC $\leq$ 500 and SLOC $\leq$ 1800) hexagonal scatter-plots of the subroutines in our corpus. Due to the skewed-data, this figure still shows 99.98% of all data points. Figure 3.7 shows the same hexagonal scatter-plots in a log-log space, allowing to show more data. The two black lines in both figures show the linear regressions before and after the log transform which will be discussed in Section 3.4.3. The logarithmic grayscale gradient of the points in the scatter-plot visualizes how many subroutines have that combination of CC and SLOC: the darker, the more data points. Figure 3.8 shows an even more zoomed in range of the scatter-plots, in these box plots we can more clearly see the variance of CC increasing as SLOC increases. Moreover the median is increasing, but so is the inter-quartile range. We have not created these plots for the full range of the data since these plots do not scale.

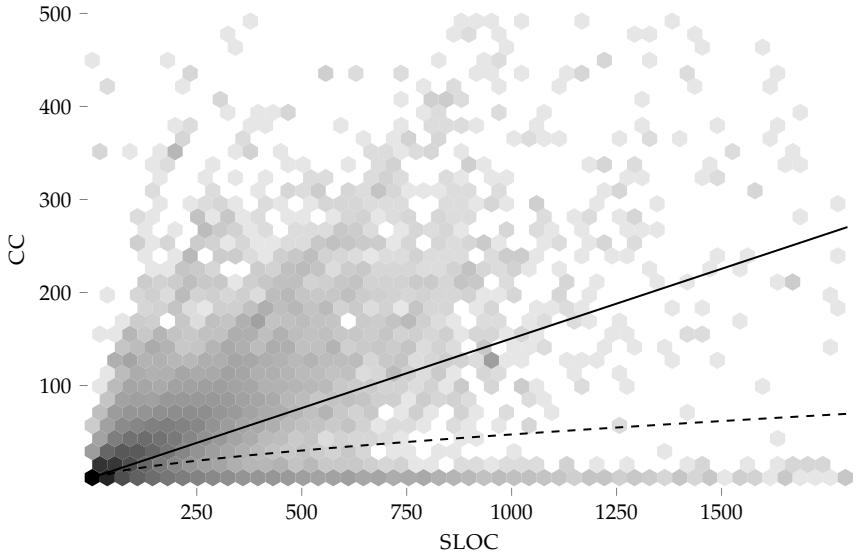
Figures 3.6 and 3.7 show a widely scattered and noisy field, with a high concentration of points in the left corner. The outline of these concentrations might hint at a positive (linear) monotone relation. However, the same outline is bounded by the minimum CC number (1) and the expected maximum CC number (CC is usually not higher than SLOC given a source code layout of one conditional statement on a single line).

We do find some points above the expected maximum CC, which we found out to be generated code and code with dozens of Boolean operators on one single line.

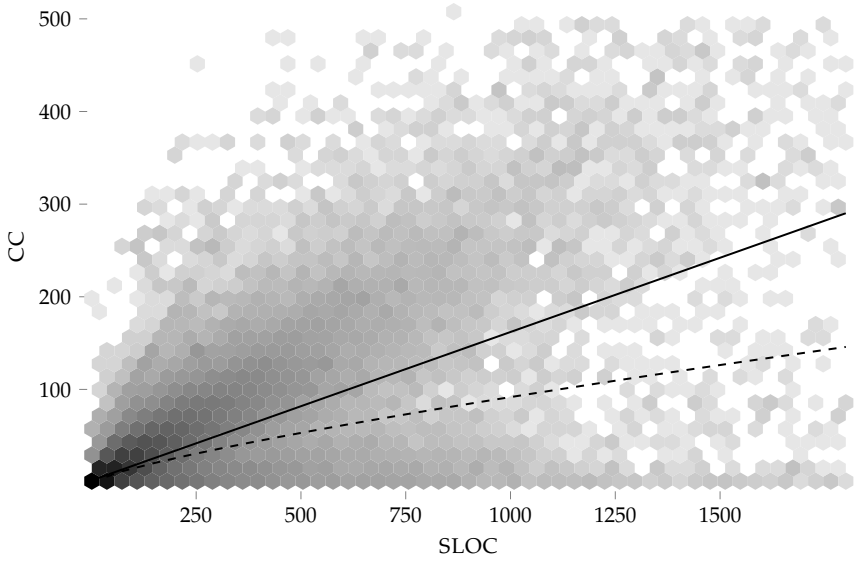
3.4.3 *Pearson correlation*

In Table 3.4, the first row shows the Pearson correlation over the whole corpus. The R^2 of SLOC and CC is 0.40 for Java and 0.43 for C. Figures 3.6(a) and 3.6(b) respectively depicts these linear fits, $\text{CC} = 0.92 + 0.15 \cdot \text{SLOC}$ and $\text{CC} = 1.73 + 0.16 \cdot \text{SLOC}$, as a solid black line. These R^2 are much lower than the related work in Table 3.1, even if we focus on the related work at the subroutine/function/method level.

The Pearson correlation after a log transform showed higher numbers, which are more in line with related work that also applies a log transform [FF79; HGR07; HH10; JHS+09]. The fit for Java, the dashed line in Figures 3.6(a) and 3.7(a), is $\log_{10}(\text{CC}) = -0.28 + 0.65 \cdot \log_{10}(\text{SLOC}) \Leftrightarrow \text{CC} = 10^{-0.28} \cdot \text{SLOC}^{0.65}$. The fit for C

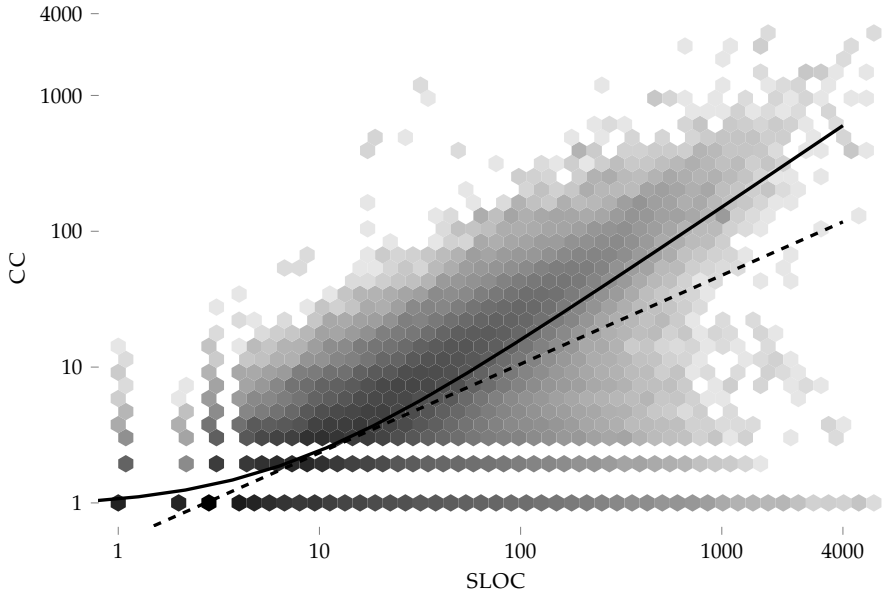


(a) Java

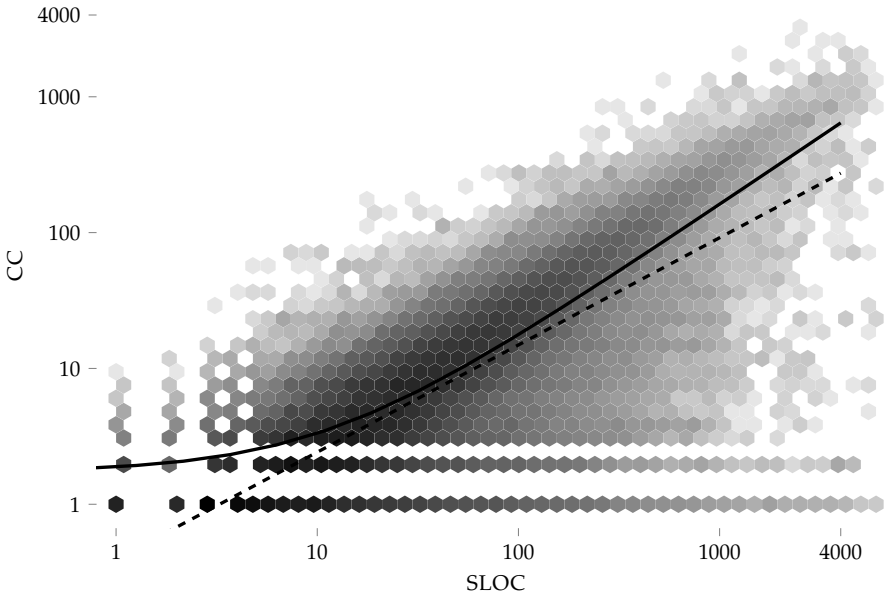


(b) C

Figure 3.6: Scatter plots of SLOC vs CC zoomed in on the bottom left quadrant. The solid and dashed lines are the linear regression before and after the log transform. The grayscale gradient of the hexagons is logarithmic.

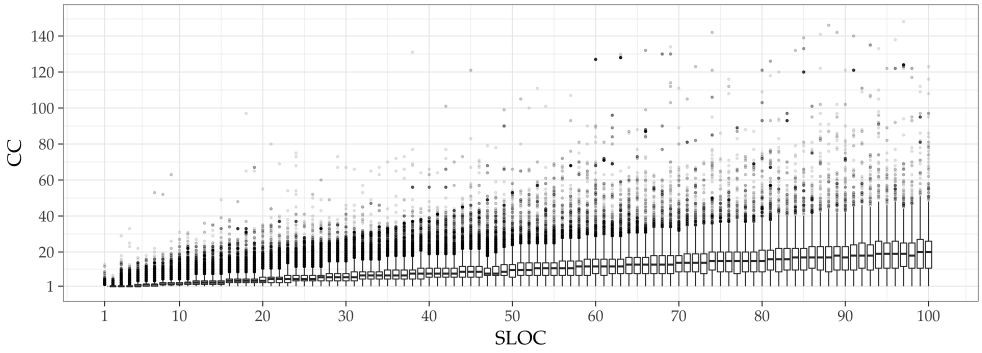


(a) Java

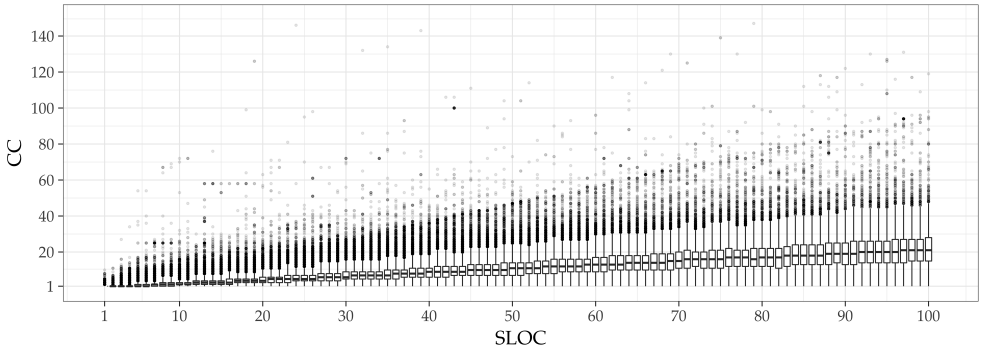


(b) C

Figure 3.7: Scatter plots of SLOC vs CC on a log-log scale. The solid and dashed lines are the linear regression before and after the log transform. The grayscale gradient of the hexagons is logarithmic.



(a) Java



(b) C

Figure 3.8: Box plots of CC per SLOC on the lower range, illustrating the wide spread of Figure 3.7(a) and Figure 3.7(b). The median is the black line in the box, bottom and top of the box are the first and third quartile, the hinges are at the traditional 1.57 $*$ inter-quartile range [CWT83].

(Figures 3.6(b) and 3.7(b)) is $\text{CC} = 10^{-0.41} \cdot \text{SLOC}^{0.79}$. More on the interpretation of this transform and the results is discussed in Section 3.5.

As discussed earlier, the data is skewed towards small subroutines and simple control flow graphs. Since 50% of Java's method and 12% of C's functions have a SLOC between 1 and 3, these points have a high influence on the correlation. We could argue that the relation – between SLOC and CC – for these smaller subroutines are less interesting. Therefore, to test Hypothesis 6, Table 3.4 also shows the Pearson correlations for parts of the tail of the SLOC variable**. Each row shows a different percentage of the tail of the data, and the minimum SLOC for that part.

**Normal quantiles do not make sense for this data since the first few buckets would hold most of the data points for only a few of the CC and SLOC values (e.g. 1–4)

Table 3.4: Correlations for part of the tail of the independent variable SLOC. All correlations have a high significance level ($p \leq 1 \times 10^{-16}$).

(a) Java methods

Min. SLOC	Coverage	R^2	$\log R^2$	ρ	Methods
1	100%	0.40	0.68	0.80	17 633 256
3	50%	0.37	0.58	0.74	8 816 628
5	40%	0.36	0.50	0.67	7 053 303
9	25%	0.34	0.38	0.60	4 408 314
11	20%	0.33	0.33	0.57	3 526 652
20	10%	0.30	0.20	0.50	1 763 326
77	1%	0.21	0.03	0.33	176 333
230	0.100%	0.14	0.00	0.21	17 634
688	0.010%	0.08	0.00	0.17	1764

(b) C functions

Min. SLOC	Coverage	R^2	$\log R^2$	ρ	Functions
1	100%	0.43	0.70	0.83	5 810 834
12	50%	0.41	0.52	0.70	2 905 417
16	40%	0.40	0.47	0.68	2 324 334
27	25%	0.38	0.37	0.63	1 452 709
33	20%	0.38	0.33	0.61	1 162 167
56	10%	0.35	0.22	0.55	581 084
220	1%	0.28	0.05	0.38	58 109
714	0.100%	0.20	0.01	0.28	5811
2695	0.010%	0.13	0.00	0.04	582

Perhaps surprisingly the higher the minimum SLOC – Table 3.4 – the worse the correlation. This directly contradicts results from Herraiz and Hassan [HH10], who reported improving correlations for higher regions of SLOC. However, Jbara et al. [JMF14] also reported decreasing correlations, except that they looked at higher CC instead of SLOC.

In three papers we cited earlier [HS90; HGR07; vdMR07] the largest subroutines are removed from the data before calculating correlation strength, as opposed to removing the smallest subroutines (see above). To be able to compare we report in Table 3.5 the effect of removing different percentages of the tail (related to Hypothesis 7). We mention the maximum SLOC which is still included in each sub-set.

We further explore removing *both* the smallest and the largest subroutines. We observed that for a fixed maximum SLOC, increasing the minimum SLOC results in lower R^2 (similarly to Table 3.4). We further observe that for a fixed minimum SLOC, increasing the maximum SLOC results in the increase of R^2 followed by the decrease

Table 3.5: Correlations for part tail of the independent variable SLOC *removed*. All correlations have a high significance level ($p \leq 1 \times 10^{-16}$).

(a) Java methods

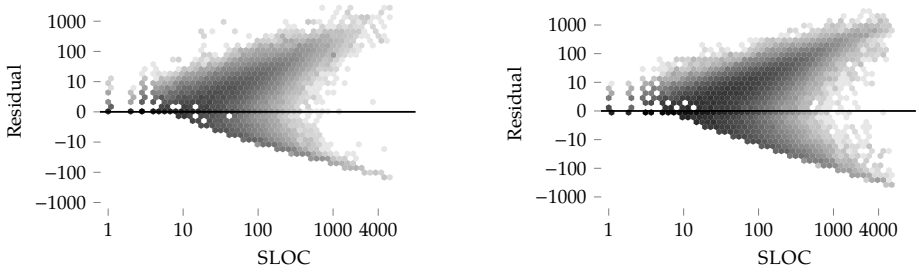
Max. SLOC	Coverage	R^2	$\log R^2$	ρ	Methods
33 851	100%	0.40	0.68	0.80	17 633 256
934	99.995%	0.53	0.68	0.80	17 632 374
688	99.990%	0.54	0.68	0.80	17 631 492
230	99.900%	0.59	0.68	0.80	17 615 622
77	99%	0.59	0.67	0.79	17 456 923
20	90%	0.51	0.55	0.74	15 869 930
11	80%	0.43	0.41	0.66	14 106 604
9	75%	0.37	0.32	0.60	13 224 942
5	60%	0.07	0.04	0.28	10 579 953
3	50%	0.00	0.00	0.02	8 816 628

(b) C functions

Max. SLOC	Coverage	R^2	$\log R^2$	ρ	Functions
44 881	100%	0.43	0.70	0.83	5 810 834
3825	99.995%	0.62	0.70	0.83	5 810 543
2693	99.990%	0.62	0.70	0.83	5 810 252
714	99.900%	0.66	0.70	0.83	5 805 023
220	99%	0.66	0.69	0.83	5 752 725
56	90%	0.56	0.61	0.79	5 229 750
33	80%	0.47	0.53	0.75	4 648 667
27	75%	0.43	0.49	0.73	4 358 125
16	60%	0.33	0.37	0.65	3 486 500
12	50%	0.26	0.28	0.58	2 905 417

(similarly to Table 3.5). Finally we observe that the optimal R^2 values are obtained when no small subroutines are eliminated and the maximum SLOC is 130 for Java ($R^2 = 0.60$) and 430 for C ($R^2 = 0.67$). While the optimal R^2 values seem to be quite close, the maximum SLOC for C exceeds the maximum SLOC for Java by more than three times. This factor is reminiscent of the apparent ratios between 1st quartile, median, mean, and 3rd quartile of the Java and C corpora in Table 3.3.

As we will discuss in Section 3.5, the increasing variance in both dimensions causes the largest subroutines have a large effect on linear correlation strength. To dig further we did read the code of a number of elements in these long tails (selected using a random number generator). For Java we read ten methods out of 1762 with SLOC > 688 and for C we also read ten functions out of the 582 with SLOC > 2695. We



(a) Java (b) C

Figure 3.9: Residual plot of the linear regressions after the log transform, both axis are on a log scale. The grayscale gradient of the hexagons is logarithmic.

observed that five out of these ten methods in Java were clearly generated code and four out of the ten sampled C functions as well.

We further analyze the strength of the linear correlation after log transform (Hypothesis 5). Figure 3.9 shows the residual plot of the dashed line shown in the scatter-plots. A residual plot displays the difference between the prediction and the actual data. For a good model, the error should contain no pattern, and have a random distribution around the zero-line. Here we clearly see the variance in CC increasing as SLOC increases. This further supports results from Table 3.4, where the prediction error for CC grows with higher SLOC.

This increasing variance we observed is a form of heteroscedasticity. Heteroscedasticity refers to the non-constant variance of the relation between two variables. The Breusch-Pagan test [BP79] confirmed ($p < 2.20 \times 10^{-16}$) that the relation between CC and SLOC is indeed heteroscedastic for both Java and C. Heteroscedasticity may bias estimated standard errors for the regression parameters [BP79] making the interpretation of the linear regression potentially error-prone.

3.4.4 *Alternative explanations*

This subsection will explorer alternative explanations to further understand the impact of different choices made by related work (Section 3.2.2).

CC *variant*

As discussed in Section 3.2.1, there is confusion on which AST nodes should be counted for CC. To understand the effect of this confusion on the correlation, we have also calculated the CC without counting the `&&` and `||` Boolean operators. The CC changed for 1.3 M of the 17.6 M Java methods, of with the CC of 74.2 K methods changed by

more than 50%. For C, 1.5 M of the 5.8 M functions had a different CC, of which the CC of 73.3 K functions changed by more than 50%. However, this change has negligible effect on correlation. For Java, the R^2 changed from 0.40 to 0.41 and for C it stayed at 0.43. Similarly small effects were observed for other ranges of Table 3.4 and 3.5.

Aggregation

To investigate Hypothesis 3 we have also aggregated CC and SLOC on file level. This A/B experiment isolates the factor of aggregation. In Table 3.6 the "None" rows repeat the R^2 before aggregation for Java and C (cf. the first rows in Tables 3.5). The "File" rows show the R^2 for the aggregated CC and SLOC before and after the log transform.

Figure 3.10 shows the hexagonal scatter plots for the aggregation on file level. The two black lines show the linear regression before and after the log transform. The dashed line is the regression after log transform. It can be observed that for larger files these regressions do not seem to fit the data, i.e. smaller files dominate the fitting of the regression line.

Since the previous experiment includes the confounding factor of header size, we now report on another A/B test to investigate Hypothesis 4. We aggregate the *subroutine values* of CC and SLOC on file level. The "$\sum$ Method" and "$\sum$ Function" rows in Table 3.6 indicate the increase of R^2 both for Java and C.

In Section 3.4.3 we showed how the non-constant variance (heteroscedasticity) causes the largest subroutines to have a large impact on the correlations. To investigate the difference between file level (Hypothesis 3) and subroutine level (Hypothesis 4) aggregation we also report the effect of removing the largest files on the correlations. Removing the 5 ‰ largest files from Java (848 files) and C (231 files) – similarly to Section 3.4.3 – improves R^2 to 0.83 (from 0.64) for Java and 0.64 for C (from 0.39).

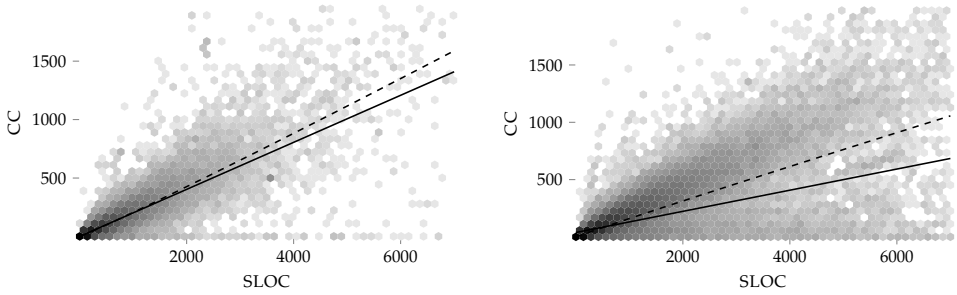
Digging further to see what kind of code could have such an large impact, we used a random number generator to sample ten large files for both corpora (SLOC > 3601 for Java and SLOC > 19934 for C). We then manually inspected the source code in these files. Five out of ten files were clearly generated code in the Java selection and nine out of ten in the C selection. Two of these generated C files, were the result of a process called "amalgamation" where the developer includes all hand-written code of a library project into a single file to help C compiler optimization or ease deployment.

3.4.5 *Spearman correlation*

Although our main hypothesis is about linear Pearson correlation, we can compute Spearman's correlation to find out if there is a monotone relation. The results are also in Table 3.4 and Table 3.5, showing reasonably high ρ values, but decreasing rapidly when we move out of the lower ranges that the distribution skews towards.

Table 3.6: Correlations (before and after a log transform) between the aggregated SLOC and CC metrics on a file level (Hypothesis 3) and after summing only the bodies of the subroutines (Hypothesis 4). The first row per language are a copy of the first rows in Tables 3.5.

Language	Aggregation	R^2	$\log R^2$
Java	None	0.40	0.68
	File	0.64	0.87
	$\sum$ Method	0.73	0.90
C	None	0.44	0.71
	File	0.39	0.84
	$\sum$ Function	0.70	0.90



(a) Java (b) C

Figure 3.10: Scatter plots of SLOC vs CC for Java and C *files*. The solid and dashed lines are the linear regression before and after the log transform. The grayscale gradient of the hexagons is logarithmic.

This indicates that for the bulk of the data it is indeed true that a new conditional leads to a new line of code; an unsurprising and much less profound observation than the acceptance or rejection of Hypothesis 1. However, it is still interesting to observe the decline of the Spearman correlation for higher SLOC which reflects the fact that many different combinations of SLOC and CC are being exercised in the larger methods of the corpus.

3.5 DISCUSSION

Here we interpret the results from Section 3.4. Note that we only have results for Java and C and we sometimes compare these informally to results on different programming languages summarized in Table 3.1.

3.5.1 *Hypothesis 1 – Strong Pearson correlation*

Compared to R^2 between 0.51 and 0.96 [CSM79; FF79; GBB90; JMF14; KS97; LH89; LV89; ONe93; WHH79] summarized in Table 3.1, our R^2 of 0.40 and 0.43 are relatively low. This is reason enough to reject the hypothesis: for Java methods and C functions there is no evidence of a strong linear correlation between SLOC and CC in these large corpora, suggesting that – at least for Java and C – CC measures a different aspect of source code than SLOC, or that other confounding factors are generating enough noise to miss the relation. Here we focus on related work with the same aggregation level and without log transforms. We conclude that these results, for different programming languages and smaller corpora, do not generalize to our corpora. For higher aggregation levels see our discussion of Hypothesis 3 below.

The cause of the low R^2 in our data seems to be the high variance of CC over the whole range of SLOC. We observe especially that the variance seems to increase when SLOC increases: the density of control flow statements for larger subroutines is not a constant. This heteroscedasticity is confirmed by the Breusch-Pagan test. Of course the shape of the distribution influences the results as well, which we investigate while answering Hypothesis 5.

Hypothesis 1: There is no evidence for strong linear correlation between CC and SLOC. Lower R^2 values can be attributed to high variance of CC for the whole range of SLOC.

3.5.2 *Hypothesis 2 – No effect of Boolean operators*

The results show that the corpora did not contain significant use of the short-circuit Boolean operators. At least there is not enough support to change the conclusion of Hypothesis 1. We can therefore not reject Hypothesis 2.

Nevertheless, the CC of 8% Java methods and 23% C functions that do use Boolean operators are influenced. It is interesting to note that these subroutines sometimes had very long lines. These subroutines would be missed when counting only SLOC or when ignoring the operators for CC.

What we conclude is that the difference between related work and our results cannot be explained by a different version of CC, since changing it does not affect the correlation. Our recommendation is that for Java and C, the CC computation should include the && and || Boolean operators, since they do measure a part of the control flow graph as discussed in Section 3.2.

Hypothesis 2: Lack of correlation can not be explained by including or excluding boolean operators in the calculation of CC.

3.5.3 *Hypothesis 3 and 4 – Effect of aggregation (sum)*

Related work [BP84; BKS+09; EBG+01; FO00; GK91; GKM+00; HS90; HGR07; HH10; JHS+09; KP87; LC87; MHL+10; MS11; MPY+05; Pai80; Sch06; SCM+79; SBV01; STU+81; TAA14; vdMR07] reported high correlations between CC and SLOC on a larger than methods/functions/subroutines level. For Java we found similar high correlation after aggregating CC and SLOC on a file level, however not for C. After removing the largest 5 ‰ files for C, we also do not find better correlations. Hypothesis 3 can therefore not be rejected for Java, but it is rejected for the C corpus. Hence, for the Java corpus we may conclude that a high R^2 is indeed caused by summing up CC. For the C corpus we investigated if another influencing factor such as the variance in the header code (see Sections 3.2.3 and 3.4.4) could explain the rejection of Hypothesis 3.

Hypothesis 4 was introduced, therefore, to investigate the impact of the header code (in files) on the correlation values as opposed to summation of the values at the subroutine level. The only difference between Hypotheses 3 and 4 is the inclusion or exclusion of SLOC outside the subroutine bodies for the entire corpus. For Java and C we both found high correlations after aggregating CC and SLOC on a subroutine level, i.e. taking the sum of the CC and SLOC for all subroutines in a file. These observations support Hypothesis 4 (now also for the C corpus) and indicate that the variance of SLOC in the header was indeed a confounding factor for the previous experiment. High correlation between the number of methods and the number of fields reported by Grechanik et al. [GMD+10] might explain why header size did not have confounding effect for Java. We conclude that Hypothesis 4 is not rejected for both Java and C.

Previously we rejected Hypothesis 1 – a strong Pearson correlation for non-aggregated data. So, we have a strong indication that the related work reporting a high correlation based on a file level aggregation is likely caused by the aggregation itself rather than a linear relation between SLOC and CC. Since we cannot literally reproduce the data of the related work, this conclusion must remain a conjecture, but the above experiments do isolate a strong effect of aggregation on our corpora.

In conclusion, the number of subroutines is a factor of system size and aggregation influences the correlation positively. Similar observation has been made for the relation between SLOC and the number of defects [VSvdB11a]. Therefore, we deem aggregated CC more unnecessary as level of aggregation grows larger (classes, packages, systems). If CC should be aggregated for another (external) reason, more advanced aggregation techniques such as econometric inequality indexes [MAL+13; VSvdB11a; VSvdB11b] should be used rather than sum.

Hypothesis 3 and 4: Summing CC and SLOC on a file level could have caused high correlations reported in related work.

3.5.4 *Hypothesis 5 – Positive effect of the log transform*

As reported in related work [FF79; HGR07; HH10; JHS+09], a log transform indeed increases the R^2 values (from 0.40 to 0.68 for Java and from 0.43 to 0.70 for C). Because of this we do not reject Hypothesis 5. This finding agrees with the earlier observation on the impact of the log transform on R^2 [Loe90].

However, what does a high Pearson correlation after log transform suggest for the relation between SLOC and CC? Does it have predictive power? Recall that the Pearson correlation estimates a linear model like this: $\text{CC} = \alpha + \beta \cdot \text{SLOC}$. Hence, if the model after the log transform is $\log_{10}(\text{CC}) = \alpha + \beta \cdot \log_{10}(\text{SLOC})$, then $\text{CC} = 10^{\alpha} \cdot \text{SLOC}^{\beta}$ which implies the non-linear and monotonic model. Note that the R^2 of 0.68 and 0.70 do not have a natural interpretation in this non-linear model. Indeed, as recognised in the literature [FWL+13; Man98] the log scale results must be retransformed to the original scale leading to "a very real danger that the log scale results may provide a very misleading, incomplete, and biased estimate of the impact of covariates on the untransformed scale, which is usually the scale of ultimate interest" [Man98]. The experiment resulting in a Spearman ρ at 0.80 and 0.83 do confirm the monotonicity as well as the correlation, but this does not help interpreting these results.

Comparing this R^2 after the log transform to the R^2 before transformation is a complex matter; indeed the literature suggests that the R^2 values are not comparable [Kvå85; Loe90]. In the lower range of SLOC and CC, the effect of the log transform is small, however as SLOC increases, so does the impact of the transform. Furthermore, the variance of the model after the transform increases a lot with higher SLOC as well (see Figure 3.9). We conclude that the observations of a R^2 being higher after transform reinforce the conclusion of Hypothesis 1 (there is no strong Pearson correlation), but do not immediately suggest that there exists an exponential relation between SLOC and CC. The variance is too high and not predictable enough.

In combination with aggregation (sum) log transform has lead to the highest R^2 values observed (cf. Table 3.6). However, the regression lines do not fit the data for larger files (cf. Figure 3.10). This is caused by the heavy skew of the distributions towards the smaller values.

What we conclude is that the relatively high correlation coefficients after a log transform in literature are reinforced by our own results. These results provide no evidence of CC being redundant to SLOC because the non-linear model cannot easily be interpreted with accuracy.

> **Hypothesis 5**: A log transform increases the R^2 values between CC and SLOC, however, interpreting the model in terms of the untransformed variables is complex.

3.5.5 *Hypothesis 6 and 7 – Positive effect of zooming*

The final try was to find linear correlation on parts of the data, in order to compensate for the shape of distributions. Our results show that zooming in on tails reduced the correlation, while zooming in on the heads improved it for the 80%–100% range. Intuitively, if we remove all elements from the tail of the distributions then we may achieve the highest R^2 (0.59 for Java and 0.67 for C).

Based on the data we reject Hypothesis 6 (hypothesizing an effect of the smallest elements) and we do not reject Hypothesis 7 (hypothesizing an effect of a long tail). These results are corroborated in Table 3.4 and Table 3.5, showing log transforms only improving correlations for the whole range.

We interpret the large effect of tail elements to the increasing variance with high SLOC (heteroscedasticity), rather than label them as "outliers". There is no reason to assume the code is strange, erroneous or false more than the elements in the prefix of the data can be considered strange. The benefit of having the two big corpora is that there are enough elements in the tail to reason about their effect with confidence.

Our analysis, however, does motivate that (depending on the goals of measuring source code) tool vendors may choose to exclude elements from the tail when designing their predictive or qualitative models. Note however that even the head of the data suffers from heteroscedasticity so the same tool vendors should still not assume a linear model between SLOC and CC.

The results for Hypothesis 6 and Hypothesis 7 support our original interpretation for the main Hypothesis 1: CC is not redundant for Java methods or C functions. Nevertheless, the data also shows enormous skew towards the smallest subroutines (2 or 3 lines), for which clearly CC offers no additional insight over SLOC. If a Java system consists largely of very small methods, then its inherent complexity is probably represented elsewhere which can be observed using OO specific metrics such as the Chidamber and Kemerer suite [CK94].

For the larger subroutines, and even the medium sized subroutines, correlation decreases rapidly. This means that for all but the smallest subroutines CC is not redundant. For example, looking at the scatter-plot in Figure 3.6 and the box plots in Figure 3.8, we see that given a Java method of 100 SLOC, CC has a range between 1 and 40, excluding the rare exceptions. In our Java corpus, there are still 104 K methods larger than or equal to 100 SLOC. For such larger Java methods, CC can be a useful metric to further discriminate between relatively simple and more complex larger methods. We refer to our previous work [VG12] and the work of Abran [Abr10] for a discussion on the interpretation of the CC metric on large subroutines.

Hypothesis 6 and 7: Large subroutines have a negative influence on the correlations. They are not always generated code, therefore, labeling them as outliers should be done with care.

3.5.6 *Comparing Java and C*

Java and C are different languages. While Java's syntax is strongly influenced by C (and C++), the languages represent different programming paradigms (respectively object-oriented programming and procedural programming). While one could write procedural code in Java (the most common model for C), OO style is encouraged and expected.

In our corpora C functions are larger and have more control flow than Java methods (Figures 3.5 and 3.6). Future work could investigate whether this difference is caused by the difference in programming paradigm and coding idioms or this is caused by another factor such as application domain.

Note that the mode of both SLOC and CC are the same for Java and C. We also observe similar shapes in the scatter plots (Figures 3.6 and 3.7): both corpora feature increasingly high variance. We must conclude that although the corpora quantitatively have a different relation between CC and SLOC, qualitatively we come to the same conclusions of a relatively weak linear correlation.

On the one hand, for the C language we observed that after aggregation to the file level the correlation strength went down. We attributed the cause to the SLOC of C header code (the code outside the function bodies) having high variance. This obscures the relation between SLOC and CC for the C language on the file level, which was confirmed by testing for an increased correlation strength after measuring only the SLOC sum of functions per file. On the other hand, for Java it appears the header code is not a confounding factor. Again, this is not the point of the current chapter, but we conjecture that the stronger encapsulation primitives which Java offers bring upon a stronger relation (cohesion) between header code and subroutine bodies.

> The differences between C and Java code do not offer additional insight for the relation between SLOC and CC in open source code, other than an increased external validity of the analysis of Hypothesis 1. Our conclusions hold for both languages.

3.5.7 *Threats to Validity*

Next to the threats to validity we have identified in the experimental setup (Section 3.3) and the previous discussion, we further discuss a few other important threats to validity here.

Construct validity

Construct validity pertains to our ability to model the abstract hypothesis using the variables we have measured [PPV00]. We do not believe our study to be subject to construct validity threats since the abstract hypothesis we have tested (Hypothesis 1)

has already been formulated in terms of measurable variables (SLOC, CC and R^2) as opposed to more abstract constructs (e.g., maintainability or development effort).

As to the use of Pearson's coefficient, this was motivated by its common use in related work which we tried to replicate. Our negative conclusions, meaning we deem the observed R^2 values significantly lower, are subject to the critical examination of the reader.

Internal validity

We have tested the tools we developed for our experiments and compared the output to manually expected results and other free and open-source metric tools. Moreover, to mitigate any unknown issues and to allow for full reproducibility, we have also published both our data and scripts online [Lan15c].

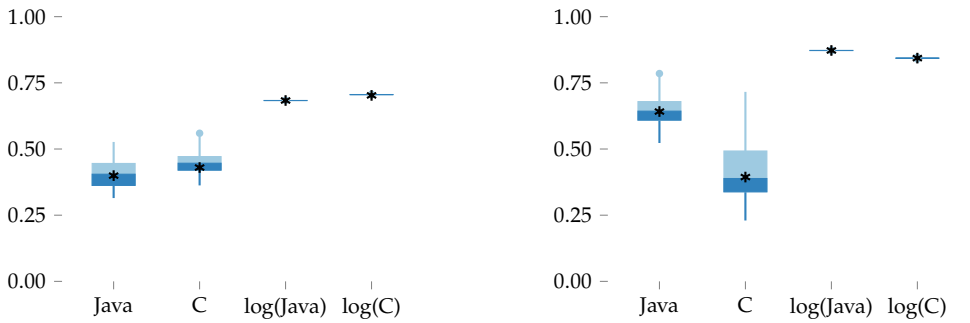
To handle the preprocessor statements in C we have used a heuristic (see Section 3.3). This heuristic filtered away 7% of the code in the corpus. We also filtered all C files with unbalanced braces which may have been introduced by the aforementioned pre-processor heuristics – not a `}` for every `{`. This removed 4 K files (0.50%) from the corpus. There is no reason to expect these filters have introduced a bias for either the SLOC or the CC variables, but without these filters the corpus would have contained invalid data.

Different from related work [HH10], we chose not to exclude all `.h` files (see Section 3.3.4). If we do ignore all `.h` files the R^2 for the subroutine level changes from 0.4301 to 0.4346, i.e. both 0.43 when rounded to two significant digits.

External validity

Both our corpora were constructed from open source software projects containing either Java or C code. Therefore, our results should not be immediately generalized to proprietary software or software written in other programming languages. We should observe that although both languages and their respective corpora are significantly different, we do arrive at similar conclusions regarding our hypotheses. We therefore conjecture that given a comparably large corpus for C-like programming languages (e.g., C++, Pascal, C#) the results should be comparable. A recent study of rank-based correlation between CC and SLOC in Scala GitHub repositories [CJ14] suggests that our results might be valid for Scala as well. While CC adaptations have also been proposed for such languages as Miranda [van95] and Prolog [Moo98], those adaptations are quite remote from McCabe's [McC76] original notion of CC and therefore the relation between CC and SLOC for these languages might be very different.

Moreover, we are aware that the size of the corpus may be a confounding factor and therefore should be investigated [RPH+13] and that our study might have been biased by presence of certain accidental data-points in our corpora. Therefore, we



(a) Subroutine (b) File

Figure 3.11: Boxplots of the R^2, for the different transformations, for 1000 randomly sampled subcorpora of half the size. The $*$ denotes the R^2 for the full corpus.

performed an additional sensitivity analysis [STC00] for which the results are reported below. To assess whether the size of the corpus has an important influence on the result, we test whether the strength of the linear correlation between SLOC and CC is similar for randomly selected sub-corpora of half the size. Figure 3.11 shows the distribution of R^2 values for 1000 random sub-corpora.

The medians of the R^2 values are very close – up to two significant digits – to the R^2 of the full corpora. However, there is a visible spread for the non-transformed variables, before and after aggregation on file level. The log transform has a clearly stabilizing effect due to compression of the tail. Because of the latter observation, we argue once more that the observed effects can be explained by the increasing variance in the tail of the data (cf. Table 3.5). Randomly selected sub-sets filter a number of elements from the tail, explaining the spread between the 1000 experiments. Moreover, the R^2 in these experiments are not contradicting our previous discussion of the hypotheses.

The above experiment mitigates the risk of the size factor confounding our observations and conclusions: it can be expected that for random sub-corpora of half the size the correlation strength is the same as for the full corpora. In contrast we believe that the large size of the corpora has made it possible to observe the relation between CC and SLOC on arbitrary real-world code with no other known biases.

3.6 CONCLUSION

The main question of this chapter was if CC correlates linearly with SLOC and if that would mean that CC is redundant. In summary, as opposed to the majority of the previous studies we did not observe a strong linear correlation between CC and SLOC

of Java methods and C functions. Therefore, we do not conclude that CC is redundant with SLOC.

Factually, on our large corpora of Java methods and C functions we observed (Section 3.4):

- CC has no strong linear correlation with SLOC on the subroutine level.
- The variance of CC over SLOC increases with higher SLOC.
- Ignoring && and || has no influence on the correlation.
- Aggregating CC and SLOC over files improves the strength of the correlation.
- A log transform improves the strength of the correlation.
- The correlation is lower for larger (SLOC) methods and functions.
- Excluding the largest methods and functions improves the strength of the correlation.
- The largest methods and functions are not just generated code, and therefore, should not be ignored when studying the relation between SLOC and CC.

From our interpretation of this data (Section 3.5) we concluded that:

- CC summed over larger code units measures an aspect of system size rather than internal complexity of subroutines. This largely explains the often reported strong correlation between CC and SLOC in literature.
- Higher variance of CC over SLOC observed in our study as opposed to the related work can be attributed to our choice for much larger corpora, enabling one to observe many more elements.
- The higher correlation after a log transform, supporting results from literature, should not be interpreted as a reason for discarding CC.
- All the linear models suffered from heteroscedasticity, i.e. non-constant variance, further complicating their interpretation.

Our work follows the ongoing trend of empirically re-evaluating (or even replicating [SCV+08]) earlier software engineering claims (cf. [KAD+14; RPF+14]). In particular we believe that studying big corpora allows to observe features of source code that would otherwise be missed [SSA15].

4 EXPLORING THE LIMITS OF STATIC ANALYSIS AND REFLECTION

Abstract

The behavior of software that uses the Java Reflection Application Programming Interface (API) is fundamentally hard to predict by analyzing code. Only recent static analysis approaches can resolve reflection under unsound yet pragmatic assumptions. We survey what approaches exist and what their limitations are. We then analyze how real-world Java code uses the Reflection API, and how many Java projects contain code challenging state-of-the-art static analysis.

Using a systematic literature review we collected and categorized all known methods of statically approximating reflective Java code. Next to this we constructed a representative corpus of Java systems and collected descriptive statistics of the usage of the Reflection API. We then applied an analysis on the abstract syntax trees of all source code to count code idioms which go beyond the limitation boundaries of static analysis approaches. The resulting data answers the research questions. The corpus, the tool and the results are openly available.

We conclude that the need for unsound assumptions to resolve reflection is widely supported. In our corpus, reflection can not be ignored for 78 % of the projects. Common challenges for analysis tools such as non-exceptional exceptions, programmatic filtering meta objects, semantics of collections, and dynamic proxies, widely occur in the corpus. For Java software engineers prioritizing on robustness, we list tactics to obtain more easy to analyze reflection code, and for static analysis tool builders we provide a list of opportunities to have significant impact on real Java code.

4.1 INTRODUCTION

Static analysis techniques are applied to support the efficiency and quality of software engineering tasks. Be it for understanding, validating, or refactoring source code, pragmatic static analysis tools exist to reduce error-prone manual labor and to increase the comprehension of complex software artefacts.

Static analysis of object-oriented code is an exciting, ongoing and challenging research area, made especially challenging by dynamic language features (a.k.a.

This chapter was previously published as: D. Landman, A. Serebrenik, and J. J. Vinju. "Challenges for static analysis of Java reflection: literature review and empirical study". In: *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*. Ed. by S. Uchitel, A. Orso, and M. P. Robillard. IEEE, 2017, pp. 507–518. DOI: 10.1109/ICSE.2017.53, and was awarded the Distinguished Paper Award of the Technical Research Papers track.

reflection). The Java Reflection API allows programmers to dynamically inspect and interact with otherwise static language concepts such as classes, fields and methods, e.g. to dynamically instantiate objects, set fields and invoke methods. These dynamic language features are useful, but their usage also wreaks havoc on the accuracy of static analysis results. This is due to the undecidability of resolving dynamic names and dynamic types.

Until 2005, the analysis of code which uses the Reflection API was considered to be out of bounds for static analysis, and handled via user annotations or dynamic analysis; handling reflection would inherently be either unsound (due to unverified assumptions) or highly inaccurate (due to over-approximation) and render the contemporary static analysis tools impractical. Then, in 2005 Livshits et al. [LWL05] published an analysis of how reflection was used in six large Java projects, proposing three unsound, yet well-motivated assumptions and using these to (partially) statically resolve the targets of dynamic method calls. Since then more tools were based on similar assumptions.

Very recently, in 2015, Livshits and several other authors of static analysis tools published the soundiness manifesto [LSS+15]. It argues for "soundy" static analysis approaches that are mostly sound, but pragmatically unsound around specific problematic language features. Java's Reflection API is one of the examples that can be handled more effectively after certain unsound assumptions are made. For future work they identified the need for empirical evidence on how these language features are used, such that tool builders can motivate the required unsound assumptions. We provide more unbiased empirical evidence on the use of reflection by focussing on the following **Main Research Question:** *What are limits of state-of-the-art static analysis tools when confronted with the Reflection API and how do these limits relate to real Java code?* Hence, we investigate the following sub-questions:

SQ1. How do static analysis approaches handle reflection; which limitations exist and which assumptions are made? (Section 4.3)

SQ2. How often are different parts (see Section 4.2) of the Reflection API used in real Java code? (Section 4.4)

SQ3. How often does real Java code challenge the limitations and assumptions identified by SQ1? (Section 4.5)

Together with answers to these questions, this chapter contributes a representative corpus of open-source Java projects [Lan16a], and a comprehensive literature overview on the relation between static analysis and Java reflection. The main question is answered with a list of challenges and suggested tactics for static analysis researchers, ordered by expected impact.

We first describe the Java Reflection API; how its features can be categorized. The resulting frame of reference is used for the interpretation of the findings in Sections 4.3–4.5, because the different API features interact differently with static analysis.

The Java Reflection API consists of objects modeling the Java type system. These meta objects are split over 8 classes – `java.lang.{Class,ClassLoader}` and `java.lang.reflection.{Array, Constructor, Field, Member, Method, Proxy}` – totaling 181 public methods. The meta objects mostly provide an immutable view of the running system's types.

Figure 4.1 summarizes the API as a context-free grammar that defines construction of references to meta objects. We use a context-free grammar as a more concise alternative to class diagrams or interface definitions. Each production in Figure 4.1 defines a number of alternatives to produce an object of the defined non-terminal. The grammar naturally groups on return type to emphasize the construction of (immutable) meta objects and compresses methods of similar intent using regular expressions. With this, we completely mapped the 181 public methods of the entire API onto 58 productions, which are further grouped into 17 categories in Table 4.1.

Next to the API listed in the `java.lang.reflection` package, there is: the `Object.getClass()` method and the literal `Object.`**`class`** language construct for class literals. There is also relevant Java expression syntax related to reflection, casts and **`instanceof`**. Class literals, such as `MyClass.`**`class`**, produce a meta object instance of the (static) type `Class<MyClass>`. They are a static alternative to `Object.getClass`. Cast and **`instanceof`** expressions also use literal types which interact with Java's execution semantics (e.g. throwing `ClassCastException`).

From the perspective of static analysis, the reflection API introduces dynamic language features for an otherwise statically resolved language. From this perspective, the API can be split in two parts. The first part (□ in Table 4.1) are the *Dynamic Language Features* that simulate statically resolved counterparts: e.g. the `<Method>.invoke` API is the dynamic equivalent of the statically resolved method invocation in Java (`obj.method()`). The second part (⬚) includes supporting methods for the dynamic language features (e.g. getting a `Method` meta object), and miscellaneous methods for accessing other elements of the Java runtime.

Even when infrequently used, a single occurrence of using a dynamic language feature does complicate static analysis of the entire program. For example, a single dynamic method invocation could in principle call any method in the currently loaded system, resulting in a highly inaccurate call graph for the entire system. For the rest of this chapter, we are primarily interested in how static analyses approximate the effect of these dynamic language features.

Modeling the supporting methods is often necessary to approximate the semantics of the dynamic language features. For example, invoking a method requires a `Method`

```
<MetaObject> ::= <Class> | <Method> | <Constructor> | <Field>
<Member> ::= <Method> | <Constructor> | <Field>

<ClassLoader> ::=
TM      <Class>.getClassLoader()
LM    | ClassLoader.getSystemClassLoader()
LM    | new ClassLoader(<ClassLoader>)
LM    | <ClassLoader>.getParent()

<Class> ::=
LC      Class.forName(<String>)
LC    | Class.forName(<String>, <Boolean>, <ClassLoader>)
LC    | <ClassLoader>.loadClass(<String>)
LM    | <Type>.class
LM    | <Object>.getClass()
TM    | <Class>.get*Interfaces()
TM    | <Class>.asSubclass(<Class>)
TM    | <MetaObject>.get*Class{es}?()
TM    | <MetaObject>.get*Type*()
P     | Proxy.getProxyClass(<Class*>)

<Method> ::=
TM      <Class>.get{Declared}?Methods()
TM    | <Class>.get{Declared}?Method(<String>, <Class*>)
TM    | <Class>.getEnclosingMethod()

<Constructor> ::=
TM      <Class>.get{Declared}?Constructors()
TM    | <Class>.get{Declared}?Constructor(<Class*>)
TM    | <Class>.getEnclosingConstructor()

<Field> ::=
TM      <Class>.get{Declared}?Fields()
TM    | <Class>.get{Declared}?Field(<String>)

<Void> ::=
M       <Field>.set*(<Object>, <Object>)
AR    | Array.set*(<Object>, <int>, <Object>)
MM    | <Member>.setAccessible(<Boolean>)
AS    | <ClassLoader>.{set}?{clear}?*AssertionStatus(<Boolean*>)
AS    | <ClassLoader>.set*AssertionStatus(<String>, <Boolean>)
```

Figure 4.1: Grammar of the Java Reflection API. A '*' inside a terminal indicates zero or more other characters, and inside a nonterminal it indicates zero or more of this nonterminal. `{X}?` indicates an optional part of a terminal. `MethodUtil.getMethod*` was elided into the – non deprecated – replacement method.

```
<Object> ::=
C       <Constructor>.newInstance(<Object*>)
C     | <Class>.newInstance()
AR    | Array.newInstance*(<Class>, <int*>)
P     | Proxy.newProxyInstance(<ClassLoader>, <Class*>, <Object>)
I     | <Method>.invoke(<Object>, <Object*>)
A     | <Field>.get*(<Object>)
AR    | Array.get*(<Object>, <int>)
DC    | <Class>.cast(<Object>)
AN    | <Method>.getDefaultValue()
TM    | <Class>.getEnumConstants()
P     | Proxy.getInvocationHandler(<Object>)
AN    | <MetaObject>.getAnnotation(<Class*>)
AN    | <MetaObject>.get*Annotations()
S     | <Class>.getSigners()

<ProtectionDomain> ::= S <Class>.getProtectionDomain()

<Boolean> ::=
SG      <Class>.isAssignableFrom(<Class>)
SG    | <Class>.isInstance(<Class>)
SG    | Proxy.isProxyClass(<Class>)
SG    | <MetaObject>.is*(<Class>) // other signature checks
SG    | <MetaObject>.equals(<Object>)
SG    | <MetaObject> == <MetaObject>
SG    | <MetaObject> != <MetaObject>
SG    | <Member>.isAccessible(<Class>)
AS    | <Class>.desiredAssertionStatus()
AN    | <MetaObject>.isAnnotationPresent(<Class>)

<String> ::=
ST      <MetaObject>.get*Name()
ST    | <MetaObject>.to*String()
ST    | <Class>.getPackage() // returns a wrapper for strings

<int> ::= SG <MetaObject>.getModifiers()

<Resource> ::= <URL> | <InputStream>
RS    | <Class>.getResource*(<String>)
RS    | <ClassLoader>.get*Resource*(<String>)
```

Figure 4.1: continued

Table 4.1: Categories for reflection productions.

Category	Description
LC Load Class	Entry to the Reflection API, returns references to meta objects from a String. Considered harmful since it can execute static initializers.
LM Lookup Meta Object	Non harmful entries to the Reflection API, returns references to meta objects.
TM Traverse Meta Object	Get references to other meta objects related to the current meta object in the type system of Java.
C Construct Object	Create a new instance of an object, equivalent to the `new <ClassName>()` Java construct.
P Proxy	Proxies are fake implementations of interfaces, where every invoke is translated to a single callback method. Very harmful for static analysis, since there is no static equivalent for this feature.
A Access Object	Read the value of an Object's field. Equivalent to the `obj.field` Java construct.
M Manipulate Object	Change the value of a field. Equivalent Java construct: `obj.field = newValue`
MM Manipulate Meta Object	The only *mutable* part of the API: changing access modifiers.
I Invoke Method	Invoke an method. Equivalent Java construct: `recv.method(args)`.
AR Array	Create, access, and manipulate arrays.
SG Signature	Test the signature of a Meta Object, for example if it is a public field.
AS Assertions	Access and manipulate the assertion flag per class.
AN Annotations	Access and iterate annotations.
RS Resources	Read resources using the ClassLoader.
ST String representations	Get the name of the meta object's elements.
S Security	Security related calls
DC Casts	Cast to a dynamically `Class` meta object. Equivalent Java construct: `(Class)obj`

The ☐ categories represent core *Dynamic Language Features* which simulate statically resolved counterparts.
The ⬚ categories represent supporting APIs comparable to normal Java library code.

meta object. Finding meta objects (with the exception of the LC productions) does not complicate static analysis on its own. It is merely an inspection of the type system. These methods can be either simulated by static analysis tools, or directly executed.

The pinnacle of dynamic behavior are `Proxy` classes P. The dynamic proxy feature allows one to instantiate objects – statically implementing a specific interface – that will dynamically forward all calls to a generic `invoke` method of another object (implementing the `InvocationHandler` interface). The proxy feature hides dynamic method invocation under a normal statically checked virtual method interface, rendering all virtual method invocations possibly dynamic.

4.3 STATIC ANALYSIS OF REFLECTION IN THE LITERATURE

To answer how reflection is handled by static analysis approaches (SQ1) we conduct a literature review. The result of the review is a list of techniques and associated properties of hard to analyse code which identify limitations and assumptions of static analysis tools. Note that the results of this review can not serve as a feature comparison between static analysis tools, because of different goals of those tools and because of our focus on the Reflection API, rather than the entire Java language.

4.3.1 *Finding and selecting relevant work*

Two commonly used literature review techniques are snowballing [WW02; Woh14] and Systematic Literature Review (SLR) [KC07]. Snowballing consists in iteratively following the citations of a small collection of serendipitously identified papers. However, several core papers have hundreds of citations, e.g. the work of Felt et al. [FCH+11] has been cited 940 times and the work of Christensen et al. [CMS03] 412 times, rendering snowballing too labor intensive. Hence, we conduct an SLR.

Initial queries

As recommended by Kitchenham and Charters [KC07] we started by considering IEEE Xplore, ACM DL, and ScienceDirect. The search results, however, contained multiple inconsistencies. In IEEE Xplore, e.g. adding an `OR` to our query reduced the number of results. ACM DL and ScienceDirect search missed papers when limited to the abstract field, even though those abstracts contained the search terms. Hence, we decided that these sources were not well-suited for SLR. Instead, we opt for Google Scholar as it provides a wide coverage of different electronic sources as recommended [KC07] and its search engine did not exhibit these peculiarities.

Following the PICO criteria [KMT06] we define our *population* as Java projects with reflection, *intervention* as static analysis and *outcomes* as approach, limitations and assumptions. We do not explicitly state the *comparison* element of PICO since our goal

Table 4.2: Inclusion criteria used to select relevant documents for manual review.

1) Papers with reflection in introduction (head) and conclusion (tail). Moreover, at least one term related to accuracy should be used. To correct for Google's stemming of JavaScript to Java, we exclude papers that mention JavaScript too often:
$P \leq 80 \wedge R_h > 0 \wedge (R_t > 0 \vee R_{t'} > 0) \wedge A > 0 \wedge S \leq 5$.

2) Thesis. A thesis discussing reflection, containing reflection code samples, and mentioning accuracy:
$P > 50 \wedge T_h > 0 \wedge R > 1 \wedge A > 0 \wedge J > 0$.

3) Proceedings with frequent mentions of reflection:
$P > 20 \wedge T_h = 0 \wedge C_h > 0 \wedge R > 5$.

4) Short papers frequently mentioning reflection. Smaller documents might have non standard layout, or be sensitive to the 10% cutoff points for the head and tail. These documents mentioning reflection at least 10 times are also included:
$P \leq 40 \wedge R \geq 10 \wedge A > 0 \wedge S \leq 5$.

5) Proceedings with reflection code samples. Similarly to 3) but with reflection code samples:
$P > 20 \wedge T_h = 0 \wedge C_h > 0 \wedge R > 0 \wedge J > 0$.

6) Large non-thesis, non-proceedings papers with frequent reflection:
$P > 80 \wedge T_h = 0 \wedge C_h = 0 \wedge R > 5$.

The $\star_h$ denotes head section, $\star_t$ tail section , $\star_{t'}$ tail section without bibliography, and P amount of pages in a PDF. A represents terms related to "accuracy", "precision" and "soundness", C for "proceedings" and "conference", J for "lang.reflect", R for "reflection", S for "javascript", and T for "thesis" and "dissertation".

consists in comparing different ways reflection is handled by static analysis techniques *with each other* as opposed to comparing them with a predefined control treatment. Based on the population, intervention and outcome we formulate the following query: `java "static analysis" +reflection`*. We do not explicitly include the outcome in the query since approaches, limitations and assumptions can be phrased in numerous ways. In October 2015 the query returned 4 K references.

Automatic selection criteria

Since manual analysis of 4 K documents is infeasible, we design six criteria to reduce the number of potentially relevant documents. To be included in the study the document should meet *at least* one of those criteria. Those criteria, presented in Table 4.2, are based on frequency of keywords in the full text, the first 10% of the text (head), the last 10% (tail), and the last 10% without the references/bibliography (tail without references). We validated all thresholds of these criteria by sampling beyond the thresholds and manually scanning the additional papers for false negatives. We picked liberal thresholds to optimize on recall (e.g. $P \leq 80$ for deciding a document is a single paper rather than a collection).

*Google has implicit `AND` and the + disables stemming

Manually Improving Accuracy

478 documents (11% of the original set) were matched by at least one of the six criteria in Table 4.2. Including the 36 documents that `pdf2text` failed to analyse we had 514 documents to read. We reviewed all documents applying the practical screen [Fin10] to exclude those meeting the following *exclusion* criteria: not about Java, not about static analysis, reflection is only recognized as a limitation, reflection is handled with an external tool, reflection is wrapped to guard against its effects, reflection is used to solve a problem, or a homonym of "reflection" was the cause of the match. We have logged the exclusion decisions in a shared online spreadsheet and reviewed each others decisions. This process produced 50 documents. Next we removed non-peer-reviewed publications: locating and substituting conference papers for equivalent technical reports, masters theses or PhD theses; locating and substituting extended journal versions for conference papers; removing non-peer-reviewed publications such as technical reports and masters thesis' without corresponding publications at the time; and finally as recommended by Kitchenham and Charters [KC07] merging duplicate documents produced by noise in Google Scholar. This results in 39 documents.

All 39 documents were then read by one author and scanned by another, producing 4 new relevant documents from the citations (all missing from the original Google Scholar results). The 4 new papers introduced Soot [VCG+99], Spark [LH03] (a plugin for Soot), WALA [FD+15], and JSA [CMS03]. Only JSA and WALA handle reflection specifically, while Soot requires plugins (such as TamiFlex [BSS+11] or Spark), and Spark requires user annotations.

While reading the documents we applied the methodological quality screen [Fin10] and identified another 10 documents to be excluded, due to the following reasons: taint analysis pushing taints through the reflection API [HDM14; YXA+15], using existing techniques for handling reflection [AL12; AL13; AFJ+09; AFJ+10; GKP+15; SR11; TPF+09], and handling reflection in generated bytecode rather than in source code [ARL+14].

4.3.2 *Documenting Properties of Static Analysis Tools*

To answer SQ1, we read the 33 (39 + 4 − 10) documents to list approaches or techniques which are involved in resolving dynamic language features of Java reflection. The end result is summarized in Table 4.3. When we could not find enough information to extract information about the properties of a tool from the respective paper, we analysed the latest version of the tool's source code and documentation (if available). As recommended by Brereton et al. one author extracted the data, and another one checked it [BKB+07].

We classified the techniques in three kinds of analysis, by the kind of information which is used to resolve reflection: *static* uses code analysis to resolve reflection (listed

in Table 4.3), *dynamic* uses information acquired at run-time for resolving reflection rather than code ([BSS+11; DRS07; HvDD+07; IC14; TB12; VCG+99; ZAG+15]) and *annotations* groups techniques based on are human-provided meta data rather than code or dynamic analysis ([BGC15; LH03; SAP+11; TLS+99; TSL+02]). Note that papers solely about dynamic analysis were excluded in an earlier stage.

Next we record the **goal** of the static analysis as mentioned in the paper (e.g. call graph construction), the name of the tool, and possible **dependency** on other related tools. We also distinguish between intra- and **inter-procedural** algorithms.

Diving further into the explanations of techniques of each static analysis tool revealed a diverse collection of mostly incomparable algorithms and heuristics in terms of functionality and quality attributes. Based on this reading we documented the authors' descriptions of properties of the analysis tools in terms of **sensitivity**. Sensitivity defines the smallest level of distinction made by the abstract (symbolic) representations of run-time values and run-time effects that static analysis tools use. Finer-grade distinctions mean more different abstract values and result in more accurate but slower analyses, while coarser-grade distinctions lead to less different abstract values and less accurate but faster analyses.

Flow sensitivity entails distinctions between subsequent assignments

Field sensitivity entails distinction between different fields in the same object

Object sensitivity entails the distinctions between individual objects, via groups of objects, to general class types, at increasing levels of indirection

Context sensitivity entails the distinction of method executions between different calling contexts of a given depth

We also record whether the analysis requires a **fixed-point** computation. Finally we identified and documented the use of three specialized measures taken by static analysis tools:

String analysis approximates run-time values of strings as accurately as possible. These results can then be used to approximate class and method names which flow into the LC, TM reflection API, after which the semantics of `invoke` and `newInstance` may be resolvable.

Casts provide information about run-time types under the assumption that no `ClassCastException` occurs. Some analyses also reason *back* from the correct-casts assumption.

Meta Objects signifies the full simulation (or execution) of the LC, LM, and TM reflection API to find out which meta objects may flow into the dynamic language features.

By inspecting Table 4.3 we observe that flow sensitivity is very common (often as a side-effect of the SSA transform), field sensitivity is used for half of the approaches (more common in DOOP and Soot), and, most analyses are inter-procedural and track at least string literals. Tracing DOOP through the years, we see more modeling of Strings, Casts and Meta Objects.

Table 4.3: Static Analysis approaches for handling reflection. For object and context sensitivity we report the sensitivity depth. For the strings column: ○ no analysis, ◔ only literals, ◓ literals and concatenations, and ● full fledged (JSA) string operations. For the remaining properties we use filled circles to summarize the coverage of a property: ○ for none, ◓ for partial, and ● for full. The table is sorted on the "Build using" and "Year" columns.

Paper	Year	Tool	Related	Kind	Goal	Sensitivity[y]				Inter-procedural	Fixed-point	Strings	Casts	Meta-Objects	Dependency
						flow[z]	field	object	context						
[LWL05]	2005	bddbddb		Static & Annotations	Call Graph[a]	○	○	0	0	○	●	◓	◓[k]	●	Datalog & bddbddb
[BS09]	2009	DOOP	[LH08; LWL05]	Static	Points to	●[b]	●	0	1, 2	●	●	◔[c]	○	○	Datalog
[Gab13]	2013	Datalaude	[LWL05]	Static	Points to	○	○	0	0	●	●	◔	○	◓	Maude & Joeq
[LTS+14]	2014	ELF	[BS09]	Static	Points to	●[b]	●	0	1, 2	●	●	◓	●	○	DOOP
[LTX15]	2015	SOLAR	[LTS+14]	Static & Annotations	Points to	●[b]	●	0	1, 2	●	●	◓	●	●[d]	DOOP & ELF
[SB15]	2015		[BS09]	Static	Points to	●[b]	○	1	1	●	●	◔	○	●	Datalog
[SBK+15]	2015	DOOP	[BS09]	Static	Points to	●[b]	●	0	1, 2	●	●	◓[e]	●[e]	●[e]	Datalog
[CMS03]	2003	JSA		Static	Call Graph	●[b]	●	0	0	●	○	●	○	○	Soot
[SR07]	2007		[CMS03]	Static & Dynamic	Class Loading	●[b]	●[f]	0	0	●	○	●[g]	○	○	Soot & JSA
[SR09]	2009		[SR07]	Static & Dynamic	Class Loading	●[b]	●[f]	0	0	●	○	●[g]	○	◓	Soot & JSA
[AL13]	2013	AVERROES		Static & Dynamic	Modeling API	○	○	0	0	○	○	◔	○	○	Soot & TamiFlex
[CFP07]	2007	ACE		Static & Dynamic	Call Graph	○	○	1	1	●	○	○	◓[k]	○	
[FCH+11]	2011	Stowaway		Static	Name	●	○	0	0	◓	○	◔	○	●	
[KYY+12]	2012	SCANDAL		Static	Taint	●	○	0	1	●	○	◔	○	○	
[HYG+13]	2013		[FCH+11]	Static	Name	●[h]	○	0	∞[h]	●[i]	○	◔	○	◓	
[WKO+14]	2014			Static	CFG	●	○	0	0	●	○	◔	○	○	
[RCT+14]	2014	FUSE		Static	Points to	●[b]	0	0	0	●	○	○	◓[k]	○	
[FD+15]	2015	WALA		Static	Multiple	●[b]	●	0/∞	0/∞	●	●	◓	●	◓	
[BJM+15]	2015	part of SPARTA	[EJM+14]	Static & Annotations	Implicit CFG	●	○	0	0	○	○	◓	○	●	Checker Framework
[CFB+15]	2015	EdgeMiner		Static	Implicit CFG	○	○	0	0	●	○	◔[j]	○	○	dx

a) Including points-to analysis.
b) After SSA transform.
c) Only for `Class.forName`.
d) Lazy
e) Only if it points to a small set of candidates (subclasses / fields / methods).
f) Only string fields.
g) JSA extended with environment information, modeling field, and tracking of objects of type Object.
h) Backwards slicing.
i) With heuristics.
j) Only for base (JRE/Android) framework.
k) Only for `newInstance`.
y) None of the papers are path sensitive.
z) The reported flow sensitivity was always intra-procedural.

Table 4.4: Reported open and resolved limitations of static analysis tools, using literature from Table 4.3.

Name	Description
CorrectCasts [LWL05]	Assumption that casts never throw `ClassCastException`
WellBehavedClassloaders [LWL05]	Assumption that all `ClassLoaders` implementations follow a specific contract, i.e. if a class with the (fully qualified) name X is requested from the LC API then a reference to a class named X is produced
ClosedWorld [LWL05]	Assumption that the classpath configured for static analysis equals that of the analysed program
IgnoringExceptions [BS09]	Not modeling the control effect of exceptions, which is relevant around common exceptions of the Reflection API (e.g. `ClassCastException`)
InaccurateIndexedCollections [BS09]	Not modeling index positions in arrays and lists, which is relevant when meta objects end up in such collections
InaccurateSetsAndMaps [SR09]	Not modeling `hashCode` and `equals` semantics in concert with hash collections, which is relevant when meta objects end up in such collections
NoMultipleMetaObjects [LTS+14]	Ignoring usage of TM API methods which return multiple meta objects in an array
IgnoringEnvironment [SR07]	Not modeling the content of configuration strings which come from `System.getEnv` for tracing LC, LM or TM methods
UndecidableFiltering [FCH+11]	Conditional control flow and arbitrary predicates are hard in general, while for code which filters meta objects even an approximate answer would greatly help
NoProxy [LTS+14]	Assumption that Proxy objects are never used. Proxy objects may invoke dynamically linked code opaquely behind any (dynamic) interface, undermining otherwise trivial assumptions of static analysis of method calls

4.3.3 *Self-reported limitations and assumptions*

The self-reported assumptions about actual code and limitations of the tools are summarized in Table 4.4. All tools discussed in the 33 studies assume well-behavedness of `ClassLoader` implementations and absence of `Proxy` classes. The other reported limitations are either resolved and fixed by a given paper, or mentioned as a known limitation of the currently described approach. We do not provide a feature comparison per tool, but rather report "common" assumptions made by static analysis tools. We choose not to extend Table 4.4 with how many tools use each assumption, to avoid it being interpreted as a (crude) comparison between incomparable tools.

> **SQ1**: State-of-the-art static analysis tools use inter-procedural, flow and field sensitive analysis. Some explicitly model Strings, Casts and Meta Objects. All tools assume well-behavedness of `ClassLoader` implementations and absence of `Proxy` classes. The techniques and their limitations are summarized in Tables 4.3 and 4.4.

4.4 How often is the reflection API used?

Regardless of the conceptual relation between reflection and static analysis, we need support for the relevance of this relation in real Java code to answer SQ2 and motivate further investigation.

Table 4.5 summarizes the related work found during the review (Section 4.3) reporting empirical observations of reflection usage. From these reports we *hypothesize that also in arbitrary Java code the usage of reflection is widespread*. This is likely true, but it may not be deduced from the reported numbers in Table 4.5, since these studies have been done on corpora selected and filtered for answering different questions.

In particular focusing only on large corpora of Android apps would not be acceptable for our current study since they are an identifiable subgroup of all Java applications. Also the much smaller SPECjvm[†] or DaCapo [BGH+06] benchmarks have been compiled to reflect typical performance characteristics of (concurrent) Java programs rather than be representative of the usage of reflection.

4.4.1 *Corpus Construction*

To test the above hypothesis and answer SQ2 we construct a corpus of the source code of 461 open-source software projects. Hunston has observed that in corpus linguistics the main issues related to corpus design pertain to its size, contents, representativeness and permanence [Hun02]. Tempero et al. have argued that the same concerns pertain to software corpora [TAD+10].

[†] `https://www.spec.org/benchmarks.html#java`

Table 4.5: Empirical observations of reflection in the literature of Table 4.3.

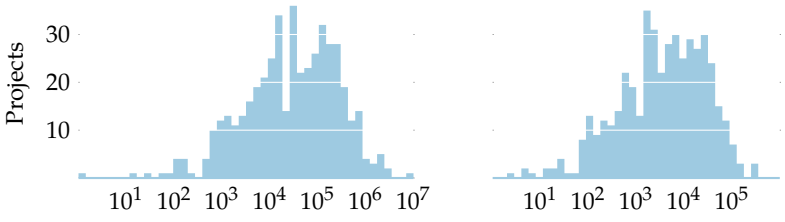
Year Ref.	Corpus	Report
2005 [LWL05]	6 applications (643 KLOC)	The accompanying technical report discusses reflection use cases, which are used to formulate the three now very popular assumptions.
2011 [FCH+11]	900 Android apps	61% use `invoke`. Reflection is also used for serialization, hidden APIs, and backwards compatibility.
2013 [HYG+13]	1.3 K Android apps	73% use `invoke`. Primarily for API calls, however, this reflects only 0.07% of all API calls.
2014 [WKO+14]	1.7 K Android apps	73% use reflection. `invoke` is most common
2014 [WKO+14]	150 Android apps	Analyzing the string argument for `forName` and `getMethod`, 17.3% use only constant strings, 25.3% use a single variable, and 38.7% use more than one variable.
2014 [LTS+14]	14 Java programs (DaCaPo benchmark and 3 other applications)	Identified 609 invocations of reflection with Soot, reports popularity of the harmful API, the kind of string operations performed on arguments, and how often the APIs return meta object arrays were used.
2015 [ZAG+15]	29 K Android apps	81.1% used either `invoke` or `newInstance`
2015 [BJM+15]	35 Android apps	142 calls to `invoke`, classifying 81% for backwards compatibility, 6% accessing hidden APIs, and 13% as unknown.

Contents of the corpus is determined by the research questions we answer using it, i.e. SQ2 and SQ3. Hence, our corpus contains Java programs. Permanence, i.e. regular corpus updates, are considered as future work. Next we discuss how size and representativeness are balanced in our corpus.

Selecting projects

To balance the corpus size with representativeness, we construct a corpus small enough to analyze while still covering a wide range of open source Java projects. We use the Software Projects Sampling (SPS) tool [NZB13] by Nagappan et al. Given a universe of projects on Ohloh/OpenHub[‡], SPS measures representativeness of a smaller *corpus* with respect to the universe in terms of diversity dimensions and

[‡]Since the access to the live OpenHub project collection is rate-limited, we used the May 2012 database snapshot when it was still called Ohloh [NZB13].



(a) Source Lines of Code (SLOC) (b) Methods

Figure 4.2: Histograms of projects size (bin width 0.15 on the log X-axis)

constructs a maximally representative corpus by iteratively adding projects that would increase the representativeness most. Diversity dimensions considered include total lines of code, project age (Young, Normal, Old, Very Old), activity (Decreasing, Stable, Increasing), and of the last 12 months, number of contributors, total code churn, and number of commits.

The entire collection contains 20 K projects, of which around 3 K have Java recorded as the main language. From this universe the SPS tool identified a sample of 468 projects, maximizing the spread of all diversity dimensions.

We tried to download the source code of the 468 projects. For 33 projects the source code was no longer available. We reran SPS to extend $435 = 468 - 33$ projects and maximize the diversity. SPS suggested 27 additional projects. The source code of two of these was not available. Repeating the procedure, SPS suggested one additional project. The resulting $461 = 468 - 33 + 27 - 2 + 1$ projects cover 99.47% of the universe.

After downloading the projects we cleaned the corpus by removing arbitrary copies of the code of projects that originate from folder-based version management. Using MD5 hashes to identify full file clones, we manually reviewed and cleaned all projects. We made the cleaned and annotated corpus openly available [Lan16a], totaling 79.4 MSLOC of Java code, to be used to reproduce the analysis results, or to benchmark static analysis research tools on systems of documented representativeness. Figure 4.2 summarizes the corpus in terms of size.

Annotated Abstract Syntax Trees

We need a precise count of actual calls into the reflection API, rendering fast *grepping* or other efficient partial parsing methods out of scope due to their inherent inaccuracy [KLN14; Moo01]. To unambiguously identify the calls to the Reflection API methods we first parsed the source code, resolved names and types, then serialized the Abstract Syntax Trees (ASTs), using the Eclipse Java Development Tools (JDT) and Rascal [BHK+15]. We deleted the 4 projects the JDT crashed on (labeled #294, #399,

#420, #455). We opt not to replace these projects as we consider the corpus as a separate contribution independent from the subsequent research.

4.4.2 *Descriptive Statistics*

To describe how the Reflection API is used by the corpus projects we make use of the context-free grammar in Figure 4.1 and categories of Table 4.1. Per category we count the percentage of projects that make use of at least one production belonging to the category. We aggregate to project level since one instance is enough to complicate static analysis and projects are a common unit for static analysis applications.

Inspecting Figure 4.3 we observe that reflection is used in almost all the projects (only 4% did not use any reflection). However, there are more use cases for reflection than just dynamic language features. The `<Type>.class` and `<Object>.getClass()` are, for example, often used as a log message prefix. The reported distributions of API method invocations over projects, should be interpreted by tool builders with the API definition itself as a frame of reference, because the API enforces certain data dependencies between different invocations into the API, e.g. `<Method>.invoke` can not be called without first retrieving an instance of an `Method` meta object, which in turn can only come from a `Class` meta object (see Figure 4.1).

We aggregated all dynamic language features API calls. Of all projects, 78% contain at least one form of these harder to analyze methods of the API. For these projects, a static analysis needs some form of reflection support. Note we only count in the Java source code of a project, reflection usage in its libraries it depends on can only increase the amount of projects impacted by the dynamic language features of reflection.

> **SQ2**: Hard to analyse parts of Reflection API are very common: 78% of all projects contain at least one usage of those.

4.5 THE IMPACT OF ASSUMPTIONS AND LIMITATIONS

In this section we answer SQ3: how often the assumptions and limitations in Table 4.4 of state-of-the-art static analysis tools are challenged by real Java code. For each identified assumption or limitation of Table 4.4 we devise one or more AST patterns and manually validate their precision in detecting occurrences of challenging code. Then we automatically identify all matches of each pattern in the corpus described in Section 4.4.1. We reuse the corpus since we look for similar representativeness and need similarly accurate unambiguously resolved classes and methods.

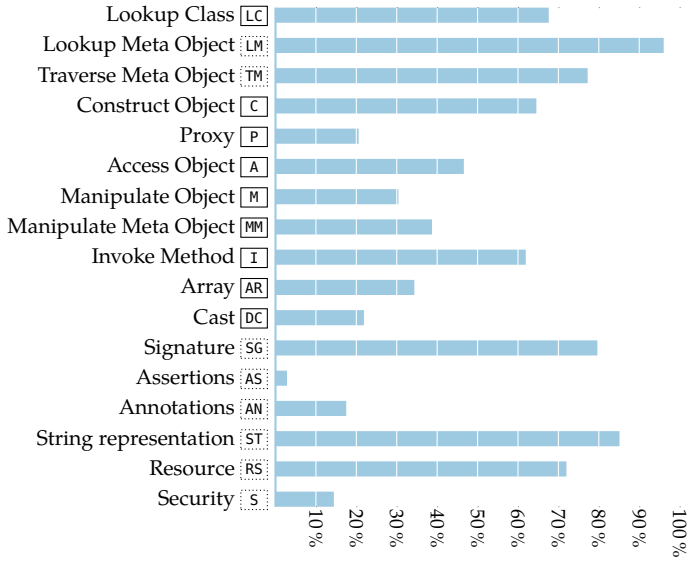


Figure 4.3: Reflection API usages, grouped per category (Table 4.1), aggregated on project level, 17 projects (3.72%) contained no reflection. 356 projects (77.90%) contained at least one of the dynamic language features (▭ categories).

4.5.1 *Detecting Patterns*

To implement pattern detectors we used the builtin AST pattern matching and traversal facilities of Rascal [KvdSV09], which have been used in many other projects [Hil15a; HKV13; LSB+16]. The pattern code is around 150 SLOC and is openly available [Lan16b].

The patterns we devised are described and motivated in Table 4.6. We strive for high precision for each pattern (a low number of false positives). Each AST pattern will capture "typical" code instances for which a clear rationale exists to relate it to the assumptions and limitations of Table 4.4.

Note that assuming each pattern is 100% exact, counting their matches will generate a lower-bound on the number of code instances which challenge static analysis tools. As a tight lower-bound more accurately answers SQ3 than a loose upper-bound would, we will not sacrifice precision for recall by generalizing patterns. Some patterns have non-empty intersections, i.e. two patterns may match on the same piece of code. This effect must be considered when interpreting the results below, next to that they are not all 100% exact.

Because the main threat to validity of this research method is the precision of the patterns, we manually estimated their precision by reading random samples of

Table 4.6: Descriptions of AST patterns used to detect the limitations identified in Table 4.4 with their rationale.

Name	Pattern description	Pattern rationale
CorrectCasts	`try` blocks with the body calling either `invoke`, `get`, or `newInstance`, and with a `catch(ClassCastException e)` case that neither contains `throw` nor calls a method with either "log" or "error" in the identifier.	Finds code which does not obviously deal with a `ClassCastException` as an unexpected error.
WellBehaved-ClassLoaders	No pattern	This is a deep semantical constraint for which we have no accurate AST pattern
ClosedWorld	No pattern	This is not a code property, as it assumes something about the classpath configuration for the static analysis
Ignoring-Exceptions-1	`try` blocks with the body calling any *dynamic language features*, with at least one `catch` block that also calls a *dynamic language feature* method.	Finds code that intents to use reflection in the "normal" path, and continues to use reflection in the "exceptional" path.
Ignoring-Exceptions-2	`for`,`while`,`do` statements with in the body a `try` block with its body using *dynamic language features*, and at least one `catch` block that does not `throw` or call a method with either "log" or "error" in the identifier	Find codes where the exceptional path is the way to continue to the next alternative way of using a *dynamic language feature* which is generated by a loop
IndexedCollections	Calls to a method which retrieves elements from the Java Collection API (e.g. `get` & `iterator`) of the containers that allow random indexing or an array access expression where the stored type is a meta object type.	This exactly identifies (using Java's type system) where meta objects may be stored in and retrieved from collections
MetaObjects-InTables	Calls to a method which retrieves elements from the Java Collection API's hash-based containers (e.g. HashMap), where the stored type is a meta object.	This exactly identifies (using Java's type system) where meta objects may be stored in and retrieved from hash collections
Multiple-MetaObjects	A call to the methods in the TM category (Table 4.1) that return arrays of meta objects	This exactly identifies the construct of interest using Java's name resolution
Environment-Strings	Any call to the methods retrieving strings from the environment [SR09] which are inlined as actual parameters in calls to reflection	Finds trivial flow of information from the environment into the Reflection API, and nothing more
Undecidable-Filtering	`for`,`while`,`do` statements with in the body a call to any of the methods in the TM or SG categories (Table 4.1) and a call to any *dynamic language feature*	Probably finds code where filtering is implemented using a loop code idiom, given that it also uses predicates and lookups from the Reflection API.
NoProxy	Any usage of the Reflection API for dynamic proxies	This is an exact query (also used in Section 4.4).

matched code in the corpus. For each pattern which is not exact by definition, we report the precision after sampling 10 instances and record the intent of the code examples as we interpreted it to confirm or deny the rationales of Table 4.6.

The patterns performed well; at least 8 out of the 10 sampled methods did challenge the limitation or assumption. In the sampled methods we observed that most of the challenging cases involve highly dynamic reflection, where the code uses complex data-dependent predicates to decide which methods to invoke or fields to modify. These predicates operated both on strings and meta objects. We also observed that exceptions were often ignored to continue with a next possible candidate.

4.5.2 *Results for Corpus Impact Analysis*

The Impact column of Table 4.7 answers SQ3, detailing for each pattern its impact in the corpus in terms of projects covered by at least a single match. Note that between the patterns the percentages are not comparable due to possible overlap. Each percentage implies a minimal amount of problematic code instances for the related assumption or limitation, so we find a lower-bound on the impact of a static analysis tool which would be able to resolve these hard cases.

Here we interpret the reported impact percentage for each limitation qualitatively: (a) the impact of CorrectCasts seems low, so we do not find evidence in this corpus that this is a bad assumption; (b) we can conclude that detailed modeling of exceptions can not be avoided; (c) we see that the combination of collections and reflection (arrays, lists, and tables) is relevant for about half of the corpus, so this is an important area of attention; (d) we find complex computations around the filtering of meta objects in almost half of the projects, which signals new opportunities for soundy assumptions for computing with meta objects; finally, (e) a significant part of the corpus is tainted directly by the use of dynamic proxies, for which no clear solution seems to be on the horizon.

> **SQ3**: Real Java code frequently challenges limitations of the existing static analysis tools, in particular, in relation to modeling of exceptions, collections, filtering of meta objects and dynamic proxies. The impact of CorrectCasts seems low.

The summary **answer to the Main Research Question** is that apart from CorrectCasts, the limitations and assumptions of static analysis tools for which we have an AST pattern are challenged in significant numbers in this corpus.

Table 4.7: Impact of limitation patterns (Table 4.6) in the corpus.

Pattern	Impact	Precision	Code intent
CorrectCasts	4%	8/10	Supplying a fallback or looping through candidates and swallowing the exception
Ignoring-Exceptions1	23%	10/10	Falling back to a less specific Meta Object, or switching to a different `ClassLoader`
Ignoring-Exceptions2	38%	9/10	Iterating through candidates and either breaking when one does not throw an exception, or continuing to the next candidates
Inaccurate-Indexed-Collections	55%	exact	Iterating through a signature of an meta object
InaccurateSets-AndMaps	38%	exact	Meta objects as function pointers in a table, mapping to objects, caching around Reflection API
NoMultiple-MetaObjects	54%	exact	Looking through candidates, performing mass updates of fields, checking signatures
Ignoring-Environment	2%	10/10	Only 9 instances found, they were all dependency injection
Undecidable-Filtering	48%	8/10	Trying different names of meta objects, filtering method and fields based on signature
NoProxy	21%	exact	Wrapping objects for caching or transactions, automatically converting between comparable interfaces

4.6 DISCUSSION

4.6.1 *Threats to validity*

A different categorization of "dynamic language features" in Section 4.2 might influence the answers to our research questions. To mitigate issues with the categorization we explicitly included a grammar fully covering the reflection API.

The SLR in Section 4.3 was conducted in 2015. To the best of our knowledge all material appeared since has been included in Section 4.3. The reading and annotating of the literature itself was a human task for which we implemented mitigating cross checks and validation steps.

Although the corpus in Section 4.4 has been constructed using state-of-the-art methods for maximum variation of meta data, the choice of meta data variables and the universe the projects are sampled from can be discussed. To the best of our

knowledge there exists no better means for sampling an unbiased and representative corpus of open-source projects.

In Section 4.5, we used AST patterns to assess the occurrence of challenging code. To mitigate the arbitrariness of the patterns, we maintained a direct trace between the patterns and literature study in Section 4.3 in Table 4.4. However, any undocumented assumptions or implicit limitations have naturally not been mapped. The patterns themselves could be inaccurate, which was discussed and mitigated in Section 4.5.

The answer to the main question, claiming a high impact of known limitations of static analysis tools, must be interpreted in context of the aforementioned threats to validity.

4.6.2 *The Dual Question of SQ3*

The question of how well static analysis tools actually do on code which uses reflection, rather than their limitations is relevant. The review in Section 4.3 and the corpus in Section 4.4 provide a starting point for answering it. However, a set of full comparative studies would be necessary, grouped by the **goal** of comparable analyses, by running the actual tools (where available) on the corpus. The respective coverage of the corpus for selecting the first 50, 100 or 200 projects are 56%, 72% and 88%. The first projects in the corpus are the most representative, so initial studies could be performed on one of these prefixes of the corpus. The configuration and execution of each tool for each project in the corpus, and the interpretation of detailed results per analysis group in this proposed study is at the scale of a community effort.

4.6.3 *Related work*

Next to the focused literature review of Section 4.3 we position this chapter in a wider field of empirical analysis of source code. Reflection and related forms of dynamic behavior are supported by many programming languages. Not surprisingly, reflection usage has been studied, e.g. for such languages as Smalltalk [CRT+13], JavaScript [RHB+11; RLB+10], PHP [Hil15b; HKV13] and Python [ÅST+14; HH09]. Despite the differences between programming languages studied as well as the methodologies used by the authors, all those papers agree with each other and with our observations made in Section 4.4: reflection mechanisms are used frequently, and they often cannot be completely resolved statically.

Even if the current observations are in line with previous work, they are unexpected. The current study is on the statically typed language Java rather than the aforementioned dynamically typed languages; for Java the use of reflection is expected to be the exception rather than commonplace. The Java language is designed to provide both clear feedback to the programmer and a built-in notion of code security, based on its static semantics. We find it surprising that reflection – the

back door to dynamic language features – is used so often and in such a way that it does undermine these design goals. Selecting Java as a platform for robust and safe software engineering provides fewer guarantees than perhaps thought.

A related topic is language feature adoption. Parnin et al. have studied adoption of Java generics [PBM13], Pinto et al. studied concurrent programming constructs [PTF+15], and Dyer et al. studied features prior to their official release [DRN+14]. Similar studies have also been conducted, e.g. for C# [CKS15] and PHP [Hil15a].

Since we have conducted our SLR in October 2015 additional papers have been published on static analysis of Java programs using reflection, witnessing the continuing attention to this topic from the research community. Harvester [RAM+16] combines static and dynamic analyses to combat malware obfuscation. Resolution of reflective calls is done by the dynamic analysis. HornDroid [CGM16] implements a simple string analysis and, similarly to DroidSafe [GKP+15], replaces reflective calls with the direct ones whenever the string analysis renders it possible. DroidRA [LBO+16] models the use of reflection with COAL [OLD+15] and reduces the resolution of reflective calls to a composite constant propagation problem.

Beyond related work for Java, without going into details, all research in and applications of static analysis techniques to dynamically typed programming languages is relevant, e.g. [AM14; SDC+12]. Our empirical observations (Section 4.5) suggest that application of the existing soundy techniques for analyzing dynamic languages to Java could have an impact.

4.6.4 *Implications for Java Software Engineers*

The data shows that reflection is not only used often, but it is also used in a way challenging to static analysis. If robustness is of high priority, then the following tactics are expected to have a positive effect: (a) do not factor out reusable reflective code in type-polymorphic methods, since the CorrectCasts assumption is highly useful, keeping casts to concrete types close to the use of dynamic language features will keep code analyzable; (b) avoid the use of dynamic proxies at any cost (c) use local variables or fields to store references to meta objects rather than collections; (d) avoid loops over bounded collections of meta objects; and (e) test for preconditions rather than to wait for exceptions such as `ClassCastException`.

Given the observations in Section 4.5, applying these tactics should lower the impact of the assumptions and limitations of static analysis tools and hence will make Java code more robust. All tactics trade more lines of code for better analyzability.

4.6.5 *Implications for Static Analysis Researchers*

For all reported challenges for static analysis tools for which we have an AST pattern, save the CorrectCasts assumption, the evidence suggests investigating opportunities

for more soundy assumptions in static analysis tools. It can also motivate Java language or API extensions which cover the current uses of the reflection API with safer counterparts. The literature survey suggests looking into combinations with dynamic analysis and user annotations. Note that the highly advanced analysis tools *already solve* a number of these challenges (such as exception handling), but further improvement to get similar accuracy for higher efficiency is warranted since these tools would run faster on a part of the corpus [SBK+15].

The negative impact of the CorrectCasts assumption seems low, so even more aggressive use of said assumption to reason back from a cast and infer more concrete details about possible semantics is warranted.

A novel soundy assumption on the semantics of dynamic proxies would have a significant impact, since currently all static analysis techniques ignore their existence completely (which is definitely unsound). For example, we observed that a useful soundy assumption might be that client code can remain "oblivious" to any proxy handlers that wrap arbitrary objects (that implement the same interface) to introduce ignorable aspects such as caching, offline serialization or transactional behavior.

We observed that exceptions are used as gotos, especially in the context of reflection. Hence, a special treatment of the code which catches these exceptions is warranted. Treating common idioms of such "error handling" should have a significant effect in the corpus, without having to use or introduce a general solution for exception handling per sé.

We see how relevant collections of meta objects (arrays, lists, and tables) are for analyzing the corpus. Since most collections of meta objects are bounded – they are acquired via bounded Reflection API methods – it should be possible to make more aggressive soundy assumptions around their usage. For instance, one can aggressively unroll iterators over meta object collections, or to soundily assume order independence.

Finally, considering the impact of UndecidableFiltering in the corpus in combination with MultiMetaObjects and the collection usage we see opportunities for the application of analysis techniques designed for dynamic languages (e.g. Javascript). Such dynamic Java code is akin to Javascript or PHP code. For example a form of determinancy analysis [AM14; SSD+13] might be ported for the Java reflection case.

4.7 CONCLUSIONS

Contemporary Java static analysis tools use pragmatic soundy techniques for dealing with the fundamental challenges around analyzing the Reflection API. Earlier work identified the need for empirical studies, relating these techniques to the way programmers actually use the Reflection API in real code.

With this chapter we contributed (a) a comprehensive survey of the literature on the features and limitations of static analysis tools targeting reflective Java projects,

(b) a representative corpus of 461 open-source Java projects, (c) an overview of the usage of the Reflection API by real Java code and (d) an AST-based analysis of how often the assumptions and limitations of the surveyed static analyses are challenged by real Java code.

The highlights among the empirical observations are that of all projects, in 78% dynamic language features are used. Moreover, 21% use dynamic proxies, 38% use exceptions for non-exceptional flow around reflection, 48% filter meta objects dynamically, and 55% store meta objects in generic collections. All those features are known to be problematic for static analysis tools. We could identify violations of the correct casts assumption in only 4% of the projects.

We conclude that (a) Java software engineers could make their code more analyzable by avoiding challenging code idioms around reflection, (b) introducing new soundy assumptions for novel static analysis techniques around the Reflection API is bound to have a significant impact in real Java code.

ACKNOWLEDGMENTS

We thank Jeroen van den Bos and Mark Hills for helpful feedback on drafts of this chapter and Anders Møller for his feedback on the topic of our research.

5 CONCLUSIONS AND PERSPECTIVES

This chapter summarizes the conclusions of Chapters 2–4 and places the results into a larger perspective.

5.1 RQ1: EXPLORING THE LIMITS OF DOMAIN MODEL RECOVERY

To understand the upper limit of reverse engineering domain models from source code, we have used an empirical study of two applications to answer the first main research research:

Research Question 1 (RQ1)

How much of a domain model can be recovered from source code under *ideal* circumstances?

This question was decomposed into two research sub-questions, for the first sub-question we wanted to understand which parts of the domain are even implemented by the application, leading to the second sub-question: can we recover those parts from the source code of that application?

5.1.1 *Result*

The first sub-question was answered by traversing the User Interface (UI) of the applications and building a domain model on the same level of abstraction as the user would interact with the application. Comparing this domain model to the reference domain model extracted from a project management reference book, we could calculate which parts of the reference domain model are implemented by the applications. We observed that the applications only implemented a small – less than 20% – part of the domain. The domain models extracted from the UI were then used as a frame of reference for the recovery of domain models from source code.

For the second sub-question, we manually constructed the domain models by reading all 36.1 KSLOC of source code. We compared these to the UI domain models and found that we could recover all concepts. Moreover, for one of the applications, we recovered more domain concepts than present in the UI. The high precision – between 79% and 92% – of the domain model recovery from source code compared to those recovered from the UI, showed that it was feasible to manually filter implementation details.

Conclusion RQ1

Domain models are recoverable from the source code of modern applications, making domain model recovery a valuable component during re-engineering.

5.1.2 *Perspective*

The results of this study is to serve as a baseline of what is possible for future work in automated domain model recovery approaches. The manually extracted models – made available online ([Lan13]) – can be used as an oracle for reverse engineering tools, enabling qualitative validation of an new approach next to the more common quantitative validation.

5.2 RQ2: EXPLORING THE RELATIONSHIP BETWEEN CC AND SLOC

To understand why the duality between the popularity and the reported redundancy between Source Lines of Code (SLOC) and Cyclomatic Complexity (CC) existed, we have used an empirical study of two large corpora of open source software to answer the second research question:

Research Question 2 (RQ2)

Is there a strong linear (Pearson) correlation between CC and SLOC metrics?

This question was translated into multiple hypotheses that might explain why almost all of the related work reported a strong linear correlation between CC and SLOC, while we could not confirm the same. Using appropriate statistical methods we then tried to reject all the hypotheses.

5.2.1 *Results*

While we tried very hard to reproduce results reported in related work, we did not find evidence for a strong linear correlation between CC and SLOC in our two corpora of Java and C software. The primary cause could be attributed to the high variance of CC over the whole range of SLOC. This lack of correlation could not be explained by an alternative interpretation of CC. However, summing CC and SLOC on a per file level – instead of a method level – could have caused the high correlation reported by releated work. It could also have been caused by the – more recently popular – log transform of the data. This log transform complicates the interpretation of what this correlation indicates for relationship between CC and SLOC.

We observed increasing variance in CC as SLOC increased, even after a log transform. This non-constant variance further complicates the interpretation of the

linear regressions as an argument for SLOC to predict CC. Moreover, for the larger methods in our corpora, we observed weak correlation at best. Looking in the tail of the data, we observed the correlation decreasing and even more so for the correlations after a log transform. Therefore, we argue that CC should remain to be used next to SLOC. Especially for the larger – arguably more interesting – methods, CC can be a useful metric to further discriminate between methods of the same SLOC.

Conclusion RQ2

Contrary to the majority of the previous studies, we do not conclude that CC is redundant to SLOC.

5.2.2 *Perspective*

Software metrics are popular in both research and industry. Our study analyzed a common critique that CC is redundant to SLOC. We have found that the main argument for this critique does not hold for the analyzed large corpora of software.

Another related critique is based on the research by El Emam et.al. [EBG+01] on the confounding impact of class size on object oriented metrics. The results of our work could motivate similar future work on how this holds for large corpora, and especially if it still holds for object oriented metrics which do not sum on a class level. As a follow-up of our study by Zhang et.al. [ZHM+16] who analyzed and reported on the strong effect summation can have on defect prediction models.

5.3 RQ3: EXPLORING THE LIMITS OF STATIC ANALYSIS AND REFLECTION

To understand how Java's Reflection Application Programming Interface (API) is supported by static analysis tools and how this translates to real world use of this API, we have used an empirical study on a corpus of open source Java software to answer the third research question:

Research Question 3 (RQ3)

What are the limits of state-of-the-art static analysis tools supporting the Reflection API and how do these limits relate to real world Java code?

We decomposed this question into three sub-questions, the first was to understand how state-of-the-art static analysis tools support reflection and what they do not support. The second sub-question was how often sections of the Reflection API are actually used. The last sub-question was to understand how often the limitations of the static analysis tools are challenged in real Java source code.

5.3.1 *Results*

After an extensive Systematic Literature Review (SLR), we identified 20 static analysis tools that introduced specific heuristics to handle the effects of the Reflection API. We observed that more recent tools have been adding deeper modeling of Java's semantics, and even more specifically around Strings and Reflection's Meta Objects. We identified 10 common assumptions and limitations, these are translated into 9 Abstract Syntax Tree (AST) patterns.

We constructed a new compact and diverse corpus of Java open source software, and in these 461 projects we found that 78% projects used the dynamic language features of the Reflection API at least once. For these projects, a static analysis tool without support for reflection would return incorrect results.

We found that certain common assumptions and limitations of static analysis tools were violated quite often. Dynamic proxies were used in 21% of the projects, exceptions as non-exceptional control flow in 38%, dynamically filtering meta objects in 48%, and storing meta objects in generic collections in 55% of all projects. We then randomly sampled cases of each violation, and proposed new heuristics for static analysis tools on how to handle these cases.

Conclusion RQ3

In our corpus, heuristics for the Reflection API are needed for 78% of the projects. Common assumptions and limitations are often violated. We propose new assumptions that could significantly improvement the soundness of static analysis tools.

5.3.2 *Perspective*

A wide variety of re-engineering tasks depends on the accuracy of static analysis tools. Until 2005, static analysis tools did not specifically model reflection. In 2005, Livshits [LWL05] introduced new (unsound) assumptions and heuristics, which allow static analysis tools to support more Java software. The recent manifesto defending this "soundiness" [LSS+15] identified the question that we have answered in our research: which parts of Java software can we already handle, and what challenge should we tackle next?

Our results can be used for future static analysis research to prioritize which assumptions and limitations to focus on first. Like the first publication proposing assumptions [LWL05], we have also proposed new assumptions for the most commonly found violations of limitations.

5.4 ADVANCING REVERSE ENGINEERING

Based on the result of the research presented in this thesis, we identify the following directions for future research:

- Automate the recovery of domain models from source code and possibly the UI [BP12; HPM03], re-using the published models for measuring the quality of the results, comparing to related work [AT10; RFJ08].
- Validate the domain models with domain experts and the original developers of the applications they were extracted from. In this way we can understand if our manual extraction was correct and why this view might differ from what the developers intended.
- Develop new measures for comparing domain models, especially taking into consideration the – sometimes transitive – relationship between concepts. For our work in Chapter 2 we could not find a suitable measure of how different the relations between two models are. This was further complicated by the difficulty of the common case of different levels of abstraction in the models.
- Analyze the effect of using more fine grained metrics for defect (or effort) prediction. It is common to predict buggy files or directories, even though some of the metrics in the model are on a method or function level, our results in Chapter 3 suggest the inevitable aggregation might hide interesting relations.
- Develop new static analysis tools with more – unsound – support for reflection features such as dynamic proxies. Such that more real world programs can be supported with a higher precision.
- Develop a new small benchmark for static analysis tools of complicated Java patterns, and construct the expected result – for example call graphs – manually. In this way, we can go beyond the more common quantitative – graph size – comparison of static analysis tools.
- Extend IDEs with warnings about code that uses reflection in a hard to analyze way. Such that software engineers can choose to write simpler reflective code. This can improve the guarantees of a later run static analysis tasks such as refactoring.

For many questions in software engineering more data helps. Analyzing large corpora of software is time consuming and error prone. Yet large corpora provide the opportunity to observe a wider spectrum of instances than possible with more controlled experiments. The primary contribution of this thesis is applying empirical studies on large corpora to answer open questions in software engineering research.

REFERENCES

[AT10] S. L. Abebe and P. Tonella. "Natural Language Parsing of Program Element Names for Concept Extraction". In: *The 18th IEEE International Conference on Program Comprehension, ICPC 2010, Braga, Minho, Portugal, June 30-July 2, 2010*. IEEE Computer Society, 2010, pp. 156–159. DOI: 10.1109/ICPC.2010.29 (cit. on pp. 22, 42, 121).

[AT11] S. L. Abebe and P. Tonella. "Towards the Extraction of Domain Concepts from the Identifiers". In: *18th Working Conference on Reverse Engineering, WCRE 2011, Limerick, Ireland, October 17-20, 2011*. Ed. by M. Pinzger, D. Poshyvanyk, and J. Buckley. IEEE Computer Society, 2011, pp. 77–86. DOI: 10.1109/WCRE.2011.19 (cit. on pp. 22, 42).

[Abr10] A. Abran. "Cyclomatic Complexity Number: Analysis of its Design". In: *Software Metrics and Software Metrology*. Wiley-IEEE Computer Society Pr, 2010. Chap. 6, pp. 131–143. ISBN: 9780470597200 (cit. on pp. 10, 59, 83).

[ÅST+14] B. Åkerblom, J. Stendahl, M. Tumlin, and T. Wrigstad. "Tracing dynamic features in python programs". In: *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India*. Ed. by P. T. Devanbu, S. Kim, and M. Pinzger. ACM, 2014, pp. 292–295. DOI: 10.1145/2597073.2597103 (cit. on p. 111).

[AL12] K. Ali and O. Lhoták. "Application-Only Call Graph Construction". In: *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings*. Ed. by J. Noble. Vol. 7313. Lecture Notes in Computer Science. Springer, 2012, pp. 688–712. DOI: 10.1007/978-3-642-31057-7_30 (cit. on p. 99).

[AL13] K. Ali and O. Lhoták. "Averroes: Whole-Program Analysis without the Whole Program". In: *ECOOP 2013 - Object-Oriented Programming - 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings*. Ed. by G. Castagna. Vol. 7920. Lecture Notes in Computer Science. Springer, 2013, pp. 378–400. DOI: 10.1007/978-3-642-39038-8_16 (cit. on pp. 99, 101).

[ARL+14] K. Ali, M. Rapoport, O. Lhoták, J. Dolby, and F. Tip. "Constructing Call Graphs of Scala Programs". In: *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*. Ed. by R. Jones. Vol. 8586. Lecture Notes in Computer Science. Springer, 2014, pp. 54–79. DOI: 10.1007/978-3-662-44202-9_3 (cit. on p. 99).

[AFJ+09] M. Alpuente, M. A. Feliú, C. Joubert, and A. Villanueva. "Defining Datalog in Rewriting Logic". In: *Logic-Based Program Synthesis and Transformation, 19th International Symposium, LOPSTR 2009, Coimbra, Portugal, September 2009, Revised Selected Papers*. Ed. by D. D. Schreye. Vol. 6037. Lecture Notes in Computer Science. Springer, 2009, pp. 188–204. DOI: 10.1007/978-3-642-12592-8_14 (cit. on p. 99).

[AFJ+10] M. Alpuente, M. A. Feliú, C. Joubert, and A. Villanueva. "Datalog-Based Program Analysis with BES and RWL". In: *Datalog Reloaded - First International Workshop, Datalog 2010, Oxford, UK, March 16-19, 2010. Revised Selected Papers*. Ed. by O. de Moor, G. Gottlob, T. Furche, and A. J. Sellers. Vol. 6702. Lecture Notes in Computer Science. Springer, 2010, pp. 1–20. DOI: 10.1007/978-3-642-24206-9_1 (cit. on p. 99).

[AM14] E. Andreasen and A. Møller. "Determinacy in static analysis for jQuery". In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*. Ed. by A. P. Black and T. D. Millstein. ACM, 2014, pp. 17–31. DOI: 10.1145/2660193.2660214 (cit. on pp. 112, 113).

[ACC+02] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, and E. Merlo. "Recovering Traceability Links between Code and Documentation". In: *IEEE Transactions on Software Engineering* 28.10 (2002), pp. 970–983. DOI: 10.1109/TSE.2002.1041053 (cit. on p. 4).

[AHM+08] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. "Using Static Analysis to Find Bugs". In: *IEEE Software* 25.5 (Sept. 2008), pp. 22–29. DOI: 10.1109/ms.2008.130 (cit. on p. 13).

[BP12] M. Bacíková and J. Porubän. "Analyzing stereotypes of creating graphical user interfaces". In: *Central Europe Journal Computer Science* 2.3 (2012), pp. 300–315 (cit. on pp. 42, 121).

[BCS+12] R. Baggen, J. P. Correia, K. Schill, and J. Visser. "Standardized code quality benchmarking for improving software maintainability". In: *Software Quality Journal* 20.2 (2012), pp. 287–307. ISSN: 0963-9314. DOI: 10.1007/s11219-011-9144-9 (cit. on p. 49).

[BBC+06] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. "Thorough Static Analysis of Device Drivers". In: *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*. EuroSys '06. Leuven, Belgium: ACM, 2006, pp. 73–85. ISBN: 1-59593-322-0. DOI: 10.1145/1217935.1217943 (cit. on p. 13).

[BJM+15] P. Barros, R. Just, S. Millstein, P. Vines, W. Dietl, M. d'Amorim, and M. D. Ernst. "Static Analysis of Implicit Control Flow: Resolving Java Reflection and Android Intents (T)". In: *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*. Ed. by M. B. Cohen, L. Grunske, and M. Whalen. IEEE Computer Society, 2015, pp. 669–679. DOI: 10.1109/ASE.2015.69 (cit. on pp. 101, 104).

[Bas93] V. R. Basili. "The Experimental Paradigm in Software Engineering". In: *Proceedings of the International Workshop on Experimental Software Engineering Issues: Critical Assessment and Future Directions*. London, UK, UK: Springer-Verlag, 1993, pp. 3–12. ISBN: 3-540-57092-6 (cit. on p. 14).

[BP84] V. R. Basili and B. T. Perricone. "Software Errors and Complexity: An Empirical Investigation". In: *Communications of the ACM* 27.1 (1984), pp. 42–52. DOI: 10.1145/69605.2085 (cit. on pp. 8, 49, 54, 81).

[BHK+15] B. Basten, M. Hills, P. Klint, D. Landman, A. Shahi, M. J. Steindorfer, and J. J. Vinju. "M3: A general model for code analytics in rascal". In: *1st IEEE International Workshop on Software Analytics, SWAN 2015, Montreal, QC, Canada, March 2, 2015*. Ed. by O. Baysal and L. Guerrouj. IEEE Computer Society, 2015, pp. 25–28. DOI: 10.1109/SWAN.2015.7070485 (cit. on pp. 63, 105).

[BKS+09] M. Bianco, D. Kaneider, A. Sillitti, and G. Succi. "Fault-Proneness Estimation and Java Migration: A Preliminary Case Study". In: *Proceedings of International Conference on SOFTWARE, SERVICES & SEMANTIC TECHNOLOGIES*. Demetra EOOD, 2009, pp. 124–131. ISBN: 978-954-9526-62-2 (cit. on pp. 57, 81).

[Big89] T. J. Biggerstaff. "Design Recovery for Maintenance and Reuse". In: *IEEE Computer* 22.7 (July 1989). Ed. by R. S. Arnold, pp. 36–49. DOI: 10.1109/2.30731 (cit. on pp. 4, 6, 22).

[BMW93] T. J. Biggerstaff, B. G. Mitbander, and D. E. Webster. "The Concept Assignment Problem in Program Understanding". In: *Proceedings of the 15th International Conference on Software Engineering, Baltimore, Maryland, USA, May 17-21, 1993*. Ed. by V. R. Basili, R. A. DeMillo, and T. Katayama. IEEE Computer Society / ACM Press, 1993, pp. 482–498 (cit. on pp. 4, 22).

[BGH+06] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. L. Hosking, M. Jump, H. B. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann. "The DaCapo benchmarks: java benchmarking development and analysis". In: *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*. Ed. by P. L. Tarr and W. R. Cook. ACM, 2006, pp. 169–190. DOI: 10.1145/1167473.1167488 (cit. on p. 103).

[BGC15] S. Blackshear, A. Gendreau, and B. E. Chang. "Droidel: a general approach to Android framework modeling". In: *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, SOAP@PLDI 2015, Portland, OR, USA, June 15 - 17, 2015*. Ed. by A. Møller and M. Naik. ACM, 2015, pp. 19–25. DOI: 10.1145/2771284.2771288 (cit. on p. 100).

[BSS+11] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini. "Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders". In: *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*. Ed. by R. N. Taylor, H. C. Gall, and N. Medvidovic. ACM, 2011, pp. 241–250. DOI: 10.1145/1985793.1985827 (cit. on pp. 99, 100).

[BS09] M. Bravenboer and Y. Smaragdakis. "Strictly declarative specification of sophisticated points-to analyses". In: *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*. Ed. by S. Arora and G. T. Leavens. ACM, 2009, pp. 243–262. DOI: 10.1145/1640089.1640108 (cit. on pp. 101, 102).

[BKB+07] P. Brereton, B. A. Kitchenham, D. Budgen, M. Turner, and M. Khalil. "Lessons from applying the systematic literature review process within the software engineering domain". In: *Journal of Systems and Software* 80.4 (2007), pp. 571–583. DOI: 10.1016/j.jss.2006.07.009 (cit. on p. 99).

[BP79] T. Breusch and A. Pagan. "A Simple Test for Heteroscedasticity and Random Coefficient Variation". In: *Econometrica* 47.5 (Sept. 1979), pp. 1287–1294. ISSN: 00129682. DOI: 10.2307/1911963 (cit. on p. 77).

[BS98] H. Bunke and K. Shearer. "A graph distance metric based on the maximal common subgraph". In: *Pattern Recognition Letters* 19.3-4 (1998), pp. 255–259 (cit. on p. 40).

[CRT+13] O. Callaú, R. Robbes, É. Tanter, and D. Röthlisberger. "How (and why) developers use the dynamic features of programming languages: the case of smalltalk". In: *Empirical Software Engineering* 18.6 (2013), pp. 1156–1194. DOI: 10.1007/s10664-012-9203-2 (cit. on p. 111).

[CGM16] S. Calzavara, I. Grishchenko, and M. Maffei. "HornDroid: Practical and Sound Static Analysis of Android Applications by SMT Solving". In: *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*. IEEE, 2016, pp. 47–62. DOI: 10.1109/EuroSP.2016.16 (cit. on p. 112).

[CFB+15] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen. "EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework". In: *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society, 2015 (cit. on p. 101).

[CKS15] P. Capek, E. Kral, and R. Senkerik. "Towards an Empirical Analysis of .NET Framework and C# Language Features' Adoption". In: *2015 International Conference on Computational Science and Computational Intelligence (CSCI)*. IEEE, Dec. 2015, pp. 865–866. DOI: 10.1109/CSCI.2015.90 (cit. on p. 112).

[CF07] A. Capiluppi and J. Fernández-Ramil. "A model to predict anti-regressive effort in Open Source Software". In: *23rd IEEE International Conference on Software Maintenance (ICSM 2007), October 2-5, 2007, Paris, France*. IEEE, 2007, pp. 194–203. DOI: 10.1109/ICSM.2007.4362632 (cit. on pp. 53, 56).

[CG07] M. M. Carey and G. C. Gannod. "Recovering Concepts from Source Code with Automated Concept Identification". In: *15th International Conference on Program Comprehension (ICPC 2007), June 26-29, 2007, Banff, Alberta, Canada*. IEEE Computer Society, 2007, pp. 27–36. DOI: 10.1109/ICPC.2007.31 (cit. on pp. 22, 42).

[CLN+87] D. B. Carr, R. J. Littlefield, W. L. Nicholson, and J. S. Littlefield. "Scatterplot Matrix Techniques for Large N". In: *Journal of the American Statistical Association* 82.398 (1987), pp. 424–436. ISSN: 01621459 (cit. on p. 66).

[CFP07] P. Centonze, R. J. Flynn, and M. Pistoia. "Combining Static and Dynamic Analysis for Automatic Identification of Precise Access-Control Policies". In: *23rd Annual Computer Security Applications Conference (ACSAC 2007), December 10-14, 2007, Miami Beach, Florida, USA*. IEEE Computer Society, 2007, pp. 292–303. DOI: 10.1109/ACSAC.2007.14 (cit. on p. 101).

[CWT83] J. M. Chambers, B. K. William S. Cleveland, and P. A. Tukey. "Comparing Data Distributions". In: *Graphical Methods for Data Analysis*. New York: Chapman and Hall, 1983. Chap. 2 (cit. on p. 74).

[Che76] P. P.-S. Chen. "The Entity-Relationship Model—Toward a Unified View of Data". In: *ACM Transactions on Database Systems* 1.1 (Mar. 1976), pp. 9–36. ISSN: 0362-5915. DOI: 10.1145/320434.320440 (cit. on p. 6).

[CM04] B. Chess and G. McGraw. "Static Analysis for Security". In: *IEEE Security and Privacy* 2.6 (Nov. 2004), pp. 76–79. ISSN: 1540-7993. DOI: 10.1109/MSP.2004.111 (cit. on p. 13).

[CK94] S. R. Chidamber and C. F. Kemerer. "A metrics suite for object oriented design". In: *IEEE Transactions on Software Engineering* 20.6 (June 1994), pp. 476–493. ISSN: 0098-5589. DOI: 10.1109/32.295895 (cit. on pp. 53, 83).

[CC90] E. J. Chikofsky and J. H. Cross. "Reverse Engineering and Design Recovery: A Taxonomy". In: *IEEE Software* 7.1 (1990), pp. 13–17 (cit. on p. 3).

[CSH06] N. Choi, I.-Y. Song, and H. Han. "A survey on ontology mapping". In: *SIGMOD Rec.* 35.3 (Sept. 2006), pp. 34–41. ISSN: 0163-5808. DOI: 10.1145/1168092.1168097 (cit. on p. 42).

[CMS03] A. S. Christensen, A. Møller, and M. I. Schwartzbach. "Precise Analysis of String Expressions". In: *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings*. Ed. by R. Cousot. Vol. 2694. Lecture Notes in Computer Science. Springer, 2003, pp. 1–18. DOI: 10.1007/3-540-44898-5_1 (cit. on pp. 97, 99, 101).

[CJ14] R. Coleman and M. A. Johnson. "A Study of Scala Repositories on Github". In: *International Journal of Advanced Computer Science and Applications* 5.7 (2014), pp. 141–148. DOI: 10.14569/IJACSA.2014.050721 (cit. on p. 85).

[CDS86] S. D. Conte, H. E. Dunsmore, and V. Y. Shen. *Software Engineering Metrics and Models*. Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc., 1986. ISBN: 0-8053-2162-4 (cit. on pp. 10, 51).

[CC94] B. Curtis and A. D. Carleton. "Seven±two software measurement conundrums". In: *Proceedings of the 1994 IEEE 2nd International Software Metrics Symposium, October 24-26, 1994, London, England, UK*. IEEE, 1994, pp. 96–105. DOI: 10.1109/METRIC.1994.344224 (cit. on pp. 51, 58).

[CSM79] B. Curtis, S. B. Sheppard, and P. Milliman. "Third Time Charm: Stronger Prediction of Programmer Performance by Software Complexity Metrics". In: *Proceedings of the 4th International Conference on Software Engineering*. ICSE '79. Munich, Germany: IEEE Press, 1979, pp. 356–360 (cit. on pp. 54, 80).

[Dan13] A. Danial. *Count Lines of Code Tool*. 2013. URL: http://cloc.sourceforge.net (visited on 02/01/2013) (cit. on p. 30).

[DMR94] J. DeBaud, B. Moopen, and S. Rugaber. "Domain Analysis and Reverse Engineering". In: *Proceedings of the International Conference on Software Maintenance, ICSM 1994, Victoria, BC, Canada, September 1994*. Ed. by H. A. Müller and M. Georges. IEEE Computer Society, 1994, pp. 326–335. DOI: 10.1109/ICSM.1994.336762 (cit. on p. 43).

[DDL99] S. Demeyer, S. Ducasse, and M. Lanza. "A hybrid reverse engineering approach combining metrics and program visualisation". In: *Sixth Working Conference on Reverse Engineering*. Institute of Electrical and Electronics Engineers (IEEE), Oct. 1999, pp. 175–186. DOI: 10.1109/WCRE.1999.806958 (cit. on p. 8).

[DRG+13] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. "Feature location in source code: a taxonomy and survey". In: *Journal of Software: Evolution and Process* 25.1 (2013), pp. 53–95. ISSN: 2047-7481. DOI: 10.1002/smr.567 (cit. on p. 4).

[DRS07] B. Dufour, B. G. Ryder, and G. Sevitsky. "Blended analysis for performance understanding of framework-based applications". In: *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2007, London, UK, July 9-12, 2007*. Ed. by D. S. Rosenblum and S. G. Elbaum. ACM, 2007, pp. 118–128. DOI: 10.1145/1273463.1273480 (cit. on p. 100).

[DRN+14] R. Dyer, H. Rajan, H. A. Nguyen, and T. N. Nguyen. "Mining billions of AST nodes to study actual and potential usage of Java language features". In: *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*. Ed. by P. Jalote, L. C. Briand, and A. van der Hoek. ACM, 2014, pp. 779–790. DOI: 10.1145/2568225.2568295 (cit. on p. 112).

[EN84] S. E. Edgell and S. M. Noon. "Effect of violation of normality on the t test of the correlation coefficient." In: *Psychological Bulletin* 95.3 (1984), pp. 576–583 (cit. on p. 67).

[EKS03] T. Eisenbarth, R. Koschke, and D. Simon. "Locating Features in Source Code". In: *IEEE Transactions on Software Engineering* 29.3 (2003), pp. 210–224. DOI: 10.1109/TSE.2003.1183929 (cit. on pp. 22, 43).

[EBG+01] K. E. Emam, S. Benlarbi, N. Goel, and S. N. Rai. "The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics". In: *IEEE Transactions on Software Engineering* 27.7 (2001), pp. 630–650. DOI: 10.1109/32.935855 (cit. on pp. 8, 56, 81, 119).

[EJM+14] M. D. Ernst, R. Just, S. Millstein, W. Dietl, S. Pernsteiner, F. Roesner, K. Koscher, P. Barros, R. Bhoraskar, S. Han, P. Vines, and E. X. Wu. "Collaborative Verification of Information Flow for a High-Assurance App Store". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*. Ed. by G. Ahn, M. Yung, and N. Li. ACM, 2014, pp. 1092–1104. DOI: 10.1145/2660267.2660343 (cit. on p. 101).

[Eva03] E. Evans. *Domain-Driven Design: Tacking Complexity In the Heart of Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Corporation, Inc., 2003. ISBN: 0321125215 (cit. on pp. 6, 7).

[FCH+11] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. "Android permissions demystified". In: *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*. Ed. by Y. Chen, G. Danezis, and V. Shmatikov. ACM, 2011, pp. 627–638. DOI: 10.1145/2046707.2046779 (cit. on pp. 97, 101, 102, 104).

[FWL+13] C. Feng, H. Wang, N. Lu, and X. M. Tu. "Log transformation: application and interpretation in biomedical research". In: *Statistics in Medicine* 32.2 (July 2013), pp. 230–239. ISSN: 1097-0258. DOI: 10.1002/sim.5486 (cit. on p. 82).

[FO00] N. Fenton and N. Ohlsson. "Quantitative analysis of faults and failures in a complex software system". In: *Software Engineering, IEEE Transactions on* 26.8 (Aug. 2000), pp. 797–814. ISSN: 0098-5589. DOI: 10.1109/32.879815 (cit. on pp. 8, 50, 55, 81).

[FB14] N. E. Fenton and J. Bieman. *Software Metrics – A Rigorous and Practical Approach (Third ed.)* CRC Press, 2014. ISBN: 978-1-4398-3823-5 (cit. on pp. 8, 9).

[FF79] A. R. Feuer and E. B. Fowlkes. "Some Results from an Empirical Study of Computer Software". In: *Proceedings of the 4th International Conference on Software Engineering*. ICSE '79. Munich, Germany: IEEE Press, 1979, pp. 351–355 (cit. on pp. 8, 49, 54, 60, 67, 71, 80, 82).

[Fin10] A. Fink. *Conducting Research Literature Reviews: From the Internet to Paper*. SAGE Publications, 2010. ISBN: 9781412971898 (cit. on p. 99).

[FD+15] S. Fink, J. Dolby, et al. *T.J. Watson Libraries for Analysis (WALA)*. 2015. URL: http://wala.sourceforge.net/wiki/index.php/Main_Page (visited on 12/10/2015) (cit. on pp. 99, 101).

[Gab13] M. A. F. Gabaldón. "Logic-based techniques for program analysis and specification synthesis". PhD thesis. Universitat Politècnica de València, Sept. 2013 (cit. on p. 101).

[GK91] G. K. Gill and C. F. Kemerer. "Cyclomatic Complexity Density and Software Maintenance Productivity". In: *IEEE Transactions on Software Engineering* 17.12 (Dec. 1991), pp. 1284–1288. ISSN: 0098-5589. DOI: 10.1109/32.106988 (cit. on pp. 55, 81).

[Gla94] R. L. Glass. "The Software-Research Crisis". In: *IEEE Software* 11.6 (Nov. 1994), pp. 42–47. ISSN: 0740-7459. DOI: 10.1109/52.329400 (cit. on p. 14).

[GKP+15] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard. "Information Flow Analysis of Android Applications in DroidSafe". In: *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society, 2015 (cit. on pp. 99, 112).

[GBB90] N. Gorla, A. Benander, and B. A. Benander. "Debugging Effort Estimation Using Software Metrics". In: *IEEE Transactions on Software Engineering* 16.2 (1990), pp. 223–231. ISSN: 0098-5589 (cit. on pp. 55, 80).

[GKM+00] T. Graves, A. Karr, J. Marron, and H. Siy. "Predicting fault incidence using software change history". In: *IEEE Transactions on Software Engineering* 26.7 (July 2000), pp. 653–661. ISSN: 0098-5589. DOI: 10.1109/32.859533 (cit. on pp. 55, 81).

[GMD+10] M. Grechanik, C. McMillan, L. DeFerrari, M. Comi, S. Crespi, D. Poshyvanyk, C. Fu, Q. Xie, and C. Ghezzi. "An Empirical Investigation into a Large-scale Java Open Source Code Repository". In: *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. Bolzano-Bozen, Italy: ACM, 2010, 11:1–11:10. ISBN: 978-1-4503-0039-1. DOI: 10.1145/1852786.1852801 (cit. on pp. 70, 81).

[HYG+13] J. Han, Q. Yan, D. Gao, J. Zhou, and R. H. Deng. "Comparing Mobile Privacy Protection through Cross-Platform Applications". In: *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*. The Internet Society, 2013 (cit. on pp. 101, 104).

[HKV07] I. Heitlager, T. Kuipers, and J. Visser. "A Practical Model for Measuring Maintainability". In: *Quality of Information and Communications Technology, 6th International Conference on the Quality of Information and Communications Technology, QUATIC 2007, Lisbon, Portugal, September 12-14, 2007, Proceedings*. Ed. by R. J. Machado, F. B. e Abreu, and P. R. da Cunha. IEEE Computer Society, 2007, pp. 30–39. DOI: 10.1109/QUATIC.2007.8 (cit. on pp. 8, 49, 50).

[HS90] S. M. Henry and C. Selig. "Predicting Source-Code Complexity at the Design Stage". In: *IEEE Software* 7.2 (1990), pp. 36–44. DOI: 10.1109/52.50772 (cit. on pp. 55, 61, 75, 81).

[HGR07] I. Herraiz, J. M. Gonzalez-Barahona, and G. Robles. "Towards a Theoretical Model for Software Growth". In: *Proceedings of the Fourth International Workshop on Mining Software Repositories*. MSR '07. Washington, DC, USA: IEEE Computer Society, 2007, 21:1–21:8. ISBN: 0-7695-2950-X. DOI: 10.1109/MSR.2007.31 (cit. on pp. 56, 60, 61, 67, 71, 75, 81, 82).

[HH10] I. Herraiz and A. E. Hassan. "Beyond lines of code: Do we need more complexity metrics?" In: *Making Software What Really Works, and Why We Believe It*. O'Reilly Media, 2010. Chap. 8, pp. 126–141 (cit. on pp. 57, 60, 61, 65, 67, 71, 75, 81, 82, 85).

[Hil15a] M. Hills. "Evolution of dynamic feature usage in PHP". In: *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015*. Ed. by Y. Guéhéneuc, B. Adams, and A. Serebrenik. IEEE Computer Society, 2015, pp. 525–529. DOI: 10.1109/SANER.2015.7081870 (cit. on pp. 107, 112).

[Hil15b] M. Hills. "Variable Feature Usage Patterns in PHP". In: *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*. Ed. by M. B. Cohen, L. Grunske, and M. Whalen. IEEE Computer Society, 2015, pp. 563–573. DOI: 10.1109/ASE.2015.72 (cit. on p. 111).

[HKV13] M. Hills, P. Klint, and J. J. Vinju. "An empirical study of PHP feature usage: a static analysis perspective". In: *International Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15-20, 2013*. Ed. by M. Pezzè and M. Harman. ACM, 2013, pp. 325–335. DOI: 10.1145/2483760.2483786 (cit. on pp. 107, 111).

[HGH08] A. Hindle, M. W. Godfrey, and R. C. Holt. "Reading Beside the Lines: Indentation as a Proxy for Complexity Metric". In: *The 16th IEEE International Conference on Program Comprehension, ICPC 2008, Amsterdam, The Netherlands, June 10-13, 2008*. Ed. by R. L. Krikhaar, R. Lämmel, and C. Verhoef. IEEE Computer Society, 2008, pp. 133–142. DOI: 10.1109/ICPC.2008.13 (cit. on pp. 53, 56).

[HvDD+07] M. Hirzel, D. von Dincklage, A. Diwan, and M. Hind. "Fast online pointer analysis". In: *ACM Transactions on Programming Languages and Systems* 29.2 (Apr. 2007). ISSN: 0164-0925. DOI: 10.1145/1216374.1216379 (cit. on p. 100).

[HH09] A. Holkner and J. Harland. "Evaluating the dynamic behaviour of Python applications". In: *Computer Science 2009, Thirty-Second Australasian Computer Science Conference (ACSC 2009), Wellington, New Zealand, January 19-23, 2009, Proceedings*. Ed. by B. Mans. Vol. 91. CRPIT. Australian Computer Society, 2009, pp. 17–25 (cit. on p. 111).

[HPM03] I. Hsi, C. Potts, and M. M. Moore. "Ontological Excavation: Unearthing the core concepts of the application". In: *10th Working Conference on Reverse Engineering, WCRE 2003, Victoria, Canada, November 13-16, 2003*. Ed. by A. van Deursen, E. Stroulia, and M. D. Storey. IEEE Computer Society, 2003, pp. 345–352. DOI: 10.1109/WCRE.2003.1287265 (cit. on pp. 42, 121).

[HDM14] W. Huang, Y. Dong, and A. Milanova. "Type-Based Taint Analysis for Java Web Applications". In: *Fundamental Approaches to Software Engineering - 17th International Conference, FASE 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*. Ed. by S. Gnesi and A. Rensink. Vol. 8411. Lecture Notes in Computer Science. Springer, 2014, pp. 140–154. DOI: 10.1007/978-3-642-54804-8_10 (cit. on p. 99).

[Hun02] S. Hunston. *Corpora in Applied Linguistics*. Cambridge applied linguistics series. Cambridge University Press, 2002. ISBN: 9783125340503 (cit. on p. 103).

[Ins08] P. M. Institute, ed. *A Guide to the Project Management Body of Knowledge*. 4th. Project Management Institute, 2008. ISBN: 9781933890517 (cit. on pp. 7, 24, 27).

[IC14] M. Islam and C. Csallner. "Generating Test Cases for Programs that Are Coded against Interfaces and Annotations". In: *ACM Transactions on Software Engineering and Methodology* 23.3 (2014), p. 21. DOI: 10.1145/2544135 (cit. on p. 100).

[JHS+09] G. Jay, J. E. Hale, R. K. Smith, D. P. Hale, N. A. Kraft, and C. Ward. "Cyclomatic Complexity and Lines of Code: Empirical Evidence of a Stable Linear Relationship". In: *Journal of Software Engineering and Applications* 2.3 (2009), pp. 137–143. DOI: 10.4236/jsea.2009.23020 (cit. on pp. 57, 60, 67, 71, 81, 82).

[JMF14] A. Jbara, A. Matan, and D. G. Feitelson. "High-MCC Functions in the Linux Kernel". In: *Empirical Software Engineering* 19.5 (2014), pp. 1261–1298. ISSN: 1382-3256. DOI: 10.1007/s10664-013-9275-7 (cit. on pp. 8, 49, 53, 57, 75, 80).

[JG98] D. N. Joanes and C. A. Gill. "Comparing measures of sample skewness and kurtosis". In: *Journal of the Royal Statistical Society: Series D (The Statistician)* 47.1 (1998), pp. 183–189. DOI: 10.1111/1467-9884.00122 (cit. on p. 68).

[KWN05] U. Kelter, J. Wehren, and J. Niere. "A Generic Difference Algorithm for UML Models". In: *Software Engineering 2005*. Ed. by P. Liggesmeyer, K. Pohl, and M. Goedicke. Vol. 64. LNI. GI, 2005, pp. 105–116 (cit. on p. 40).

[KS97] C. F. Kemerer and S. A. Slaughter. "Determinants of Software Maintenance Profiles: An Empirical Investigation". In: *Journal of Software Maintenance* 9.4 (July 1997), pp. 235–251. ISSN: 1040-550X (cit. on pp. 55, 80).

[KAD+14] F. Khomh, B. Adams, T. Dhaliwal, and Y. Zou. "Understanding the impact of rapid releases on software quality". In: *Empirical Software Engineering* 20.2 (2014), pp. 336–373. ISSN: 1382-3256. DOI: 10.1007/s10664-014-9308-x (cit. on p. 87).

[KYY+12] J. Kim, Y. Yoon, K. Yi, and J. Shin. "ScanDal: Static Analyzer for Detecting Privacy Leaks in Android Applications". In: *MoST 2012: Mobile Security Technologies*. Ed. by H. Chen, L. Koved, and D. S. Wallach. Los Alamitos, CA, USA: IEEE, May 2012 (cit. on p. 101).

[KP87] B. Kitchenham and L. Pickard. "Towards a constructive quality model. Part 2: Statistical techniques for modelling software quality in the ESPRIT REQUEST project". In: *Software Engineering Journal* 2.4 (July 1987), pp. 114–126. ISSN: 0268-6961. DOI: 10.1049/sej.1987.0015 (cit. on pp. 55, 81).

[KC07] B. Kitchenham and S. Charters. *Guidelines for performing Systematic Literature Reviews in Software Engineering*. Tech. rep. EBSE 2007-001. Keele University and Durham University Joint Report, 2007 (cit. on pp. 52, 97, 99).

[KMT06] B. Kitchenham, E. Mendes, and G. H. Travassos. "A Systematic Review of Cross- vs. Within-Company Cost Estimation Studies". In: *Proceedings of the 10th International Conference on Evaluation and Assessment in Software Engineering*. EASE'06. UK: British Computer Society, 2006, pp. 81–90 (cit. on p. 97).

[KLV13] P. Klint, D. Landman, and J. J. Vinju. "Exploring the Limits of Domain Model Recovery". In: *2013 IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, September 22-28, 2013*. IEEE Computer Society, Sept. 2013, pp. 120–129. DOI: 10.1109/ICSM.2013.23 (cit. on pp. 16, 21).

[KvdSV09] P. Klint, T. van der Storm, and J. J. Vinju. "RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation". In: *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2009, Edmonton, Alberta, Canada, September 20-21, 2009*. IEEE Computer Society, 2009, pp. 168–177. DOI: 10.1109/SCAM.2009.28 (cit. on pp. 13, 28, 63, 107).

[KDG07] A. Kuhn, S. Ducasse, and T. Gîrba. "Semantic clustering: Identifying topics in source code". In: *Information & Software Technology* 49.3 (2007), pp. 230–243. DOI: 10.1016/j.infsof.2006.10.017 (cit. on pp. 22, 42).

[KLN14] J. Kurs, M. Lungu, and O. Nierstrasz. "Bounded Seas - - Island Parsing Without Shipwrecks". In: *Software Language Engineering - 7th International Conference, SLE 2014, Västerås, Sweden, September 15-16, 2014. Proceedings*. Ed. by B. Combemale, D. J. Pearce, O. Barais, and J. J. Vinju. Vol. 8706. Lecture Notes in Computer Science. Springer, 2014, pp. 62–81. DOI: 10.1007/978-3-319-11245-9_4 (cit. on p. 105).

[Kvå85] T. O. Kvålseth. "Cautionary Note about R^2". In: *The American Statistician* 39.4 (1985), pp. 279–285. ISSN: 00031305 (cit. on pp. 60, 82).

[Lan13] D. Landman. *cwi-swat/project-planning-domain*. Apr. 2013. DOI: 10.5281/zenodo.208212 (cit. on pp. 17, 18, 24, 118).

[Lan15a] D. Landman. *A Curated Corpus of Java Source Code based on Sourcerer (2015)*. Feb. 2015. DOI: 10.5281/zenodo.208213 (cit. on pp. 17, 62).

[Lan15b] D. Landman. *A Large Corpus of C Source Code based on Gentoo packages*. Feb. 2015. DOI: 10.5281/zenodo.208215 (cit. on pp. 17, 63).

[Lan15c] D. Landman. *cwi-swat/jsep-sloc-versus-cc*. Feb. 2015. DOI: 10.5281/zenodo.293795 (cit. on pp. 18, 85).

[Lan16a] D. Landman. *A corpus of Java projects representing the 2012 Ohloh universe*. Mar. 2016. DOI: 10.5281/zenodo.162926 (cit. on pp. 18, 92, 105).

[Lan16b] D. Landman. *cwi-swat/static-analysis-reflection*. Oct. 2016. DOI: 10.5281/zenodo.163326 (cit. on pp. 18, 107).

[LSB+16] D. Landman, A. Serebrenik, E. Bouwers, and J. J. Vinju. "Empirical analysis of the relationship between CC and SLOC in a large corpus of Java methods and C functions". In: *Journal of Software: Evolution and Process* 28.7 (2016), pp. 589–618. DOI: 10.1002/smr.1760 (cit. on pp. 16, 49, 107).

[LSV14] D. Landman, A. Serebrenik, and J. J. Vinju. "Empirical Analysis of the Relationship between CC and SLOC in a Large Corpus of Java Methods". In: *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*. IEEE Computer Society, 2014, pp. 221–230. DOI: 10.1109/ICSME.2014.44 (cit. on pp. 16, 49).

[LSV17] D. Landman, A. Serebrenik, and J. J. Vinju. "Challenges for static analysis of Java reflection: literature review and empirical study". In: *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*. Ed. by S. Uchitel, A. Orso, and M. P. Robillard. IEEE, 2017, pp. 507–518. DOI: 10.1109/ICSE.2017.53 (cit. on pp. 17, 91).

[Leh80] M. Lehman. "Programs, life cycles, and laws of software evolution". In: *Proceedings of the IEEE* 68.9 (1980), pp. 1060–1076. DOI: 10.1109/proc.1980.11805 (cit. on p. 3).

[LSF03] T. . Lethbridge, J. Singer, and A. Forward. "How software engineers use documentation: the state of the practice". In: *IEEE Software* 20.6 (Nov. 2003), pp. 35–39. DOI: 10.1109/ms.2003.1241364 (cit. on p. 6).

[LH89] J. Lewis and S. Henry. "A methodology for integrating maintainability using software metrics". In: *Proceedings of Conference on Software Maintenance-1989*. IEEE, Oct. 1989, pp. 32–39. DOI: 10.1109/ICSM.1989.65191 (cit. on pp. 55, 80).

[LH03] O. Lhoták and L. J. Hendren. "Scaling Java Points-to Analysis Using SPARK". In: *Compiler Construction, 12th International Conference, CC 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*. Ed. by G. Hedin. Vol. 2622. Lecture Notes in Computer Science. Springer, 2003, pp. 153–169. DOI: 10.1007/3-540-36579-6_12 (cit. on pp. 99, 100).

[LH08] O. Lhoták and L. J. Hendren. "Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation". In: *ACM Transactions on Software Engineering and Methodology* 18.1 (2008). DOI: 10.1145/1391984.1391987 (cit. on p. 101).

[LC87] H. Li and W. Cheung. "An Empirical Study of Software Metrics". In: *IEEE Transactions on Software Engineering* SE-13.6 (June 1987), pp. 697–708. ISSN: 0098-5589. DOI: 10.1109/TSE.1987.233475 (cit. on pp. 54, 81).

[LBO+16] L. Li, T. F. Bissyandé, D. Octeau, and J. Klein. "DroidRA: taming reflection to support whole-program analysis of Android apps". In: *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*. Ed. by A. Zeller and A. Roychoudhury. ACM, 2016, pp. 318–329. DOI: 10.1145/2931037.2931044 (cit. on p. 112).

[LTS+14] Y. Li, T. Tan, Y. Sui, and J. Xue. "Self-inferencing Reflection Resolution for Java". In: *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*. Ed. by R. Jones. Vol. 8586. Lecture Notes in Computer Science. Springer, 2014, pp. 27–53. DOI: 10.1007/978-3-662-44202-9_2 (cit. on pp. 101, 102, 104).

[LTX15] Y. Li, T. Tan, and J. Xue. "Effective Soundness-Guided Reflection Analysis". In: *Static Analysis - 22nd International Symposium, SAS 2015, Saint-Malo, France, September 9-11, 2015, Proceedings*. Ed. by S. Blazy and T. Jensen. Vol. 9291. Lecture Notes in Computer Science. Springer, 2015, pp. 162–180. DOI: 10.1007/978-3-662-48288-9_10 (cit. on p. 101).

[LLL08] R. Lincke, J. Lundberg, and W. Löwe. "Comparing Software Metrics Tools". In: *Proceedings of the 2008 International Symposium on Software Testing and Analysis*. ISSTA '08. Seattle, WA, USA: ACM, 2008, pp. 131–142. ISBN: 978-1-60558-050-0. DOI: 10.1145/1390630.1390648 (cit. on p. 51).

[LV89] R. K. Lind and K. Vairavan. "An Experimental Investigation of Software Metrics and Their Relationship to Software Development Effort". In: *IEEE Transactions on Software Engineering* 15.5 (May 1989), pp. 649–653. ISSN: 0098-5589. DOI: 10.1109/32.24715 (cit. on pp. 55, 80).

[LBN+09] E. Linstead, S. K. Bajracharya, T. C. Ngo, P. Rigor, C. V. Lopes, and P. Baldi. "Sourcerer: mining and searching internet-scale software repositories". In: *Data Mining and Knowledge Discovery* 18.2 (2009), pp. 300–336. DOI: 10.1007/s10618-008-0118-x (cit. on pp. 15, 17, 50, 61).

[LRB+07] E. Linstead, P. Rigor, S. K. Bajracharya, C. V. Lopes, and P. Baldi. "Mining concepts from code with probabilistic topic models". In: *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*. Ed. by R. E. K. Stirewalt, A. Egyed, and B. Fischer. ACM, 2007, pp. 461–464. DOI: 10.1145/1321631.1321709 (cit. on pp. 22, 42).

[LSS+15] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis. "In Defense of Soundiness: A Manifesto". In: *Communications of the ACM* 58.2 (Jan. 2015), pp. 44–46. ISSN: 0001-0782. DOI: 10.1145/2644805 (cit. on pp. 92, 120).

[LWL05] V. B. Livshits, J. Whaley, and M. S. Lam. "Reflection Analysis for Java". In: *Programming Languages and Systems, Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005, Proceedings*. Ed. by K. Yi. Vol. 3780. Lecture Notes in Computer Science. Springer, 2005, pp. 139–160. DOI: 10.1007/11575467_11 (cit. on pp. 13, 92, 101, 102, 104, 120).

[Loe90] C. Loehle. "Proper Statistical Treatment of Species-Area Data". In: *Oikos* 57.1 (1990), pp. 143–145. ISSN: 00301299 (cit. on pp. 60, 82).

[MHL+10] Y.-T. Ma, K.-Q. He, B. Li, J. Liu, and X.-Y. Zhou. "A Hybrid Set of Complexity Metrics for Large-Scale Object-Oriented Software Systems". In: *Journal of Computer Science and Technology* 25.6 (2010), pp. 1184–1201. ISSN: 1000-9000. DOI: 10.1007/s11390-010-9398-x (cit. on pp. 57, 81).

[MS11] R. Malhotra and Y. Singh. "On the Applicability of Machine Learning Techniques for Object Oriented Software Fault Prediction". In: *Software Engineering: An International Journal* 1 (1 2011), pp. 24–37 (cit. on pp. 53, 54, 57, 81).

[Man98] W. G. Manning. "The logged dependent variable, heteroscedasticity, and the retransformation problem". In: *Journal of Health Economics* 17.3 (1998), pp. 283–295. ISSN: 0167-6296. DOI: 10.1016/S0167-6296(98)00025-3 (cit. on p. 82).

[MPY+05] C. L. Martín, J. L. Pasquier, C. Yáñez-Márquez, and A. Gutierrez-Tornes. "Software Development Effort Estimation Using Fuzzy Logic: A Case Study". In: *Sixth Mexican International Conference on Computer Science (ENC 2005), 26-30 September 2005, Puebla, Mexico*. IEEE Computer Society, 2005, pp. 113–120. DOI: 10.1109/ENC.2005.47 (cit. on pp. 56, 81).

[McC76] T. J. McCabe. "A Complexity Measure". In: *IEEE Transactions Software Engineering* 2.4 (1976), pp. 308–320 (cit. on pp. 10, 51, 52, 70, 85).

[MT04] T. Mens and T. Tourwe. "A survey of software refactoring". In: *IEEE Transactions on Software Engineering* 30.2 (Feb. 2004), pp. 126–139. DOI: 10.1109/tse.2004.1265817 (cit. on p. 13).

[MHS05] M. Mernik, J. Heering, and A. M. Sloane. "When and how to develop domain-specific languages". In: *ACM Computing Surveys* 37.4 (2005), pp. 316–344. DOI: 10.1145/1118890.1118892 (cit. on p. 21).

[Moo01] L. Moonen. "Generating Robust Parsers Using Island Grammars". In: *Proceedings of the Eighth Working Conference on Reverse Engineering, WCRE'01, Stuttgart, Germany, October 2-5, 2001*. Ed. by E. Burd, P. Aiken, and R. Koschke. IEEE Computer Society, 2001, p. 13. DOI: 10.1109/WCRE.2001.957806 (cit. on p. 105).

[Moo98] T. T. Moores. "Applying complexity measures to rule-based Prolog programs". In: *Journal of Systems and Software* 44.1 (1998), pp. 45–52. DOI: 10.1016/S0164-1212(98)10042-0 (cit. on p. 85).

[MAL+13] K. Mordal, N. Anquetil, J. Laval, A. Serebrenik, B. Vasilescu, and S. Ducasse. "Software quality metrics aggregation in industry". In: *Journal of Software: Evolution and Process* 25.10 (2013), pp. 1117–1135. DOI: 10.1002/smr.1558 (cit. on pp. 58, 81).

[MK93] J. C. Munson and T. M. Kohshgoftaar. "Measurement of data structure complexity". In: *Journal of Systems and Software* 20.3 (1993), pp. 217–225. ISSN: 0164-1212. DOI: 10.1016/0164-1212(93)90065-6 (cit. on p. 70).

[Mye90] B. A. Myers. "Taxonomies of visual programming and program visualization". In: *Journal of Visual Languages & Computing* 1.1 (1990), pp. 97–123. DOI: 10.1016/S1045-926X(05)80036-9 (cit. on p. 4).

[Mye77] G. J. Myers. "An Extension to the Cyclomatic Measure of Program Complexity". In: *SIGPLAN Notices* 12.10 (Oct. 1977), pp. 61–64. ISSN: 0362-1340. DOI: 10.1145/954627.954633 (cit. on pp. 10, 51).

[NZB13] M. Nagappan, T. Zimmermann, and C. Bird. "Diversity in Software Engineering Research". In: *ESEC/FSE*. Saint Petersburg, Russia: ACM, 2013, pp. 466–476. ISBN: 978-1-4503-2237-9. DOI: 10.1145/2491411.2491415 (cit. on p. 104).

[OLD+15] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. D. McDaniel. "Composite Constant Propagation: Application to Android Inter-Component Communication Analysis". In: *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*. Ed. by A. Bertolino, G. Canfora, and S. G. Elbaum. IEEE Computer Society, 2015, pp. 77–88. DOI: 10.1109/ICSE.2015.30 (cit. on p. 112).

[OWK03] D. Ohst, M. Welle, and U. Kelter. "Differences between versions of UML diagrams". In: *SIGSOFT Software Engineering Notes* 28.5 (Sept. 2003), pp. 227–236. ISSN: 0163-5948. DOI: 10.1145/949952.940102 (cit. on p. 40).

[ONe93] M. B. O'Neal. "An Empirical Study of Three Common Software Complexity Measures". In: *Proceedings of the 1993 ACM/SIGAPP Symposium on Applied Computing: States of the Art and Practice*. SAC '93. Indianapolis, Indiana, USA: ACM, 1993, pp. 203–207. ISBN: 0-89791-567-4. DOI: 10.1145/162754.162867 (cit. on pp. 55, 80).

[Pai80] M. Paige. "A metric for software test planning". In: *Conference Proceedings of COMPSAC 80*. IEEE, Oct. 1980, pp. 499–504 (cit. on pp. 54, 81).

[Par92] R. E. Park. *Software Size Measurement: A Framework for Counting Source Statements*. Tech. rep. CMU/SEI-92-TR-020. Software Engineering Institute, Sept. 1992 (cit. on pp. 9, 10).

[PBM13] C. Parnin, C. Bird, and E. R. Murphy-Hill. "Adoption and use of Java generics". In: *Empirical Software Engineering* 18.6 (2013), pp. 1047–1089. DOI: 10.1007/s10664-012-9236-6 (cit. on p. 112).

[Pea95] K. Pearson. "Note on regression and inheritance in the case of two parents". In: *Proceedings of the Royal Society of London* 58 (1895), pp. 240–242 (cit. on p. 66).

[Pee+15] J. Peek et al. *Linguist: Language Savant*. 2015. URL: https://github.com/github/linguist (visited on 02/10/2015) (cit. on p. 63).

[PPV00] D. E. Perry, A. A. Porter, and L. G. Votta. "Empirical Studies of Software Engineering: A Roadmap". In: *Proceedings of the Conference on The Future of Software Engineering*. ICSE '00. Limerick, Ireland: ACM, 2000, pp. 345–355. ISBN: 1-58113-253-0. DOI: 10.1145/336512.336586 (cit. on p. 84).

[PSR+05] I. Philippow, D. Streitferdt, M. Riebisch, and S. Naumann. "An approach for reverse engineering of design patterns". In: *Software & Systems Modeling* 4.1 (Feb. 2005), pp. 55–70. DOI: 10.1007/s10270-004-0059-9 (cit. on p. 8).

[PTF+15] G. Pinto, W. Torres, B. Fernandes, F. C. Filho, and R. S. M. de Barros. "A large-scale study on the usage of Java's concurrent programming constructs". In: *Journal of Systems and Software* 106 (2015), pp. 59–81. DOI: 10.1016/j.jss.2015.04.064 (cit. on p. 112).

[PFD11] D. Posnett, V. Filkov, and P. T. Devanbu. "Ecological inference in empirical software engineering". In: *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. Ed. by P. Alexander, C. S. Pasareanu, and J. G. Hosking. Lawrence, KS, USA: IEEE Computer Society, Nov. 2011, pp. 362–371. DOI: 10.1109/ASE.2011.6100074 (cit. on p. 58).

[RPH+13] F. Rahman, D. Posnett, I. Herraiz, and P. Devanbu. "Sample Size vs. Bias in Defect Prediction". In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2013. Saint Petersburg, Russia: ACM, 2013, pp. 147–157. ISBN: 978-1-4503-2237-9. DOI: 10.1145/2491411.2491418 (cit. on p. 85).

[RW02] V. Rajlich and N. Wilde. "The Role of Concepts in Program Comprehension". In: *10th International Workshop on Program Comprehension (IWPC 2002), 27-29 June 2002, Paris, France*. IEEE Computer Society, 2002, pp. 271–278. DOI: 10.1109/WPC.2002.1021348 (cit. on p. 22).

[RAM+16] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden. "Harvesting Runtime Values in Android Applications That Feature Anti-Analysis Techniques". In: *23nd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society, 2016 (cit. on p. 112).

[RFJ08] D. Ratiu, M. Feilkas, and J. Jürjens. "Extracting Domain Ontologies from Domain Specific APIs". In: *12th European Conference on Software Maintenance and Reengineering, CSMR 2008, April 1-4, 2008, Athens, Greece*. IEEE Computer Society, 2008, pp. 203–212. DOI: 10.1109/CSMR.2008.4493315 (cit. on pp. 22, 42, 121).

[RCT+14] T. Ravitch, E. R. Creswick, A. Tomb, A. Foltzer, T. Elliott, and L. Casburn. "Multi-App Security Analysis with FUSE: Statically Detecting Android App Collusion". In: *Proceedings of the 4th Program Protection and Reverse Engineering Workshop, PPREW@ACSAC 2014, New Orleans, LA, USA, December 9, 2014*. Ed. by M. D. Preda and J. T. McDonald. ACM, 2014, 4:1–4:10. DOI: 10.1145/2689702.2689705 (cit. on p. 101).

[RPF+14] B. Ray, D. Posnett, V. Filkov, and P. Devanbu. "A Large Scale Study of Programming Languages and Code Quality in Github". In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York, NY, USA: ACM, 2014, pp. 155–165. ISBN: 978-1-4503-3056-5. DOI: 10.1145/2635868.2635922 (cit. on p. 87).

[RHB+11] G. Richards, C. Hammer, B. Burg, and J. Vitek. "The Eval That Men Do - A Large-Scale Study of the Use of Eval in JavaScript Applications". In: *ECOOP 2011 - Object-Oriented Programming - 25th European Conference, Lancaster, UK, July 25-29, 2011 Proceedings*. Ed. by M. Mezini. Vol. 6813. Lecture Notes in Computer Science. Springer, 2011, pp. 52–78. DOI: 10.1007/978-3-642-22655-7_4 (cit. on p. 111).

[RLB+10] G. Richards, S. Lebresne, B. Burg, and J. Vitek. "An analysis of the dynamic behavior of JavaScript programs". In: *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010*. Ed. by B. G. Zorn and A. Aiken. Toronto, Ontario, Canada: ACM, June 2010, pp. 1–12. DOI: 10.1145/1806596.1806598 (cit. on p. 111).

[RJB99] J. Rumbaugh, I. Jacobson, and G. Booch. "State Machine View". In: *The Unified Modeling Language Reference Manual*. Addison-Wesley Professional, 1999. Chap. 6, pp. 67–80. ISBN: 020130998X (cit. on p. 6).

[STC00] A. Saltelli, S. Tarantola, and F. Campolongo. "Sensitivity analysis as an ingredient of modeling". In: *Statistical Science* 15.4 (2000), pp. 377–395 (cit. on p. 86).

[SR07] J. Sawin and A. Rountev. "Improved Static Resolution of Dynamic Class Loading in Java". In: *Seventh IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2007), September 30 - October 1, 2007, Paris, France*. IEEE Computer Society, 2007, pp. 143–154. DOI: 10.1109/SCAM.2007.24 (cit. on pp. 101, 102).

[SR09] J. Sawin and A. Rountev. "Improving static resolution of dynamic class loading in Java using dynamically gathered environment information". In: *Automated Software Engineering* 16.2 (June 2009), pp. 357–381. DOI: 10.1007/s10515-009-0049-9 (cit. on pp. 101, 102, 108).

[SR11] J. Sawin and A. Rountev. "Assumption Hierarchy for a CHA Call Graph Construction Algorithm". In: *11th IEEE Working Conference on Source Code Analysis and Manipulation, SCAM 2011, Williamsburg, VA, USA, September 25-26, 2011*. IEEE Computer Society, 2011, pp. 35–44. DOI: 10.1109/SCAM.2011.20 (cit. on p. 99).

[SSD+13] M. Schäfer, M. Sridharan, J. Dolby, and F. Tip. "Dynamic Determinacy Analysis". In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '13. Seattle, Washington, USA: ACM, 2013, pp. 165–174. ISBN: 978-1-4503-2014-6. DOI: 10.1145/2491956.2462168 (cit. on p. 113).

[Sch06] N. Schneidewind. "Software reliability engineering process". In: *Innovations in Systems and Software Engineering* 2.3-4 (2006), pp. 179–190. ISSN: 1614-5046. DOI: 10.1007/s11334-006-0007-7 (cit. on pp. 56, 81).

[SLB+11] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki. "Reverse engineering feature models". In: *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*. Ed. by R. N. Taylor, H. C. Gall, and N. Medvidovic. ACM, 2011, pp. 461–470. DOI: 10.1145/1985793.1985856 (cit. on p. 43).

[SCM+79] S. B. Sheppard, B. Curtis, P. Milliman, M. A. Borst, and T. Love. "First-year results from a research program on human factors in software engineering". In: *American Federation of Information Processing Societies (AFIPS) Conference Proceedings*. Vol. 48. New York, NY, USA: AFIPS Press, June 1979, pp. 1021–1027. DOI: 10.1109/AFIPS.1979.59 (cit. on pp. 8, 49, 54, 81).

[She88] M. Shepperd. "A Critique of Cyclomatic Complexity As a Software Metric". In: *Software Engineering Journal* 3.2 (Mar. 1988), pp. 30–36. DOI: 10.1049/sej.1988.0003 (cit. on pp. 51, 52, 54, 58).

[She07] D. J. Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures*. 4th ed. Chapman & Hall/CRC, 2007. ISBN: 9781584888147 (cit. on pp. 60, 67).

[SCV+08] F. J. Shull, J. C. Carver, S. Vegas, and N. Juristo. "The role of replications in Empirical Software Engineering". In: *Empirical Software Engineering* 13.2 (2008), pp. 211–218. ISSN: 1382-3256. DOI: 10.1007/s10664-008-9060-1 (cit. on p. 87).

[SSA15] J. Siegmund, N. Siegmund, and S. Apel. "Views on Internal and External Validity in Empirical Software Engineering". In: *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*. Ed. by A. Bertolino, G. Canfora, and S. G. Elbaum. IEEE Computer Society, 2015, pp. 9–19. DOI: 10.1109/ICSE.2015.24 (cit. on p. 87).

[SB15] Y. Smaragdakis and G. Balatsouras. "Pointer Analysis". In: *Foundations and Trends in Programming Languages* 2.1 (2015), pp. 1–69. ISSN: 2325-1107. DOI: 10.1561/2500000014 (cit. on p. 101).

[SBK+15] Y. Smaragdakis, G. Balatsouras, G. Kastrinis, and M. Bravenboer. "More Sound Static Handling of Java Reflection". In: *Programming Languages and Systems - 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30 - December 2, 2015, Proceedings*. Ed. by X. Feng and S. Park. Vol. 9458. Lecture Notes in Computer Science. Springer, 2015, pp. 485–503. DOI: 10.1007/978-3-319-26529-2_26 (cit. on pp. 101, 113).

[Spe04] C. Spearman. "The Proof and Measurement of Association between Two Things". In: *The American Journal of Psychology* 15.1 (1904), pp. 72–101. ISSN: 00029556 (cit. on p. 67).

[SAP+11] M. Sridharan, S. Artzi, M. Pistoia, S. Guarnieri, O. Tripp, and R. Berg. "F4F: taint analysis of framework-based web applications". In: *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*. Ed. by C. V. Lopes and K. Fisher. ACM, 2011, pp. 1053–1068. DOI: 10.1145/2048066.2048145 (cit. on p. 100).

[SDC+12] M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip. "Correlation Tracking for Points-To Analysis of JavaScript". In: *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings*. Ed. by J. Noble. Vol. 7313. Lecture Notes in Computer Science. Springer, 2012, pp. 435–458. DOI: 10.1007/978-3-642-31057-7_20 (cit. on p. 112).

[SBV01] G. Succi, L. Benedicenti, and T. Vernazza. "Analysis of the effects of software reuse on customer satisfaction in an RPG environment". In: *IEEE Transactions on Software Engineering* 27.5 (May 2001), pp. 473–479. ISSN: 0098-5589. DOI: 10.1109/32.922717 (cit. on pp. 56, 81).

[STU+81] T. Sunohara, A. Takano, K. Uehara, and T. Ohkawa. "Program Complexity Measure for Software Development Management". In: *Proceedings of the 5th International Conference on Software Engineering, San Diego, California, USA, March 9-12, 1981*. Ed. by S. Jeffrey and L. G. Stucki. IEEE Computer Society, 1981, pp. 100–106 (cit. on pp. 54, 81).

[SM05] A. Sutton and J. I. Maletic. "Mappings for Accurately Reverse Engineering UML Class Models from C++". In: *12th Working Conference on Reverse Engineering, WCRE 2005, Pittsburgh, PA, USA, November 7-11, 2005*. IEEE Computer Society, 2005, pp. 175–184. DOI: 10.1109/WCRE.2005.21 (cit. on p. 42).

[TAA14] Y. Tashtoush, M. Al-Maolegi, and B. Arkok. "The Correlation among Software Complexity Metrics with Case Study". In: *International Journal of Advanced Computer Research* 4.2 (15 2014), pp. 414–419 (cit. on pp. 57, 81).

[TAD+10] E. D. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. "The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies". In: *17th Asia Pacific Software Engineering Conference, APSEC 2010, Sydney, Australia, November 30 - December 3, 2010*. Ed. by J. Han and T. D. Thu. IEEE Computer Society, 2010, pp. 336–345. DOI: 10.1109/APSEC.2010.46 (cit. on p. 103).

[TB12] A. Thies and E. Bodden. "RefaFlex: safer refactorings for reflective Java programs". In: *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*. Ed. by M. P. E. Heimdahl and Z. Su. ACM, 2012, pp. 1–11. DOI: 10.1145/2338965.2336754 (cit. on p. 100).

[Tip95] F. Tip. "A survey of program slicing techniques". In: *Journal of Programming Languages* 3.3 (1995), pp. 121–189 (cit. on p. 4).

[TLS+99] F. Tip, C. Laffra, P. F. Sweeney, and D. Streeter. "Practical Experience with an Application Extractor for Java". In: *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '99), Denver, Colorado, USA, November 1-5, 1999*. Ed. by B. Hailpern, L. M. Northrop, and A. M. Berman. ACM, 1999, pp. 292–305. DOI: 10.1145/320384.320414 (cit. on p. 100).

[TSL+02] F. Tip, P. F. Sweeney, C. Laffra, A. Eisma, and D. Streeter. "Practical extraction techniques for Java". In: *ACM Transactions on Programming Languages and Systems* 24.6 (2002), pp. 625–666. DOI: 10.1145/586088.586090 (cit. on p. 100).

[TTB+07] P. Tonella, M. Torchiano, B. D. Bois, and T. Systä. "Empirical studies in reverse engineering: state of the art and future trends". In: *Empirical Software Engineering* 12.5 (2007), pp. 551–571. DOI: 10.1007/s10664-007-9037-5 (cit. on pp. 3, 4).

[TPF+09] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. "TAJ: effective taint analysis of web applications". In: *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*. Ed. by M. Hind and A. Diwan. ACM, 2009, pp. 87–97. DOI: 10.1145/1542476.1542486 (cit. on p. 99).

[TT95] J. Troster and J. Tian. "Measurement and Defect Modeling for a Legacy Software System". In: *Annals of Software Engineering* 1 (1995), pp. 95–118. DOI: 10.1007/BF02249047 (cit. on p. 60).

[TM94] R. J. Turver and M. Munro. "An early impact analysis technique for software maintenance". In: *Journal of Software Maintenance* 6.1 (1994), pp. 35–52. DOI: 10.1002/smr.4360060104 (cit. on p. 4).

[VCG+99] R. Vallée-Rai, P. Co, E. Gagnon, L. J. Hendren, P. Lam, and V. Sundaresan. "Soot - a Java bytecode optimization framework". In: *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative Research*. Ed. by S. A. MacKay and J. H. Johnson. Mississauga, Ontario, Canada: IBM, Nov. 1999, p. 13. DOI: 10.1145/781995.782008 (cit. on pp. 99, 100).

[van95] K. G. van den Berg. "Software Measurement and Functional Programming". PhD thesis. University of Twente, Enschede, the Netherlands, June 1995. ISBN: 9090082514 (cit. on p. 85).

[vdMR07] M. J. van der Meulen and M. A. Revilla. "Correlations Between Internal Software Metrics and Software Dependability in a Large Population of Small C/C++ Programs". In: *Proceedings of the The 18th IEEE International Symposium on Software Reliability*. ISSRE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 203–208. ISBN: 0-7695-3024-9. DOI: 10.1109/ISSRE.2007.6 (cit. on pp. 56, 61, 75, 81).

[vDKV99] A. van Deursen, P. Klint, and C. Verhoef. "Research Issues in the Renovation of Legacy Systems". In: *Fundamental Approaches to Software Engineering, Second Internationsl Conference, FASE'99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings*. Ed. by J. Finance. Vol. 1577. Lecture Notes in Computer Science. Springer, 1999, pp. 1–21. DOI: 10.1007/978-3-540-49020-3_1 (cit. on pp. 3, 6).

[VSvdB11a] B. Vasilescu, A. Serebrenik, and M. G. J. van den Brand. "By No Means: A Study on Aggregating Software Metrics". In: *2nd International Workshop on Emerging Trends in Software Metrics*. WETSoM. ACM, 2011, pp. 23–26 (cit. on pp. 58, 81).

[VSvdB11b] B. Vasilescu, A. Serebrenik, and M. G. J. van den Brand. "You can't control the unfamiliar: A study on the relations between aggregation techniques for software metrics". In: *IEEE 27th International Conference on Software Maintenance, ICSM2011*. Sept. 2011, pp. 313–322. DOI: 10.1109/ICSM.2011.6080798 (cit. on pp. 58, 81).

[VG12] J. J. Vinju and M. W. Godfrey. "What Does Control Flow Really Look Like? Eyeballing the Cyclomatic Complexity Metric". In: *12th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2012*. Riva del Garda, Italy: IEEE Computer Society, Sept. 2012, pp. 154–163. DOI: 10.1109/SCAM.2012.17 (cit. on pp. 49, 83).

[vMV95] A. von Mayrhauser and A. M. Vans. "Program Comprehension During Software Maintenance and Evolution". In: *IEEE Computer* 28.8 (1995), pp. 44–55. DOI: 10.1109/2.402076 (cit. on p. 49).

[WS07] K. Wang and W. Shen. "Improving the Accuracy of UML Class Model Recovery". In: *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*. 2007, pp. 387–390 (cit. on p. 42).

[WW02] J. Webster and R. T. Watson. "Analyzing the Past to Prepare for the Future: Writing a Literature Review". In: *Management Information Systems (MIS) Quarterly* 26.2 (June 2002), pp. xiii–xxiii. ISSN: 0276-7783 (cit. on p. 97).

[Wei79] M. Weiser. "Program slices: formal, psychological, and practical investigations of an automatic program abstraction method". PhD thesis. University of Michigan, 1979 (cit. on p. 4).

[WKO+14] E. R. Wognsen, H. S. Karlsen, M. C. Olesen, and R. R. Hansen. "Formalisation and analysis of Dalvik bytecode". In: *Science of Computer Programming* 92 (2014), pp. 25–55. DOI: 10.1016/j.scico.2013.11.037 (cit. on pp. 101, 104).

[Woh14] C. Wohlin. "Guidelines for snowballing in systematic literature studies and a replication in software engineering". In: *18th International Conference on Evaluation and Assessment in Software Engineering, EASE*. ACM, 2014, 38:1–38:10. DOI: 10.1145/2601248.2601268 (cit. on pp. 52, 97).

[WRH+12] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering*. Springer Nature, 2012. DOI: 10.1007/978-3-642-29044-2 (cit. on p. 14).

[WHH79] M. R. Woodward, M. A. Hennell, and D. Hedley. "A Measure of Control Flow Complexity in Program Text". In: *IEEE Transactions on Software Engineering* 5.1 (Jan. 1979), pp. 45–50. ISSN: 0098-5589. DOI: 10.1109/TSE.1979.226497 (cit. on pp. 54, 80).

[YXA+15] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck. "AppContext: Differentiating Malicious and Benign Mobile App Behaviors Using Context". In: *ICSE*. Florence, Italy: IEEE, 2015, pp. 303–313. ISBN: 978-1-4799-1934-5 (cit. on p. 99).

[YM13] L. Yu and A. Mishra. "An Empirical Study of Lehman's Law on Software Quality Evolution". In: *International Journal of Software & Informatics* 7.3 (2013), pp. 469–481 (cit. on p. 58).

[ZHM+16] F. Zhang, A. E. Hassan, S. McIntosh, and Y. Zou. "The Use of Summation to Aggregate Software Metrics Hinders the Performance of Defect Prediction Models". In: *IEEE Transactions on Software Engineering* (2016), pp. 1–1. DOI: 10.1109/tse.2016.2599161 (cit. on p. 119).

[ZAG+15] Y. Zhauniarovich, M. Ahmad, O. Gadyatskaya, B. Crispo, and F. Massacci. "StaDynA: Addressing the Problem of Dynamic Code Updates in the Security Analysis of Android Applications". In: *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy, CODASPY 2015, San Antonio, TX, USA, March 2-4, 2015*. Ed. by J. Park and A. C. Squicciarini. ACM, 2015, pp. 37–48. DOI: 10.1145/2699026.2699105 (cit. on pp. 100, 104).

SUMMARY

Reverse Engineering Source Code:
Empirical Studies of Limitations and Opportunities

The goal of software renovation is to modernize software. One way to achieve this is to first reverse engineer the essential concepts and abstractions used in the software and then use these during renovation. Reverse engineering can use several sources: users, documentation, or source code. We have focused on reverse engineering from source code. Scaling reverse engineering to large software systems requires at the very least partially automated analysis. Automation often comes at the cost of over- or underapproximation. We have formulated three research questions to explore limits and opportunities for these approximations.

To answer the first question, we have explored the limits of domain model recovery by manually recovering two domain models from two software systems. Comparing these models to a manually constructed reference domain model based on a reference book of the domain and two manually constructed reference applications models we found that most domain information could be recovered – with high quality – by reading the source code of the software system. This motivates future work in automating the domain model recovery from source code.

In trying to automate domain model recovery, we have identified challenges that hold for a wider range of reverse engineering methods than just domain model recovery. The second and third question address these challenges in the broader context of reverse engineering.

To answer the second question, we have explored the opportunity of using both Cyclomatic Complexity (CC) and Source Lines of Code (SLOC) for automating reverse engineering. Metrics, such as CC and SLOC, are used in a wide variety of reverse-engineering methods to filter methods or files of interest. Almost all of the literature on the relation between the two metrics – identified using a Systematic Literature Review (SLR) – claim a strong linear correlation between them (R^2 between 0.51 and 0.96). This is often interpreted as indication that CC and SLOC measured the same property. Often this is further interpreted that measuring CC and SLOC next to each other is redundant. In two large corpora – with 362 MSLOC of Java and 186 MSLOC of C – we did not observe a strong correlation (R^2 of 0.40 and 0.44). We have identified two transformations of the data that did increase the correlations to the more commonly reported strengths. However these transformations complicate the interpretation of the relationship between CC and SLOC. Our final interpretation is that there is a lack evidence for CC being redundant to SLOC, which supports the continued used of both metrics next to each other.

In order to answer the final question, we have explored the limits of statically analyzing Java – with respect to the Reflection Application Programming Interface (API) – for a corpus of 462 Java projects (80 MSLOC). Using a SLR of all static analysis approaches – that published new heuristics for handling reflection – we have identified the common assumptions and limitations. Analyzing the corpus revealed that 78% of all projects use the parts of the Reflection API that are hard to model with static analysis. Common challenges for analysis tools such as "non-exceptional exceptions", "programmatic filtering meta objects", "semantics of collections", and "dynamic proxies" widely occur in the corpus. We support Java software engineers with tactics to obtain more easy to analyze reflection code. We also propose new opportunities for static analysis tools to significantly impact the analyses of real Java code.

All three results have been obtained with empirical studies on corpora of open source software. The corpora and the scripts used to analyze them are available online to support critique from other researchers and enable future work on different challenges with the same corpora. We have used empirical studies to both answer open questions and identify new opportunities in reverse engineering research and practice.

SAMENVATTING

Reverse Engineering Source Code:
Empirical Studies of Limitations and Opportunities

Bestaande software kan gemoderniseerd worden door deze te renoveren. Softwarerenovatie vereist kennis over de software – zoals concepten en abstracties – die vaak niet meer expliciet beschikbaar zijn. Door middel van reverse engineering kunnen dit soort interne aspecten weer zichtbaar gemaakt worden. Hiervoor staan allerlei bronnen ter beschikking: gebruikers, documentatie, of broncode. Wij richten ons in dit proefschrift op het reverse engineeren van kennis uit broncode.

Het reverse engineeren van grotere softwaresystemen vereist automatisering. Echter, automatisering levert slechts een benadering van de gewenste kennis op. Wij formuleren drie onderzoeksvragen die de grenzen en mogelijkheden daarvan verkennen.

Voor de eerste vraag zoeken we de grenzen op van het reverse engineeren van een domeinmodel uit broncode. Een domeinmodel is een beschrijving van de concepten in een domein en de relatie daartussen. We maken eerst met de hand twee domeinmodellen voor twee softwaresystemen in het domein van projectplanning. Bovendien stellen we uitgaande van een handboek uit het domein van projectplanning handmatig een domeinmodel op dat als referentiemodel dient. We hebben de twee uit de code geëxtraheerde domeinmodellen vervolgens vergeleken met het referentiemodel. Daarnaast vergelijken we dezelfde modellen met domeinmodellen verkregen uit de grafische gebruikersomgeving van de betreffende softwaresystemen. Conclusie: de meeste domeinkennis kan verkregen worden uit de broncode zelf. Dit motiveert nader onderzoek naar het automatiseren van broncode gedreven reverse engineering van domeinmodellen.

Tijdens het automatiseren van het reverse engineeren van domeinmodellen, ontdekten wij nieuwe uitdagingen die voor een veel breder scala aan benaderingen van reverse engineering gelden. De tweede en derde vraag richten zich daarom op deze uitdagingen in de bredere context van reverse engineering.

Voor de tweede vraag onderzoeken we de mogelijkheid om de Cyclomatic Complexity (CC) en Source Lines of Code (SLOC) metrieken naast elkaar te gebruiken voor het automatiseren van reverse engineering. Metrieken zoals CC en SLOC proberen moeilijk meetbare eigenschappen van broncode te benaderen en worden zeer veel gebruikt. Om, bijvoorbeeld, interessante methodes of bestanden te filteren uit een grotere verzameling. De meeste literatuur – geïdentificeerd met een Systematic Literature Review (SLR) – beweert echter dat deze twee metrieken sterk met elkaar lineair correleren (R^2 tussen 0.51 en 0.96). Deze correlatie wordt vaak geïnterpreteerd als

bewijs dat ze aan elkaar gelijk zijn. In twee grote corpora – bestaand uit 362 MSLOC Java en 186 MSLOC C code – observeren wij deze sterke correlatie niet (R^2 van 0.40 en 0.44). Wegens gebrek aan bewijs dat CC redundant is ten opzichte van SLOC, concluderen wij dat het wel degelijk zin heeft om CC en SLOC naast elkaar te blijven gebruiken.

Voor de derde en laatste vraag onderzoeken wij de grenzen van het statisch analyseren van Java broncode. Vooral de Reflectie Application Programming Interface (API) is moeilijk te analyseren. Deze API staat een programma toe zijn eigen interne structuur te bekijken en te wijzigen. Door middel van een SLR maken wij eerst een overzicht van alle benaderingen voor statische analyse die nieuwe heuristieken voor reflectieve code geïntroduceerd hebben. Op basis van dit overzicht inventariseren wij veelvoorkomende aannames en beperkingen. In een door ons geconstrueerd corpus van Java software gebruikt 78% van alle projecten moeilijk te analyseren onderdelen van de Reflection API. Daarnaast kwamen tegenvoorbeelden van de volgende aannames en beperkingen veel voor: "niet exceptionele excepties", "programmatische filtering van meta objecten", "semantiek van verzamelingen", en "dynamische proxies". Voor Java software ingenieurs geven wij strategieën om hun reflectieve code makkelijker analyseerbaar te maken. Voor ontwikkelaars van tools voor statische analyse presenteren wij mogelijkheden om de analyse van reflectieve Java code te verbeteren.

Om tot deze drie resultaten te komen hebben wij gebruik gemaakt van empirisch onderzoek op grote corpora van open source software. Deze corpora en bijhorende analyse-code stellen wij online beschikbaar; hiermee kunnen andere onderzoekers onze resultaten repliceren en bekritiseren. Bovendien zijn onze corpora te gebruiken in nieuw onderzoek. Tenslotte hebben wij gebruik gemaakt van empirische studies voor het beantwoorden van open vragen en identificeren van nieuwe mogelijkheden voor het onderzoek in en de praktijk van de reverse engineering.

J. van den Bos. *Gathering Evidence: Model-Driven Software Engineering in Automated Digital Forensics*. Faculty of Science, UvA. 2014-01

D. Hadziosmanovic. *The Process Matters: Cyber Security in Industrial Control Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-02

A.J.P. Jeckmans. *Cryptographically-Enhanced Privacy for Recommender Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-03

C.-P. Bezemer. *Performance Optimization of Multi-Tenant Software Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2014-04

T.M. Ngo. *Qualitative and Quantitative Information Flow Analysis for Multi-threaded Programs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-05

A.W. Laarman. *Scalable Multi-Core Model Checking*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-06

J. Winter. *Coalgebraic Characterizations of Automata-Theoretic Classes*. Faculty of Science, Mathematics and Computer Science, RU. 2014-07

W. Meulemans. *Similarity Measures and Algorithms for Cartographic Schematization*. Faculty of Mathematics and Computer Science, TU/e. 2014-08

A.F.E. Belinfante. *JTorX: Exploring Model-Based Testing*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-09

A.P. van der Meer. *Domain Specific Languages and their Type Systems*. Faculty of Mathematics and Computer Science, TU/e. 2014-10

B.N. Vasilescu. *Social Aspects of Collaboration in Online Software Communities*. Faculty of Mathematics and Computer Science, TU/e. 2014-11

F.D. Aarts. *Tomte: Bridging the Gap between Active Learning and Real-World Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2014-12

N. Noroozi. *Improving Input-Output Conformance Testing Theories*. Faculty of Mathematics and Computer Science, TU/e. 2014-13

M. Helvensteijn. *Abstract Delta Modeling: Software Product Lines and Beyond*. Faculty of Mathematics and Natural Sciences, UL. 2014-14

P. Vullers. *Efficient Implementations of Attribute-based Credentials on Smart Cards*. Faculty of Science, Mathematics and Computer Science, RU. 2014-15

F.W. Takes. *Algorithms for Analyzing and Mining Real-World Graphs*. Faculty of Mathematics and Natural Sciences, UL. 2014-16

M.P. Schraagen. *Aspects of Record Linkage*. Faculty of Mathematics and Natural Sciences, UL. 2014-17

G. Alpár. *Attribute-Based Identity Management: Bridging the Cryptographic Design of ABCs with the Real World*. Faculty of Science, Mathematics and Computer Science, RU. 2015-01

A.J. van der Ploeg. *Efficient Abstractions for Visualization and Interaction*. Faculty of Science, UvA. 2015-02

R.J.M. Theunissen. *Supervisory Control in Health Care Systems*. Faculty of Mechanical Engineering, TU/e. 2015-03

T.V. Bui. *A Software Architecture for Body Area Sensor Networks: Flexibility and Trustworthiness*. Faculty of Mathematics and Computer Science, TU/e. 2015-04

A. Guzzi. *Supporting Developers' Teamwork from within the IDE*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-05

T. Espinha. *Web Service Growing Pains: Understanding Services and Their Clients*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-06

S. Dietzel. *Resilient In-network Aggregation for Vehicular Networks*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-07

E. Costante. *Privacy throughout the Data Cycle*. Faculty of Mathematics and Computer Science, TU/e. 2015-08

S. Cranen. *Getting the point — Obtaining and understanding fixpoints in model checking*. Faculty of Mathematics and Computer Science, TU/e. 2015-09

R. Verdult. *The (in)security of proprietary cryptography*. Faculty of Science, Mathematics and Computer Science, RU. 2015-10

J.E.J. de Ruiter. *Lessons learned in the analysis of the EMV and TLS security protocols*. Faculty of Science, Mathematics and Computer Science, RU. 2015-11

Y. Dajsuren. *On the Design of an Architecture Framework and Quality Evaluation for Automotive Software Systems*. Faculty of Mathematics and Computer Science, TU/e. 2015-12

J. Bransen. *On the Incremental Evaluation of Higher-Order Attribute Grammars*. Faculty of Science, UU. 2015-13

S. Picek. *Applications of Evolutionary Computation to Cryptology*. Faculty of Science, Mathematics and Computer Science, RU. 2015-14

C. Chen. *Automated Fault Localization for Service-Oriented Software Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-15

S. te Brinke. *Developing Energy-Aware Software*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-16

R.W.J. Kersten. *Software Analysis Methods for Resource-Sensitive Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2015-17

J.C. Rot. *Enhanced coinduction*. Faculty of Mathematics and Natural Sciences, UL. 2015-18

M. Stolikj. *Building Blocks for the Internet of Things*. Faculty of Mathematics and Computer Science, TU/e. 2015-19

D. Gebler. *Robust SOS Specifications of Probabilistic Processes*. Faculty of Sciences, Department of Computer Science, VUA. 2015-20

M. Zaharieva-Stojanovski. *Closer to Reliable Software: Verifying functional behaviour of concurrent programs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-21

R.J. Krebbers. *The C standard formalized in Coq*. Faculty of Science, Mathematics and Computer Science, RU. 2015-22

R. van Vliet. *DNA Expressions – A Formal Notation for DNA*. Faculty of Mathematics and Natural Sciences, UL. 2015-23

S.-S.T.Q. Jongmans. *Automata-Theoretic Protocol Programming*. Faculty of Mathematics and Natural Sciences, UL. 2016-01

S.J.C. Joosten. *Verification of Interconnects*. Faculty of Mathematics and Computer Science, TU/e. 2016-02

M.W. Gazda. *Fixpoint Logic, Games, and Relations of Consequence*. Faculty of Mathematics and Computer Science, TU/e. 2016-03

S. Keshishzadeh. *Formal Analysis and Verification of Embedded Systems for Healthcare*. Faculty of Mathematics and Computer Science, TU/e. 2016-04

P.M. Heck. *Quality of Just-in-Time Requirements: Just-Enough and Just-in-Time*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2016-05

Y. Luo. *From Conceptual Models to Safety Assurance – Applying Model-Based Techniques to Support Safety Assurance*. Faculty of Mathematics and Computer Science, TU/e. 2016-06

B. Ege. *Physical Security Analysis of Embedded Devices*. Faculty of Science, Mathematics and Computer Science, RU. 2016-07

A.I. van Goethem. *Algorithms for Curved Schematization*. Faculty of Mathematics and Computer Science, TU/e. 2016-08

T. van Dijk. *Sylvan: Multi-core Decision Diagrams*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2016-09

I. David. *Run-time resource management for component-based systems*. Faculty of Mathematics and Computer Science, TU/e. 2016-10

A.C. van Hulst. *Control Synthesis using Modal Logic and Partial Bisimilarity – A Treatise Supported by Computer Verified Proofs*. Faculty of Mechanical Engineering, TU/e. 2016-11

A. Zawedde. *Modeling the Dynamics of Requirements Process Improvement*. Faculty of Mathematics and Computer Science, TU/e. 2016-12

F.M.J. van den Broek. *Mobile Communication Security*. Faculty of Science, Mathematics and Computer Science, RU. 2016-13

J.N. van Rijn. *Massively Collaborative Machine Learning*. Faculty of Mathematics and Natural Sciences, UL. 2016-14

M.J. Steindorfer. *Efficient Immutable Collections*. Faculty of Science, UvA. 2017-01

W. Ahmad. *Green Computing: Efficient Energy Management of Multiprocessor*

Streaming Applications via Model Checking. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2017-02

D. Guck. *Reliable Systems – Fault tree analysis via Markov reward automata*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2017-03

H.L. Salunkhe. *Modeling and Buffer Analysis of Real-time Streaming Radio Applications Scheduled on Heterogeneous Multiprocessors*. Faculty of Mathematics and Computer Science, TU/e. 2017-04

A. Krasnova. *Smart invaders of private matters: Privacy of communication on the Internet and in the Internet of Things (IoT)*. Faculty of Science, Mathematics and Computer Science, RU. 2017-05

A.D. Mehrabi. *Data Structures for Analyzing Geometric Data*. Faculty of Mathematics and Computer Science, TU/e. 2017-06

D. Landman. *Reverse Engineering Source Code: Empirical Studies of Limitations and Opportunities*. Faculty of Science, UvA. 2017-07