Effective and Efficient Reuse with Software Libraries

Lars Heinemann

Technische Universität München

Institut für Informatik
der Technischen Universität München

Effective and Efficient Reuse with Software Libraries

Lars Heinemann

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr. Daniel Cremers
Prüfer der Dissertation:	
1.	Univ.-Prof. Dr. Dr. h.c. Manfred Broy
2.	Prof. Martin Robillard, Ph.D.
	McGill University, Montréal, Kanada

Die Dissertation wurde am 26.07.2012 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 20.11.2012 angenommen.

Abstract

Research in software engineering has shown that software reuse positively affects the competitiveness of an organization: the productivity of the development team is increased, the time-to-market is reduced, and the overall quality of the resulting software is improved.

Today's code repositories on the Internet provide a large number of reusable software libraries with a variety of functionality. To analyze how software projects utilize these libraries, this thesis contributes an empirical study on the extent and nature of software reuse in practice. The study results indicate that third-party code reuse plays a central role in modern software development and that reuse of software libraries is the predominant form of reuse. It shows that many of today's software systems consist to a considerable fraction of code reused from software libraries.

While these libraries represent a great reuse opportunity, they also pose a great challenge to developers. Often these libraries and their application programming interfaces (APIs) are large and complex, leading to cognitive barriers for developers. These obstacles can cause unintended reimplementation in a software system. This negatively affects the system's maintainability due to the unnecessary size increase of the code base and may even compromise its functional correctness, as library functionality is often more mature.

To investigate the applicability of analytic methods for finding missed reuse opportunities, this work presents an empirical study on detecting functionally similar code fragments. It reports on the considerable challenges of the automated detection of reimplemented functionality in real-world software systems. The results of the study emphasize the need for constructive approaches to avoid reimplementation during forward-engineering.

Motivated by these results, this thesis introduces an API method recommendation system to support developers in using large APIs of reusable libraries. The approach employs data mining for extracting the intentional knowledge embodied in the identifiers of code bases using the APIs. The knowledge obtained is used to assist developers with context-specific recommendations for API methods. The system suggests methods that are useful for the current programming task based on the code the developer is editing within the integrated development environment (IDE). It thereby makes reuse of software libraries both more effective and efficient. As opposed to existing approaches, the proposed system is not dependent on the previous use of API entities and can therefore produce sensible recommendations in more cases. We evaluate the approach with a case study showing its applicability to real-world software systems and quantitatively comparing it to previous work.

In addition, we present an approach that transfers the idea of API recommendation systems to the field of model-based development. We introduce a recommendation system that assists developers with context-dependent model element recommendations during modeling. We instantiate and evaluate the approach for the Simulink modeling language. To the best of our knowledge, this is the first work to adopt recommendation systems to the field of model-based development.

Acknowledgements

I am grateful to all the people that supported me on the way to this thesis. First, I want to thank Manfred Broy for providing me with the opportunity to work at his chair and for supervising this thesis. Working at the chair in teaching and research has both extended and sharpened my view on computer science in a way I did not expect. The professional environment at his chair is outstanding in many ways, in particular with respect to the challenging research projects involving key players from both industry and academia. I am also grateful to Martin Robillard for co-supervising this thesis and his valuable feedback on my work.

In addition, I want to express my gratitude to the complete staff at the chair of Prof. Broy. I very much enjoyed working with all of you in the last three and a half years. I especially thank Silke Müller for the organizational support, the IT department for always providing me with all I needed in terms of hardware and software resources, Florian Deißenböck for the most direct and honest review feedback I ever received, Elmar Jürgens for being a great example regarding his remarkable writing skills, Benjamin Hummel for a constantly open door for discussing research ideas, Veronika Bauer for sharing an office with me, and Andreas Vogelsang for his numerous and always helpful paper reviews.

My scientific work in the last years included the publication of a number of research papers. I am thankful for the co-authorship in joint paper projects of Veronika Bauer, Florian Deißenböck, Mario Gleirscher, Markus Herrmannsdörfer, Benjamin Hummel, Maximilian Irlbeck, Elmar Jürgens, Klaus Lochmann, Daniel Ratiu, and Stefan Wagner.

I also worked in interesting and challenging research projects at the chair of Prof. Broy together with a number of fellow researchers. I want to express my gratitude to Veronika Bauer, Sebastian Eder, Benedikt Hauptmann, Markus Herrmannsdörfer, Benjamin Hummel, Maximilian Junker, Klaus Lochmann, and Daniel Méndez-Fernández for being great colleagues during research projects.

Further thanks go to the proof-readers of this thesis for insightful discussions and valuable feedback: Veronika Bauer, Jonas Eckhardt, Sebastian Eder, Henning Femmer, Bendikt Hauptmann, Benjamin Hummel, Maximilan Junker, Elmar Jürgens, Klaus Lochmann, Birgit Penzenstadler, and Andreas Vogelsang.

Last but not least I want to thank my family and friends for their support during my entire education and especially this thesis project.

Contents

1 Introduction

Software reuse addresses the use of existing artifacts for the construction of new software. Research in software engineering has shown that it positively affects the competitiveness of an organization in multiple ways: the productivity of the development team is increased, the time-to-market is reduced, and the overall quality of the resulting software is improved [46, 66].

While software reuse can take many forms, a particularly attractive form are externally developed artifacts, for which the complete development and maintenance is outsourced to third parties. Fueled by both the free/open source movements and commercial marketplaces for software components in combination with the evolution of the Internet as a commodity resource in software development, a large number of reusable libraries and frameworks is available for building new software [101]. The Internet itself has become an interesting reuse repository [43] and is emerging as "a de facto standard library for reusable assets" [26]. For example, the Java library search engine *findJAR.com*[1] indexes more than 140,000 Java class libraries[2] and the Apache Maven repository[3] contains about 40,000 unique artifacts[4].

To effectively use a software library, and thus achieve high reuse rates, profound knowledge about the library's content is required. Unfortunately, recent research indicates that developers still reinvent the wheel and reimplement general purpose functionality in their code instead of reusing mature and well-tested library functionality [52, 57]. While there are conscious and valid reasons not to reuse a certain artifact (*e.g.*, legal constraints), unconscious obstacles cause reuse opportunities to be missed inadvertently. For instance, Kawrykow and Robillard analyzed 10 open source systems and found 400 reimplemented methods that would have been available in the included libraries of the project [57]. Such reimplementation is problematic for multiple reasons. First, it is a waste of development resources. Second, more code has to be maintained due to the unecessary size increase of the software. Finally, the home-grown solution to a problem may exhibit bugs not present in the library implementation [52].

Unfortunately, the sheer number of libraries included by many software projects, as well as the size and complexity of their application programming interfaces (APIs) make it difficult for developers to find the right library element for a given programming problem and can thus cause reimplementation of library functionality. Recent research shows that finding relevant API elements is one of the major difficulties faced by developers using unfamiliar APIs [21]. As an illustration for this problem, Table 1.1 shows six widely-used Java libraries and frameworks with different types of functionality. The rightmost column shows the number of API types[5] in each of these libraries. For

[1] http://www.findjar.com/
[2] as of May, 2012
[3] http://search.maven.org/
[4] as of May, 2012
[5] classes, interfaces, enumerations and annotations listed in the API reference documentation

Table 1.1: Common Java libraries/frameworks and number of API types

Library	Version	Description	API types
Java Standard API	1.7	Java Development Platform	4,024
Eclipse Platform	3.6	Platform for general purpose applications	3,588
Spring Framework	2.0	Application framework	1,680
JFreeChart	1.0.13	Charting and imaging library	586
Dom4j	2.0	XML, XPath and XSLT library	159
Apache Commons IO	2.0.1	Utilities for IO	98

instance, the Apache Commons IO library, which is solely concerned with IO functionality consists of almost 100 types. Application frameworks such as Eclipse can contain as much as several thousand types.

Furthermore, even developers who are familiar with these libraries and their APIs are still challenged by their evolution because successful libraries are often continuously enhanced with new functionality. An illustrative example is given by the Java Standard API, which is used by virtually every Java program. The number of types listed in its reference documentation grew by 160% over the last six releases, as shown in Figure 1.1. In other words, even a developer who is familiar with the complete API in version 1.2, but now has to switch to the latest version, does not even know half of the API. These numbers document the challenges faced by developers using these libraries and their APIs.

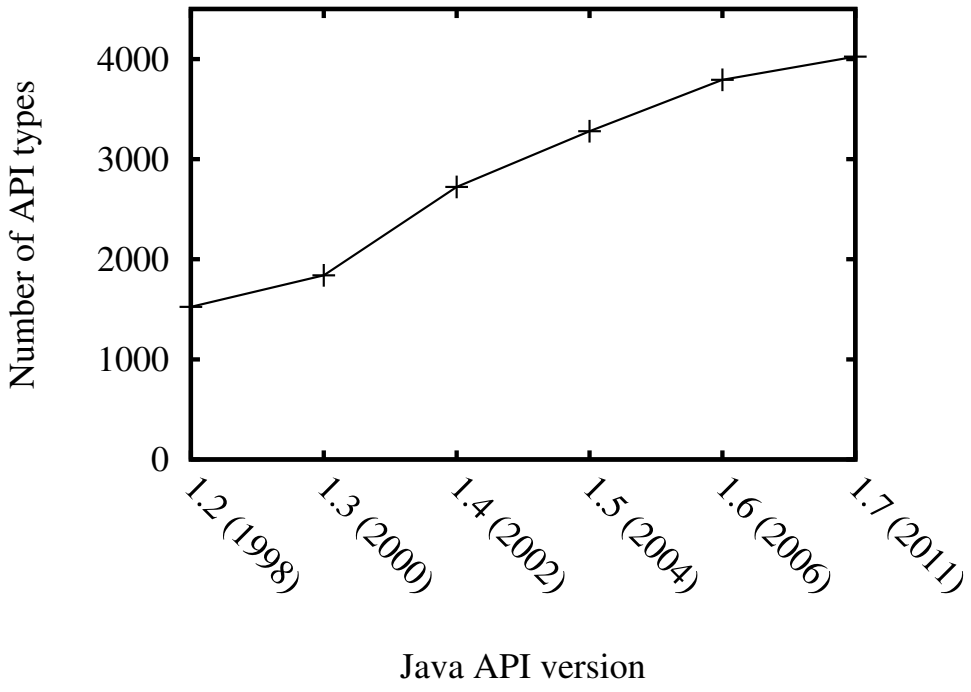

Figure 1.1: Number of API types for different versions of the Java API

1.1 Problem Statement

Code repositories on the Internet provide a large amount of reusable code with a variety of functionality. However, it remains largely unknown to what extent software projects make use of these reuse opportunities and how third-party code contributes to software projects. Thus, the reuse potential exhibited by these code repositories cannot be determined and we cannot adequately judge the role of third-party reuse in practice.

While the available third-party code represents a great reuse opportunity, it also poses a great challenge to developers. Often, reusable libraries and their APIs are large and complex, creating cognitive barriers for developers [25, 92]. For successful reuse of a library element, a developer has to (1) anticipate the existence of a solution to his current problem and (2) actively search for a suitable solution. If these challenges are not overcome, reuse opportunities are missed and thus the benefits of reuse are lost.

Reimplementation in software can be considered a quality defect. It negatively affects maintainability due to the unnecessary size increase of the code base and may even compromise functional correctness, as library functionality is often more mature [52]. In the course of quality assurance measures it is thus desirable to detect such reimplementation. However, manual detection of this type of redundancy is infeasible due the number and size of the libraries as well as the size of the including software systems. It is unknown whether analytic methods can help to automatically detect such reimplementation in real-world software systems. Ideally, reimplementation is avoided during the development of software. To ensure effective and efficient reuse of libraries, proper support is required that assists developers using the available artifacts during the construction of new software.

Recently, model-based development has gained increased attention in software engineering due to its promise of productivity improvements. The development activities shift their focus from code to models, which become the primary artifacts created and maintained by the developers. Data flow and processing languages often have a notion of reuse and libraries of reusable model elements are used to construct models from existing parts. The challenges exhibited by these modeling libraries are similar to those in code-based development. It remains unclear if methods to support library reuse can be adopted to model-based development.

1.2 Contribution

To characterize third-party code reuse in practice, this thesis contributes an empirical study showing how open source projects achieve reuse and to what extent they utilize the reuse opportunities on the Internet. The study results indicate that reuse of software libraries is the predominant form of reuse and that almost half of the analyzed projects reuse more code than they develop from scratch, thus providing evidence for the central role of third-party code reuse in modern software development.

To investigate the applicability of analytic methods for finding missed reuse opportunities, this work presents an empirical study on detecting functionally similar code fragments. It reports on the considerable challenges of the automated detection of reimplemented functionality in real-world

software systems. The results of the study emphasize the need for constructive approaches to avoid reimplementation during forward-engineering.

Motivated by these results, this thesis introduces an API method recommendation system to support developers in using large APIs of reusable libraries. The approach employs data mining for extracting the intentional knowledge embodied in the identifiers of code bases using the APIs. The knowledge obtained is used to assist developers with context-specific recommendations for API methods. The system suggests methods that are useful for the current programming task based on the code the developer is editing within the integrated development environment (IDE). It thereby makes reuse of software libraries both more effective and efficient. As opposed to existing approaches, the proposed system is not dependent on the previous use of API entities and can therefore produce sensible recommendations in more cases. We evaluate the approach with a case study showing its applicability to real-world software systems and quantitatively comparing it to previous work.

To address the challenge of working with large modeling libraries, we present an approach that transfers the idea of API recommendation systems to the field of model-based development. We introduce a recommendation system that assists developers with context-dependent model element recommendations during modeling. We instantiate and evaluate the approach for the Simulink modeling language. To the best of our knowledge, this is the first work to adopt recommendation systems to the field of model-based development.

1.3 Outline

The remainder of this document is structured as follows. Chapter 2 presents the foundations for this thesis and sets this work into the context of different reuse approaches. The current state of the art is discussed in Chapter 3. Chapter 4 presents the empirical study on code reuse in open source projects. Chapter 5 presents the study on the automated detection of functionally similar code fragments. The API method recommendation system is presented in Chapter 6 and the recommendation system for model-based development is described in Chapter 7. Chapter 8 discusses the tool support that was developed in the context of this thesis. Finally, Chapter 9 discusses the conclusions that can be drawn from this work and Chapter 10 highlights directions for future work.

Previously Published Material

Parts of the contributions presented in this thesis have been published in the proceedings of international workshops and conferences [17, 33–36].

2 Preliminaries

This chapter introduces the preliminaries for this thesis. First, it motivates software reuse, presents different approaches to software reuse, discusses potential obstacles to reuse, and explains how this thesis integrates into that context. It then lays the required foundation in the area of data mining and outlines the fields of recommendation systems in software engineering and model-based development.

2.1 Why Software Reuse?

Reusing existing code for the development of new software provides a number of benefits for software development which are listed in the following.

Increased Productivity Studies have shown that software reuse has a positive effect on productivity [10, 56, 66]. Less code has to be developed from scratch and thus the ratio between input and output of the development process is increased.

Improved Overall Quality Reusing mature and well-tested software elements can increase the overall quality of the resulting software system [56, 66]. Reusable software elements are often used multiple times and are therefore proven solutions that can provide better quality characteristics compared to newly developed code. A user of a mature software module can benefit from the collective experience of previous users of the module, as many bugs as well as deficiencies in the documentation have already been discovered [10].

Reduced Time-To-Market In dynamic markets where multiple companies offer similar products, the time in which a product can be brought to the market is an important factor influencing the competitiveness of an organization. Reusing software can help to significantly reduce the time-to-market [56, 66]. In case the software is used in-house, the earlier completion results in an earlier payback of the development efforts [10].

2.2 Software Reuse Approaches

To set the contributions of this thesis into perspective, this section presents different approaches to software reuse. First, we discuss the dimensions along which reuse can be categorized and then briefly describe several reuse approaches.

2.2.1 Dimensions of Software Reuse Approaches

Karlsson classifies software reuse approaches along the three dimensions *scope*, *target* and *granularity* [56]. We briefly repeat them here.

Scope Reuse can have different *scopes*: *General reuse* aims at reusing domain-independent artifacts in a broad variety of application domains. *Domain reuse* has the goal of reusing artifacts specific to a certain application domain (*e.g.*, *accounting*). *Product-line* reuse tries to reuse artifacts between multiple applications of the same type within a product family.

Target The target reuser can be either *internal*, *i.e.*, the artifact is created and reused within the same organization or *external*, *i.e.*, the reused artifact is created outside the organization. In case of the latter, the development and maintenance of the artifact are "outsourced" to third parties.

Granularity Software reuse can occur at different levels of *granularity*. A simple classification of granularity is fine-grained and coarse-grained. Examples for fine-grained reuse are general purpose and domain-independent utility functions (*e.g.*, file handling). Coarse-grained reuse means reusing larger artifacts such as a database.

2.2.2 Reuse Approaches

A number of different approaches to achieve reuse in software projects have been proposed in the literature. We list them and briefly discuss their individual characteristics as well as their intent in this section.

Design and Code Scavenging Design and code scavenging is an ad hoc approach to software reuse. Developers take parts from existing software and integrate them into the new software. Typically, individual code fragments are reused by copying and adapting them as necessary. In case of design scavenging, large fragments are copied and internals are removed until only the underlying design remains [61]. The act of reusing artifacts in which the internals are revealed and may be modified is also referred to as *white-box reuse*.

Software Libraries Software libraries are collections of code entities, typically with well-defined and documented interfaces, intended to be reused. In contrast to reuse by design and code scavenging, software libraries are reused "as is". The internals are hidden to the reuser and no modifications are applied to the reusable artifact. This form of reuse is also referred to as *black-box reuse*.

Software Frameworks Software frameworks are incomplete applications that can be reused by completing them to a custom application through the implementation of designated extension points (or *hook methods*) [23]. A major underlying principle of framework reuse is the *inversion of control*, meaning that the framework calls the code that is written by developers instead of the other way around (as with library reuse). A popular example is the Eclipse Framework[1] for building rich client applications.

Component-based Development The idea of components which are assembled to larger applications was first introduced by McIlroy in 1968 [72]. In component-based development, components are exclusively developed with the intention to be reused. There exist a multitude of different notions of a component. It can be given by a programming language unit such as a package, subsystem or class [61].

Application Generators Application generators transform a specification of a program into an executable application [15]. The input specification is typically formulated in a domain specific language (DSL). Reuse is achieved by running the generator multiple times on different input specifications. Application generators thereby "reuse complete software system designs" [61]. Similar to software product lines, the commonalities of applications are exploited to achieve reuse. Application generators contain these commonalities and are built only once but used repeatedly to generate multiple applications [61].

Design Patterns Design patterns are solutions to common recurring design problems in object-oriented programming [28]. Reuse occurs by applying a pattern repeatedly to instances of the general problem. Patterns describe the objects involved in a design and how they collaborate. A design pattern can be viewed as a "reusable micro-architecture" [29].

Product Lines Software product lines are a systematic approach to achieve reuse within a family of similar products. A strong need for software product lines has been growing in the area of embedded systems where the demand for variability of products is steadily increasing. Product line engineering involves planning for reuse, creating reusable parts and reusing them. The wish for mass customization is addressed with the concept of *managed variability*. Thereby, commonality and variability of the applications are explicitly modeled [85]. A common way to model applications in a product line are so-called *feature models*. A feature model consists of a feature hierarchy and a set of constraints between the features [7]. A constraint could, for instance, prescribe that for feature A to be present in an application, feature B is required as well. Product lines can only be applied within an organization and typically involve significant education efforts for developers [56].

[1] http://www.eclipse.org/

2.3 Obstacles for Software Reuse

Successfully reusing a software artifact requires three principal steps [56]:

1. searching for candidate artifacts
2. determining the most suitable artifact
3. potentially adapting the chosen artifact to individual needs

Despite the known benefits of software reuse regarding productivity and quality, various obstacles within these steps can inhibit successful reuse. Some of these issues are conscious while others are unconscious. This section discusses obstacles from both of these categories.

2.3.1 Conscious Obstacles

Developers can deliberately decide not to reuse a certain software artifact for multiple reasons, which are discussed in the following.

Not Invented Here The *not invented here* phenomenon denotes the situation where a developer dislikes to use an artifact created by a third party and prefers to implement the artifact herself. A possible reason is the lack of trust in the solution from others [89] or a personal interest for the programming task at hand. According to Leach, one way to address this problem is for management to reward reuse and make developers aware that they can focus their creativity on more challenging problems [64].

Licensing/Legal Issues Licensing and legal issues can prevent the reuse of an artifact if the license of the provided artifact is in contradiction with the situation in the reusing organization. For instance, a license may prohibit employment of the artifact in certain application areas (*e.g.*, safety-critical applications). Moreover, licenses can require the integrating organization to release the software reusing the artifact under a certain license (*e.g.*, so-called *viral* licenses). Finally, commercial organizations often provide some warranty for their products. When integrating third-party artifacts, control over them is limited. However, the integrating organization may be held responsible for any harm caused by the artifact integrated in the product.

Non-Functional Requirements Not Met Another important reason not to reuse an artifact is given, if it provides the desired functional behavior but fails to meet certain non-functional requirements (*e.g.*, performance, resource utilization).

Problems Adapting Solution Reuse may also be prevented if the existing solution cannot be adapted to the context where it is to be employed. The adaptation effort may exceed the effort for developing the functionality from scratch.

Loss of Control By reusing third-party artifacts, the development organization looses control over parts of the software system. As an example, during maintenance, changes to the reused parts may be difficult or impossible.

2.3.2 Unconscious Obstacles

Besides the conscious reasons not to reuse an artifact, there are also unconscious factors that can inhibit reuse, although it would be desirable.

Existence of Solution Unanticipated A necessary prerequisite for reuse to occur is that the developers anticipate the existence of a solution. If an artifact is not assumed to exist for a given problem, the developer will not search for it and reuse cannot happen.

Problems Finding Solution Another hurdle is the task of finding an existing solution to a given problem. The developer may assume that there is likely an artifact useful for the problem but she may fail to locate a suitable artifact in a reuse repository, *e.g.*, due to weaknesses in search capabilities.

2.4 Perspective of this Thesis

This thesis addresses the *unconscious obstacles* that are inhibitive for successful reuse in practice. The goal is to foster reuse during the construction of software and thus limit the negative effects of these obstacles. Reuse is assisted by supporting developers in searching for reusable artifacts which is a necessary prerequisite for successful reuse.

With regards to the dimensions of reuse as identified by Karlsson (see Section 2.2.1), this thesis is positioned as follows:

Scope The focus of this thesis is on general reuse as the code repositories on the Internet provide a large amount of freely reusable general purpose code.

Target As external reuse of free third-party code is particularly attractive, we focus on this target. However, many aspects of the work in this thesis also apply to internal reuse.

Granularity We focus on *fine-grained* reuse on the level of methods as the most basic form for black-box reuse of existing code.

Due to the prominent role of object-oriented languages in many application areas of modern software development, the code-centric considerations and contributions are specifically targeted at object-oriented systems, however, where appropriate, it is stated if and how the findings or principles apply to other programming paradigms or could be adapted for them.

Within the software development life cycle, the work in this thesis is targeted at the module implementation phase. We assume, that in the module design phase, appropriate libraries for reuse are identified and integrated into the overall system design. The goal of the contributions in this thesis is to make the usage of software libraries during development and maintenance of a software system more effective and efficient.

2.5 Data Mining Background

This section introduces the background in the field of data mining as a foundation for the contributions on recommendation systems presented in this thesis. We give a brief overview and discuss the specific techniques this work is based on.

2.5.1 Overview

Data mining is an integral part of knowledge discovery—the process of discovering new and ideally useful information from data. The purpose of data mining is to discover patterns within the analyzed data. The obtained knowledge can then be used for decision support. Application areas are diverse and include financial forecasting, medical diagnosis and targeted marketing. A well-known example is the shopping cart analysis to optimize the alignment of the business to customer needs [11].

2.5.2 Recommendation Systems

Recommendation systems aim at giving sensible recommendations to users for items they might like [98]. These systems have been introduced in response to the information overload buyers face in many markets. The basic idea is "to harness the opinions of millions of people" in order to "help all of us find more useful and interesting content" [47]. A well-known form of a recommendation is the phrase "Customers who bought this item also bought these items" [47]. A famous application is the recommendation functionality on the Amazon Online bookstore[2]. When browsing books, recommendations are displayed for other books that may be of interest. Different techniques are employed for implementing recommendation systems, including association rules and collaborative filtering, which are introduced in the following.

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

2.5.3 Association Rules

In data mining, association rules are typically employed for shopping basket analysis [11]. The goal is to identify regularities in shopping transaction data (*i.e.*, detailed information on purchases) which can then be used, for instance, to identify cross-selling potential[3] or better align the business to customer needs. A common form of such a regularity is the fact that certain items are often purchased together [47]. A transaction (or shopping basket) is typically given as a set of purchased items where quantities are abstracted. An example is:

$$\{paper, envelope, stamp\}$$

Association rules are of the form $I \rightarrow j$, where I is a set of items and j an item. An example is:

$$\{paper, envelope\} \rightarrow stamp$$

In the example, customers who bought paper and envelopes "typically" also bought stamps.

The quality of association rules is expressed in terms of *support* and *confidence*. The support of an association rule $I \rightarrow j$ corresponds to the fraction of baskets which contain all elements of $I \cup j$ with respect to the number of all baskets. The confidence of an association rule $I \rightarrow j$ is given by the ratio between the number of baskets containing all elements of $I \cup j$ and the number of baskets that contain all elements of I. More formally, this can be written as follows:

$$support(I \rightarrow j) = \frac{baskets\ containing\ all\ elements\ of\ I \cup j}{overall\ number\ of\ baskets}$$

$$confidence(I \rightarrow j) = \frac{baskets\ containing\ all\ elements\ of\ I \cup j}{baskets\ containing\ all\ elements\ of\ I}$$

The Apriori algorithm [1] mines association rules from a given set of shopping baskets. It computes frequent item sets in the shopping baskets that have a certain *support threshold*. From the frequent item sets, association rules are built that have a desired *confidence threshold*.

The mined association rules can then be used to recommend additional items for shopping baskets that they are applicable to. An association rule $I \rightarrow j$ is *applicable* to a given shopping basket B iff all elements in I are contained in B. More formally, this can be expressed as follows:

$$applicable(I \rightarrow j, B) \Leftrightarrow I \subseteq B$$

The threshold parameters of the algorithm influence the number and quality of the obtained association rules.

[3] Cross-selling refers to the sale of products which complement each other. The potential lies in selling an additional complementing product to a customer.

Table 2.1: Example data base for users and movies

	Joe	Ann	Mike
Moby Dick	0	0	1
The Time Machine	1	1	0
Star Wars	0	1	1
Once Upon a Time in the West	1	1	0

2.5.4 Vector Space Model

The *vector space model* is used in data mining to represent items. The items are encoded as an n-dimensional vector. An important data mining problem is to find similar items. This can be mapped to the problem of finding nearest neighbors in the vector space model, for which algorithms exist. As an example, in information retrieval, natural language documents are represented as vectors where the components of the vector denote the weight of a term. The weight can, for instance, be computed by the number of occurrences of the term within that document [11].

2.5.5 Collaborative Filtering

The basic idea of collaborative filtering (CF) is that users with similar preferences regarding items will rate other items similarly as well [102]. CF uses a database of users, items and a *like*-relation between users and items, to recommend items to users. Based on what a user already likes, similar users are searched and additional items which the similar users like are returned as recommendations.

As an example, let us assume that we have data about which movies users bought (*i.e.*, *binary ratings*). The content of the data base is illustrated in Table 2.1. A "1" indicates that a user bought a particular movie and a "0" denotes that a user has not bought that movie yet. Imagine, we want to produce a recommendation for Joe. In the database, the user Ann is most similar to Joe, as she bought two movies which Joe bought as well. Consequently, we can recommend the movie *Star Wars* to Joe, which he has not bought yet.

A common approach for implementing a CF-based recommendation system is to encode the items which a user likes as a vector and define similarity among users as the similarity between the corresponding vectors in the vector space (see Section 2.5.4). A vector similarity measure that is often used is the *cosine similarity*, which corresponds to the cosine of the angle between the vectors. It is computed as follows:

$$cosine\_similarity(\vec{a}, \vec{b}) = \frac{\vec{a} \cdot \vec{b}}{|\vec{a}| \times |\vec{b}|}$$

To obtain recommendations, the *k-nearest-neighbor* algorithm can be used. The k most similar users are determined according to the similarity measure. Their item preferences are aggregated and the set of recommended items is derived. The value of k influences the number and quality of the recommendations.

2.5.6 Data Mining in Software Engineering

During software development, a number of repositories are used to store a variety of data. Examples include source code in version control systems, bug data bases and communication logs (*e.g.*, e-mails). The field of *Mining Software Repositories* (MSR) which has gained increased interest in the past years tries to leverage this data for understanding software evolution and for decision support in software projects [32, 54]. An example is the detection of change coupling patterns, where the goal is to determine which artifacts often change together. If during maintenance, a developer changes one artifact, she may be notified to consider other artifacts that have been changed along with this artifact in the past [120].

2.6 Recommendation Systems in Software Engineering

The basic idea of recommendation systems in software engineering is to leverage the *collective wisdom* of developers to assist individual developers in their daily work. The knowledge of many developers is utilized with the help of machine learning to create a shared knowledge base, which is then used to give task-specific recommendations to individual developers. This principle is illustrated in Figure 2.1.

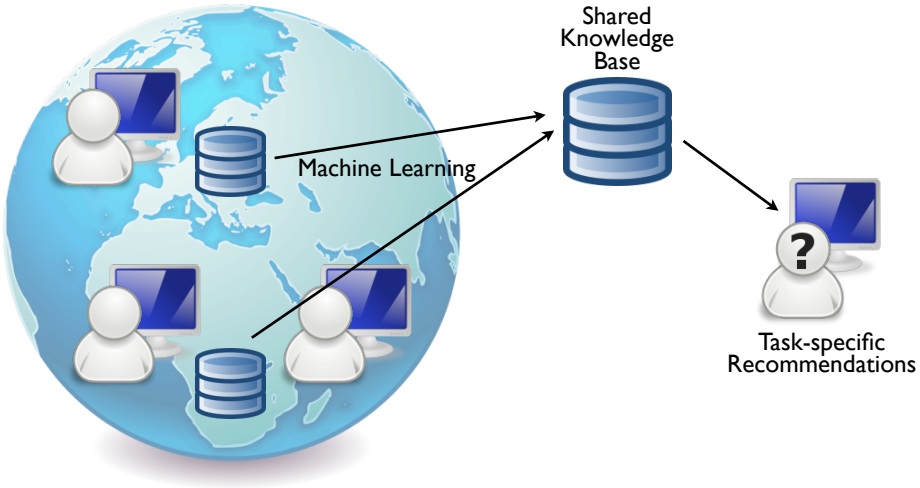

Figure 2.1: Recommendation systems in software engineering

The knowledge used by recommendation systems can be obtained in different ways. A common approach is to perform data mining on software repositories such as version control systems and bug databases (*cf.*, Section 2.5.6) and extract knowledge from them. The major advantage is that this process can be performed completely automatically. Moreover, it is not relevant where the developers are located physically. The type of knowledge that can be used is diverse and many different objects can be recommended by recommendation systems. The goal of the recommendations is to help developers in decision making within their tasks. Often, recommendation systems aim at assisting developers who seek for certain information [93]. The types of objects recommended include code, artifacts, people (*e.g.*, the *ideal* assignee for a bug [4]), tools [31], and refactorings [106].

Robillard et al. [93] identified the following three principal design dimensions for recommendation systems in software engineering:

- The *nature of the context* defines the query input of the recommendation system. It can be explicit (*e.g.*, entering text or selecting graphical elements), implicit (*e.g.*, recorded developer interaction with the IDE) or a combination of both.
- The design of the *recommendation engine* determines which additional data the recommendation system uses, besides the context, to produce recommendations. Examples include a project's source code and the change history stored in the version control system. Very common is also a ranking mechanism that orders recommendations with regards to their expected relevance for the user.
- The *output mode* of a recommendation system can be either *pull mode* (recommendations are made in response to an explicit user request) or *push mode* (recommendations are made proactively). Moreover, the *output presentation* can be either *batch*, in which case a list of recommendations is presented in a separated IDE view, or *inline*, in which case recommendations are presented as annotations directly on top of the artifacts the developer is working with.

In addition, Robillard et al. identified cross-dimensional features of recommendation systems in software engineering: *Explanations*, affecting both the recommendation engine and the output mode, give information to the user *why* a certain recommendation was made in order to improve transparency of the system. *User feedback*, influencing all three dimensions, can for instance be used to allow users to flag recommendations as inappropriate in order to exclude them from future recommendations.

A comprehensive overview of recommendation systems in software engineering is provided by Robillard et al. in [93]. The focus of this thesis is the recommendation of API elements to developers. A detailed discussion of existing work in the area of *code* recommendation systems is given in Chapter 3.

2.7 Model-Based Development

Model-based development has recently gained increased attention due to its promise of improved productivity. As opposed to code-based development, where the focus is on source code, models become primary development artifacts and, in order to obtain a running system, the models are interpreted or executable code is generated from them. The main development and maintenance activities thus shift their focus from code to models. Instead of writing code, developers create and maintain models represented in a domain specific language.

Mathworks Simulink[4] is a development environment for modeling and simulating dynamic systems which utilizes a block diagram notation [16]. Simulink has become a prevalent technology in the embedded systems domain, especially in the automotive field. With TargetLink[5], a C code generator for Simulink models, executable code can be generated automatically. Simulink models

[4] http://www.mathworks.com/products/simulink/
[5] http://www.dspace.de/en/pub/home/products/sw/pcgs/targetli.cfm

are constructed from so called *blocks*, the atomic functional units of the Simulink language, which compute output signals from input signals.

As an illustration, Figure 2.2 shows a Simulink subsystem for an air condition (A/C) control. The subsystem has four inputs for data from different sensors in a vehicle. The output of the subsystem is the temperature control of the A/C which is computed from the inputs using several blocks. The subsystem thus implements a control loop for the temperature in the vehicle.

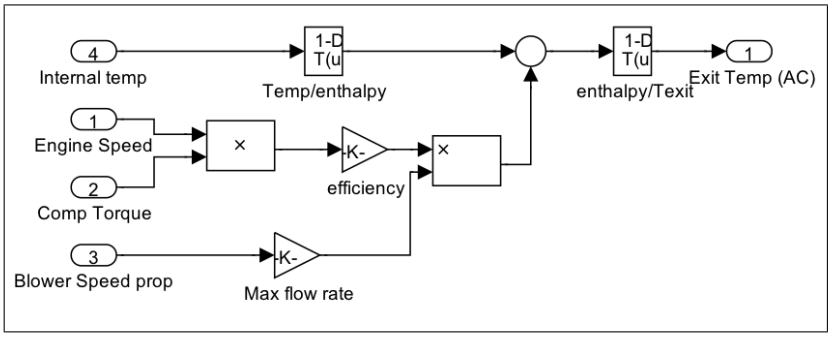

Figure 2.2: A/C control subsystem (Source: http://www.mathworks.com)

2.8 ConQAT: Continuous Quality Assessment Toolkit

ConQAT[6] is a flexible open source toolkit for the quality analysis of software systems. Within this thesis, it is used as a basis for various analyses and the tool support. ConQAT employs a modular architecture that supports the extension with new analysis capabilities by a plug-in mechanism. Its graphical user interface is built on top of the Eclipse Platform which also allows for an extension of its user interface with additional features in the form of plug-ins.

Figure 2.3 shows a high-level overview of the main parts of ConQAT. The *ConQAT Engine* implements a large variety of analysis capabilities for source code written in diverse programming languages as well as models. The *Dashboard Construction* is the graphical editor for editing quality analysis configurations. *Clone Detection* and *Architecture Analysis* refer to user interface capabilities specific to the detection of duplicated code and conformance analysis of software architectures. In the course of this thesis, we will build on both the *ConQAT Engine* and the *Dashboard Construction*.

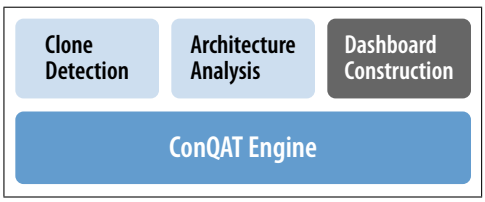

Figure 2.3: Overview of ConQAT

[6] http://www.conqat.org/

3 State of the Art

This chapter presents existing work in the research areas of this thesis organized according to the structure of this thesis. We first discuss related work regarding the quantification of software reuse in open source projects in Section 3.1. Section 3.2 presents related work on the detection of functional similarity in programs. In Section 3.3, we summarize previous work on developer support for software reuse. Finally, Section 3.4 discusses existing approaches related to recommendation systems in model-based development.

Each of the sections describes existing work, identifies their limitations and points to the chapters of this thesis that provide contributions to the identified problems.

3.1 Quantifying Reuse in Software Projects

Several authors have worked on quantifying the extent of software reuse in real-world software projects with varying intent.

In [99], Sojer and Henkel investigate the usage of existing open source code for the development of new open source software by conducting a survey among 686 open source developers. They analyze the degree of code reuse with respect to several developer and project characteristics. They report that software reuse plays an important role in open source development. Their study reveals that the share of functionality that is based on reused code, as estimated by the developers, ranges between 0% and 99% with a mean of 20% and a median of 30%. Since Sojer and Henkel used a survey to analyze the extent of code reuse, their results may be subject to inaccurate estimates of the respondents.

Haefliger et al. [30] analyzed code reuse within six open source projects by performing interviews with developers as well as inspecting source code, code modification comments, mailing lists and project web pages. Their study revealed that all sample projects reuse software. Moreover, the authors found that by far the dominant form of reuse within their sample was black-box reuse. In their sample of 6 MLOC project code (excluding reused code), 55 components, which in total account for 16.9 MLOC, were reused. Of the 6 MLOC, only about 38 kLOC were reused in a white-box fashion. The developers also confirmed that this form of reuse occurs only infrequently and in small quantities. The authors treated whole components as reusable entities. However, in practice not always the whole component may be actually reused, since only parts of its functionality might be used by the integrating program. This may result in an overestimation of the actual reuse rate. Since they use code repository commit comments for identifying white-box reuse, their results are sensitive with regards to the accuracy of these comments.

In [80], Mockus investigates large-scale code reuse in open source projects by identifying components that are reused among several projects. The approach looks for directories in the projects that

share a certain fraction of files with equal names. He investigates how many of the files are reused among the sample projects and identifies what type of components are reused the most. In the studied projects, about 50% of the files were used in more than one project. However, libraries reused in a black-box fashion are not considered in his study. While Mockus' work quantifies how often code entities are reused, the reuse rates achieved in the reusing projects are not analyzed. Moreover, reused entities that are smaller than a group of files are not considered.

In [65], Lee and Litecky report on an empirical study that investigates how organizations employ reuse technologies and how different criteria influence the reuse rate in organizations using Ada technologies. They surveyed 500 Ada professionals from the ACM Special Interest Group on Ada with a one-page questionnaire. They found a broad spectrum of reuse rates from under 10% to over 80% whereas the majority of the projects (about 55%) exhibited a reuse rate below 20%. The authors determined the amount of reuse with a survey. Therefore their results may be inaccurate due to subjective judgement of the respondents.

In [111], von Krogh et al. report on an exploratory study that analyzes knowledge reuse in open source software. The authors surveyed the developers of 15 open source projects to find out whether knowledge is reused among the projects and to identify conceptual categories of reuse. They analyze commit comments from the code repository to identify accredited lines of code as a direct form of knowledge reuse. Their study reveals that all the considered projects do reuse software components. Like Haefliger et al., von Krogh et al. rely on commit comments of the code repository with the aforementioned potential drawbacks.

Basili et al. [6] investigated the influence of reuse on productivity and quality in object-oriented systems. Within their study, they determine the reuse rate for 8 projects developed by students with a size ranging from about 5 kSLOC to 14 kSLOC. The students had to implement a management information system and were provided with existing libraries containing GUI functionality, general utilities and a data base implementation. The reuse rates ranged between 0% and about 46% with a mean of 26%.

Ma et al. [69] studied how the Java Standard API is used by a corpus of 76 open source software projects. They employ a tool called Jepends to find API usages, *i.e.*, static use dependencies of API classes or methods. The authors analyze the used entities regarding their usage frequency. In particular, they present the most used and unused entities of the Java Standard API. The analyzed projects combined used about 52% of the classes in the Java Standard API and 21% of the methods, which means that about half of the classes in the Java API are not used at all by the projects in the corpus. The goal of their approach is to assess the utilization and thereby judge the effectiveness of an API.

Lämmel et al. [63] analyzed API usage in a corpus of 1,476 open source Java projects to inform API migration research. They found that the studied projects use between 1 and 27 different APIs with a median of 4. The projects used from 1 to 10,850 distinct API methods from all APIs (median 199.5).

Raemaekers et al. [86] analyzed the usage frequency of third-party libraries within a corpus of 106 open source and 178 proprietary Java systems. The authors present the top 10 of the most frequently used libraries for the analyzed systems. Based on the usage frequency, they propose a risk assessment of library usage for a project, whereby the assumption is that an 'uncommon' (less frequently used) library exposes a higher risk than a commonly used library.

Problem Many existing studies use either interviews or surveys for quantifying reuse and therefore their results are sensitive with regards to the subjective judgement of the interviewees. Other studies either do not analyze real-world software projects or analyze reuse at a coarse level (*e.g.*, groups of files). Other studies focus on the API producers' view on reuse by investigating what fraction of a library is reused by a project corpus.

We currently lack detailed and accurate empirical data on the extent of software reuse in real-world software projects. We do not know what reuse rates can be achieved with the reusable artifacts that are available in code repositories on the Internet.

Contribution Chapter 4 presents an empirical study providing quantitative data on the extent and nature of software reuse in open source Java projects. It thereby sheds light on the question whether reuse of open source code is successful in practice.

3.2 Detection of Functional Similarity in Programs

Duplicated functionality within or among software systems can have two causes. First, code may be copied and pasted, which is known as *cloning*, or functionally similar code may be developed independently. While in the first case the copying and pasting of code is typically a conscious act, the second case may be caused due to unawareness of the existing solution.

Research in software maintenance has shown that many programs contain a significant amount of cloned code. Such cloned code is considered harmful for two reasons: (1) multiple, possibly unnecessary, duplicates of code increase maintenance costs [59, 95] and, (2) inconsistent changes to cloned code can create faults and therefore lead to incorrect program behavior [53]. The negative impact of clones on software maintenance is not due to copy&paste but caused by the semantic coupling of the clones. Hence, functionally similar code, independent of its origin, suffers from the same problems clones are known for. In fact, the reimplementation of existing functionality can be seen as even more critical, since it is a missed reuse opportunity.

The manual identification of functionally similar code is infeasible in practice due to the size of today's software systems. Tools are required to automatically detect such similar code. A study by Juergens et al. [52] showed that existing clone detection tools are limited to finding cloned code, *i.e.*, they are not capable of finding redundant code that has been developed independently. As a consequence, we do not know to what degree real-world software systems contain similar code beyond the code clones originating from copy&paste programming. Manual analysis of sample projects [52] as well as anecdotal evidence, however, indicate that programs indeed contain many similarities not caused by copy&paste.

The problem of detecting functionally similar code is related to various research areas. This section first discusses theoretical work and then focuses on different practical approaches related to detecting functionally similar code fragments in real-world software.

Theoretical Work The problem of functional program equivalence, *i.e.*, deciding whether two programs compute the same function, is undecidable in general, which follows from Rice's theorem [115]. Thus, we can conclude that it is not possible to build a tool that is capable of detecting functionally equal programs with 100% precision.

Equivalence of programs has been discussed in terms of operational semantics by Pitts [83] and Raoult and Vuillemin [87] as well as input/output behavior by Zakharov [114] and Bertran et al. [8]. Pitts introduces a contextual preorder on expressions of ML programs which is used as a basis for defining a contextual equivalence. The focus is on a method for proving contextual equivalence of ML functions. Raoult and Vuillemin prove that two different notions of program equivalence, *i.e.*, fixed-point semantics and operational semantics coincide for recursive definitions. Zakharov defines the functional equivalence of Turing machines. Bertran et al. introduces a notion of input/output equivalence of distributed imperative programs with synchronous communications. The goal of the approach is to allow for the decomposition of distributed program simplification proofs via communication elimination.

In [78], Milner introduces a simulation relation between programs. It abstracts from the data representation and sequence controlling of a program. He proposes a technique for proving simulation between programs. The goal of his work is twofold: First, the notion of simulation leads to a closer definition of what an algorithm is. Second, proving simulation between programs can simplify the task of proving the correctness of a program.

In [45], Ianov introduces so-called *program schemata*, which are formal abstractions of programs. These schemata abstract away the details of the operations of the program while preserving the sequential and control properties. He defines equivalence on these schemata and presents a method for deciding the equivalence of program schemata. The notion of equivalence is not defined with respect to the computed function of the program but rather represents a very strong notion of equivalence [96].

The basic idea of *program transformation* is to transform a program into a semantically equivalent one by applying only semantic-preserving transformations. A possible application is the use for program optimizers. In program optimization, the goal is to find resource intensive parts of the program and to replace them with semantically equivalent calls to libraries that are more efficient. Since program equivalence is undecidable, the more specific notion of *algorithmic equivalence* is introduced. Two programs are algorithmically equal if one can be obtained from the other by applying transformations like variable renaming, semantic-invariant statement reordering or operand permutations of commutative operators [74].

Code Retrieval In the field of software reuse, behavior-based retrieval approaches are used to find reusable software entities. The search query is represented by a semantic abstraction of the software entity to be retrieved. The semantic abstraction can be for instance given in the form of formal specifications or input-output tuples.

Based on contracts (*i.e.*, formal models of functional behavior), Fischer et al. [24] define *plug-in compatibility* denoting whether a component can be safely reused in a given context without modification. The basic idea of the reuse approach is that a component satisfies a query with pre-

and postcondition if the precondition of the query implies the precondition of the component and the postcondition of the component implies that of the query.

In [84], Podgurski and Pierce introduce a method called *behavior sampling* for the automated retrieval of software components from a software component library with the goal of reuse. The search query is specified by supplying the required target interface, input samples as well as the expected outputs. Candidate functions are identified and executed with the given inputs and the outputs are compared to those provided by the searcher. The authors discuss the probabilistic basis for behavior sampling based on the operational distribution of the input. They argue that with this approach it is "very unlikely" that a routine is retrieved that does not satisfy the searcher's needs.

A similar approach is employed by Mittermeir [79] for classifying components on the basis of tuples with a characteristic pair of input and output. The components are organized in a classification tree according to the I/O-tuples. The user is subsequently prompted with a choice between two tuples requiring to choose according to the desired behavior. The search process ends when only components remain that provide the requested effect.

Static Similarity Detection There has been significant research dedicated to detecting syntactically similar code fragments (called *code clones*). An overview of the research in the area of program representation based clone detection can be found in the survey from Koschke [59]. Clone detectors, such as CCFinder [55], CloneDetective [51] and Deckard [48] are effective in finding clones created by *copy&paste programming*. However, in [50], Juergens et al. showed that code with similar behavior from independent origin cannot be detected with clone detection.

A number of approaches have been proposed that aim at statically detecting similarity of code which was not caused by copy&paste. These similarities of independent origin are also referred to as *semantic clones* [60] or *type-4 clones* [94].

In [97], Sager et al. propose an approach for detecting similar Java classes using tree algorithms for understanding software evolution steps, *e.g.*, how classes changed between two releases. The comparison of classes is based on an intermediate representation. For this purpose, the abstract syntax tree of a class is constructed and converted into a language-independent information interchange model. These tree structured models are then compared with tree similarity analysis. They evaluate various tree similarity algorithms, from which the tree edit distance yielded the best results.

In [58], Kim et al. use a semantic-based static analyzer for detecting semantic clones. Their approach compares abstract memory states for identifying semantic clones. An abstract memory state maps abstract memory addresses to guarded values. Thereby, a guarded value is a set of pairs with a guard and a symbolic value. The guard represents the condition under which the address has that value. The clone detection process consists of clone candidate identification, the computation of abstract memory states for all candidates, and the computation of the similarity between memory states.

In [57], Kawrykow and Robillard propose a method for detecting "imitations" of API methods. Their approach detects client code that reimplements methods available in the API of the types the code is using. A method is represented by the set of program elements it is using. More specifically, for each method, it is determined, which fields, methods and types are referenced by its body. On

this abstraction, they define a potential replacement relation that allows them to automatically detect code that could be replaced with calls to existing library methods.

Kuhn et al. [62] introduced an approach for identifying "topics" in source code. They use *latent semantic indexing* and analyze linguistic information in identifiers and comments for grouping source code artifacts that are semantically related. Their heuristic is to cluster artifacts that use similar vocabulary. They call this technique *Semantic Clustering* and their goal is to support program comprehension activities. As the result of their case study they conclude that their approach "provides a useful first impression of an unfamiliar system".

McMillan et al. [73] suggested a method for detecting similar software applications. They use the concept of *semantic anchors* given by calls to API methods to precisely define the semantic characteristics of an application. The authors argue that the method calls to widely-used APIs have precise semantics and can thus effectively be used to determine the degree of similarity between programs. Similar to Kuhn et al., they use latent semantic indexing to reduce the number of dimensions in the similarity space.

Dynamic Similarity Detection In [49], Jiang and Su employ random testing to find functionally similar code fragments, where the term functional similarity is based on input and output behavior. Therefore, program states are not considered. Their approach uses a code chopper that extracts code fragments as candidates for function comparison. A code transformer turns the code fragments into compilable and executable units. An input generator generates random values for all input variables of a code fragment. The code fragments are clustered by separating fragments with different outputs for the same input. Finally, a code filter reduces false-positives, *e.g.*, by filtering code fragments with too much overlap in source code lines. The authors applied their approach to the Linux kernel and detected 32,996 clusters of functionally similar code fragments which in total account for 624,000 lines of C code.

Problem Existing theoretical work provides valuable insights into the problem of program equivalence and the theoretical limitations associated with its undecidability. However, it provides little guidance on how functional similarity could be detected automatically in real-world software systems. Other related work focuses either on different use cases (*e.g.*, code retrieval), is limited to representation-based similarity (classic clone detection or specific cases of imitated API methods) or targets high-level similarity. The only dynamic approach is targeted at procedural C systems.

We currently do not know if and how functional similarity in object-oriented systems can be detected automatically. It is unclear if the existing approach proposed for procedural systems can be adapted to detect functionally similar code in object-oriented systems.

Contribution Chapter 5 presents the transfer of an existing approach for detecting functional similarity in procedural programs to the field of object-oriented programs. It reports on a case study with 5 open source systems and an artificial set of highly similar programs and discusses the challenges faced with the approach. It identifies potential causes for the low detection rates and gives important directions for future research.

3.3 Developer Support for Library Reuse

The use of large software libraries poses a considerable cognitive challenge to software developers [25]. An illustrative example is shown in Figure 3.1, which shows how the state-of-the-art Java IDE Eclipse[1] allows browsing the Java Standard class library. As can be seen, the developer faces an overwhelming number of API entities.

Two major directions have been proposed to increase API accessibility, thus fostering reuse: classic code retrieval and code recommendation systems. Code retrieval allows querying code entities from code repositories. Prominent examples are keyword-based code search engines on the Internet such as Koders[2]. Advanced code retrieval approaches use signatures [116], specifications [24], test cases [43], or combinations thereof [90] to search for reusable entities. These methods, however, require users "to derive abstractions of what they actually want in order to find the artifacts that are potentially useful" [76]. On the language level, the *vocabulary problem* [27] often hampers the effectiveness of the listed tools, as developers unfamiliar with an API may formulate their query in a terminology differing from the one employed by the API. Recent approaches address this issue by enabling free-form natural language queries. SNIFF [14], a Java search engine, enriches the API with its documentation. Apatite [22] provides an interface for associative browsing of the Java and Eclipse APIs and supports the developer by additionally providing API items frequently related to their query. Hill et al. [38] propose an approach for contextual code search, which enriches query results with natural language phrases drawn from the context of the proposed method.

Despite their increase in effectiveness, code retrieval approaches fail to overcome a considerable obstacle: the user has to abandon the current task to actively formulate a query, which disrupts his workflow. Code recommendation systems address this issue by using the current development context to automatically extract queries and recommend code entities that may be useful for the task at hand. This can even happen proactively, *i.e.*, without the user anticipating the existence of a relevant code entity. A general introduction to recommendation systems in software engineering is given by Robillard et al. in [93]. While the recommended objects of recommendation systems in software engineering can be very diverse, including people and tools [31], we mainly focus on the recommendation of code.

In the remainder of this section, we summarize approaches that assist developers in using libraries and their APIs more effectively.

Classic IDE Code Completion Code completion tools (or code assists) offer a list of available variables, types and methods during code editing in an IDE as illustrated in Figure 3.2, which shows the code assist of the Eclipse IDE. Partially entered names are used to narrow down the proposed completions. The focus is on syntactical applicability and visibility of the suggestions. The order of the proposals is usually regarding the type of item (*e.g.*, method, variable, type) and furthermore alphabetical. The quality of the suggestions heavily depends on the programming language. Strongly typed languages typically enable better suggestions, since the type system allows inferring what suggestions are syntactically applicable.

[1] http://www.eclipse.org/
[2] http://www.koders.com/

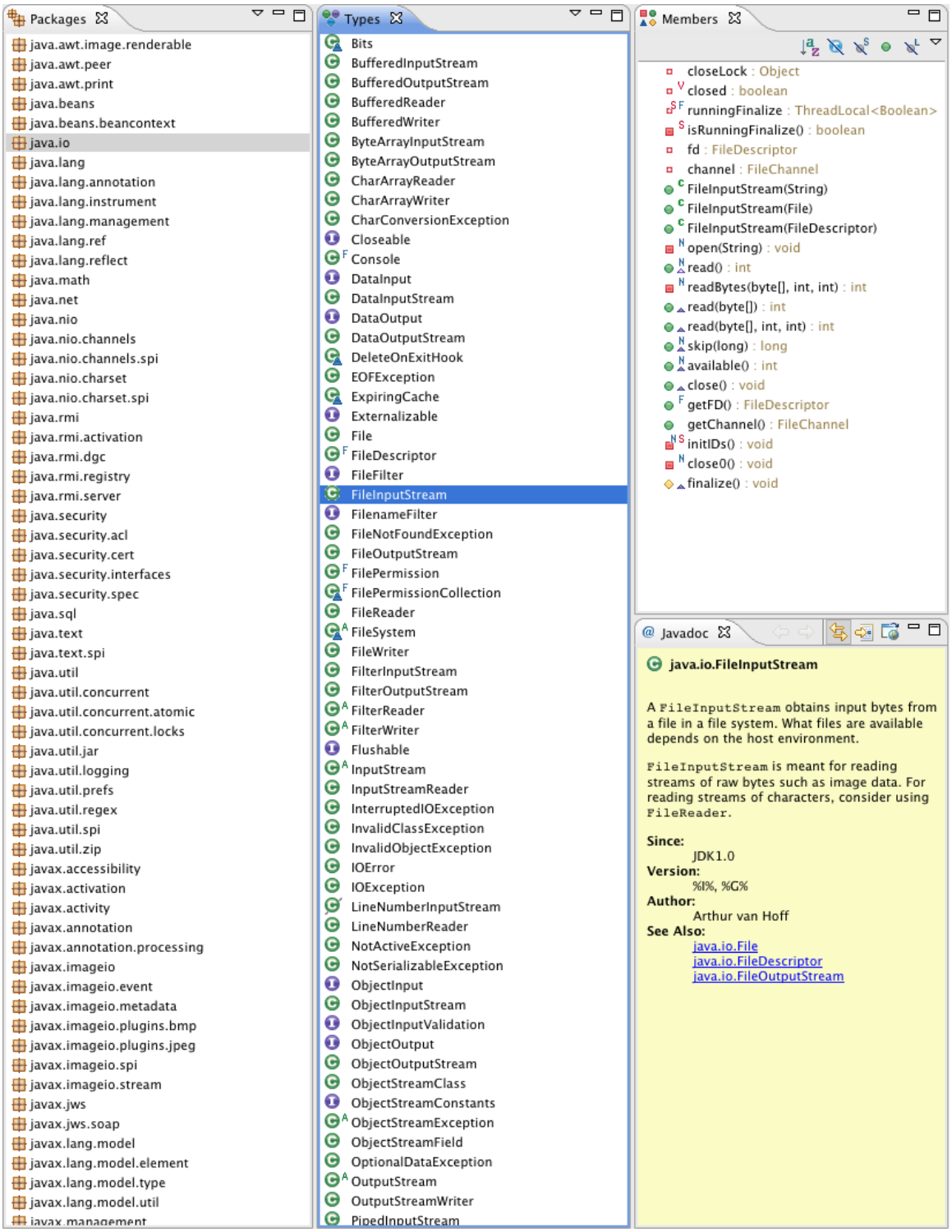

Figure 3.1: Browsing the Java standard class library in Eclipse

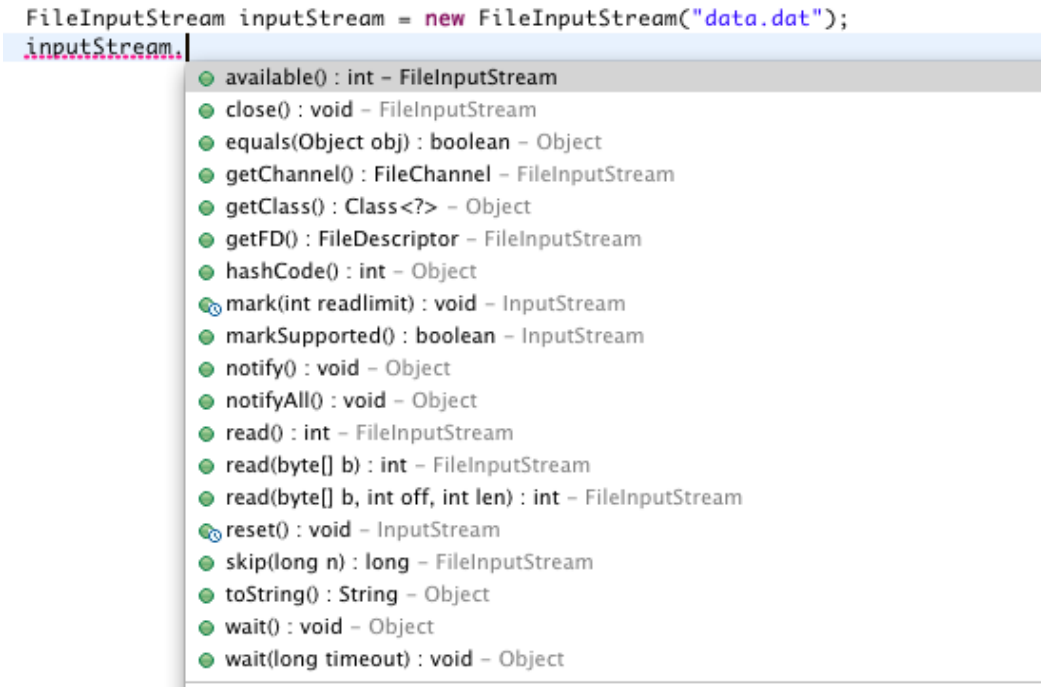

Figure 3.2: Eclipse code assist

Enhanced Code Completion Enhanced code completion approaches aim at improving "classic" code completion systems by producing more relevant completion proposals and help developers find the "right" API entity more quickly. This is usually achieved by sorting the proposals by their relevance to the current programming task instead of alphabetically.

In [12], Bruch et al. introduce an example-based code completion system. They enhance current code completion systems by making context-sensitive method recommendations. Their approach is based on mining knowledge from an example code base. They present three different methods for using the information in the code repository: frequency of method calls, association rule mining and a modification of the k-nearest-neighbor algorithm. As the approach is based on classic code completion, they recommend only methods applicable to an expression of a class type.

Robbes and Lanza [91] developed an approach that utilizes recorded program histories to improve code completion. They use different information from change-based software repositories to improve the order of completion proposals. As an example, they rank recently changed methods higher in a list of proposals, assuming that a programmer is likely to employ a method recently defined or modified.

Code Example Recommendation Systems Code example recommendation systems suggest complete code examples or snippets that illustrate the usage of an API.

Bajracharya et al. [5] proposed *structural semantic indexing* which utilizes API usage similarities extracted from code repositories to associate words with source code entities. Their goal is to improve the retrieval of API usage examples from code repositories by addressing the vocabulary

problem. This is done by sharing terms among code entities that have similar usage of APIs and are thus functionally similar.

Mapo, developed by Zhong et al. [119], mines API usage patterns from existing source code. The patterns are given by methods that are frequently called together and that follow certain sequential rules. The patterns are used for answering queries for methods by returning example snippets illustrating the usage of that method.

Holmes and Murphy introduced Strathcona [40], a tool for retrieving useful code snippets from a repository of code examples. It extracts the structure of the code under development and matches it to the code in the repository and returns code snippets similar to the code being edited. The repository can be created automatically from existing software systems. The authors developed several heuristics for structure matching of code, including inheritance and call/use relations.

Hill and Rideout [39] developed a plug-in for the Java Editor jEdit that completes a partial method by searching the code base for syntactically similar code fragments and suggesting to replace it with the best match. The search for similar code fragments is a clone detection which uses the k-nearest-neighbor algorithm based on a feature vector taking into account—among others—the length, complexity and tokens of a method.

Takuya and Masuhara [104] introduced Selene, a code recommendation tool based on an associative search engine. It uses as the query all tokens in a file being edited and searches for files with similar tokens in a code repository. The similarity measure used is based on the cosine similarity taking into account term frequencies and inverse document frequencies. Moreover, the tokens near the cursor position obtain a higher weight. In the IDE, the system then displays for each of the similar files, the code segment with a length of 20 lines that has the highest similarity with the edited code.

API Recommendation Systems API recommendation systems propose API elements, *e.g.*, methods or types, based on different aspects of the development context.

CodeBroker [113], introduced by Ye and Fischer, infers queries for reusable components from Javadoc comments and signatures of the code currently being developed. It matches the comments in the code against those of repository code. Signature matching is used to ensure the syntactic compatibility with the code context. While other recommendation systems use the code context to make recommendations, CodeBroker requires the desired functionality to be described in terms of comments and/or signatures.

Parseweb, proposed by Thummalapenta and Xie [108], and Prospector, suggested by Mandelin et al. [70], can answer queries of the form *source class type* $\rightarrow$ *destination class type*. The result is a method sequence that transforms an object of the source type to an object of the destination type. As opposed to other approaches, these tools require the source and destination type of a required method sequence to be known. Alnusair et al. [2] use an explicit ontological representation of source code for answering queries of this type. They use a model of the inter-relationships and dependencies between program elements to improve the relevance of the recommendations. They compare their tool to Parseweb, Prospector and Strathcona and obtain better results in a case study with 10 sample query scenarios.

McCarey et al. developed Rascal [71], which suggests API methods based on a set of already employed methods within a class. It mines the method usage of existing classes and uses collaborative filtering to recommend API methods for unfinished classes under development, in which classes are interpreted as users and the called methods are treated as items.

Zhang et al. [117] suggested a parameter recommendation system called Precise which is specifically targeted at recommending actual parameters for API method calls. The authors argue that many API methods require parameter values to be passed and that most of the values that are actually passed are complex and thus cannot be recommended by existing code completion tools. Precise mines existing software source code and recommends parameters in new development contexts using the k-nearest neighbor approach.

Javawock from Tsunoda et al. [109] employs an approach similar to Rascal. The main difference is that the items for the collaborative filtering are API classes instead of methods. Based on an incomplete Java program and its API class usage, the system suggests additional API classes for use. The authors also investigate *item-based* collaborative filtering in addition to the *user-based* variant done by Rascal.

Code Conjurer [44] by Hummel et al. automatically retrieves reusable components based on JUnit test cases written in advance according to the test-driven development methodology. The test cases are written as if the desired component already existed. The tool then extracts the signature of the "virtual component" and matches it against the component repository to obtain a set of syntactically compatible candidate components. The previously implemented test cases are then executed on these candidates. Components passing the tests are returned as recommendations.

Duala-Ekoko and Robillard [20] proposed an approach that makes it easier for developers to find API types and methods that are not accessible from the type the developer is currently working with (as proposed by "classic" code assists). They argue that useful methods are often located in helper types and are not easily discovered by developers unfamiliar with the API. They enhance the Eclipse code assist by using the structural relationships of API elements to make available additional useful types and methods that are not reachable from the type being used.

Zheng et al. [118] address the use case, in which an old library is replaced with a new library. Thereby, the task for the developer knowing the old library is to identify counterparts of API elements of the old library within the new library. Their approach utilizes results from Web search to recommend related API elements of different libraries. They construct a Web query by concatenating the API element in the old library and the name of the new library. They evaluate their prototype with the JDK and .NET libraries, recommending corresponding classes for two examples and show that sensible recommendations are produced.

Other API Usage Recommendation Approaches Mileva et al. [75] proposed an approach that analyzes the API usage of existing projects and extracts the usage frequency of API entities (*e.g.*, classes) over time. If the collective usage of an API element is decreasing, they may infer that the item is problematic, *e.g.*, exhibits a defect. Their goal is to utilize the *wisdom of the crowds* to produce recommendations expressing that projects are increasingly avoiding a particular API element, which may trigger an improvement to the user's code base by removing the references to the problematic element. In contrast to other recommendation systems, their system aims at

dissuading users from using certain API elements. In a case study with 200 projects they found that the past usage trend of an API entity can indeed be used to predict its usage in the future and that their recommendations are thus sensible.

Enhanced API Browsing Holmes and Walker [41] use API usage analysis to assist both API producers and consumers. For API producers it might be interesting to know if an API is used as intended. For API consumers it is useful to know which parts of a large API are more relevant than others, thus enabling a more directed search for API elements. They present a tool called "PopCon" which allows querying and browsing an API, presenting "popular" API elements more prominently (*i.e.*, at the beginning of a list of elements) than less frequently used items. They determine the popularity of API elements by analyzing the number of uses among a corpus of software projects using the API.

Problem Existing approaches for addressing the cognitive challenges due to the information overload originating from large and complex APIs are subject to several limitations:

- Classic code search approaches require the developers to derive abstractions of what they are looking for and actively formulate a query.
- API browsing techniques are context-unaware and thus disregard the context in which the developer is currently working.
- Code completion focuses on syntactically applicable API entities and thus potentially misses entities relevant to the problem at hand.
- Example recommendation systems suggest code snippets. The developer has to manually identify the relevant API entities within the recommended snippet.
- Existing API recommendation systems focus on the structural context, as given for instance by type usage or inheritance relations, to derive recommendations. However, in case of general purpose types these approaches are not effective.

We lack a recommendation system that suggests API elements based on the development context within the IDE that is not dependent on structural context such as the usage of API-specific types or methods.

Contribution Chapter 6 introduces an API method recommendation approach that suggests API methods based on the identifiers used in the code that is being edited in the IDE. The recommendation system is evaluated in a case study with 9 open source Java systems. The results of the evaluation indicate that the recommendations are better compared to an existing structure-based approach.

3.4 Recommendation Systems for Model-Based Development

Current work on recommendation systems in software engineering focuses on code-based development. We discuss the related research areas *model search* and *model clone detection*. These topics are related to model recommendation systems, since they share the common problem of defining and automatically determining similarity of models or model fragments. The individual notions of similarity differ notably among the approaches, depending on the use case.

Content-Based Model Search In content-based model search, the task is to find models in a repository that are similar to a given query model (fragment). Therefore, a notion of similarity between (partial) models is required. Existing approaches [9, 19] use different graph-matching techniques to retrieve similar models.

Model Clone Detection The goal of model clone detection is to detect duplicated models or model parts originating from copy&paste editing operations. As for content-based model search, a notion of similarity for models is needed. Deissenboeck et al. [18] introduced a model clone detection approach for models based on data-flow graphs. To avoid the exponential complexity of the maximum common subgraph problem, they developed a heuristic algorithm for finding clone pairs. Liu et al. [68] use a suffix tree-based algorithm to detect duplications in UML sequence diagrams.

Problem Current recommendation systems in software engineering are limited to code-based development. Related approaches target different use cases, such as model search or model clone detection.

We thus need approaches to tackle the cognitive problem of large modeling libraries in model-based development. As with code-based development, it is a challenge for developers to find a useful modeling element for a specific task at hand.

Contribution Chapter 7 presents a novel approach for the context-dependent recommendation of model elements from large modeling libraries during model-based development. The chapter presents an evaluation of two variants of the approach applied to the Simulink modeling language.

4 Empirical Study on Third-Party Code Reuse in Practice

Practitioners and researchers alike are disappointed by the failure of reuse in form of a software components subindustry as imagined by McIlroy over 40 years ago [72]. However, we observe that reuse of existing third-party code is a common practice in almost all software projects of significant size. Software repositories on the Internet provide solutions for many recurring problems in the form of code, frameworks and libraries. Popular examples are the frameworks for web applications provided by the Apache Software Foundation[1] and the Eclipse platform for the development of rich client applications[2]. Search engines like Koders[3] provide search capabilities and direct access to millions of source code files written in a multitude of programming languages. Open source software repositories like Sourceforge[4], hosting more than 300,000 projects, offer the possibility for open source software projects to conveniently share their code with a world-wide audience.

Despite the widely recognized importance of software reuse and its proven positive effects on quality, productivity and time to market [46, 66], it remains largely unknown to what extent current software projects make use of the extensive reuse opportunities provided by code repositories on the Internet. Literature is scarce on how software reuse occurs in software projects. We consider this lack of empirical knowledge about the extent and nature of software reuse in practice problematic and argue that a solid basis of data is required in order to assess the success of software reuse.

This chapter presents quantitative data on third-party code reuse in 20 open source projects, which was acquired with different types of static code analysis techniques. The data describes the reuse rate (*ratio between reused code and newly developed code*) of each project, the ratio between white-box reuse (*internals revealed, potential modifications*) and black-box reuse (*internals hidden, no modifications*) as well as a categorization of the functionality that is reused. The provided data helps to substantiate the discussion about the success or failure of software reuse and supports practitioners by providing them with a benchmark for software reuse in 20 successful open source projects. The study results emphasize the role of library reuse in software projects and demonstrates the reuse potential given by the available libraries.

Parts of the content of this chapter have been published in [35].

[1] http://www.apache.org/
[2] http://www.eclipse.org/
[3] http://www.koders.com/
[4] http://sourceforge.net/

4.1 Terms

This section introduces the fundamental terms for the study presented in this chapter.

Software Reuse In the scope of this study, software reuse is considered as the utilization of code developed by third parties besides the functionality provided by the operating system and the programming platform. We distinguish between two reuse strategies, namely *black-box* and *white-box* reuse. Our definitions of these strategies follow the notions of Ravichandran and Rothenberger [88]:

White-box Reuse We consider the reuse of code as white-box reuse, if it is incorporated in the project files in source form, *i.e.*, the internals of the reused code are exposed to the developers of the software. This implies that the code may potentially be modified.

Black-box Reuse We consider the reuse of code as black-box reuse, if it is incorporated in the project files in binary form, *i.e.*, the internals of the reused code are hidden from the developers and maintainers of the software. This implies that the code is reused *as is*, *i.e.*, without modifications.

Reuse Rate The reuse rate is defined as the ratio between the amount of reused code and the overall amount of code. The reuse rate for white-box reuse is defined as the ratio between the amount of reused lines of source code and the total amount of lines of code (incl. reused source code). For black-box reuse, the reuse rate is given by the ratio between the size of the reused binary code (in bytes) and the size of the binary code of the whole software system (incl. reused binary code).

4.2 Study Design

This section presents the design of the empirical study consisting of the goal of the study and the research questions that were investigated.

4.2.1 Study Goal

We use the goal definition template proposed by Wohlin et al. in [112] for defining the research objective of this study:

We analyze	*open source projects*
for the purpose of	*understanding the state of the practice in software reuse*
with respect to its	*extent and nature*
from the viewpoint of	*the developers and maintainers*
in the context of	*Java open source software.*

4.2.2 Research Questions

To achieve the study goal, we chose a set of Java open source systems as study objects and investigated the following four research questions.

RQ 1 *Do open source projects reuse software?*

The first research question of the study asks whether open source projects reuse software at all, according to our definition.

RQ 2 *What is the extent of white-box reuse?*

For those projects that do reuse existing software, we ask how much of the code is reused in a white-box fashion, as defined in Section 4.1. We use as metrics the number of copied lines of code from external sources as well as the reuse rate for white-box reuse.

RQ 3 *What is the extent of black-box reuse?*

We further ask how much of the code is reused in a black-box fashion according to our definition in Section 4.1. For this question, we use as metrics the number and accumulated size of all included libraries, the aggregated byte code size of the reused classes from these libraries, and the reuse rate for black-box reuse. Although not covered by our definition of software reuse (see Section 4.1), we separately measure the numbers for black-box reuse of the Java API, since one could argue that this is also a form of software reuse.

RQ 4 *What type of functionality is reused?*

We investigate functional categories in which the reused code can be divided and determine how much code is reused from each category. Here, our metrics are the aggregated byte code size of the reused classes within each of the categories per project and the overall distribution of the reused code among the categories.

4.3 Study Objects

This section describes how we selected the projects that were analyzed in the study and how they were preprocessed in advance to the analyses.

Selection Process

We chose 20 projects from the open source software repository Sourceforge[5] as study objects. We used the following procedure for selecting the study objects with the web-based search and browsing interface of the Sourceforge repository[6]: We searched for Java projects with the development status *Production/Stable*. We then sorted the resulting list by the number of weekly downloads in descending order. We stepped through the list beginning from the top and selected each project that was a standalone application, purely implemented in Java, based on the Java SE Platform and had a source download. All of the 20 study objects selected by this procedure were among the 50 most downloaded projects. Thereby, we obtained a set of successful projects in terms of user acceptance. The application domains of the projects were diverse and included accounting, file sharing, e-mail, software development and visualization. The size of the downloaded packages (zipped files) had a broad variety, ranging from 40 KB to 53 MB.

Table 4.1 shows overview information about the study objects. The column *LOC* denotes the total number of lines in Java source files in the downloaded and preprocessed source package as described below. The column *Size* shows the size of the byte code.

Table 4.1: The 20 studied Java applications

System	Version	Description	LOC	Size (KB)
Azureus/Vuze	4504	P2P File Sharing Client	786,865	22,761
Buddi	3.4.0.3	Budgeting Program	27,690	1,149
DavMail	3.8.5-1480	Mail Gateway	29,545	932
DrJava	20100913-r5387	Java Programming Env.	160,256	6,199
FreeMind	0.9.0 RC 9	Mind Mapper	71,133	2,352
HSQLDB	1.8.1.3	Database Engine	144,394	2,032
iReport-Designer	3.7.5	Visual Reporting Tool	338,819	10,783
JabRef	2.6	BibTeX Manager	109,373	3,598
JEdit	4.3.2	Text Editor	176,672	4,010
MediathekView	2.2.0	Media Management	23,789	933
Mobile Atlas Creator	1.8 beta 2	Atlas Creation Tool	36,701	1,259
OpenProj	1.4	Project Management	151,910	3,885
PDF Split and Merge	0.0.6	PDF Manipulation Tool	411	17
RODIN	2.0 RC 1	Service Development	273,080	8,834
soapUI	3.6	Web Service Testing Tool	238,375	9,712
SQuirreL SQL Client	20100918_1811	Graphical SQL Client	328,156	10,918
subsonic	4.1	Music Streamer	30,641	1,050
Sweet Home 3D	2.6	Interior Design	77,336	3,498
TV-Browser	3.0 RC 1	TV Guide	187,216	6,064
YouTube Downloader	1.9	Video Download Utility	2,969	99
Overall			**3,195,331**	**100,085**

[5] http://sourceforge.net/

[6] the project selection was performed on October 5th, 2010

Preprocessing

We deleted test code from the projects following a set of simple heuristics (*e.g.*, folders named test/tests). In few cases, we had to remove code that was not compilable. For one project, we omitted code that referenced a commercial library.

We also added missing libraries that we downloaded separately in order to make the source code compilable. We either obtained the libraries from the binary package of the project or from the library's website. In the latter case we chose the latest version of the library.

4.4 Study Implementation and Execution

This section details how the study was implemented and executed on the study objects. All automated analyses were implemented in Java on top of the open source quality analysis framework ConQAT (see Section 2.8), which provides—among others—clone detection algorithms and basic functionality for static code analysis.

Detecting White-Box Reuse

As white-box reuse involves copying external source code into the project's code, we used a combination of clone detection and manual code inspections for its detection. Clone detection tools can automatically find duplicated code fragments caused by copy&paste programming.

We used the clone detection algorithm presented in [42] to find duplications between a selection of libraries and the study objects. The potential sources of white-box reuse are not limited to libraries available at compile time, but can virtually span all existing Java source code. The best approximation of *all existing Java source code* is probably provided by the indices of the large code search engines, such as Koders[7]. Unfortunately, access to these engines is typically limited and does not allow to search for large amounts of code, such as the 3 MLOC of our study objects. Consequently, we only considered a selection of commonly used Java libraries and frameworks as potential sources for white-box reuse. We selected 22 libraries that are commonly reused based on our experience with both own development projects and systems we analyzed during earlier studies. The libraries are listed in Table 4.2 and comprise more than 6 MLOC. For the sake of presentation, we treated the Apache Commons as a single library, although it consists of 39 individual libraries that are developed and versioned independently. The same holds for Eclipse, where we chose a selection of its plug-ins.

We configured the clone detection to compute all clones consisting of at least 15 statements in which formatting and identifier names were normalized. This allowed us to also find partially copied files (or files that are not fully identical due to further independent evolution), while keeping the rate of false-positives low. All clones reported by our tool were also inspected manually to remove any remaining false-positives.

[7] http://www.koders.com/

Table 4.2: The 22 libraries used as potential sources for white-box reuse

Library	Description	Version	LOC
ANTLR	Parser Generator	3.2	66,864
Apache Ant	Build Support	1.8.1	251,315
Apache Commons	Utility Methods	5/Oct/2010	1,221,669
log4j	Logging	1.2.16	68,612
ASM	Byte-Code Analysis	3.3	3,710
Batik	SVG Rendering and Manipulation	1.7	366,507
BCEL	Byte-Code Analysis	5.2	48,166
Eclipse	Rich Platform Framework	3.5	1,404,122
HSQLDB	Database	1.8.1.3	157,935
Jaxen	XML Parsing	1.1.3	48,451
JCommon	Utility Methods	1.0.16	67,807
JDOM	XML Parsing	1.1.1	32,575
Berkeley DB Java Edition	Database	4.0.103	367,715
JFreeChart	Chart Rendering	1.0.13	313,268
JGraphT	Graph Algorithms and Layout	0.8.1	41,887
JUNG	Graph Algorithms and Layout	2.0.1	67,024
Jython	Scripting Language	2.5.1	252,062
Lucene	Text Indexing	3.0.2	274,270
Spring Framework	J2EE Framework	3.0.3	619,334
SVNKit	Subversion Access	1.3.4	178,953
Velocity Engine	Template Engine	1.6.4	70,804
Xerces-J	XML Parsing	2.9.0	226,389
Overall			**6,149,439**

As the body of potential sources for white-box reuse used in the clone detection is very limited, we complemented it with a manual inspection of the source code of all study objects, searching the source code files for indication whether some of them have been copied from external sources. The size of the study objects only allows a very shallow inspection based on the names of files and directories (which correspond to Java packages). For this, we scanned the directory trees of the projects for files residing in separate source folders or in packages that were significantly different from the package names used for the project itself. The files found this way were then inspected and their origin identified based on header comments or a web search. Of course, this step can only find large scale reuse, where multiple files are copied into a project and the original package names are preserved (which are typically different from the project's package names). However, during this inspection we are not limited to the 22 selected libraries, but potentially can find other reused code as well.

Detecting Black-Box Reuse

The primary way of black-box reuse in Java programs is the inclusion of class libraries. Technically, these are Java Archive Files (JAR), which are zipped files containing the byte code of the Java

classes. First, to obtain an overview of the extent of black-box reuse, we counted the number of JAR files included in the files of each project and computed their accumulated size in bytes. Second, we determined the black-box reuse rate on the class level, *i.e.*, we computed the accumulated size of the classes that were actually reused by a project. Ideally, one would measure the reuse rate based on the source code of the libraries. However, obtaining the source code for such libraries is error-prone as many projects do not document the exact version of the libraries used. In certain cases, the source code of libraries is not available at all. To avoid these problems and prevent measurement inaccuracies, we performed the analysis of black-box reuse directly on the Java byte code stored in the JAR files.

While JAR files are the standard way of packaging reusable functionality in Java, the JAR files themselves are not directly reused. They merely represent a container for Java types (classes, interfaces, enumerations and annotations) that are referenced by other types. Hence, the type is the main entity of reuse in Java[8]. Our black-box reuse analysis determines which types from libraries are referenced from the types of the project code. The dependencies are defined by the Java Constant Pool [67], a part of the Java class file that holds information about all referenced types. References are method calls and all type usages induced for example by local variables or inheritance. Our analysis transitively traverses the dependency graph. Therefore, also the types that are referenced by reused types are included in the resulting set of reused types. The analysis approach ensures that, in contrast to counting the whole library as reused code, only the subset that is actually referenced by the project is considered. The rationale for this is that a project can incorporate a large library but only use a small fraction of it. To quantify black-box reuse, the analysis measures the size of the reused types by computing their aggregated byte code size. The black-box analysis is technically based on the BCEL library[9], which provides byte code processing functionality.

It has to be noted that our analysis can lead to an overestimation of reuse because we always include whole types, although only specific methods of a type may actually be reused by the project's code. Moreover, a method may reference certain types but the method itself could be unreachable. On the other hand, our approach can lead to an underestimation of reuse since the implementations of interfaces are not considered as reused, unless they are discovered on another path of the dependency search. Details regarding this potential error are discussed in the section about the threats to validity (Section 4.7).

Although reuse of the Java API is not covered by our definition of software reuse (see Section 4.1), we also measured reuse of the Java API, since potential variations in the reuse rates of the Java API are worthwhile to investigate. Since every Java class inherits from `java.lang.Object` and thereby (transitively) references a significant part of the Java API classes, even a trivial Java program exhibits—according to our analysis—a certain amount of black-box reuse. To determine this *baseline*, we performed the analysis for an artificial minimal Java program that only consists of an empty `main` method. This baseline of black-box reuse of the Java API consisted of 2,082 types and accounted for about 5 MB of byte code. We investigated the reason for this rather large baseline and found that `Object` has a reference to `Class` which in turn references `ClassLoader` and `SecurityManager`. These classes belong to the core functionality for running Java applications. Other referenced parts include the Reflection API and the Collection API. Due to the special role of

[8] It is interesting to note that, in addition to JAR files, Java provides a *package* concept that resembles a logical modularization concept. Packages, however, cannot directly be reused.

[9] http://jakarta.apache.org/bcel/

the Java API, we captured the numbers for black-box reuse of the Java API separately. All black-box reuse analyses were performed with a Sun Java Runtime Environment for Linux 64 Bit in version 1.6.0.20.

Categories of Reused Software

For the 293 Java libraries collectively reused in black-box fashion by all study objects, we investigated their purpose by means of web search. In a discussion with other researchers, we consensually determined a set of functional categories. Through an agreement of at least two researchers we assigned each library to one of these categories.

For measuring the amount of reuse in each category, we extended the bytecode analysis to compute the amount of bytecode reused per Java library (technically for each JAR file). Together with the category assignment we then determined the amount of bytecode reused by the study objects in each of the categories.

4.5 Results

This section presents the results of the study separately for each research question.

RQ 1: Third-Party Reuse

The reuse analyses revealed that 18 of the 20 projects (90%) reuse software from third parties. *HSQLDB* and *YouTube Downloader* were the only projects for which no reuse—neither black-box nor white-box—was found. We investigated the reason for this by asking the developers of both projects. In case of *YouTube Downloader*, according to the developer, the need for any special reusable library did not arise, since everything needed could be found in the Java API. The developer of *HSQLDB* mentioned three reasons why no software was reused by this project. First, he mentioned that existing components do not provide the required quality characteristics (*e.g.*, performance). Concretely, he gave as an example the Collection implementations of the JDK or alternative libraries which provide an inferior memory consumption compared to the project's own implementation. The second reason mentioned was an unnecessary size increase of the software distribution, since often only parts of a library are actually used. Finally, the developer mentioned version incompatibilities as a problem of library reuse, since in user deployments, the own code may depend on a library in a different version than other code in that deployment. It has to be noted that HSQLDB itself can be employed as a reusable library.

RQ 2: White-Box Reuse

We attempt to answer this question by a combination of automatic techniques (clone detection) and manual inspections. The clone detection between the code of the study objects and the libraries

from Table 4.2 reported 337 clone classes[10] with 791 clones overall. These numbers only include clones between a study object and one or more libraries. Clones within the study objects or the libraries were not considered. As we had *HSQLDB* both in our set of study objects and the libraries used, we discarded all clones between these two.

The results for the duplicated code found by clone detection and during manual inspection are summarized in Table 4.3. The last column gives the overall amount of white-box reused code relative to the project's size in LOC. For 11 of the 20 study objects no white-box reuse could be found. For another 5 of them, the reuse rate is below 1%. However, there are also 4 projects with white-box reuse rates in the range of 7% to 10%.

Table 4.3: White-box reuse found by clone detection and manual inspection

System	Clone Detection (LOC)	Manual Inspection (LOC)	Overall (%)
Azureus/Vuze	1040	57,086	7.39%
Buddi			—
DavMail			—
DrJava			—
FreeMind			—
HSQLDB			—
iReport-Designer	298		0.09%
JabRef		7,725	7.06%
JEdit	7,261	9,333	9.39%
MediathekView			—
Mobile Atlas Creator		2,577	7.02%
OpenProj	87		0.06%
PDF Split and Merge			—
RODIN	382		0.14%
soapUI	2,120		0.89%
SQuirreL SQL Client			—
subsonic			—
Sweet Home 3D			—
TV-Browser	513		0.27%
YouTube Downloader			—
Overall	11,701	76,721	

Manual inspection of the detected clones led to the observation that typically all clones are in just a few of the files which are almost completely covered by clones. Hence, the unit of reuse (according to our findings) is the file/class level; single methods (or sets of methods) were not copied. Most of the copied files were not completely identical. These changes are caused either by minor modifications to the files after copying them to the study objects, or (more likely) due to different versions of the libraries used. As the differences between the files were minor, we counted the entire file as copied if the major part of it was covered by clones.

[10] A clone class is a set of code fragments in which every pair of the code fragments is in a clone relationship.

By manual inspection of the study objects, we found entire libraries copied in four of the study objects. These libraries were either less well-known (GNU ritopt), no longer available as individual project (microstar XML parser), or not released as an individual project but rather extracted from another project (OSM JMapViewer). All of these could not be found by the clone detection algorithm, as the corresponding libraries were not part of our original set.

RQ 3: Black-Box Reuse

Table 4.4 shows for each project the number of included libraries and their accumulated byte code size. It gives a general impression on the extent of black-box reuse in the study objects. Since we count the number of JAR files included in a project, reusable frameworks like Eclipse, which consist of a number of individual JARs, do not count as a single library but contribute with the according number of their JAR files to the library count. The number of reused libraries ranged between 0 and 74 with a mean of 15.8 and a median of 6. The accumulated byte code size of the included libraries ranged between about 2 MB and about 88 MB with a mean of about 19 MB and a median of about 9 MB.

Table 4.4: Number of included libraries per project

System	# Libraries	Accumulated Size (KB)
Azureus/Vuze	3	2,088
Buddi	12	5,240
DavMail	14	12,232
DrJava	7	10,548
FreeMind	5	25,776
HSQLDB	0	0
iReport-Designer	74	67,832
JabRef	11	5,540
JEdit	2	5,264
MediathekView	2	1,192
Mobile Atlas Creator	2	8,500
OpenProj	23	18,972
PDF Split and Merge	4	1,900
RODIN	38	61,368
soapUI	60	90,068
SQuirreL SQL Client	31	22,880
subsonic	1	29,308
Sweet Home 3D	4	9,652
TV-Browser	23	5,580
YouTube Downloader	0	0
Overall	**316**	**383,940**

Figure 4.1 illustrates the absolute bytecode size distributions between the project code (own), the reused parts of the libraries (3rd party) and the Java API in descending order of the total amount of bytecode. The horizontal line indicates the baseline usage of the Java API. The reuse of third-party

libraries ranged between 0 MB and 42.2 MB. The amount of reuse of the Java API was similar among the analyzed projects and ranged between 12.9 MB and 16.6 MB. The median was 2.4 MB for third-party libraries and 13.3 MB for the Java API. The project *iReport-Designer* reused the largest amount of functionality in a black-box fashion both from libraries and the Java API. The project with the smallest extent of black-box reuse was *YouTube Downloader*.

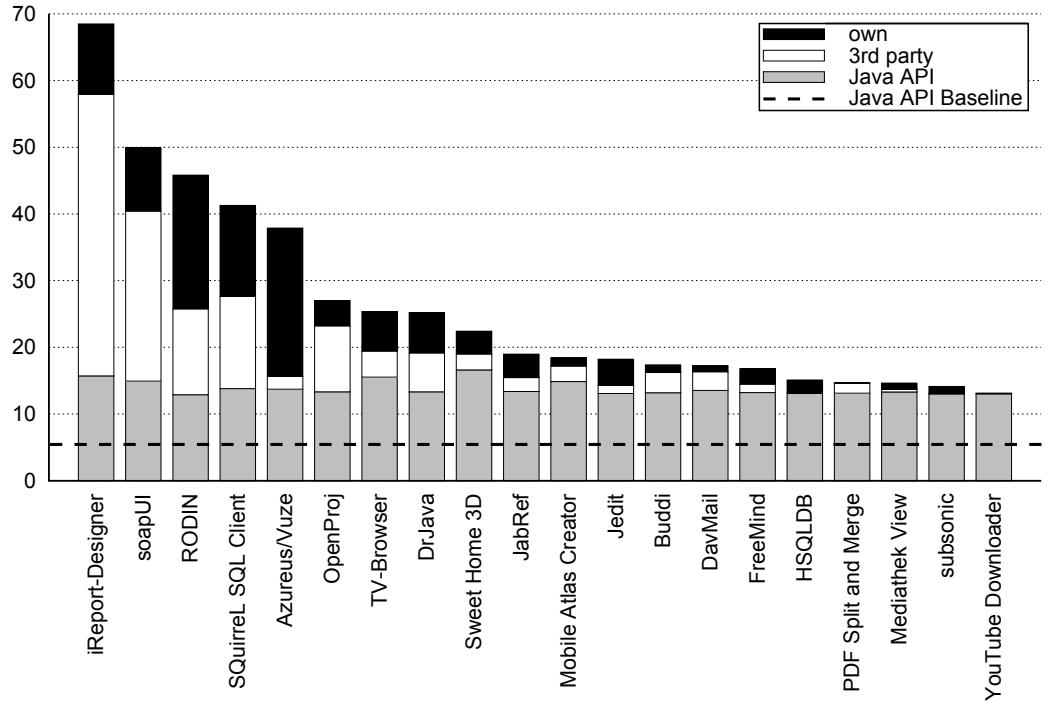

Figure 4.1: Absolute bytecode size distribution (MB)

Figure 4.2 is based on the same data but shows the relative distributions of the bytecode size. The projects are ordered descending by the total amount of relative reuse. The relative reuse from third-party libraries was between 0% and 61.7% with a median of 11.8%. The relative amount of reused code from the Java API ranged between 23.0% and 99.3% with a median of 73.0%. Overall (third-party and Java API combined), the relative amount of reused code ranged between 41.3% and 99.9% with a median of 85.4%. The project *iReport-Designer* had the highest black-box reuse rate. *YouTube Downloader* used the most code from the Java API relative to its own code size. For 19 of the 20 projects, the amount of reused code was larger than the amount of own code. Of the overall amount of reused code in the sample projects, 34% stemmed from third-party libraries and 66% from the Java API.

Figure 4.3 illustrates the relative byte code size distributions between the own code and third-party libraries, without considering the Java API as a reused library. The projects are ordered descending by reuse rate. The relative amount of reused library code ranged from 0% to 98.9% with a median of 45.1%. For 9 of the 20 projects the amount of reused code from third-party libraries was larger than the amount of own code.

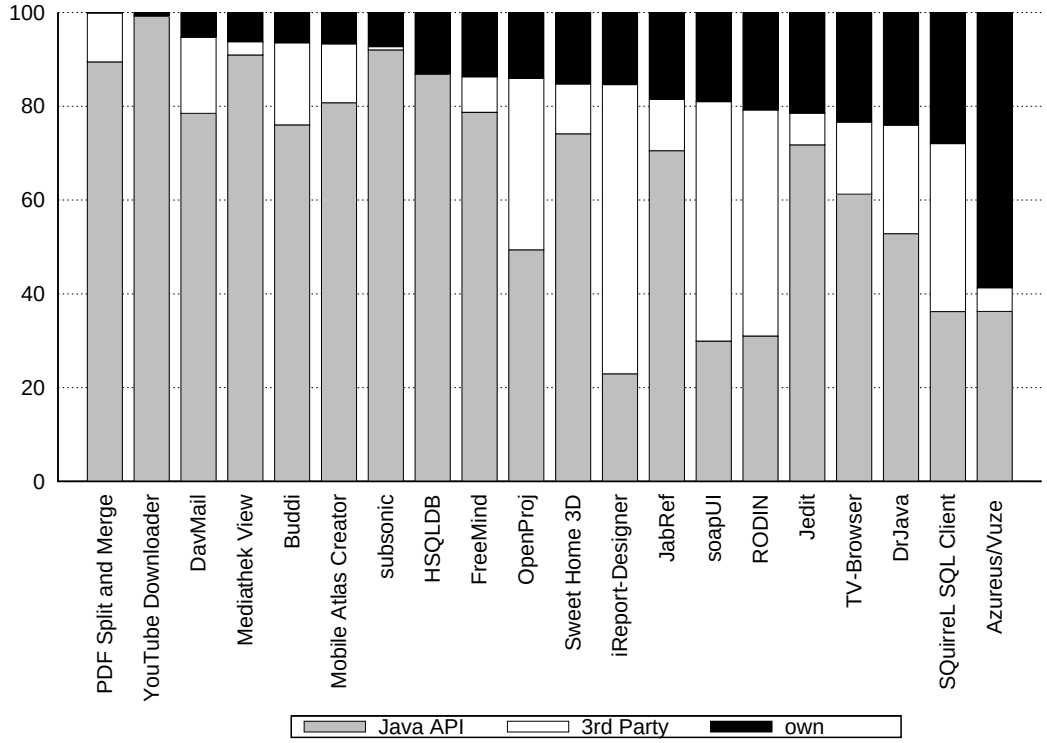

Figure 4.2: Relative bytecode size distribution (%)

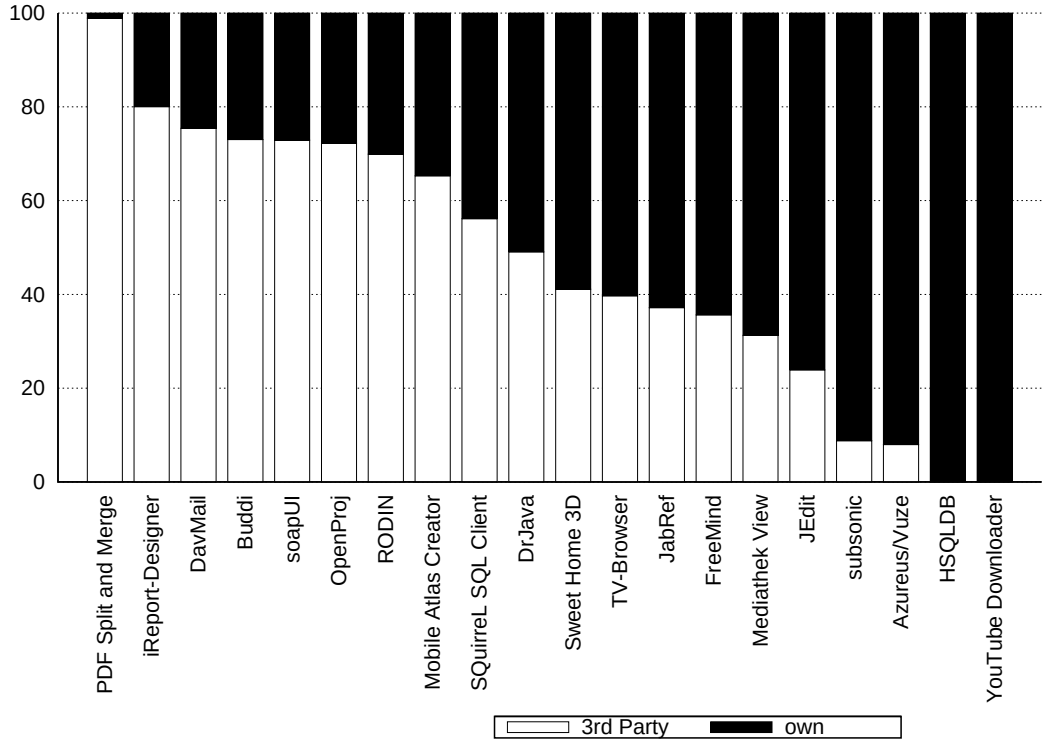

Figure 4.3: Relative bytecode size distribution (%) without Java API

RQ 4: Types of Reused Functionality

The set of functional categories for reused libraries that we developed is illustrated in Table 4.5. This categorization reflects a high-level view according to typical technical domains of software applications. To avoid fragmentation, we introduced a category named *Other* for libraries with very specific purposes that did not warrant a separate category.

Table 4.5: Categories overview

Category	Typical purpose of libraries	Examples
RCP	Platform libraries for rich client development	NetBeans, Eclipse
Persistency	Database connectivity/abstraction, file handling	Hibernate, SQL Leonardo
Networking	Communication, data transfer, protocols	WSDL4J, Apache Axis
Text/XML	Text processing, XML validation/manipulation	CSS Parser, dom4j
UI/graphics	Widget handling, layout, graphics, image proc.	Eclipse SWT, SwingX
Misc	Utilities, frequently used data types/algorithms	JCommon, Log4j
Other	Specific; not warranting an own category	Event-B, BeanShell

Since the white-box analysis revealed only small amounts of reuse, we limited the categorization to the code reused in a black-box manner.

Figure 4.4 shows the overall distribution of the code collectively reused by all study objects according to the categories. The analysis revealed that most of the reused software can be found in the categories *Text/XML*, *RCP* and *UI/graphics*.

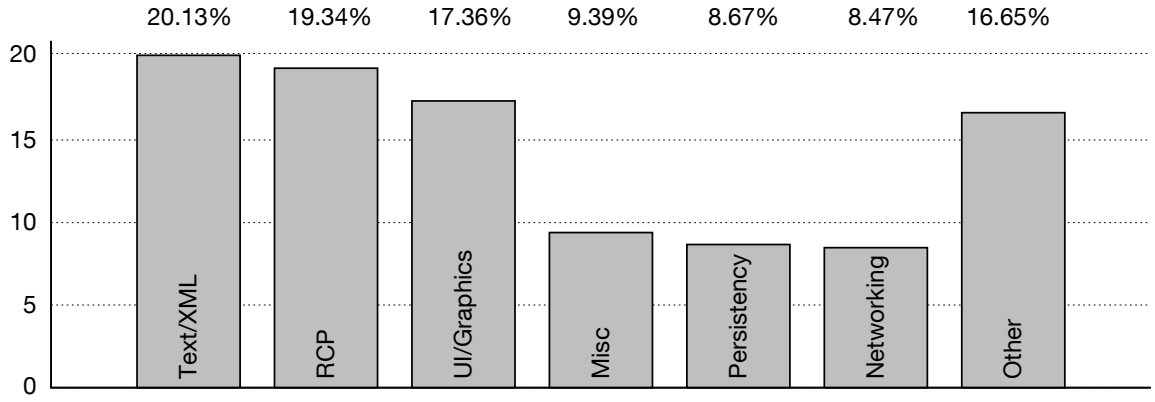

Figure 4.4: Overall partitioning of black-box reuse by category (%)

Figure 4.5 illustrates the partitioning of black-box reuse into the functional categories for each study object. The two study objects that did not exhibit any black-box reuse (*YouTube Downloader* and *HSQLDB*) have been omitted. The diagram shows that 6 of the remaining 18 study objects reuse code out of at least 5 of the 7 categories. Both, *iReport-Designer* and *SQuirreL SQL Client* reuse code from all 7 categories.

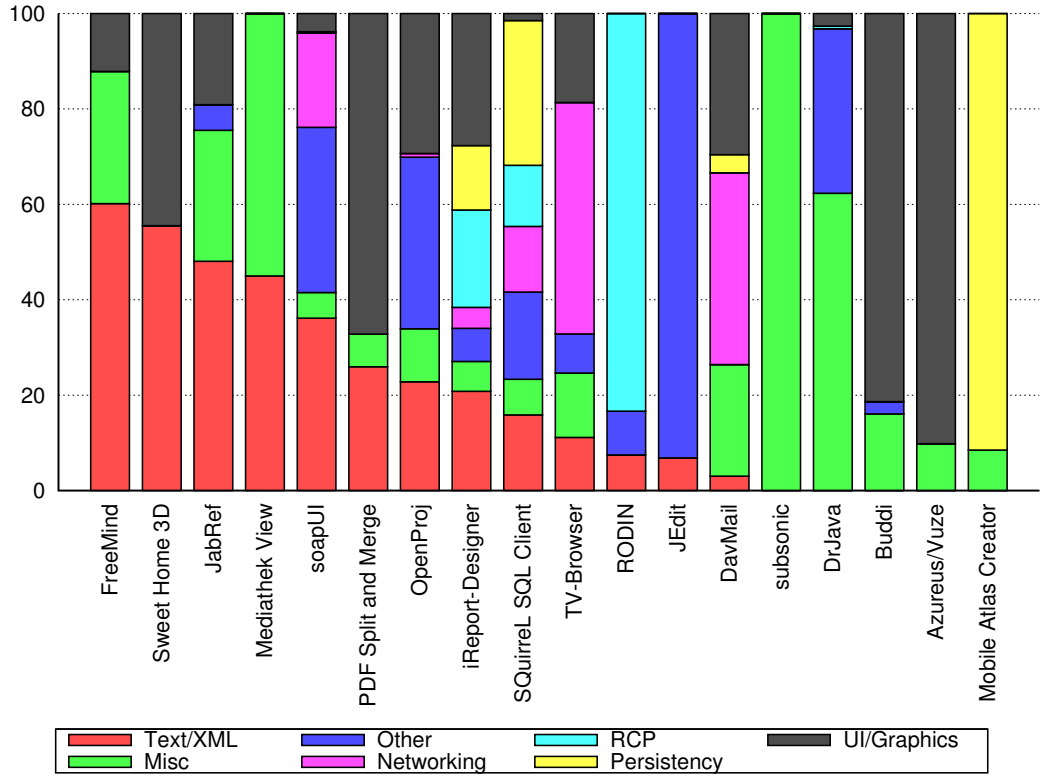

Figure 4.5: Relative partitioning of black-box reuse by category (%)

4.6 Discussion

The data presented in the previous sections lead to interesting insights into the current state of third-party code reuse in open source Java development, but also open new questions that were not part of our study setup. We discuss both in the following sections.

4.6.1 Extent of Reuse

Our study reveals that software reuse is common among open source Java projects. The results indicate that black-box reuse is the predominant form of reuse. None of the 20 analyzed projects has less than 40% black-box reuse when including the Java API. Even when not considering the Java API, the median reuse rate is still above 40% and only for 4 projects it is below 10%. In contrast, white-box reuse is only found in about half of the projects at all and never exceeds 10% of the code.

This difference can probably be explained by the increased maintenance efforts that are commonly associated with white-box reuse as described by Jacobson et al. [46] and Mili et al. [77]. The detailed results of RQ 2 also revealed that larger parts consisting of multiple files were mostly copied if either the originating library was no longer maintained or the files were never released as an individual library. In both cases, the project's developers would have to maintain the reused code in any case.

It also seems that the amount of reused third-party libraries seldom exceeds the amount of code reused from the Java API. The only projects for which this is not the case are *iReport-Designer*, *RODIN* and *soapUI*, from which the first two are built upon NetBeans[11] respectively Eclipse, which provide rich platforms on top of the Java API.

Our findings suggest that the early visions of off-the-shelf reusable components that only have to be connected by small amounts of glue code and would lead to reuse rates beyond 90% are not realistic today. On the other hand, the reuse rates we found are high enough to have a significant impact on the development effort. We would expect that reuse of software, as it is also fostered by the free/open source movement, has a considerable influence to the rich set of applications available today.

4.6.2 Influence of Project Size on Reuse Rate

The amount of reuse varies significantly between the different study objects. While *PDF Split and Merge* is just a very thin wrapper around existing libraries, there are also large projects that have (relatively) small reuse rates. An example is *Azureus* which has a reuse rate of less than 10% without counting the Java API.

Motivated by a study by Lee and Litecky [65], we investigated a possible correlation between code size and reuse rate in our data set. Their study was based on a survey in the domain of commercial Ada development on 73 samples and found a *negative influence* of software size on the rate of reuse. We used the Spearman correlation coefficient [100], which is commonly employed in statistics to measure a statistical dependence between two variables. To compute the correlation for two variables, the values are converted to ranks according to their order. The computation of the correlation coefficient for two variables X and Y with according ranks x_i and y_i is computed as follows ($\bar{x}$ and $\bar{y}$ denote the mean of all x_i and y_i respectively):

$$\rho = \frac{\sum_i (x_i - \bar{x}) \times (y_i - \bar{y})}{\sqrt{\sum_i (x_i - \bar{x})^2 \times (y_i - \bar{y})^2}}$$

Table 4.6 shows the data for the Spearman correlation analysis. It contains three pairs of columns, in which each pair consists of the values and the according rank in ascending order. We correlate the project's own code in LOC with (1) the reuse rate including the JDK and (2) the reuse rate excluding the JDK.

For the reuse rate without the Java API (only third-party code), we found a Spearman correlation coefficient of 0.05 with the size of the project's own code (two-tailed p-value: 0.83). Thus, we can infer no dependence between these values. If we use the overall reuse rate (including the Java API), the Spearman coefficient is -0.93 (p-value < 0.0001), which indicates a significant and strong negative correlation. This confirms the results of [65] that project size typically reduces the reuse rate.

[11] http://www.netbeans.org/

Table 4.6: Data for correlation analysis

Project code (LOC)	rank	Reuse rate w/ JDK	rank	Reuse rate w/o JDK	rank
17538	1	0.998859523	20	0.989174121	20
101652	2	0.992618534	19	0	1.5
954490	3	0.947203025	18	0.754337614	18
955457	4	0.9377561	17	0.312635204	6
1075674	5	0.927364138	14	0.087788004	4
1176411	6	0.93537777	16	0.730306903	17
1288919	7	0.933242913	15	0.653001684	13
2080938	8	0.868566747	13	0	1.5
2408909	9	0.863042884	12	0.356238033	7
3581469	10	0.847607608	10	0.410948024	10
3684752	11	0.814853823	8	0.371843857	8
3977736	12	0.85950907	11	0.722422042	15
4106744	13	0.785141462	5	0.238963072	5
6209692	14	0.766357252	4	0.396879078	9
6347528	15	0.759826214	3	0.490682724	11
9046302	16	0.792345466	6	0.698959178	14
9945395	17	0.809989505	7	0.728817759	16
11041900	18	0.846251529	9	0.800447744	19
11179693	19	0.720348931	2	0.56154093	12
23307292	20	0.413104894	1	0.079545657	3

4.6.3 Types of Reused Functionality

The first observation regarding the types of functionality that is reused is that most applications reuse functionality from different categories. We conclude from this finding that the developers are aware of the positive effects of reuse in general and do not only pick libraries or frameworks for a single purpose.

The distribution of reuse across the functional categories is fairly balanced. The strongest category is *Text/XML* with 20.13%, the weakest is *Networking* with 8.47% (see Figure 4.4). Hence, the categories appear to be roughly equally important for the analyzed applications. The majority of the analyzed libraries has a technical nature such as networking or XML processing and does not address business domains like accounting or finance (domain-specific libraries were included in the category *Other* that accounts for 16.65% of the reused code). While this is obviously influenced by the selection of the analyzed systems, we still consider this finding rather astonishing. Given the amount of functionality that the Java runtime environment already provides in technical areas (particularly for XML processing), one would expect that there is little need for additional libraries.

4.7 Threats to Validity

This section discusses potential threats to the internal and external validity of the results presented in this study.

4.7.1 Internal Validity

The amount of reuse measured in our study fundamentally depends on the definition of *software reuse* and the techniques used to measure it. We discuss possible flaws that can lead to an over- or underestimation of the actual reuse or otherwise threaten our results.

Overestimation of Reuse

The measurement of white-box reuse is dependent on the results of a clone detection, which could contain false-positives. Thus, not all reported clones may indicate actual reuse. To mitigate this, we manually inspected the clones found. Additionally, for both the automatically and manually found duplicates, it is not known whether the code was copied *into* the study objects or rather *from* them. However, all findings were manually verified. For example, by checking the header comments, we ensured that the code was actually copied from the library into the study object.

Our estimation of black-box reuse is based on static references in the byte-code. We consider a class as completely reused if it is referenced, which may not be the case. For example, the method holding the reference to another class might never be called. Another possibility would be to use dynamic analysis and execution traces to determine the amount of reused functionality. However, this approach has the disadvantage that only a finite subset of all execution traces could be considered, leading to a potentially large underestimation of reuse.

Underestimation of Reuse

The application of clone detection was limited to a fixed set of libraries. Thus, copied code could be missed as the source it was taken from was not included in our comparison set. Additionally, the clone detection might miss actual clones (low recall) due to weak normalization settings (the code segments are normalized prior to comparison to be able to detect copied code that is subject to certain changes such as different formatting or identifier renaming). To address this, we chose settings that yield higher recall (at the cost of precision). Moreover, the manual inspection of the study objects' code for further white-box reuse is inherently incomplete; due to the large amounts of code, only the most obvious copied parts could be found.

The static analysis used to determine black-box reuse misses certain dependencies, such as method calls performed via Java's reflection mechanism or classes that are loaded based on configuration information. Additionally, our analysis cannot penetrate the boundaries created by Java interfaces. The actual implementations used at run time (and their dependencies) might not be included in our reuse estimate. To mitigate this, one could search for an implementing class and include the first match into the further dependency search and the result set. However, preliminary experiments

showed that this approach leads to a large overestimation. For example a command line program that references an interface that is also implemented by a UI class could lead to the false conclusion that the program reuses UI code.

There are many other forms of software reuse that are not covered by our approach. One example are reusable generators. If a project uses a code generator to generate source code from models, this would not be detected as a form of reuse by our approach. Moreover, there are many other ways in which software components can interact with each other, besides use-dependencies in the source code. Examples are inter-process communication, web services that utilize other services via SOAP calls, or the integration of a database via an SQL interface.

Categorization of Reused Code

For the categorization of the reused code, we developed our own set of categories. This choice could potentially bias the results. Additionally, our assignment of a JAR file to one of these categories could be wrong as we might have misinterpreted the purpose of the library. We tried to mitigate this by making all decisions by at least two researchers. Additionally, a third researcher made an independent categorization of a sample of 10% of the libraries (29 in total). The inter-rater agreement between the initial categorization and the sample was evaluated using Cohen's Kappa coefficient [110], which measures to what extent two raters agree when classifying items in multiple disjoint categories. The computation of Cohen's Kappa κ is computed as follows (in which $Pr(a)$ denotes the fraction of classifications on which the raters agreed and $Pr(e)$ the probability of a rater agreement by chance):

$$\kappa = \frac{Pr(a) - Pr(e)}{1 - Pr(e)}$$

For the sample of 29 libraries, there was an agreement for 24 of them. Thus, we obtain the following Kappa value:

$$\kappa = \frac{0.82 - 0.20}{1 - 0.20} \approx 0.78$$

The obtained Kappa value of $\kappa = 0.78$ indicates *substantial agreement* making us confident that we chose sensible categories. Our approach also assigns a single category to each JAR file, which might be imprecise. Often a library serves multiple purposes or contains additional code that would be categorized differently from its core. In such cases, we tried to use the category that corresponds to the majority of the code and captures the library's *intent*. Still, a more precise categorization (maybe on the type level) could lead to different results.

4.7.2 External Validity

While we tried to use a comprehensible way of sampling the study objects, it is not clear to what extent they are representative for the class of open source Java programs. First, the choice of Sourceforge as source for the study objects could bias our selection, as a certain kind of open source developers could prefer other project repositories (such as Google Code[12]). Secondly, we selected the projects from the 50 most downloaded ones, which could also bias our results.

[12] http://code.google.com/

As the scope of the study are open source Java programs, the transferability of the results to other programming languages or commercially developed software is unclear. Especially the programming language is expected to have a huge impact on reuse, as the availability of both open source and commercial reusable code heavily depends on the language used.

4.8 Summary

This chapter presented an empirical study on the extent and nature of software reuse in open source Java projects. The study analyzed the amount of software reuse in 20 software projects and found that most reuse is achieved by including libraries in a black-box fashion as opposed to copying&pasting source code. A considerable fraction of the overall amount of code in many of these systems is reused from software libraries. Moreover, the study showed that code with diverse functionality is reused, indicating that the availability of reusable code is well established in the area of Java development.

5 Challenges of the Detection of Functionally Similar Code Fragments

Due to the size and complexity of many software libraries, it appears unrealistic that developers can completely overlook them. In consequence, reuse opportunities are missed and library functionality is unconsciously reimplemented [52,57]. The detection of such missed reuse opportunities would be beneficial for quality assurance. Reimplemented code could be replaced with the corresponding library functionality, which is often more mature than the newly implemented solution to a problem.

This chapter presents our experiences with a fully automated dynamic approach to detect functional similarity. The approach executes candidate code fragments with random input data and compares the output values. It is comparable to the approach pursued by Jiang&Su [49]. While they experienced high detection rates applying their approach to code of the Linux kernel, we were unable to produce significant results using this approach for diverse Java systems. As our results differ from Jiang&Su's results, this chapter details our insights regarding challenges of dynamic detection of functionally similar code in Java programs and provides a detailed comparison to their work.

Parts of the content of this chapter have been published in [17].

5.1 Terms

This section introduces the central terms used in this chapter.

Clone We define a *clone* as a syntactically similar code fragment which typically results from *copy&paste* and potential modification.

The research community introduced several specific clone types which impose constraints on the differences between the code fragments [59]. *Type-1 clones* are clones that may differ in layout and comments. *Type-2 clones* may additionally differ in identifier names and literal values. *Type-3 clones* allow a certain amount of statement changes, deletions or insertions. *Type-4 clones* are clones that are functionally similar but not necessarily syntactically similar and may be developed by different programmers [95].

In addition to the known clone types, we define for this thesis *type-1.5 clones* as type-1 clones that may be subject to consistent variable renaming. The clone types form an inclusion hierarchy, *i.e.*, all type-2 clones are also type-3 clones, but there are type-3 clones that are not type-2 (and analogously for the other types).

Simion We follow the definition from Juergens et al. [52] and define a *simion* as a functionally similar code fragment regarding its interface behavior. The interface of a code fragment consists of the variables that are read (the input) and the the variables that are written (the output) by the code. Two fragments are simions if they compute the same output for all input values except a bounded number of exceptions (*i.e.*, we allow for different behavior in corner cases or error conditions). Type-4 clones as defined by [95] are comparable to simions. However, we do not use this term, since the term *clone* implies that one instance is derived from the other.

Simions are not necessarily also syntactically similar. As an illustration, Listing 5.1 shows two alternative implementations of the factorial function whose syntactic representation differs significantly.

Listing 5.1: Two functionally equivalent but syntactically different implementations of the factorial function

```
1  //Recursive implementation of the factorial function
2  public static int factorial(int n) {
3    if (n == 0) {
4      return 1;
5    } else if (n > 0) {
6      return n * factorial(n - 1);
7    }
8    throw new IllegalArgumentException();
9  }
10
11 //Iterative implementation of the factorial function
12 public static int factorial(int n) {
13   if (n < 0) {
14     throw new IllegalArgumentException();
15   }
16   int result = 1;
17   for (int i = 1; i <= n; i++) {
18     result = result * i;
19   }
20   return result;
21 }
```

It has to be noted that while clones may be (and often are) simions, even textually identical fragments of code may emit different behavior because of the type binding implied by the surrounding context.

Chunk A *chunk* is a code fragment that is compared for functional similarity. It consists of a set of input parameters, a statement sequence, and a set of output parameters.

5.2 Dynamic Detection Approach

Our approach for dynamically detecting functionally similar code fragments follows in principle the approach of Jiang&Su [49]. The fundamental assumption is that two functionally similar code fragments will produce the same output for the same randomly generated input. However, we exclude syntactic clones from the simion detection, since these can be found by existing clone detection tools. The main difference to the approach of Jiang&Su is that our approach targets object-oriented systems written in Java whereas they address imperative C programs.

The detection procedure can be divided in five principal phases that are executed in a pipeline fashion. We implemented a prototype of this pipeline based on ConQAT (see Section 2.8).

Figure 5.1 illustrates the detection pipeline with its five phases, each consisting of several processing steps. The input of the pipeline is the source code of the analyzed projects and their included libraries. The output is a set of equivalence classes of functionally similar code fragments. The pipeline phases are detailed in the following sections.

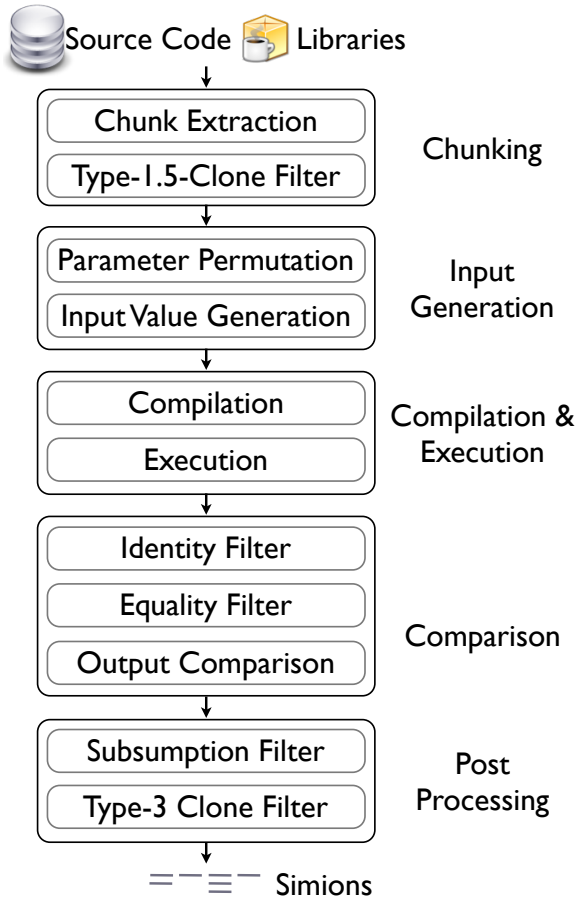

Figure 5.1: Simion detection pipeline

5.2.1 Chunking

The goal of the *chunking* phase is to divide the source code into chunks. It consists of the *chunk extraction* step and the *type-1.5-clone filter*.

Chunk Extraction

The *chunk extraction* step extracts a set of chunks from each source code file. The chunks are used as candidates for the simion detection. For Java systems, which consist of classes, it is a challenge to extract those chunks, since a code fragment arbitrarily cut out of the source files will in most cases not represent a compilable unit on its own. Therefore, we developed several strategies for extracting chunks from Java classes. We use the AST of each class and extract only valid statement sequences (*i.e.*, blocks are always extracted as a whole). The next challenge is to determine what will be considered as the input and output parameters of the chunk. In a Java class, several different types of variables occur. During the chunk extraction, we derive the input and output parameters from the declared and referenced variables in the statements of the chunk with the following heuristics:

- *Local variables* with a scope that is nested in the statements of the chunk do not have to be considered as input or output variables. All other referenced local variables become output variables. Local variables that are referenced but not declared in the chunk statements additionally become input variables.
- *Method parameters* are treated in the same way as referenced but not declared local variables, *i.e.*, they become input and output parameters. They are also output parameters, as the code after the chunk might reference these values.
- Referenced *instance variables (non-static)* become input as well as output parameters.
- Referenced *class variables (static)* are treated as a local variable since some of the called methods might rely on the value of these globals.

Since we want to compare individual chunks for functional similarity, we have to be able to compile their statement sequences and execute them separately. We therefore process each statement sequence and apply several transformations to obtain a static function with the input and output parameters of the chunk. In case of code within non-static methods, two chunks are extracted for the same statement sequence. This is used to provide different input signatures for the same piece of code. For example, code referencing the attributes of the class could either use an input object of the class' type or inputs for each of the attributes.

Example Listing 5.2 shows a simple Java class named `RentalPriceCalculator` that allows calculating the price for a car rental. The class has a field that defines the rental fee per day (`dailyRentalFee`). The constant `TAX_RATE` represents the tax rate in the country the company is operating. The rental price for a given rental duration is computed in the method `calculatePrice`.

Listing 5.2: Example Java class

```
1  public class RentalPriceCalculator {
2
3    private static final double TAX_RATE = .19;
4    private double dailyRentalFee;
5
6    public RentalPriceCalculator(double dailyRentalFee) {
7      this.dailyRentalFee = dailyRentalFee;
8    }
9
10   public double calculatePrice(int rentalDays) {
11     double price = rentalDays * dailyRentalFee;
12     price = price * (1 + TAX_RATE);
13     return price;
14   }
15
16 }
```

Listing 5.3 shows the two chunks that are extracted for the `RentalPriceCalculator` class. The first chunk treats the rental fee and the rental days as input parameters whereas the second chunk has a parameter of the type `RentalPriceCalculator` and the rental days parameter. These are two alternative interfaces for the functionality of the price calculation.

Listing 5.3: Extracted chunks

```
1  // Chunk #1
2  // Inputs  : dailyRentalFee(double), rentalDays(int)
3  // Outputs : out1(double)
4  double price = rentalDays * dailyRentalFee;
5  price = price * (1 + TAX_RATE);
6  out1 = price;
7
8  // Chunk #2
9  // Inputs  : in1(RentalPriceCalculator), rentalDays(int)
10 // Outputs : out1(double)
11 double price = rentalDurationDays * in1.dailyRentalFee;
12 price = price * (1 + TAX_RATE);
13 out1 = price;
```

There are cases when we cannot create a compilable chunk at all. Examples include statement sequences with branch statements (*e.g.*, `continue`) where the target of the branch statement is not in the sequence or calls to constructors of non-static inner classes.

Non-static methods that could be static since they do not reference any non-static methods or fields are converted to static methods by adding the static keyword to the method declaration. Thereby

more chunks do not need an additional input parameter with the type of the surrounding class and thus represent a more generally reusable fragment.

In cases where a statement sequence contains a `return` statement, the resulting chunk gets a single output parameter with the type of the return value of the surrounding method. The code with the `return` statement is changed to a local variable declaration with the corresponding initialization.

The performance of the detection heavily depends on the number of chunks compared for similarity. Since Jiang&Su experienced performance problems due to high numbers of chunks, we developed three different chunk extraction strategies:

- The *method-based* strategy utilizes the structure of the code given by the methods and considers the statement sequences of all methods in the code. This strategy has a slight variation for determining the output parameters. If the method has a non-void return value, the chunk gets one output parameter with the return type of the method. Otherwise the chunk gets one output parameter with the type of the surrounding class.

- The *sliding window* strategy extracts chunks by identifying all possible statement subsequences that represent valid fragments of the abstract syntax tree (AST) with a certain minimal length. Thus, it can cover cases where an arbitrary statement sequence is functionally similar to another one. However, the number of chunks is quadratic in the number of statements of a method.

- The *intent-based* strategy utilizes the programmer's intent by interpreting blank lines or comments in statement sequences as a logical separator between functional units. It extracts one chunk that includes all statements in a method and all chunks that can be formed from the statement sequences separated by blank or comment lines.

Figure 5.2 schematically illustrates these chunking strategies.

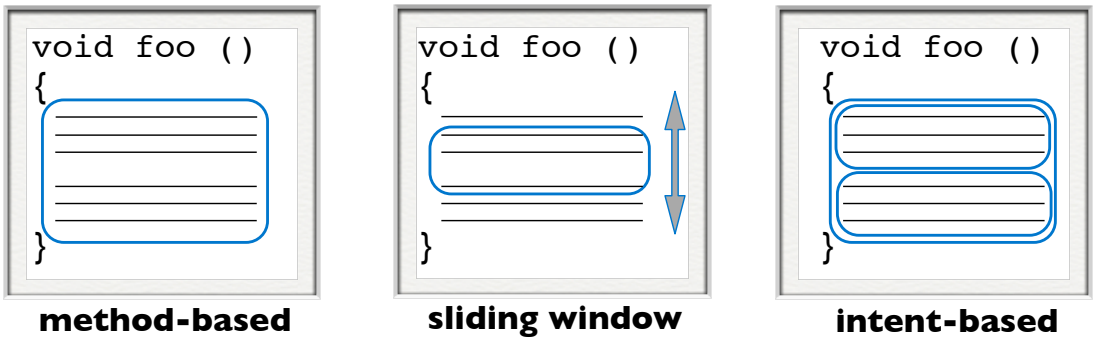

Figure 5.2: Schematic illustration of chunking strategies

Chunks that have no input or no output variables are excluded from further processing. An example is a statement sequence without variable declarations that solely calls static methods without parameters.

Type-1.5-Clone Filter

While code clones may be simions with regards to our definition, their detection with a simion detector is not worthwhile because they could be detected more easily with a clone detection tool. The type-1.5-clone filter discards chunks that are type-1.5 clones. This removes the clones from the results and improves detection performance as these chunks are not further processed.

5.2.2 Input Generation

The goal of the *input generation* phase is to supply the chunk code with different randomly generated input data values for multiple executions. It involves the permutation of parameters and the actual generation of input values.

Parameter Permutation

To cover cases where two code fragments compute the same function but have different parameter ordering, we additionally apply a parameter permutation. For each chunk, we generate *additional (virtual)* chunks where all input parameters of the same type are permutated. To cope with the combinatorial explosion, we constrain the number of additionally generated permutations to 25.

Input Value Generation

Similar to Jiang&Su, we employ a random testing approach for generating input values. We generate code for constructing random values for the input parameters of the chunks. Since the chunks must be executed on the same inputs for comparison, the input generation uses a predefined set of values for primitive parameters (all Java primitives, their corresponding wrapper types, and String). Input values for parameters with composite types are generated with a recursive algorithm. All constructors of the type are used for creating instances. The algorithm is applied recursively to the parameters of the constructors, *i.e.*, again, random values are generated and passed as arguments to the constructor. In case of 1-dimensional arrays and Java 5 collections where the component type is known, we generate collections of several randomly chosen but fixed sizes and apply the algorithm recursively for the elements of the collection. For chunks with many parameters or constructors with many arguments, a large number of inputs would be generated by this approach. Therefore, we constrain the overall number of input values generated for one chunk to 100.

5.2.3 Compilation & Execution

The goal of the *compilation & execution* phase is to perform one execution for each chunk and combination of input data values and to store the produced output values.

Compilation

To instrument the chunk code for simion detection, the *compilation* step wraps the chunk code with additional code for creating input values and handling errors during execution. Moreover, at the end of the chunk, we add code for storing the values of the output parameters after execution. If an exception occurred during the execution, a special output value *error* is used. Moreover, we generate code for handling non-terminating chunks (*e.g.*, due to infinite loops), which stops execution after a timeout of 1 second. The code is compiled in a static method within a copy of the original class code and with all compiled project code and libraries on the classpath. This means that the statements in the chunk have access to all static methods and fields from the original context as well as their dependencies.

Execution

After compilation, the chunks are executed in groups of at most 20 chunks in a separate Java Virtual Machine with a security manager configured to prevent certain unsafe or sensitive actions performed by a Java program. This ensures that the chunk execution does not have unwanted side effects (*e.g.*, deletion of files). The result of the execution step is a list of execution data objects that hold data about the chunk and a list of input and output values of the chunk execution.

5.2.4 Comparison

Preliminary experiments with the simion detection pipeline revealed that the majority of the identified simions is not relevant. A large fraction are false-positives, *i.e.*, our analysis identified them as functionally similar although they are not. This is caused by the random nature of the generated input data. Often two (or more) chunks are executed with input data that *triggers* only very specific execution paths. With respect to these paths the chunks are similar, although they are not for *reasonable* input data. An example are two string processing methods where one trims the string (deletes leading and trailing whitespace) and one replaces characters with a unicode code point above 127 with the unicode escape sequence used in Java. If both chunks are executed for a string without leading or trailing whitespace and without characters outside the ASCII range, they both simply return the input string. As a result, they are identified as a simion. We address both problems, clones and false-positives, with additional filter steps in our detection pipeline.

Identity Filter

The identity filter discards all chunks that *behave* like the identity function for the generated input data, *i.e.*, for each input value set, they return the same output value set. This heuristic excludes chunks for which the randomly generated input data is not capable of triggering *interesting* execution paths.

Equality Filter

The equality filter discards all chunks that generate the same fixed output data for all input data sets. The rationale behind this is that the chunk execution is apparently independent of the input data. Again, this is caused by the inherently limited quality of the randomly generated input data. An example is a chunk that has two input parameters: a string and an integer value i. If i is less than the length of the string (in characters), the chunk returns a new string with the first i characters of the input string. If i is greater than the length of the string, it returns the empty string. As this dependency between the two parameters is unknown to the input data generator, it could possibly generate only data sets where i is greater than the length of the chosen string. Consequently, all executions of the chunk return the empty string.

Output Comparison

The *output comparison* step uses the execution data objects to compare the chunks for functional similarity. Chunks that do not provide *valid* output data for at least 3 inputs (*i.e.*, either throw an exception or have a time-out), are discarded at this step. To make the comparison efficient, we use a hash-based approach. For each chunk, it computes an MD5 digest from the output values which is then used for the comparison. This requires only moderate space in memory even for large output data. This digest can be thought of as a functional fingerprint of the chunk. To construct the MD5 digest, we transform each output object to a string representation and append it to the MD5 digest. We use the XStream XML serialization library[1] to transform an arbitrary object into a string with its XML serialization. The MD5 digest of each chunk is used as a key into a hash map holding the chunks. If two chunks have the same MD5 digest, we have identified a pair of functionally similar code chunks. This is done for eliminating the otherwise quadratic effort of comparing all chunks for equal MD5 digests. While collisions lead to false-positives, we consider the comparison correct, since collisions are very unlikely in practice. The result of the output comparison step is a set of equivalence classes of chunks with similar functionality.

5.2.5 Post Processing

To further improve the precision of the detection, we apply two additional filters after the comparison. These filters are discussed in the following.

Subsumption Filter

The subsumption filter discards simions that are entirely covered by a larger simion (in terms of its length and position in the source code). For example, if two methods are identified to be simions, it is usually not worth reporting that parts of them are also simions. This type of filter is also commonly used in classic clone detectors.

[1] http://xstream.codehaus.org/

Type-3 Clone Filter

Additionally, at the end of the pipeline, a type-3-clone-detector is run to determine which of the simions could also be detected by a clone detector that takes into account insertion, modification, and deletion of a certain amount of statements. The filter calculates the statement-level edit-distance between chunks and is configured to filter all chunks with an edit-distance less or equal to 5. Both clone filters are implemented with ConQAT's clone detection algorithm [51]. We cannot filter type-3 clones earlier, as they are not guaranteed to be functionally equivalent. Thus, it is unclear which of the instances should be filtered as each could be a potential simion of another chunk.

5.3 Case Study

To assess the difficulty of the simion detection problem and to evaluate how capable the approach is of detecting simions, we conduct a case study with open source software systems and a collection of highly similar student programs.

5.3.1 Research Questions

We split the research problem into three major research questions about the difficulty of the problem of detecting simions, the effects of technical challenges, and the effectiveness of the proposed approach.

RQ 1 *How difficult is the simion detection problem?*

As a first step, we want to characterize the problem of simion detection. Since the detection pipeline has to be executed multiple times for each chunk, we look at the number of chunks extracted by the different chunking strategies. Moreover, since, for each chunk, multiple input parameter valuations have to be generated by the random test generation, we investigate the number of input parameters of the chunks.

RQ 2 *How do technical challenges affect the detection?*

To compare the functionality of two chunks, we need to transform them into executable code and generate useful input data. This provokes a number of technical challenges that affect the detection approach. This includes the generation of input values for the chunks, which need to be meaningful to trigger interesting functionality. Furthermore, the generation of useful input data for project-specific data types and the emulation of certain operations used by the chunks such as I/O or GUI operations is challenging. Finally, even after overcoming these limitations, there might still be problems that prevent the compilation of the extracted chunk. We investigate how many chunks need to be disregarded during detection because of these challenges.

RQ 3 *How effective is our approach in detecting simions?*

We ask if the approach is able to detect functionally similar code with a reasonable recall. Moreover, as we investigated the technical challenges in RQ 1, we are interested in the share of the code of real-world systems that we are able to analyze. Finally, we want to know how many simions we can find in realistic systems. We are also interested in the precision and ask if the detected simions are really simions according to our definition.

5.3.2 Data Collection Procedure

In Section 5.2, we described three different chunking strategies. The general idea is to run a complete simion detection with all strategies on a set of software systems to collect the needed data for answering the research questions. For practical reasons, however, we cannot perform a complete detection using the sliding window chunking strategy because it creates far more chunks than feasible to analyze. Therefore, we collect data using that strategy only for RQ 1.

For RQ 1 and RQ 2, we employ no filters, since we are only interested in determining the difficulty of the problem and technical challenges regardless of the precision of the results. For RQ 3, where we are interested in the amount of detected simions, we use all filters.

RQ 1: Problem Difficulty

ConQAT writes the total number of extracted chunks into its log file, which answers the first part of RQ 1. For the second part, we extend the ConQAT configuration to count for each chunk the number of input and output parameters and to write aggregated statistics to a separate output file. In both cases, we use separate configurations for the different chunking strategies.

RQ 2: Technical Challenges

To answer RQ 2, we use two different configurations. The first one is similar to RQ 1, using a specific statistics processor for collecting the required data. Our input generator logs the number of chunks for which no input could be generated. Additionally, our configuration determines and counts the types of the input parameters and aggregates these values.

Another part of the configuration counts the number of chunks that contain method calls to I/O, networking, SQL, or UI code. We cannot execute chunks containing such calls successfully, as the expected files, network peers, or databases are not available, or the required UI initialization was not performed. We identify calls to these groups by the package the corresponding class resides in, *e.g.*, a call to *java.io.File.canRead()* would be considered as I/O. These packages also contain methods that can be safely called even without the correct environment being set up (such as methods from *java.io.StringReader*), so we expect to slightly overestimate these numbers. On the other hand, we only count methods that are directly called from the chunk. Methods that are called indirectly (from other methods called from the chunk) are not included in these numbers. However, as we are only interested in the magnitude of this problem, we consider this heuristic sufficient.

The second configuration is a slightly simplified detection pipeline, that uses the approach described in Section 5.2 to generate code and tries to compile the chunks. Statistics on the number of chunks that could not be compiled are reported. For both configurations, we disabled the type-1.5 clone filter and the permutation step, as these distort the statistics slightly.

RQ 3: Effectiveness of our Approach

For the last research question in this study, we utilize the full simion detection pipeline to count the number of simions detected by our implementation. Our code is instrumented to report the number of chunks *lost* (*i.e.*, not processed further) at the steps of the pipeline. We manually assess a sample of the detected simions to determine the precision of the detection pipeline.

5.3.3 Analysis Procedure

This section describes for each research question how we analyzed the data.

RQ 1: Problem Difficulty

For analyzing the difficulty of the problem, we report the total number of chunks per chunking strategy to show the order of magnitude. To make the numbers more comparable and to allow an estimate for the simion detection in systems of other sizes, we give the number relative to the source lines of code (SLOC, number of non-blank and non-comment lines) and calculate the mean value. We plot the relative distribution of the input parameters per chunk and study object for each strategy.

RQ 2: Technical Challenges

To analyze the technical challenges, we show the relative distribution and calculate the mean per strategy for a set of metrics, that characterize different technical challenges. We give the values for all relative metrics rounded to full percentages. First, we analyze the difficulty of generating inputs by two metrics. One is the number of chunks for whose input parameters we cannot generate values. The other is the number of inputs of project-specific data types, because it is especially hard to generate meaningful input for them. Second, for the execution, there are certain types of methods that are hard to emulate during random testing. We analyze the number and share of calls to I/O, network, SQL, and UI. Third, the chunks need to be compiled before they can be executed. Hence, we investigate the fraction of chunks that cannot be compiled. For these challenges, we add qualitative, manual analysis of the chunks that cannot be further used in the detection approach to get more insights into the reasons.

RQ 3: Effectiveness of our Approach

For the analysis of the effectiveness of our approach, we use two different types of study objects. The first type is a set of programs of which we know that they have to exhibit similar functionality because they were produced according to the same specification. These study objects show whether the detection approach works in principle. We expect to get at least as many simions reported as there are implementations of the specification. The second type of study objects are real-world, large systems of which we do not know beforehand of any simions. We show the change in chunks during the execution of the detection pipeline in absolute and relative terms. We round the relative values to percentages with two decimal places to be able to differentiate small results. For the method-based chunking strategy and the real-world systems, we analyze the precision by manually inspecting the reported simions. We assign each simion to one of the following two categories:

- *false-positive*: Falsely identified simions (code fragments without similar functional behavior)
- *true-positive*: Correctly identified simions

5.3.4 Study Objects

We chose two different types of study objects: (1) a large number of Java programs that are functionally similar and (2) a set of real-world Java systems. The study objects of type 1 are small programs, which are easy to analyze and built according to the same specification so that we can be sure that they exhibit largely the same functionality. We used the same set of student programs as Juergens et al. in [52]. We will refer to it as *Info1*. These programs are implementations of a specification of an e-mail address validator by computer science undergraduate students. We only include programs passing a defined test suite to ensure a certain similarity in functionality. This results in 109 programs with a size ranging from 8 to 55 statements.

The second type of study objects represents real Java systems to show realistic measurements for chunks and simions. As selection criteria, we chose systems that cover a broad range of application domains, sizes, and functionalities. Furthermore, we chose systems we are already familiar with to support the interpretation of the results. The selection resulted in the five open source Java systems that represent libraries, GUI applications, and servers, shown in Table 5.1 together with their size in SLOC.

Table 5.1: Size of open source study objects

Project	SLOC
Commons Lang	17,504
Freemind	51,762
Jabref	74,586
Jetty	29,800
JHotDraw	78,902
Overall	**252,554**

5.3.5 Results

This section presents the results of our study separately for each research question.

RQ 1: Problem Difficulty

Table 5.2 shows the absolute and relative numbers of chunks extracted for each study object and the different chunking strategies. The sliding window strategy extracts the highest number of chunks with up to 2.68 chunks per SLOC and 1.44 chunks/SLOC on average. The intent-based strategy creates less chunks with at most 0.40 chunks/SLOC and 0.25 chunks/SLOC on average. The smallest number of chunks is extracted using the method-based strategy. It creates at most 0.09 chunks/SLOC and 0.05 chunks/SLOC on average.

Table 5.2: Total number of chunks for different extraction strategies

Object	Sl. Win.		Intent		Method	
	Total	per SLOC	Total	per SLOC	Total	per SLOC
Commons Lang	7,940	0.45	1,843	0.11	1,538	0.09
Freemind	80,816	1.56	20,632	0.40	1,984	0.04
Jabref	133,556	1.79	21,388	0.27	2,085	0.03
Jetty	22,006	0.74	7,713	0.26	1,457	0.05
JHotDraw	211,283	2.68	16,221	0.21	2,813	0.04
Mean	–	1.44	–	0.25	–	0.05

Figure 5.3 shows the relative distribution of the number of input parameters per chunk for the intent-based chunking strategy. For all projects, at least about half (48–87%) of the chunks have one or two input parameters. The fraction of chunks with more than 10 parameters is for all projects below 3%.

The relative distribution of the number of input parameters per chunk for the method-based chunking strategy is illustrated in Figure 5.4. More than about three quarters of the chunks (78–90%) have one or two input parameters. Except for *Commons Lang*, more than half of the chunks (59–73%) have exactly one input parameter.

Overall, the number of chunks is large, especially for the sliding window strategy. Thus, the detection approach needs to be able to cope with several thousand chunks. The number of input parameters per chunk, however, is mostly small. Most chunks have only 1 to 3 input parameters for which input data needs to be generated.

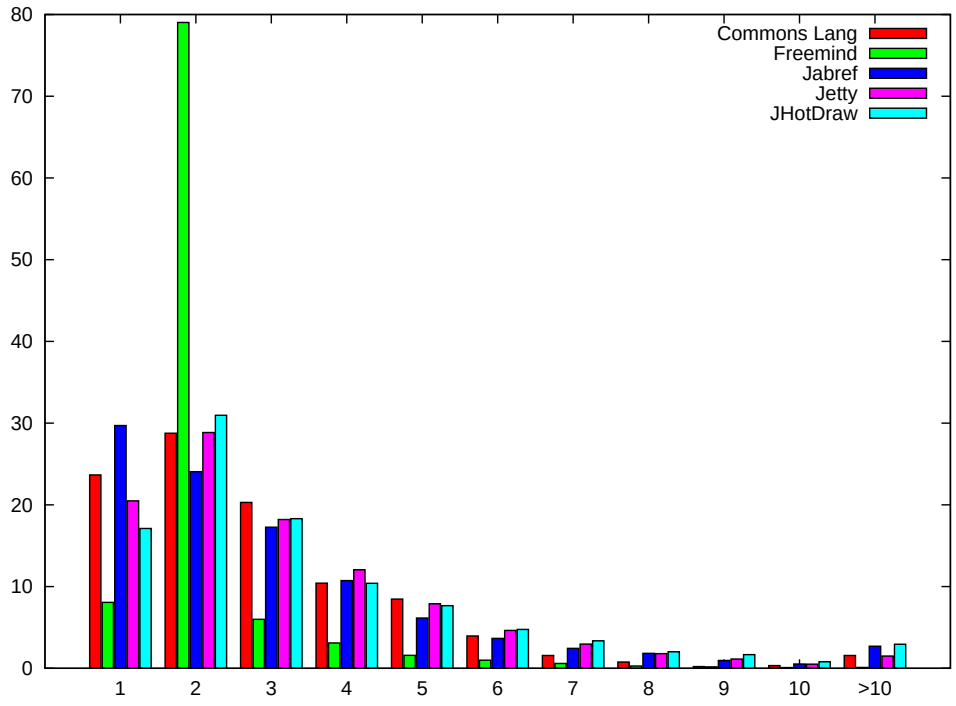

Figure 5.3: Relative distribution of number of input parameters per chunk (%): intent chunking

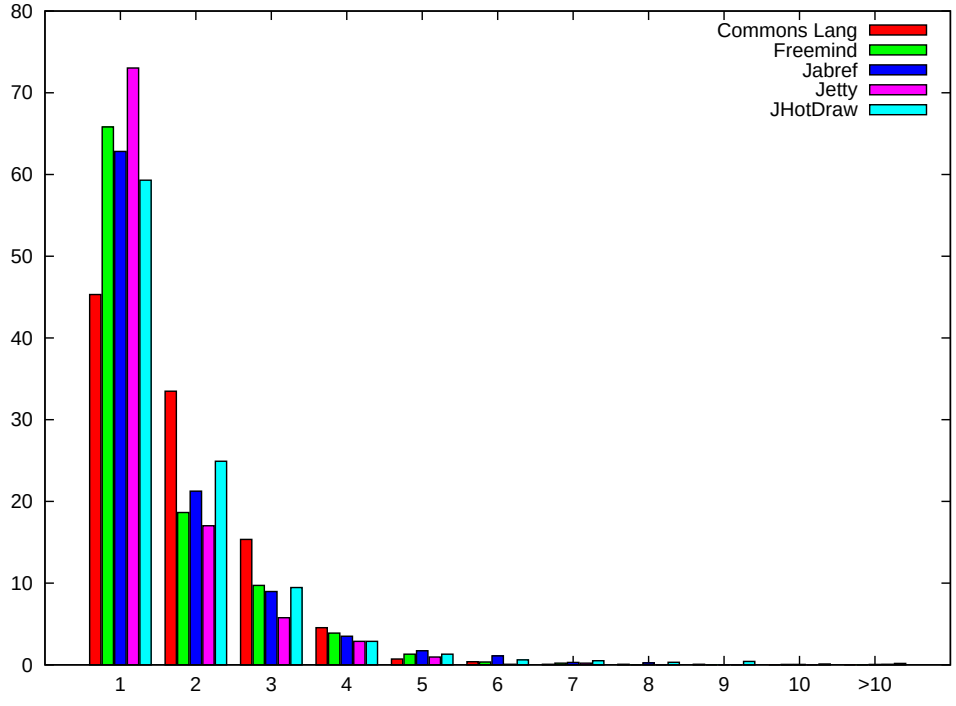

Figure 5.4: Relative distribution of number of input parameters per chunk (%): method chunking

RQ 2: Technical Challenges

The first pair of columns in Table 5.3 shows the fraction of chunks for which the approach cannot construct input values. The two main cases where no input can be generated are chunk parameters that refer to (1) an interface or abstract class and (2) a collection with an unknown component type[2]. In the first case, it is unclear which implementation should be chosen to obtain an object that implements the interface. In the second case, we do not know what type of object to put in the collection. For all systems, the fraction of chunks for which no input can be generated is higher for the intent-based chunking strategy compared to the method-based strategy. In case of *Freemind* and the intent-based chunking strategy for as much as 94% of the chunks no input could be generated. A manual inspection revealed that *Freemind* uses untyped Collections. Except for *Commons Lang*, where primitive types are dominant, input generation failed for more than 30% of the chunks with both chunking strategies.

Table 5.3: Fraction of chunks with technical challenges

	No Input		Proj. Specific		Specific Env.	
Project	**Meth.**	**Int.**	**Meth.**	**Int.**	**Meth.**	**Int.**
Commons Lang	20%	37%	60%	60%	1%	5%
Freemind	65%	94%	79%	75%	11%	27%
Jabref	32%	41%	61%	81%	17%	44%
Jetty	36%	66%	72%	87%	7%	15%
JHotDraw	43%	55%	76%	86%	24%	60%

We determined how many chunks have parameters (either input or output) of project-specific data types. The results are shown in the second pair of columns in Table 5.3. The analysis revealed that a considerable fraction of the chunks (60-87%) refer to project-specific data types. With our approach, these would not qualify as candidates for cross-project simions, since the other project would not have the same data types.

To estimate the use of methods that require a specific environment, we determined for each chunk whether it contains direct calls to methods from one of the groups I/O, networking, SQL, or UI. The relative numbers of chunks containing calls to a category are shown in Figure 5.5 for each of the categories. The results show that the numbers depend on the application domain. The library *Commons Lang* has only a couple of calls to I/O code, the HTTP server *Jetty* has the highest number of calls to networking code, and the drawing tool *JHotDraw* dominates the UI group. The most common groups are UI (if the application has a user interface at all), followed by I/O.

In the execution step, a chunk containing methods from at least one of these groups is likely to fail as the expected environment is not provided. The last pair of columns in Table 5.3 lists how many chunks are affected by such methods for both of the chunking strategies. Overall, as many as 24% of the methods can be affected (*JHotDraw*), or for the intent-based chunking strategy more than 60% of the chunks, thus having a significant impact on the number of chunks we can process by our dynamic approach. Interestingly, the relative numbers for the intent-based strategy are higher

[2]This second issue especially applies to code that is not targeted at version 1.5 or above of the Java language specification which supports Generics and thereby allows to specify the component type within the declaration of a collection type.

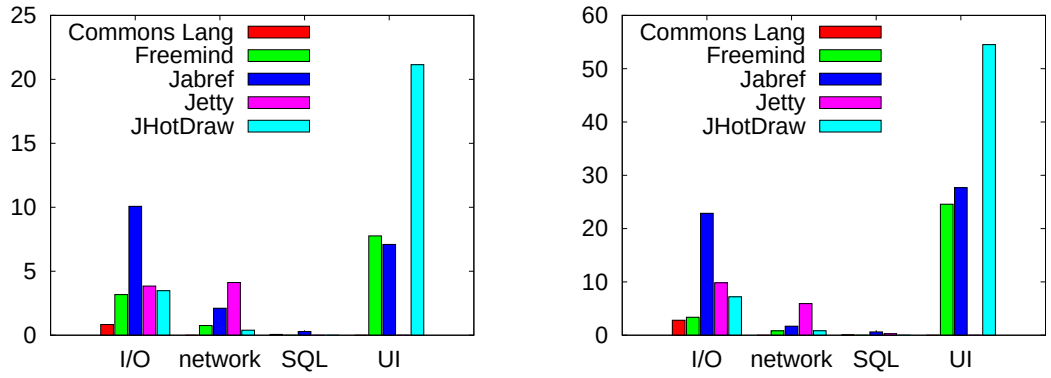

Figure 5.5: Fraction of chunks calling complex methods (%) for method chunking (left) and intent chunking (right)

than for the method-based strategy in all cases. This suggests, that the methods containing I/O or UI code are typically longer than the remaining methods and thus produce more chunks.

Table 5.4: Compileable chunks

Project	Method	Intent
Commons Lang	96%	97%
Freemind	92%	99%
JabRef	90%	82%
Jetty	83%	88%
JHotDraw	93%	84%

Finally, we checked how many of the chunks we were able to make compilable by providing a suitable context. The relative number of chunks we could make compilable of those for which at least one input could be generated is shown in Table 5.4. These numbers do not indicate a principal limitation, as each of the chunks we extracted is a valid subtree of the AST and thus can be executed in a suitable context. They rather document limitations in our tool. An inspection of the problematic chunks revealed weaknesses in chunks dealing with generic data types, anonymous inner classes, method local classes, and combinations thereof. Still, we are able to automatically generate a context for at least 82% in all cases and up to 99% for *Freemind*. These numbers could be improved by using more advanced algorithms for generating the context for a chunk. The other results from RQ 2 and those of RQ 3 presented next suggest, however, that the chunks we lost because we can not make them compile are not the main bottleneck of the detection pipeline.

RQ 3: Effectiveness of our Approach

Tables 5.5 and 5.6 summarize the analysis results for all study objects discussed previously plus the additional *Info1* system (gray column). For each study object, the tables depict all pipeline steps along with the number of resulting chunks *after* the step's execution (column *Abs.*). Additionally, column *Rel.* shows the relative number of chunks with respect to the original number of chunks created by the chunk extraction step. The absolute delta between the rows shows how many chunks are *lost*[3] in each pipeline step. The tables' last row *Type-3-Clone* reports the total number of simions found after all steps have been processed. We do not report exact processing times, as the load of the machines we were using for the analysis varied. Each individual run for one system took between 1 and 30 hours (depending on number of chunks).

For the *Info1* data set, we found 105 simions with the method-based chunking strategy and 418 simions with intent-based chunking. The higher number for the intent-based strategy is expected, as certain sub-steps of the implementation can be also seen as individual simions that are not removed by the subsumption filter if they occur more often than the surrounding simion. As the data set consists of 109 implementations of the same functionality, we would not expect to find substantially more simions with any other approach. Hence, the recall for purely algorithmic code (no I/O, no custom data structures, etc.) is good.

For the *real* software systems (not including the *Info1* set), the analysis discovered a total of 153 simions with the method-based chunking strategy and 294 simions with the intent-based strategy. Compared to the size of the analyzed systems, this number seems small. We discuss the number of simions in more detail in Section 5.6. We consider the overall number of simions too small to discuss the differences between both chunking strategies, however.

RQ 2 quantified the numbers of chunks that are discarded due to different technical reasons, such as missing input. A chunk can be affected, however, by more than one of these issues. To understand how many chunks are affected by none of these problems, we have to look at the entire processing pipeline. On average, about 28% of the chunks for both strategies remain in the pipeline after the execution step and all preceding steps. This is still a significant part of the systems in which we can in principle find simions using our approach. Yet, this also means that more than two thirds of the chunks are lost before the actual comparison can be performed and are thus not receptive for our dynamic detection.

Another observation is that while the intent-based chunking strategy results in the higher number of simions found, the relative number of simions found in comparison to the number of input chunks is four times higher for the method-based strategy. One explanation is that the method boundary chosen by the developer more likely encapsulates a reusable code fragment. Fragments from within a method are less likely to be reusable by another developer. Thus, the probability of finding a duplicate of its functionality is lower. For practical application this means that the intent-based strategy is preferred in terms of results (more simions), but regarding the required computation time (which scales linear in the number of chunks) the method-based approach is more effective.

Tables 5.5 and 5.6 also hint at the amount of cloning. In the method-based case, on average, 27% of the chunks are discarded early on as type-1.5 clones (for the intent-based strategy this number is

[3] In the permutation step, additional chunks are created.

Table 5.5: Analysis results for the method-based chunking strategy

	Comm. Lang		Freemind		JabRef		Jetty		JHotDraw		Info1	
	Abs.	Rel. [%]	Abs.	Rel. [%]	Abs.	Rel. [%]	Abs.	Rel. [%]	Abs.	Rel. [%]	Abs.	Rel. [%]
Chunk Extr.	1,538	100.00	1,984	100.00	2,085	100.00	1,457	100.00	2,813	100.00	240	100.00
Type-1.5-Clone	1,100	71.52	1,541	77.67	1,647	78.99	1,034	70.97	1,936	68.82	230	95.83
Permutation	1,601	104.10	1,916	96.57	2,245	107.67	1,255	86.14	3,472	123.43	231	96.25
Input Gen.	1,265	82.25	643	32.41	1,542	73.96	742	50.93	1,879	66.80	134	55.83
Compilation	1,215	79.00	530	26.71	1,324	63.50	568	38.98	1,586	56.38	133	55.42
Execution	1,066	69.31	189	9.53	621	29.78	313	21.48	660	23.46	133	55.42
Identity	1,066	69.31	189	9.53	621	29.78	313	21.48	660	23.46	133	55.42
Equality	947	61.57	165	8.32	522	25.04	178	12.22	579	20.58	133	55.42
Comparison	108	7.02	15	0.76	94	4.51	19	1.30	34	1.21	105	43.75
Subsumption	108	7.02	13	0.66	90	4.32	19	1.30	30	1.07	105	43.75
Type-3-Clone	54	3.51	11	0.55	55	2.64	15	1.03	18	0.64	105	43.75

Table 5.6: Analysis results for the intent-based chunking strategy

	Comm. Lang		Freemind		JabRef		Jetty		JHotDraw		Info1	
	Abs.	Rel. [%]	Abs.	Rel. [%]	Abs.	Rel. [%]	Abs.	Rel. [%]	Abs.	Rel. [%]	Abs.	Rel. [%]
Chunk Extr.	1,843	100.00	20,632	100.00	21,388	100.00	7,713	100.00	16,221	100.00	1,969	100.00
Type-1.5-Clone	1,598	86.71	20,586	99.78	20,794	97.22	7,527	97.59	14,643	90.27	1,938	98.43
Permutation	4,690	254.48	26,705	129.43	61,836	289.12	17,131	222.11	51,683	318.62	3,259	165.52
Input Gen.	2,823	153.17	2,858	13.85	33,559	156.91	5,301	68.73	26,347	162.43	1,910	97.00
Compilation	2,772	150.41	2,432	11.79	22,837	106.77	4,227	54.80	16,527	101.89	1,883	95.63
Execution	2,399	130.17	556	2.69	3,998	18.69	2,173	28.17	9,887	60.95	1,841	93.50
Identity	2,298	124.69	424	2.06	3,480	16.27	1,734	22.48	9,352	57.65	1,825	92.69
Equality	2,014	109.28	388	1.88	2,818	13.18	1,392	18.05	8,694	53.60	1,730	87.86
Comparison	85	4.61	4	0.02	232	1.08	36	0.47	131	0.81	752	38.19
Subsumption	77	4.18	4	0.02	196	0.92	32	0.41	67	0.41	420	21.33
Type-3-Clone	59	3.20	2	0.01	159	0.74	28	0.36	46	0.28	418	21.23

lower with about 4%). From the simions found, 41% (or 22% for intent-based) could be found by a type-3 clone detector. The absolute numbers provide an even clearer picture. There are 17 times as many chunks removed by the type-1.5 clone filter as there are simion instances reported (9 times for intent-based), so the cloning problem seems to be worse than the problem of independent reimplementation (simion). Additionally, clones (even of type-3) are much easier to detect than simions (both in terms of the involved algorithms and the required processing time) and the detection of clones is also possible for code where we can not find suitable input or a compilation context. From a quality improvement standpoint, this indicates that detecting simions should only be performed after all clones have already been filtered.

To determine the precision of our pipeline, we manually assessed the 153 simions detected with the method-based chunking strategy for the real-world software systems among the study objects (all except *Info1*). Table 5.7 shows the results of the manual assessment of these simions.

The manual inspection revealed that, for all projects, the detected simions contain a significant amount of false-positives. The fraction of false-positives ranged from 39% to 100% for the 5 study objects. On average, about two thirds (67%) of the detected simions were false-positives. Thus, the amount of real simions detected by our pipeline using the method-based chunking drops from 153 to 51 for the 5 projects combined.

The false-positives were in most cases functions that produce the same output for a small number of input values. However, these were mostly *trivial* corner cases regarding the input data (*e.g.*, `null` or the empty String) and the manual inspection of these functions revealed that they differ significantly in their behavior for *interesting* input data values, thus not qualifying as simions according to our definition.

The true-positives for *Commons Lang* were mostly simple functions that used different data types (such as primitive types and their corresponding wrapper types) but performed an equal function. Other true-positives included cases where one function simply delegated to another one (in some cases with additional error handling in the delegating function), thus trivially exhibiting equal functionality. Although these are valid simions according to our definition, we do not consider them worthwhile to detect, since such delegation is often consciously done, for instance to provide the same functionality under a different name or for backward compatibility reasons after renaming a method. Figure 5.6 shows three examples of the detected simions in the project *Commons Lang*.

Table 5.7: Manual assessment of simions for method-based chunking

Project	Simions detected	True-positives	False-positives
Commons Lang	54	33 (61%)	21 (39%)
Freemind	11	7 (64%)	4 (36%)
JabRef	55	9 (16%)	46 (84%)
Jetty	15	0 (0%)	15 (100%)
JHotDraw	18	2 (11%)	16 (89%)
Overall	**153**	**51 (33%)**	**102 (67%)**

```java
public static int stringToInt(String str, int defaultValue) {
    try {
        return Integer.parseInt(str);
    } catch (NumberFormatException nfe) {
        return defaultValue;
    }
}
```

```java
public static int stringToInt(String str, int defaultValue) {
    return toInt(str, defaultValue);
}
public static int toInt(String str, int defaultValue) {
    if(str == null) {
        return defaultValue;
    }
    try {
        return Integer.parseInt(str);
    } catch (NumberFormatException nfe) {
        return defaultValue;
    }
}
```

```java
public static boolean toBoolean(int value, int trueValue, int falseValue) {
    if (value == trueValue) {
        return true;
    } else if (value == falseValue) {
        return false;
    }
    // no match
    throw new IllegalArgumentException("The Integer did not match either specified value");
}
```

```java
public static boolean toBoolean(Integer value, Integer trueValue, Integer falseValue) {
    if (value == null) {
        if (trueValue == null) {
            return true;
        } else if (falseValue == null) {
            return false;
        }
    } else if (value.equals(trueValue)) {
        return true;
    } else if (value.equals(falseValue)) {
        return false;
    }
    // no match
    throw new IllegalArgumentException("The Integer did not match either specified value");
}
```

```java
public static boolean xor(boolean[] array) {
    // Validates input
    if (array == null) {
        throw new IllegalArgumentException("The Array must not be null");
    } else if (array.length == 0) {
        throw new IllegalArgumentException("Array is empty");
    }

    // Loops through array, comparing each item
    int trueCount = 0;
    for (int i = 0; i < array.length; i++) {
        // If item is true, and trueCount is < 1, increments count
        // Else, xor fails
        if (array[i]) {
            if (trueCount < 1) {
                trueCount++;
            } else {
                return false;
            }
        }
    }

    // Returns true if there was exactly 1 true item
    return trueCount == 1;
}
```

```java
public static Boolean xor(Boolean[] array) {
    if (array == null) {
        throw new IllegalArgumentException("The Array must not be null");
    } else if (array.length == 0) {
        throw new IllegalArgumentException("Array is empty");
    }
    boolean[] primitive = null;
    try {
        primitive = ArrayUtils.toPrimitive(array);
    } catch (NullPointerException ex) {
        throw new IllegalArgumentException("The array must not contain any null elements");
    }
    return xor(primitive) ? Boolean.TRUE : Boolean.FALSE;
}
```

Figure 5.6: Sample of detected simions

5.4 Threats to Validity

The validity of the findings reported in this case study is subject to a number of potential threats, which will be discussed in the following.

One possible threat to the validity of our results are errors in our implementation of the detection pipeline. To mitigate this, we integrated excessive logging in our tool and inspected samples of the reported or excluded chunks at every pipeline step during development. Additionally, we included the student implementations of the same specification, for which a lower bound for the number of simions is known, into our study objects. This ensures that our approach and implementation are capable of finding at least certain simions.

To improve the validity of the results, we chose study objects we were familiar with from earlier experiments in a code analysis and quality context. This helped us to interpret the results compared to an entirely unknown system. Still, we attempted to select study objects of different types and sizes, to improve transferability of our results to other Java systems.

A threat to the internal validity is that the different filters used can distort the results for individual technical challenges. We mitigated this threat by separate configurations for collecting different data. The configuration for the results of RQ 2 uses less filters, so we could get the complete results.

The manual assessment of the simions in order to determine the precision of our approach is inherently subjective and may therefore be biased in either direction.

5.5 Comparison to Jiang&Su's Approach

To the best of our knowledge, the work of Jiang&Su [49] is the only published attempt of dynamically detecting semantically equivalent code from a large code base (see Section 3.2). In their case study with the Linux Kernel, the authors found 32,996 clusters of functionally similar code fragments which in total account for about 624,000 lines of code.

The considerable differences in the amount of detected simions raises the question what reasons this discrepancy has. Thus, we compare our approach and results in detail to their paper in this section. The most obvious difference is that our tool chain works on Java code, while their tool, called *EqMiner*, deals with the C language. For a more systematic comparison, we structure the comparison according the different phases of the detection pipeline.

Chunk Extraction Phase

This phase is called *code chopper* in EqMiner. Their approach corresponds to the *sliding window* chunking strategy with a minimal window size of 10 statements. They report, that for long functions, the quadratic number of chunks created this way is too large, which matches our observation. To mitigate this, EqMiner "randomly selects up to 100 code fragments from all code fragments in each function". We expect, that this random selection can cause relevant chunks to be missed. Thus, we

employ a strategy based on logical separation found in the syntax, which also helps to reduce the number of chunks, but better captures the programmer's intent compared to random chopping.

Input Generation Phase

The type system of C essentially consists of primitive types, pointers, and record types (structs). Consequently, the input generator used in EqMiner supports generation of all of these types, including dynamic allocation of structs to provide a pointer to a struct, but the generation of arrays is not supported. In Java, there are also primitive types and arrays, but instead of structured data (structs) and pointers, Java has classes and object references. While similar in the intent, this complicates input generation. Furthermore, the presence of abstract types and interfaces leads to situations where a suitable concrete implementation can not be easily generated as input. As long as non-abstract classes are used, we follow the approach from [81] by picking a random constructor and recursively generating inputs for its parameters.

Chunk Execution Phase

This phase in our tool corresponds to the *code transformation* and *code execution* steps in EqMiner. The obvious difference is that to make a fragment of Java code compile requires slightly different surrounding code than with C code. Especially the access to local attributes and methods within the same class requires additional considerations. The main difference, however, is how we treat function calls within the extracted chunk. In EqMiner, Jiang&Su "view each callee as a random value generator and ignore its side-effects besides assignments through its return values (*i.e.*, the random values)". We found this approach too limiting for Java code, as we might miss interdependencies between method calls (for example, code might rely on getting the same value from a getter method that was earlier passed to the corresponding setter method). Instead, we just execute the original method, which means that the entire context of the chunk must be reconstructed, including the surrounding class with its attributes and methods. Actually executing the methods also requires protection of the execution environment of unwanted side-effects. For example, feeding code that deletes files with random data could cause problems during analysis. Using Java's security manager, however, we could ensure that the program does not cause these kinds of problems.

Result Comparison Phase

EqMiner treats code chunks as equivalent, if they produce the same output for ten different random inputs. Jiang&Su also report, that a common pattern found in the largest clusters of equivalent code is that the (single) output of the chunk is exactly the input value. So, the chunk is essentially equivalent to the identity function (or a projection of inputs). Our experiments also showed this pattern and manual inspection of these chunks revealed that these fragments typically influence the system by other means (for example, by calling functions with side-effects) and would not be considered equivalent by a developer. All chunks following this pattern were filtered out by our detection pipeline prior to comparison. Also chunks that return the same result for all random inputs are discarded by our tool. As the results for RQ 3 indicate, both cases are frequent.

Study Objects

Jiang&Su evaluated their tool on a sorting benchmark and the Linux kernel. Both systems do not contain code that performs I/O operations (actually the kernel *offers* system calls for performing I/O) or deal with GUIs. Additionally, a huge part of the kernel deals with process scheduling, device drivers, or memory management, which are all not known to require complex string processing. Contrary, as shown by RQ 2, our study objects spend lots of code on I/O and GUI tasks, and string processing is essential for parts of them. As I/O and GUI are tricky for input generation, and string algorithms are often hard to differentiate with only 10 inputs, the number of valid chunks and false-positives is affected by our choice of systems.

5.6 Discussion

The low number of simions raises the question whether there are no simions in those systems or rather our detection approach is flawed. Ideally, we would answer this question by calculating the recall, *i.e.*, the fraction of all known simions in a system we are able to detect. For realistic systems, such as our study objects, the number of existing simions is not known (and practically infeasible to determine manually by inspection for even a small part of them). For artificial benchmarks, such as the *Info1* set, we have a good estimate of the number of simions. Yet, while our recall is good for this study object, the comparison with the numbers from Tables 5.5 and 5.6 shows that they are not representative for realistic systems.

Intuition and experience tells us, that developers tend to *reinvent the wheel* and consequently we would expect many simions. One explanation for the low rates could be that we only analyzed simions within a single system. Maybe developers know their *own* code base well enough to reuse code (either by referencing or cloning it) instead of reimplementing it as a simion. First experiments with detection of simions between two independent projects, however, did not reveal substantially higher simion rates. One explanation is given by Table 5.3, which reports high rates of chunks with project-specific data types. As we only find simions with the same I/O signature, these chunks can not be cross-project simions.

In our opinion, another reason for the low detection rates is that the notion of I/O equivalence is inappropriate. Often, code encountered in practice might be intuitively considered a simion, but does not exhibit identical I/O behavior. Reasons are differences in special cases, the error handling, or the kind of data types used at the interfaces. An extreme example would be databases from different vendors. While they basically provide the same functionality and interface (SQL), migrating from one database to another is typically far from trivial as the behavior is *not* the same in all details. Thus, we believe that there should be a better definition of a simion than the I/O behavior of code chunks. Finding a suitable definition and exploiting it for simion detection is one of the main open questions for future work in this area.

5.7 Summary

In this chapter, we presented our experience with an approach for detecting functionally similar code fragments in Java systems, which was inspired by an existing approach for C systems. We evaluated the approach with 5 open source systems and an artificial system with independent implementations of the same specification. In contrast to existing work targeting C systems, we experienced low detection results. In our opinion, this is mainly due to the limited capability of the random testing approach. In many cases, input generation either fails to generate valid input or the generated input is not able to achieve sufficient code coverage. There is also reason to believe that similarities are missed due to the chunking, *e.g.*, if code fragments perform a similar computation but use different data structures at their interfaces.

In conclusion, the results of the case study showed that such an approach faces considerable challenges when applied to real-world software. Thus, our findings emphasize the importance of alternatives as given by more light-weight constructive approaches whose goal is to *avoid* reimplemtented functionality already during forward-engineering.

6 Developer Support for Library Reuse

To prevent missed reuse during software development and thus foster effective and efficient reuse of software libraries, this chapter proposes an API method recommendation system which provides context-dependent suggestions for API methods during code editing within an IDE.

The proposed approach is based on the identifiers used in a program and utilizes the developer's intention embodied in the names of variables, types and methods. We investigate the impact of several variation points of the recommendation algorithm and evaluate the approach for recommending methods from the Java and Eclipse APIs in 9 open source systems. Furthermore, we compare the recommendations to those of a structure-based recommendation system and describe a metric for predicting the expected precision of a recommendation. The findings indicate that the proposed approach performs significantly better than the structure-based approach.

Parts of the content of this chapter have been published in [34, 36].

6.1 Motivation

Existing reuse recommendation systems utilize structural program information, *e.g.*, the types and methods used in the code, to infer what functionality might be needed next [70, 71, 108]. However, cases where no methods or types are used in the context or only general purpose types are used, are not supported well by these approaches.

Our approach takes an alternative route compared to structure-based approaches and is based on the intentional knowledge embodied in the identifiers in the context of an API method call.

Consider the example code snippet in Listing 6.1, which was taken from an open source system. In this example, the only type involved is `float` and the only methods are simple getters and setters. Thus, when relying exclusively on structural information, a reliable recommendation is hard to determine.

Listing 6.1: Example code snippet from an open source system

```
1 if (angle != getAngle()) {
2   float angleDelta = angle - getAngle();
3   super.setAngle(angle);
4   ...
```

In contrast, the identifiers (*i.e.*, the names of the variables, types and methods in a program chosen by a developer) in the given program contain valuable information about the intent of the developer. In the example above, it can be determined from the identifiers that the code deals with angles and thus recommending trigonometric functions could be helpful.

6.2 Terms

This section introduces the central terms for this chapter.

Development Context The development context refers to the code being edited within an IDE. In the scope of this thesis, this context is given by the source code preceding a cursor position within a source code file.

API Method An API method is a method (static or non-static) that is declared in a class that belongs to an API, such as the Java API, and has protected or public visibility. An API method is uniquely identified by its signature (consisting of the name of the method, the return type and the list of parameter types). This means that methods with the same name but different parameters are considered as distinct methods.

Recommendation Set A recommendation set is a set of API methods recommended in a given development context.

Recommendation Rate The recommendation rate denotes the fraction of correct recommendation sets produced by a method recommendation system when trying to predict method calls in existing code from the context of the method call. A recommendation set is considered *correct*, if the method actually employed is in the recommendation set. The recommendation rate is used to assess the appropriateness of the recommendations produced by a recommendation system.

6.3 Approach

This section describes our API method recommendation approach, which uses data mining for learning term-method associations that are used for recommending API methods during the development of new code.

We implemented the approach in Java on top of ConQAT (see Section 2.8). Our implementation is targeted at Java programs, but could be adapted to other (object-oriented and procedural) programming languages.

The approach has a training phase that builds an association index by analyzing existing software systems. This index is then used to answer queries that are formed of a set of terms extracted from the development context. Both the up-front index creation and the index query share a number of common processing steps as illustrated in Figure 6.1.

The subsequent sections first describe the common steps of index creation and index query and then present the processing steps particular to the both phases.

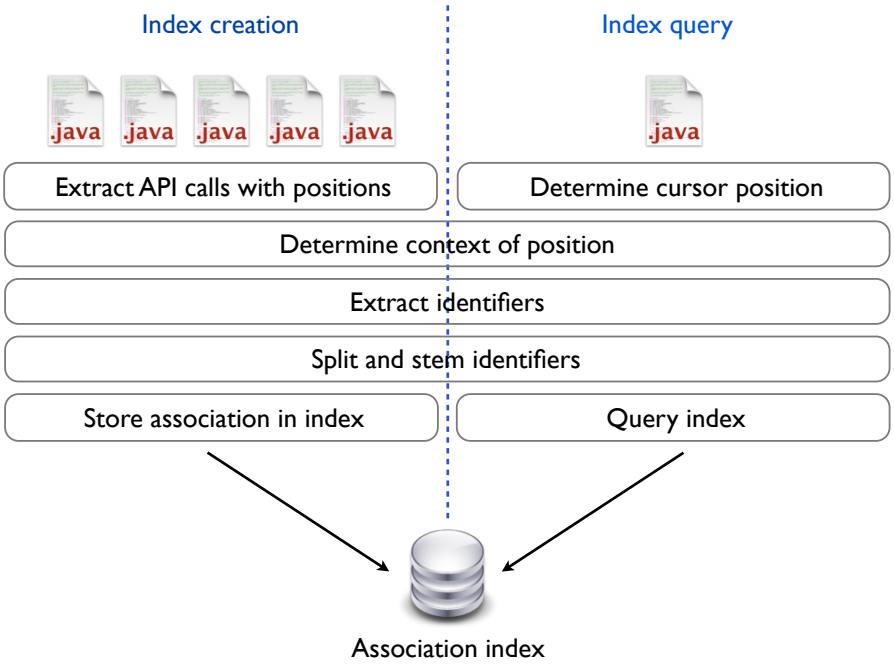

Figure 6.1: Overview of our approach

6.3.1 Common Processing Steps

Determine context of position

The context relative to a given position in the source code consists of the intent contained in the identifiers preceding the position. In case of index creation, this position is given by the first character of the method call, while in case of the index query, it is the position of the cursor within the source code editor. The context determination requires the position to be within a method body, since we want to avoid considering unrelated code in other methods. This also means that for positions at the beginning of a method body, no context can be determined. The context of a position is given by a configurable amount of identifiers preceding the code position, called the *lookback*. The amount of lookback creates a trade-off: on the one hand, a lookback that is too large can result in taking potentially unrelated identifiers into account while on the other hand a lookback that is too small can result in a context that is not very descriptive.

Extract identifiers

From the context, we determine the program identifiers used. We consider all program elements, including variables, types, and methods. Thereby, we include identifiers referring to program elements both from the source code and from the API employed. We extract the set of distinct identifiers used, *i.e.*, it does not matter how often an identifier occurs.

Split and stem identifiers

The extracted identifiers are split into words according to the camel case notation, which is recommended by the official Java Code Conventions [103]. The following example illustrates how an identifier is split into a set of words:

$$split(errorMessageDialog) = \{error, message, dialog\}$$

Identifier parts that only consist of a single character are discarded. The split words are reduced to their stem by using an English *stemmer*. The English language represents parts of speech, tense, and number by inflected (*i.e.*, morphologically varied) words. Stemming is commonly used in information retrieval to match queries also to documents containing derived words [13]. For instance, the word *read* shall be also matched with the word *reading*. The following example illustrates the word stemming:

$$stem(reading) = stem(reads) = stem(read) = read$$

The result of these common processing steps is a set of context terms, in which multiple occurrences of the same term are treated as one.

Parameters

The common processing steps have three parameters, which influence how the context terms are derived from a source code position.

- **Lookback ($\mathbb{N}^+$).** The number of distinct identifiers preceding the source code position that are considered for the extraction of terms can be configured with the parameter *lookback*.
- **Stopwords (preserved, removed).** In information retrieval, stopwords are words that occur very frequently and are not useful to find documents depending on a search query. These words are typically not indexed by search engines. Examples include articles (*e.g.*, "the", "a", "an") and prepositions (*e.g.*, "in", "for", "to") [13]. The parameter *stopwords* of our approach denotes whether these words shall be removed from the set of terms associated with an API method. We used the list of 119 stopwords from [107].
- **Keywords (included, excluded).** Since keywords carry information about the structure of the code, they can serve as a simple extension to our approach for considering structural information in addition to the intentional knowledge from identifiers. For instance, the keyword `catch` indicates an exception handling code section and can therefore add important information to a method association in our index. The parameter *keywords* specifies whether keywords that occur in the context of the method call should be considered in addition to identifiers. Technically, we wrap the keywords with brackets (which are non-identifier characters) and treat them just as normal terms during subsequent processing. For instance, the keyword `catch` is transformed to `<catch>`. Due to the brackets, we obtain a *unique term* and avoid a meaningless relation to an equal term embodied in an identifier.

Example

The following example illustrates the identifier extraction process for a given position (denoted by | in the snippet):

```
1 try {
2   readFile();
3 }
4 catch (IOException e) {
5   String errorMessage = e.getMessage();
6   |
7 }
```

From this snippet, using a lookback of 5 identifiers, the following set of terms is extracted:

$$\{io, except, string, error, messag, get\}$$

⌟

6.3.2 Index Creation

The index creation analyzes API method calls in Java classes. We use the Eclipse Java Compiler (ECJ) to build and traverse the abstract syntax tree (AST) of each class and process all API method calls. We determine if the method call targets an API method by checking if the declaring class of the method belongs to the API under consideration. Settings for included and excluded prefixes are used to determine which classes belong to the API. Given the character position of the API method call in the source code file, we extract the context terms as described in the previous sections and store an entry in the index that associates the set of terms with the API method that was called. The method is represented by its signature consisting of the fully qualified name of its declaring class, the method name, and the list of parameter types. In case of inheritance, the most "specific" type in the inheritance hierarchy declaring the method is considered as declaring class. The index consists of a list of these association entries.

Parameters

The index creation has the following parameters that influence which method calls are considered as API method calls.

- **Included prefixes.** A set of Strings that defines which classes are considered as API classes. All methods declared in these classes are considered as API methods. Each class that matches at least one include prefix is considered as an API class. An example setting for this parameter is: `{"java.io", "java.lang"}`.
- **Excluded prefixes.** A set of Strings that defines which classes are not considered as API classes. If a class matches one of the exclude patterns, its methods are not considered for mining association index entries.

Example

We extend the previous example with an API method call to illustrate the complete index creation.

```
1 try {
2   readFile();
3 }
4 catch (IOException e) {
5   String errorMessage = e.getMessage();
6   JOptionPane.showMessageDialog(null, errorMessage);
7 }
```

From this snippet, using a lookback of 5 identifiers, the following association entry would be created, associating the set of terms with a method for opening a message dialog:

$$\{io, except, string, error, messag, get\} \rightarrow \\ javax.swing.JOptionPane\#showMessageDialog(\ Component, Object)$$

⌟

6.3.3 Index Query

A query to the index consists of a set of terms extracted from the code preceding the current cursor position within a source code editor. The extraction of the set of context terms is done as described in the section on the common processing steps.

For a query to the index, given by a set of terms, we determine those association entries in the index whose term set is most similar to the query term set. For this, a notion of *similarity* between sets of terms is required. We experimented with two alternative similarity measures known from data mining [105]: The *Jaccard* and the *cosine* similarity.

Jaccard Similarity

The Jaccard similarity (js) is defined for arbitrary sets and is given by the ratio between the size of the intersection and the size of the union of the sets:

$$js(T_1, T_2) = \frac{|(T_1 \cap T_2)|}{|(T_1 \cup T_2)|}$$

Example

$$js(\{file, input, read\}, \{file, write\}) = \frac{|\{file\}|}{|\{file, input, read, write\}|} = 0.25$$

⌟

Cosine Similarity

The cosine similarity (cs) is defined as the cosine of the angle between two vectors in a vector space (see Section 2.5). We interpret term sets as vectors in the n-dimensional boolean vector space $\mathbb{B}^n$ such that each component corresponds to one term and thus n is the overall number of terms. A component of "1" indicates that a term is in the set, a component of "0" means that the term is not in the set. The cosine similarity for two vectors a and b is given by:

$$cs(\vec{a}, \vec{b}) = \frac{\sum_{i=1}^{N} a_i \times b_i}{\sqrt{\sum_{i=1}^{N} (a_i)^2} \times \sqrt{\sum_{i=1}^{N} (b_i)^2}}$$

We can reformulate the cosine similarity for sets of terms (which is also referred to as *binary cosine similarity*):

$$cs\_terms(T_1, T_2) = \frac{|T_1 \cap T_2|}{\sqrt{|T_1|} \cdot \sqrt{|T_2|}}$$

Example

$$\begin{aligned} & cs\_terms(\{file, input, read\}, \{file, write\}) \\ = & \frac{|\{file\}|}{\sqrt{|\{file, input, read\}|} \cdot \sqrt{|\{file, write\}|}} \approx 0.41 \end{aligned}$$

⌟

Both similarity measures have the range $[0.0..1.0]$ where 1.0 represents the highest possible similarity and 0.0 the lowest possible similarity respectively.

Based on this similarity measure, a set of method recommendations is built by successively considering the most similar index entries with decreasing similarity. For each similar entry, the associated method is added to the recommendation set until it contains a configurable number of recommendations. Since different term sets can be associated with the same method, we may have to consider more entries than the number of required recommendations. Moreover, since multiple entries can have the same similarity to the query, it can occur that we cannot add all methods associated with these entries to the recommendation set because we already have reached the desired amount of recommendations. In that case we have to arbitrarily choose from the entries with equal distance until we have obtained the required number of recommendations.

6.4 Case Study

This section presents the case study that we performed to evaluate our approach.

6.4.1 Reimplementation of Rascal

For a quantitative comparison to our approach, we chose Rascal [71] as a representative for structure-based approaches. It is most similar to our approach, since it also recommends single API methods based on the development context. Instead of identifiers in the context, it uses the methods already employed in a class to derive what method might be needed next. Since Rascal is not publicly available and we could not obtain the implementation from the authors, we carefully reimplemented the approach as described in their paper. In this section, we briefly describe their approach and present the results of an evaluation of our reimplementation.

Approach

Rascal uses collaborative filtering (see Section 2.5.5) which is used to predict the preferences of a user regarding how they "like" a particular item based on what other items the user likes and what items similar users like. Adopted for API method recommendation, Rascal interprets Java classes as users and API methods as items. A user-item preference database is created from existing software that stores for each class the API method calls it contains. The method usage of a class is modeled as a vector in the vector space $\mathbb{N}^n$. Each component of a vector represents as a natural number how often a particular method was called by the class and thus n is the overall number of distinct methods used collectively by all classes. For computing a recommendation set of methods, a set of already employed methods in a class under development is matched against the database. Depending on a query vector, they compute the nearest neighbors with different approaches, of which cosine similarity performs best. For the recommendation, they use content-based filtering to predict how a query user likes a particular item. They determine how a query user likes the items that are collectively used by its nearest neighbors. The recommendation set then consists of the 5 items liked most.

Reevaluation

To assess the validity of our reimplementation, we first attempted to reproduce the results reported by McCarey et al. [71] by reenacting their evaluation setup as closely as possible. They tried to predict method calls to the Java Swing API[1] in classes taken from 30 GUI applications. Since they did not report what exact applications were used in their case study, we used 5 open source Java applications from our study objects (see Table 6.1) that use the Swing API. We created the user-item preference database from them. As in the paper from McCarey et al., we consecutively removed Swing methods from the end of the class and queried the index with the sequence of the remaining Swing methods in the class. The recommendation set was considered correct, if the removed method was in the recommendation set. McCarey et al. report a recommendation rate of

[1] http://docs.oracle.com/javase/6/docs/technotes/guides/swing/

43% achieved in their experiment. The user-item preference database created in their experiment contained 228 classes with calls to 761 distinct Swing methods.

The user-item preference database created for the 5 Java applications in our reevaluation contained 1,014 classes with a total of 22,963 calls to 1,942 distinct Swing methods. The average fraction of correct recommendations for the systems ranged from 40% to 44%. This indicates that our reimplementation of Rascal achieves recommendation rates close to the number reported by McCarey et al. for the cosine similarity based implementation (43%). Thus, we are confident that we correctly reimplemented their approach.

6.4.2 Research Questions

To evaluate our approach, we investigate the following six research questions.

RQ 1 *How do the parameters of the approach impact the recommendation rate?*

We investigate how the ratio of correct recommendations depends on the parameters of our approach. The best parameter settings are selected to optimize the performance of the system and used as parameter configuration for the following research questions.

RQ 2 *How does ignoring well-known methods affect the recommendation rate?*

For employing a recommendation system in practice, recommending API methods that are already well-known to a developer is not desirable. We analyze how the rate of correct recommendations is affected when ignoring well-known methods in index creation and index query.

RQ 3 *How does the approach perform for APIs other than the Java API?*

To assess the transferability of our approach, we determine the quality of the recommendations for the Eclipse API.

RQ 4 *How does the approach compare to a method usage-based approach?*

To further evaluate our approach, we analyze how the recommendation results compare to those from the method usage-based approach Rascal. Furthermore, we investigate the potential of a hybrid approach.

RQ 5 *Can we use the similarity measure to derive a confidence level for recommendations?*

We investigate whether the similarity measures of the approaches can be used for deriving a meaningful confidence level for a recommendation set, indicating the likelihood that a recommendation produced by the system is relevant to the user.

RQ 6 *How are the recommendation rates for cross-project vs. intra-project recommendations?*

To investigate the impact of the consistency of identifiers, we compared the recommendation rates for cross-project recommendation (*i.e.*, mining associations from a set of projects and evaluating the recommendations for another project not contained in the set) and intra-project recommendation (*i.e.*, mining associations from parts of a project and evaluating the recommendations for the other part of the same project).

6.4.3 Study Objects

Table 6.1 lists the study objects that we used for our case study together with the version and a description.

Table 6.1: Overview of study objects

System	Version	Description
DrJava	stable-20100913-r5387	Java Programming Environment
FreeMind	0.9.0 RC 9	Mind Mapper
HSQLDB	1.8.1.3	Relational Database Engine
JabRef	2.6	BibTeX Reference Manager
JEdit	4.3.2	Text Editor
SoapUI	3.6	Web Service Testing Tool
MyTourbook	11.3	Bike Tour Visualization and Analysis Tool
Rodin	1.4.0	Event-B Modeling and Verification Environment
RssOwl	2.0.6	RSS / RDF / Atom News Feed Reader

Table 6.2: Details of studied Java and Eclipse RCP applications

Studied API	System	LOC	MCAPI	DM	BLRR
Java	DrJava	160,256	21,090	2,026	11.8%
	FreeMind	71,133	8,725	1,439	12.9%
	HSQLDB	144,394	9,735	1,100	24.0%
	JabRef	109,373	21,350	1,691	18.1%
	JEdit	176,672	17,341	1,934	10.5%
	SoapUI	238,375	24,659	2,500	11.3%
Eclipse	MyTourbook	238,963	18,865	1,160	12.3%
	Rodin	273,080	5,924	1,122	7.5%
	RssOwl	174,643	12,774	1,199	12.6%

Table 6.2 shows for each study object which API was studied (column *Studied API*) and a set of metric values. The column *LOC* shows the size of each study object in number of lines of code, which ranges between about 71 kLOC and 273 kLOC. The column *MCAPI* denotes the total number of method calls to the studied API with values between 5,924 and 24,659. The column *DM* shows the overall number of distinct API methods called within the source code. The study objects call from 1,100 to 2,500 distinct methods. To put the measurements of the recommendation rates into

perspective as well as to be able to assess the difficulty of the recommendation problem, we also computed a baseline recommendation rate for a trivial approach that always recommends the 5 API methods used most frequently within that project. The values of the baseline rate are presented in column *BLRR*, ranging between 7.5% and 24.0%.

We used the study objects with the different APIs for the research questions as follows.

Java API: For RQ 1, RQ 2, RQ 4, RQ 5, and RQ 6 we studied the Java API. We used 6 popular Java projects of different application types from the open source project repository SourceForge[2]. All projects were among the 100 most downloaded Java applications with the development status Production/Stable[3].

Eclipse API: For RQ 3, we studied the API provided by Eclipse for building Rich Client Platform (RCP) applications. We used 3 open source Eclipse RCP applications. To be representative, we chose applications from completely different domains—geovisualization, software modeling, and news reading. Since we also needed the binaries of the applications for our analyses, we were restricted to applications where we could easily download or compile the binaries. We considered as API all callable methods that are defined by classes whose full qualified name starts with `org.eclipse`.

6.4.4 Design and Procedure

This section first describes the basic procedure used for the evaluation and then discusses the steps for each individual research question.

Basic Procedure

To evaluate the proposed approach, we used the most common way for evaluating recommendation systems, namely the *evaluation on historical datasets* [47]. The basic idea is to randomly split the data into training data and evaluation data, train the recommendation system on the training data and evaluate the recommendations using the evaluation data. In our case of API method recommendation, we use as historical datasets software systems already developed. We use the code of existing systems to assess how appropriate recommendations would have been during the construction of these systems. Therefore, we "remove" method calls from a set of classes and use the recommendation system to "guess" the removed method call from its context—in case of our approach given by the identifiers preceding the method call.

To quantify the appropriateness of the recommendations, we measured the probability that a given method call is actually part of the top N recommendations (this probability metric is also referred to as *hit ratio*). It is important to note that this metric can only be interpreted with respect to the value of N.

[2] http://sourceforge.net/
[3] as of April 27th, 2011

For our case study, we define the *recommendation rate* as follows. Let $MCWC$ be the set of method calls with a non-empty context, *i.e.*, where the recommendation system is applicable and can make a recommendation. Let $m(c)$ be the method targeted by the method call c, ctx the function that yields the context for a given method call and $query$ the function that returns a set of recommended methods for a given context. The recommendation rate is then given by:

$$RR = \frac{|\{c \in MCWC \mid m(c) \in query(ctx(c))\}|}{|MCWC|}$$

Intuitively, the recommendation rate is the fraction of method calls that can be "predicted" by the recommendation system based on the context of each method call.

Example

Let us assume the source code of a software system contains three method calls c_1, c_2, c_3 with an appropriate context, *i.e.*, enough identifiers according to the configured lookback can be extracted. The method calls are targeted at two API methods m_1 and m_2 as follows:

$$\begin{aligned} m(c_1) &= m_1 \\ m(c_2) &= m_2 \\ m(c_3) &= m_2 \end{aligned}$$

The API provides another method m_3 which is not called in the source code of the system. The contexts of the three method calls are given by

$$\begin{aligned} ctx(c_1) &= \{file, read\} \\ ctx(c_2) &= \{file, write\} \\ ctx(c_3) &= \{content, lines\} \end{aligned}$$

The recommendation system is queried with the three contexts and returns the following recommendations:

$$\begin{aligned} query(ctx(c_1)) &= query(\{file, read\}) = \{m_1\} \\ query(ctx(c_2)) &= query(\{file, write\}) = \{m_2\} \\ query(ctx(c_3)) &= query(\{content, lines\}) = \{m_1, m_3\} \end{aligned}$$

Thus:

$$\begin{aligned} m(c_1) &\in query(ctx(c_1)) \\ m(c_2) &\in query(ctx(c_2)) \\ m(c_3) &\notin query(ctx(c_3)) \end{aligned}$$

The recommendation rate is then given by:

$$RR = \frac{|\{c_1, c_2\}|}{|\{c_1, c_2, c_3\}|} = \frac{2}{3}$$

⌟

For the research questions RQ 1 to RQ 5, we mined the association index and the user-item preference database respectively from half of the project files and used it to predict the method calls in the other half of the files (*intra-project recommendation*). The sets of files were determined randomly with a fixed random seed ensuring equal results in consecutive runs. For RQ 6, we mined the index from a set of "training projects" and produced recommendations for a different project not contained in the training set (*cross-project recommendation*). The recommendation sets contained $N = 5$ methods each[4].

The following paragraphs detail the specific study design and procedure separately for each research question.

RQ 1 To determine the impact of the algorithm parameters, we ran our analysis with different configurations. First, we determined the influence of different lookback values and similarity measures. We then used the best settings (w.r.t. the recommendation rates achieved) for these parameters to evaluate the influence of the parameters stopwords and keywords.

RQ 2 We analyzed how the recommendation rate is affected when a set of well-known methods is ignored completely during index creation and query. We approximated this set of well-known methods by the set of all methods of the 20 types used most frequently in 76 open source applications as reported in a study about the usage of the Java API by Ma et al. [69]. This list of well-known classes is shown in Table 6.3.

RQ 3 To assess the transferability of our approach to other APIs, we evaluated the recommendation rate for the 3 Eclipse RCP applications in Table 6.1.

RQ 4 For the quantitative comparison of our approach with the reimplementation of Rascal, we evaluated the recommendation rate for the 6 Java applications and both approaches. Both approaches have different sets of cases where they are applicable. This has to be taken into account for a quantitative comparison. Figure 6.2 illustrates the applicability for method predictions of both our approach and Rascal. The outer rectangle *MC* denotes the set of all method calls in the classes used for evaluation. The set *MCAPI* contains all method calls to the API under consideration, which is the Java API in the comparison. As previously described, our approach needs a context to derive identifiers from, and Rascal needs at least one method call to be able to make recommendations. Therefore, there are cases where Rascal can make recommendations and our approach is unable to do so, and vice versa. This is illustrated by the two rectangles *Our approach applicable* and *Rascal applicable*. We perform two comparisons. First, we evaluate the recommendation rates for both approaches for all cases where they are applicable. Second, we determine the recommendation rate for cases where both approaches are applicable, *i.e.*, we consider the method calls in the dark gray intersection in Figure 6.2.

We also evaluated in how many cases one of the approaches can predict a method and the other cannot. From these numbers, we computed the recommendation rate of a hypothetical ideal hybrid

[4] In our opinion, a developer is willing to inspect 5 potentially useful method recommendations. This is also the value used by McCarey et al. in [71] and thus allows us to directly compare the recommendation rate values.

Table 6.3: List of well-known classes from [69]

java.io.File
java.io.IOException
java.lang.Class
java.lang.Exception
java.lang.Integer
java.lang.Object
java.lang.String
java.lang.StringBuffer
java.lang.Throwable
java.net.URL
java.sql.Connection
java.sql.SQLException
java.util.ArrayList
java.util.Collection
java.util.Iterator
java.util.List
java.util.Map
java.util.Properties
java.util.Vector
javax.swing.text.Element

approach that assumes a perfect oracle which can decide which recommendation set should be returned as an answer to a query. The oracle always decides for the approach that has the correct recommendation. In case both recommendation sets are correct, an arbitrary approach is chosen. To build a real hybrid approach, we would need to implement such an oracle. RQ 5 takes a step in this direction by investigating a confidence level for recommendation sets.

RQ 5 To implement an oracle, we attempted to compute a confidence level for both approaches that allows us to decide for an approach depending on the query and the recommendation set produced by both tools.

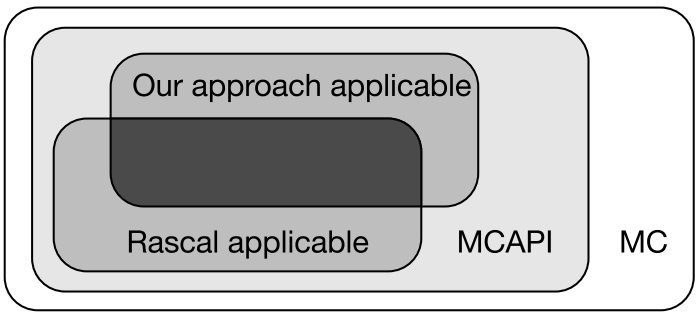

Figure 6.2: Applicability of approaches

To compute a confidence level for a given recommendation set, we used the similarity between the query and the nearest neighbor (in case of Rascal) or the most similar association entry (in case of our approach) respectively for a query. For Rascal, the similarity measure is given by the cosine similarity and for our approach it is the Jaccard similarity. Formally, this can be expressed as follows. Let $nn(q)$ be the nearest neighbor or most similar index entry for a query q. Then the confidence of the recommendation for given a query q is given by:

$$c(q) = similarity(q, nn(q))$$

We related the confidence level to the recommendation rate. For both approaches, we computed the average recommendation rate over all 6 Java applications within 10 confidence intervals from 0 to 1 in steps of 0.1. The average was computed over all method calls in all applications.

RQ 6 For this research question, we used 6 additional software systems to obtain an index of sufficient diversity with regards to identifier naming. These 6 additional systems are shown in Table 6.4. We only used these additional systems for mining associations but not for the validation.

We performed 6 evaluations for the Java API with the study objects from Table 6.1, in which we used an index constructed from 11 projects and evaluated the recommendations for the remaining project.

Table 6.4: The 6 additional software systems used for the evaluation of cross-project recommendation

System	Version	Description	LOC
Azureus/Vuze	4504	P2P File Sharing Client	786,865
iReport-Designer	3.7.5	Visual Reporting Tool	338,819
OpenProj	1.4	Project Management	151,910
RODIN	2.0 RC 1	Service Development	273,080
SQuirreL SQL Client	Snapshot-20100918_1811	Graphical SQL Client	328,156
TV-Browser	3.0 RC 1	TV Guide	187,216

6.4.5 Results

This section presents the results of the case study separately for each research question.

RQ 1: Influence of Parameters

Table 6.5 presents the recommendation rates for lookback values ranging from 1 to 6 identifiers and the two similarity measures. The rate of correct recommendations varies between 24.8% and 47.0%. The difference between the two distance measures is on average below 1%, whereby the Jaccard measure yields slightly better results. The rate of correct recommendations varies only moderately for different lookback values. On average, the best recommendation rate is achieved

Table 6.5: Recommendation rate regarding lookback and similarity measure

Distance	Lookback	DrJava	Freemind	HSQLDB	Jabref	JEdit	SoapUI	Avg.
Jaccard	1	33.6%	33.1%	24.8%	37.0%	34.7%	37.0%	33.4%
	2	38.9%	38.3%	34.5%	45.3%	38.8%	46.8%	40.5%
	3	39.4%	39.4%	33.9%	45.8%	39.6%	46.5%	40.8%
	4	40.5%	39.0%	35.9%	47.0%	39.6%	45.6%	41.3%
	5	38.7%	38.8%	36.8%	46.1%	38.6%	44.9%	40.7%
	6	39.4%	38.5%	38.1%	44.9%	38.2%	43.9%	40.5%
Cosine	1	33.6%	33.1%	24.8%	37.0%	34.7%	36.9%	33.4%
	2	38.7%	37.4%	32.9%	45.1%	38.6%	46.9%	39.9%
	3	39.2%	39.0%	33.8%	45.7%	39.3%	46.5%	40.6%
	4	40.0%	39.0%	35.4%	47.0%	39.7%	45.4%	41.1%
	5	38.5%	38.8%	36.7%	46.2%	38.9%	45.0%	40.7%
	6	39.3%	38.3%	37.8%	45.0%	38.5%	43.9%	40.5%

with a lookback of 4 for both distance measures. Consequently, for the following experiments, we use a lookback of 4 identifiers and the Jaccard similarity.

Table 6.6 shows the results regarding the impact of stopwords. Removing stopwords has a very small effect on the recommendation rate. In all cases, the difference is below 0.6%. For two of the 6 study objects, the rate slightly increases while for the other four, the rate slightly decreases. Consequently, the stopwords parameter is set to preserved during subsequent analyses.

Table 6.6: Recommendation rates with stopwords preserved/removed

Project	Stopwords	
	preserved	removed
DrJava	40.5%	39.7%
Freemind	39.0%	39.7%
HSQLDB	35.9%	36.0%
Jabref	47.0%	46.7%
JEdit	39.6%	39.0%
SoapUI	45.6%	45.2%
Average	41.3%	41.1%

Table 6.7 presents the recommendation rates when keywords are excluded vs. included. The results show that taking keywords into account has a notable positive effect on the recommendation rates for all study objects. The analysis including keywords is performing 3.8% to 5.6% better compared to the analysis with excluded keywords. Thus, we configure the parameter keywords to included for the next experiments.

Based on the outcomes of RQ 1, the parameter configuration for the remaining analyses is as follows:a lookback of 4 identifiers, stopwords preserved, and keywords included.

Table 6.7: Recommendation rates with keywords excluded/included

Project	Keywords	
	excluded	included
DrJava	40.5%	44.3%
Freemind	39.0%	43.1%
HSQLDB	35.9%	41.6%
Jabref	47.0%	52.1%
JEdit	39.6%	45.1%
SoapUI	45.6%	51.2%
Average	41.3%	46.2%

RQ 2: Well-known Methods

Table 6.8 displays the results of the analysis when including vs. ignoring well-known methods. Ignoring well-known methods decreases the recommendation rate between 0.7% and 11.4%.

Table 6.8: Recommendation rates with well-known methods included/ignored

Project	Well-known methods	
	included	ignored
DrJava	44.3%	42.8%
Freemind	43.1%	32.2%
HSQLDB	41.6%	30.1%
Jabref	52.1%	50.5%
JEdit	45.1%	44.4%
SoapUI	51.2%	48.3%
Average	46.2%	41.4%

RQ 3: Other APIs

Table 6.9 shows the recommendation rates of our approach applied to the three Eclipse RCP applications. The resulting recommendation rate ranges between 49.4% to 67.8%.

Table 6.9: Other APIs

Project	RR
MyTourbook	67.8%
Rodin	49.4%
RssOwl	57.5%

RQ 4: Comparison to Rascal

Table 6.10 shows the results of the comparison to Rascal regarding the individual applicability of both approaches. Column $Appl$ shows in how many cases each of the approaches can make recommendations. Rascal is applicable in 96.1% to 98.4% of the cases, whereas our approach can be applied in 68.9% to 89.1% of the cases. The columns RR_{appl} show the recommendation rates for the approaches taking only those method calls into account where the approach is applicable. For a better comparison, RR_{glob} shows the "global" recommendation rate, normalized for all method calls, *i.e.*, it is the product of columns $Appl$ and RR_{appl}. Globally, the recommendation rate for Rascal ranges between 24.4% and 32.0%, while our approach recommends the correct method in 33.1% to 43.1% of the cases.

Table 6.10: Comparison to Rascal

Project	Rascal			Our approach		
	$Appl$	RR_{appl}	RR_{glob}	$Appl$	RR_{appl}	RR_{glob}
DrJava	98.0%	30.5%	29.9%	79.9%	44.3%	35.4%
Freemind	97.2%	25.1%	24.4%	76.9%	43.1%	33.1%
HSQLDB	98.4%	32.5%	32.0%	89.1%	41.6%	37.1%
Jabref	97.8%	26.7%	26.1%	82.8%	52.1%	43.1%
JEdit	97.4%	25.3%	24.6%	73.5%	45.1%	33.1%
SoapUI	96.1%	26.9%	25.9%	68.9%	51.2%	34.8%

Table 6.11 shows the recommendation rates for the approaches in cases where both of them are applicable. For these calls, Rascal showed recommendation rates ranging from 19.1% to 33.4%. Our approach was able to recommend the correct method in 41.2% to 51.7% of the cases. The recommendation rate of the ideal hybrid approach is shown in column *Hybrid*. It ranges from 47.5% to 58.8%, which is an increase of 5.5% to 12.4% compared to our approach.

Table 6.11: Comparison to Rascal (shared applicability)

Project	Appl	RR		
	Shared	Rascal	Our appr.	Hybrid
DrJava	78.7%	31.9%	43.7%	56.1%
Freemind	75.2%	19.1%	42.0%	47.5%
HSQLDB	88.2%	33.4%	41.2%	52.1%
Jabref	81.9%	25.0%	51.7%	58.8%
JEdit	71.8%	26.7%	45.2%	54.0%
SoapUI	66.7%	24.7%	50.2%	57.4%

RQ 5: Confidence Level

The results of the confidence level analysis for the 6 Java applications are shown in Figure 6.3. For both Rascal and our approach the average recommendation rate relation is monotonically increasing

with the confidence interval, *i.e.*, the higher the computed confidence level, the more likely the recommendation set is correct.

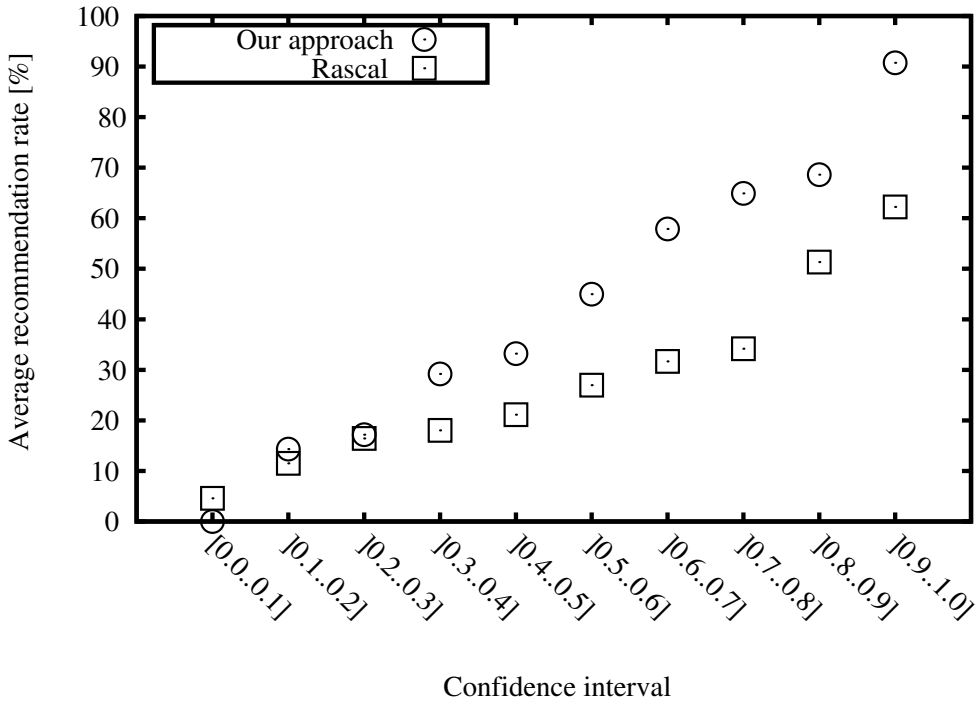

Figure 6.3: Average recommendation rate vs. confidence

RQ 6: Cross-Project Recommendation

The results of the cross-project recommendation analysis is shown in Table 6.12. For all study objects, the intra-project recommendation rate is higher than the cross-project recommendation rate. The difference of recommendation rate ranges between 5.7% and 16.2%, indicating a significant decrease.

Table 6.12: Cross-project recommendation

Project	Intra	Cross
DrJava	44.3%	33.2%
Freemind	43.1%	37.0%
HSQLDB	41.6%	34.7%
Jabref	52.1%	44.2%
JEdit	45.1%	39.4%
SoapUI	51.2%	35.0%
Average	46.2%	37.3%

6.5 Discussion

According to the results, our approach includes the correct recommendation in the returned set of 5 methods in about every second case. This is a significant improvement over a naive approach that always returns the 5 most frequent methods, which on average succeeds only in 1 out of 9 cases.

A central parameter of our approach is the lookback. As expected, taking too many identifiers into account decreases the recommendation rate, as potentially unrelated identifiers are considered. Taking too few identifiers also has a negative effect, since the descriptive power of the context is reduced. Consequently, there seems to be an optimum value in between, which we determined with our experiments.

In RQ 2, we investigated the effect of excluding well-known methods both from mining and the recommendation. As expected, the recommendation rates drop, since well-known methods are used frequently in the source code and are therefore easier to recommend. For the majority of the projects (4 of 6), the decrease is only moderate (below 3%). For two of the projects, we experienced a significant, although not threatening, reduction in the recommendation rate.

The results are consistently good for both the Java and the Eclipse API, indicating that the approach is not limited to a specific API. The slightly better results for the Eclipse API could be caused by having a more specific purpose than the Java API. However, to really answer this question, further experiments with other APIs are required.

When comparing our results to method usage-based approaches—here represented by Rascal—we find our approach to return the correct suggestion in 5.5% to 17.0% more of the cases. The reduced recommendation rates of Rascal compared to the numbers reported in [71] and our reevaluation of Rascal are likely to be caused by the different APIs used. The original experiments were limited to the Swing API, which is a subset of the Java API used in our setup. As the number of possible methods increases, the recommendation rate is expected to drop. Additionally, in the evaluation from [71], the class for which the recommendations are retrieved is part of the training set, while in our experimental setup, the files used for index creation and recommendation are disjunct. While we consider our setup more realistic, it can have an impact on the recommendation rate.

Given that both our approach and Rascal are applicable in slightly different contexts and recommendation rates of about 50% still allow further improvement, a combined approach using both method usage and identifier contexts seems feasible. The results of RQ 4 suggest, that such a hybrid approach could add another 5.5% to 12.4% to the recommendation rates of our approach. The question of which results to use if both approaches are applicable for a context is partially answered by RQ 5. Our results show that the confidence level we suggested is a good predictor for the expected recommendation rate of both algorithms. Based on this, a recommendation system could calculate recommendations with both algorithms and present the one with the higher confidence level to the user. However, how good exactly this approach would be compared to the ideal hybrid approach is an open question for further research.

The confidence level determined in RQ 5 opens the path to another interesting application. A proactive recommendation system could be configured to actively suggest recommendations if the confidence level is sufficiently high, thus not interrupting programming if the results are unlikely to

help. Evaluating and assessing such a setting would require more details on the interaction of the recommendation system with the user.

Regarding the application in an interactive development environment, we measured the times for building and querying the index for each study object. Index construction time was between 30s and 231s. Typically, the construction of the index is performed only once in a preparation step and thus its duration is less important than the query time. The average index query took between 5ms and 26ms, which is sufficient for an interactive setup.

6.6 Threats to Validity

This section discusses the threats to the internal and external validity of the results presented in this study.

6.6.1 Internal Validity

The choice of returning 5 methods in the recommendation set is arbitrary and obviously affects the results. However, returning 5 methods seemed to be a suitable compromise from a programmer's perspective and is the same value used in the original evaluation of Rascal [71]. Additionally, as all numbers reported are based on 5 methods in the recommendation set, the numbers are comparable to each other.

The context for recommending methods also includes the identifiers in the line of the method call up to the position of the method call. Cases in which this line contains the variable declaration for a very specific return type limit the choice of possible methods and thus increase the probability of a correct recommendation set. However, a user might not know the return type of an appropriate method in advance.

Our evaluation is based on counting in how many cases the method actually used in the code is contained in the recommendation set produced for its context. This might be different from the recommendation rate actually perceived by a user. However, we argue that this is a one-sided error, as the method actually used in the code should be correct in any case, while a user might even find other methods in the recommendation set useful, even if they were not used in the implementation. Thus, the recommendation rates found in a study with subjects would be expected to be higher rather than lower.

6.6.2 External Validity

The results of our experiments might be biased by the choice of the study objects (which is also referred to as *selection bias*). We tried to mitigate this threat by choosing applications from different application domains. Obvious questions are also how the results transfer to commercial software systems (rather than open source) and other programming languages besides Java. While both are interesting questions for further research, we did not try to answer them in this study.

Our comparison to Rascal could be invalidated by an incorrect reimplementation, which could lead to an error in both directions. However, as explained in Section 6.4.1, we could reproduce the recommendation rates achieved in [71] with our implementation in a similar evaluation setup, which makes us confident that our implementation of Rascal resembles the published algorithm near enough for a valid comparison.

6.7 Summary

This chapter described an API method recommendation algorithm based on the identifiers in the development context. We experimentally determined optimal settings for the algorithm's variation points. With these settings, recommendation rates vary between about 42% and 68% if the context allows application. Compared to method usage-based approaches, the algorithm is correct in about 6% to 17% more of the cases. Finally, the confidence level can be used to predict the expected precision of both algorithms, opening the path to a hybrid approach and a more goal-oriented interaction with a user.

7 Beyond Code: Reuse Support for Model-Based Development

Modeling languages for data flow and processing models, as used, for instance, for control systems, often have a notion of *reuse*. Model elements can be composed hierarchically and reuse occurs by using elements multiple times. Moreover, the modeling language toolkit often provides collections of reusable elements in form of libraries. Since the amount of reusable elements can grow large, just as with classical code-based development, it is a challenge to find a reusable element for a given task. For code based development, recommendation systems have been proposed to alleviate this. These systems suggest library elements, such as methods or classes useful for the task at hand [34, 71, 109]. We transfer this idea to the field of model-based development.

In case of Mathworks Simulink (see Section 2.7), the development environment offers a large variety of predefined blocks arranged in block libraries. To use these block libraries, the Simulink development tool suite provides a library browser that allows navigating a hierarchical categorization of the blocks. Figure 7.1 shows a screenshot of the Simulink library browser. The 16 standard Simulink libraries consist of 135 blocks. In addition to the provided libraries, Simulink can be extended with custom libraries. A number of additional libraries are provided by Mathworks as well as by third party vendors.

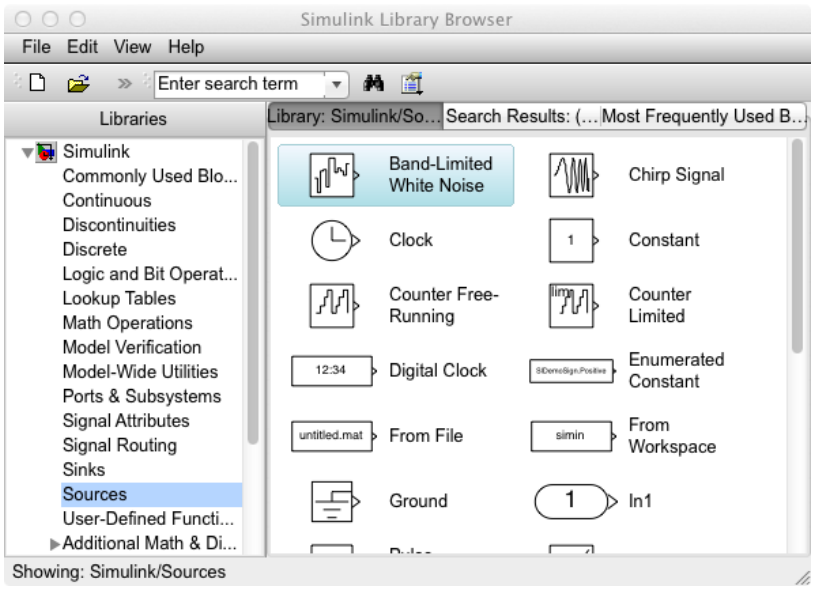

Figure 7.1: Simulink Library Browser

This chapter introduces an approach for the context-dependent recommendation of elements from reusable modeling libraries during model-based development. It uses data mining techniques to

extract knowledge from existing models, which is used to produce recommendations for unfinished models under development. We instantiate the approach for the Simulink modeling language and evaluate the quality of the recommendations with a case study using 165 model files from a public repository. We compare two variants for producing recommendations: association rules (ARs) and collaborative filtering (CF), which were introduced in Section 2.5.

Parts of the content of this chapter have been published in [33].

7.1 Approach

Our approach assumes that the modeling language has the concept of functional elements, which can be used for composing models. We abstract a model as the set of distinct elements used in it. Hence, we do not consider how often the elements are used or how they are interconnected. Our approach analyzes the distinct elements used in the models of a training corpus during the *training phase*. In the *recommendation phase*, elements that are not used yet are recommended for unfinished models. We implemented two variants of the model recommendation approach: an AR-based and a CF-based variant.

7.1.1 AR-based Recommendation System

Training phase We adopt ARs for model-based development by considering the set of distinct elements used in a model as shopping baskets and the individual elements as items. The mining process analyzes a corpus of training models by extracting the set of elements employed in the models. It uses the Apriori algorithm [1] to mine association rules, which are stored in a file for later use in the recommendation phase.

Recommendation phase Using the mined association rules, the recommendation system suggests additional model elements for partial models. For this, the recommendation system is given the set of elements employed so far in an incomplete model. The system iterates over all association rules and recommends the associated element of each applicable rule if it is not yet employed in the model.

7.1.2 CF-based Recommendation System

Training phase For the CF-based recommendation system, we consider the models as users and the model elements as items. As in the AR-based variant, we abstract how many times an element is used in a model. The training phase only consists of the extraction of the element usage for each model. We store a list of sets of model elements for later use in the recommendation phase.

Recommendation phase For a query, given as a set of model elements, the k most similar sets from the training phase are determined. All elements that the neighbors collectively use and that are not already used in the query model are returned as recommendations.

7.1.3 Instantiation for Simulink Models

Simulink models are a hierarchical composition of Simulink subsystems, in which the atomic units are Simulink blocks. For AR mining, each Simulink subsystem is considered as a shopping basket. Consequently, the ARs associate library blocks. As an example, the following AR means that models using the blocks $Gain$, $Integrator$ and $Constant$ typically also use the Sum block:

$$\{Gain, Integrator, Constant\} \rightarrow Sum$$

For CF, we consider Simulink subsystems as users and blocks as items. The block usage of a subsystem is encoded as an n-dimensional boolean vector, where each component refers to a specific block and thus n corresponds to the overall number of different blocks among all models. Within the vector, a component of "1" indicates that a particular block was employed (one or more times) in the model and a "0" means that it was not employed.

7.2 Case Study

As an evaluation of the proposed approach, we performed a case study with a set of Simulink model files with the goal of assessing the adequacy of the recommendations and determining the influence of the parameters for both approach variants.

7.2.1 Study Objects

We used as study objects 165 Simulink model files downloaded from the Matlab Central File Exchange[1]. We excluded uses of the blocks $Inport$, $Outport$ and $SubSystem$ completely, since these elements do not provide functionality on their own and occur very often in the models. We also filtered models that used only one type of block, since no association rules can be mined from them. Thereby, we obtained 1103 subsystems, which collectively used 335 distinct library blocks. Table 7.1 shows information on the number of distinct blocks used per subsystem.

Table 7.1: Number of distinct blocks used per subsystem

min	max	mean	p25	median	p75
2	16	4.72	3	4	6

To illustrate the recommendation problem, Figure 7.2 shows a log-scale diagram of the block usage frequency among the study objects. The distribution can be compared to the *long tail* effect in marketing [3]. While there are few blocks that are used very frequently, there is a large amount of blocks that are used infrequently but in sum account for a large fraction of the usages. As an illustration, if we exclude the 20 most used blocks, still about 35% of all block usages employ none of the "popular" blocks. In other words, during modeling, in 35% of the cases, a modeler needs a "non-commodity" block and could thus benefit from a recommendation system.

[1] http://www.mathworks.com/matlabcentral/fileexchange/

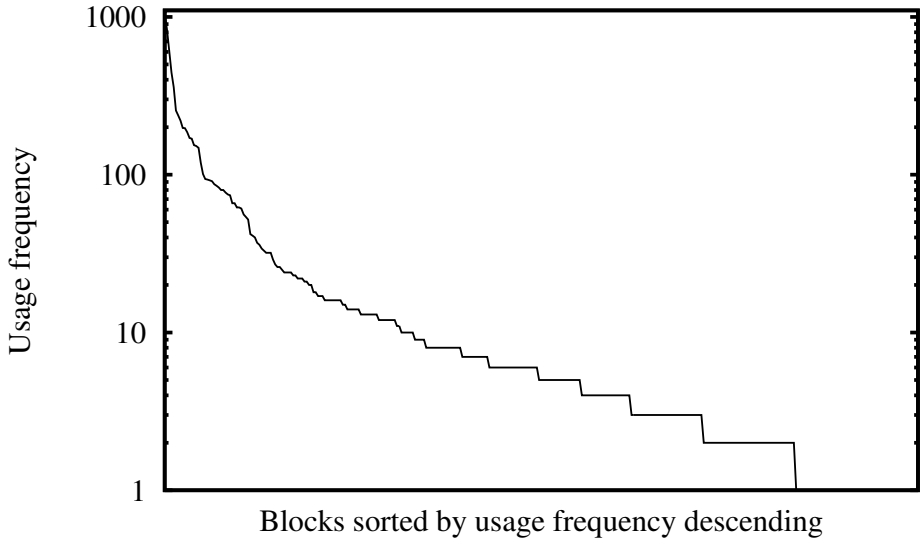

Figure 7.2: Log-scale diagram of block usage in study objects

7.2.2 Design and Procedure

Similar to the evaluation of the code-based recommendation system (see Chapter 6), we evaluated the model-based recommendation system based on *historical datasets*, in our case given by already developed Simulink models. More specifically, we performed a 10-fold cross validation, *i.e.*, we conducted 10 evaluations where in each we used 90% of the 1103 subsystems as the training set and the remaining 10% as the test set. To evaluate the approach for a given subsystem in the test set, we created an artificial unfinished subsystem by randomly removing half of its blocks and queried the recommendation system with the remaining half. Depending on the recommendations made and the blocks actually employed (removed previously), we measured the accuracy of the recommendations with precision and recall, two standard metrics for evaluating information retrieval systems [37]. In addition, we computed the F-measure, combining precision and recall into a single metric. Applied to the recommendation scenario, these metrics are computed as follows:

$$\text{precision} = \frac{correct\ recommendations}{total\ recommendations}$$

$$\text{recall} = \frac{correct\ recommendations}{actually\ employed\ blocks}$$

$$\text{F-measure} = \frac{2 \times precision \times recall}{precision + recall}$$

In addition to the two approach variants, we evaluated a trivial baseline recommendation system, which always recommends a certain amount of the most frequently used blocks. For the AR-based variant, we chose, as a result of preliminary experiments, 1% as support threshold. We evaluated different values for the confidence threshold. For the CF-based variant, we evaluated different values for k.

7.2.3 Results

Baseline Recommendation System Figure 7.3 shows the precision, recall and F-measure values for the baseline recommendation system. The results depend on the number of blocks recommended from the top used blocks. Returning more blocks increases the recall while decreasing the precision. For a value of 3 recommended blocks, the recommendation system performs best and achieves an F-measure of 0.19. For this setting the precision is 0.17 and the recall is 0.21.

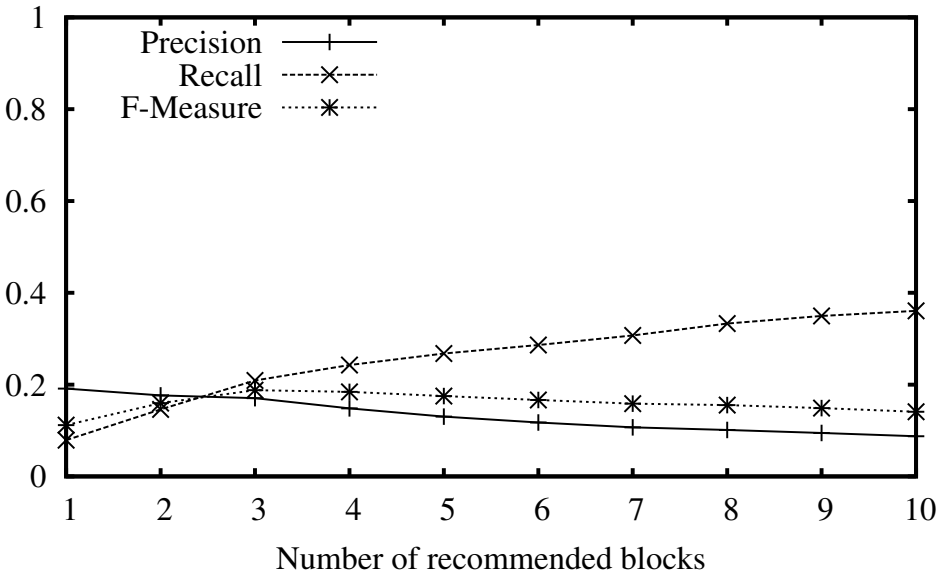

Figure 7.3: Baseline recommendation system

AR-based Recommendation System Figure 7.4 shows the results for the AR-based recommendation system for different values of the confidence threshold. As expected, the precision increases and the recall decreases with an increasing confidence threshold. For a confidence threshold of 0.4, the F-measure reaches the best value of 0.31. For this optimal setting, precision and recall are 0.32 and 0.30 respectively. Table 7.2 shows information on the number of recommendations returned per query for this setting.

CF-based Recommendation System Figure 7.5 shows the results for the CF-based recommendation system for different values of k (number of neighbors considered). The diagram shows that for increasing values of k, the recall is increasing whereas the precision is decreasing. The F-measure has its optimum value of 0.56 for $k = 1$, where the precision is 0.66 and the recall is 0.49. Information on the number of recommendations returned per query for $k = 1$ is shown in Table 7.2.

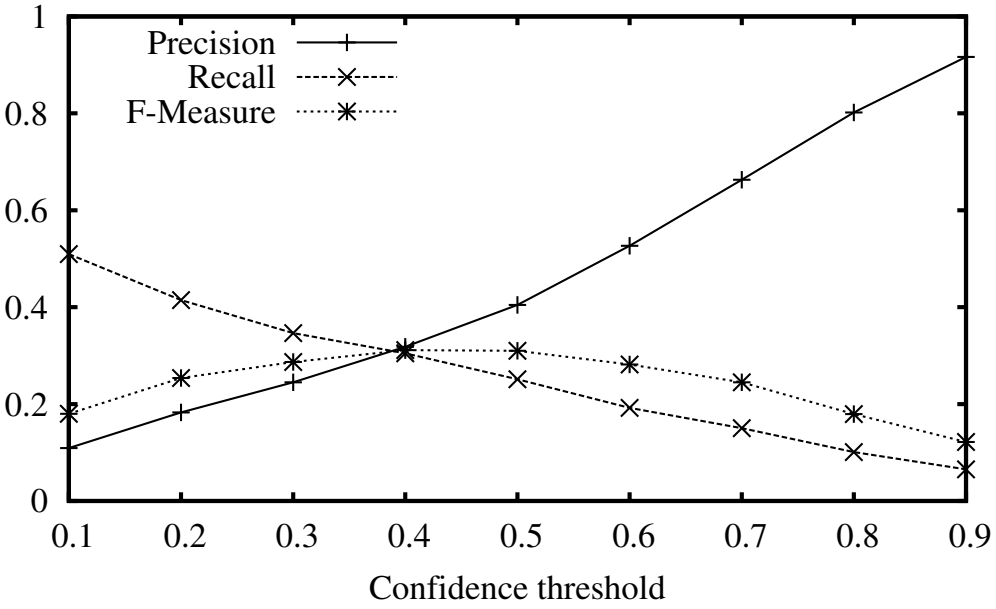

Figure 7.4: AR-based recommendation system

Table 7.2: Number of recommendations per query

	min	max	mean	p25	median	p75
AR (conf=0.4)	0	11	2.26	1	2	3
CF (k=1)	0	9	1.75	1	1	2

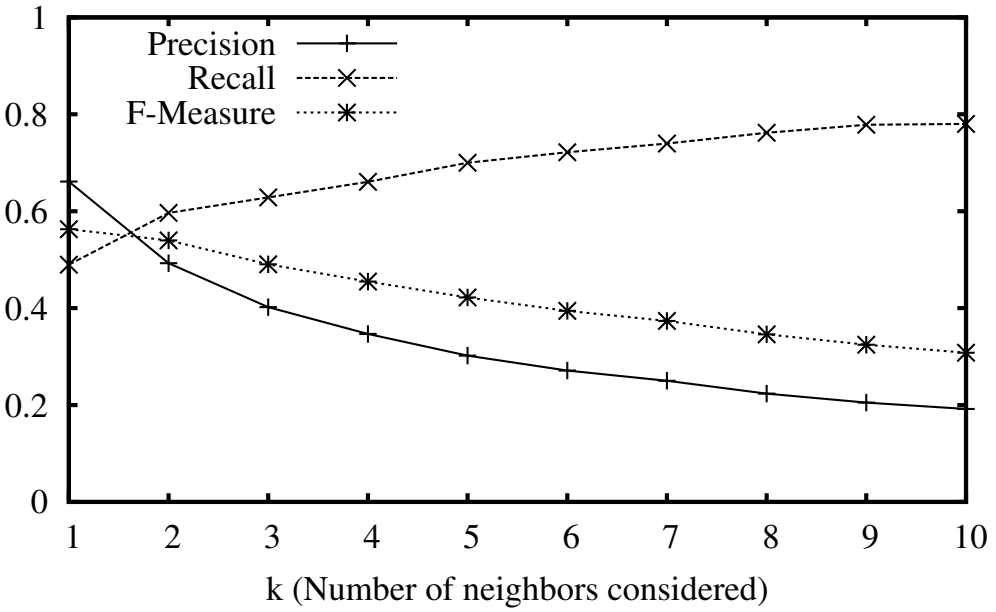

Figure 7.5: CF-based recommendation system

7.3 Discussion

The results show that both the AR-based and the CF-based variant of the recommendation system clearly outperform the baseline approach. The results also make the trade-off between precision and recall for both variants apparent, which depends on the chosen parameter values. While there is an optimum of the parameters with regards to the F-measure, the variability allows adjusting the recommendations to a user's preference. For instance, an unexperienced user might want to rather obtain more recommendations while accepting to also receive potentially irrelevant recommendations.

Moreover, it can be seen that the AR-based variant in general achieves better values for precision in comparison to the CF-based variant whereas for recall, the relation is the other way round. In terms of F-measure, however, taking the trade-off between precision and recall into account, the CF-based variant outperforms the AR-based variant.

7.4 Threats to Validity

This section discusses the threats to the internal and external validity of the study results.

7.4.1 Internal Validity

The so called *twinning problem* denotes duplicate or near duplicate values in the data set. If, during cross validation, one twin is in the training data set and the corresponding twin is in the test data set, better evaluation results are measured by mistake. We mitigated this with a simple duplicate detection on the Simulink files. We excluded files whose content was identical to another file.

We performed an automated evaluation by predicting removed blocks from subsystems. We thus do not know how useful the recommendations would be perceived by a user. However, we assume that users may consider additional recommended blocks as useful even if they are not used in the model.

The evaluation assumed that half of the blocks of a subsystem were already employed to produce recommendations. However, a user would ideally like to get recommendations with less blocks already employed.

We determined the query set for the evaluation randomly. However, the resulting partial subsystem might not correspond to an intermediate state as it would occur during modeling. This can lead to an error in both directions, *i.e.*, better or worse recommendation quality during real use.

7.4.2 External Validity

It is unclear how representative the study objects are for all Simulink models. However, since the models of the case study collectively used a total of 335 distinct blocks, we assume a certain diversity among the study objects.

Since we restricted our evaluation to Simulink models, we do not know how the approach transfers to other model-based development approaches. We consider this an important direction for future work.

7.5 Summary

We presented an approach for recommending useful model elements during model-based development that assists developers in using large and complex modeling libraries. Our results show that our approach can produce recommendations for Simulink models with satisfactory precision and recall.

8 Tool Support

Reuse of large and complex libraries should ideally be supported by appropriate tools. This chapter presents the recommendation system tooling that was developed as a proof-of-concept for the proposed approaches. We present two recommendation systems, one for code-based development and one for model-based development.

8.1 API Method Recommendation

The tooling for the recommendation of API methods was developed as a plug-in for the Eclipse Java IDE[1]. This enables a seamless integration of the recommendation system into the workflow of a Java developer. As introduced in Chapter 6, the API method recommendation approach consists of the *recommendation mining* and the process of *obtaining recommendations*. The following two sections present how these were realized within the tooling.

8.1.1 Recommendation Mining

The recommendation mining feature allows mining term-method associations from a set of Java projects in the workspace via a context menu action. Figure 8.1 shows how to initiate the mining process for a number of selected projects.

The mining process allows the user to set the necessary parameters as illustrated in Figure 8.2. The include and exclude prefixes define which classes are considered as belonging to the API and are used to decide whether a certain method call is considered as an API call. Furthermore, the output file which will hold the associations needs to be specified. Finally, the dialog allows setting whether keywords are taken into account and how many identifiers preceding the method call should be considered as context.

8.1.2 Obtaining recommendations

After the recommendation mining process is completed, the API method recommendations can be requested during the editing of Java code. A keyboard shortcut in the editor computes the context-dependent recommendations and displays them in a dedicated view, as depicted in Figure 8.3.

The view shows a list of API methods that are recommended for the current code context. The number of recommendations produced can be configured as a user preference. More relevant methods (with a higher confidence) occur earlier in the presented list. To allow the user to learn more about a

[1] http://www.eclipse.org/

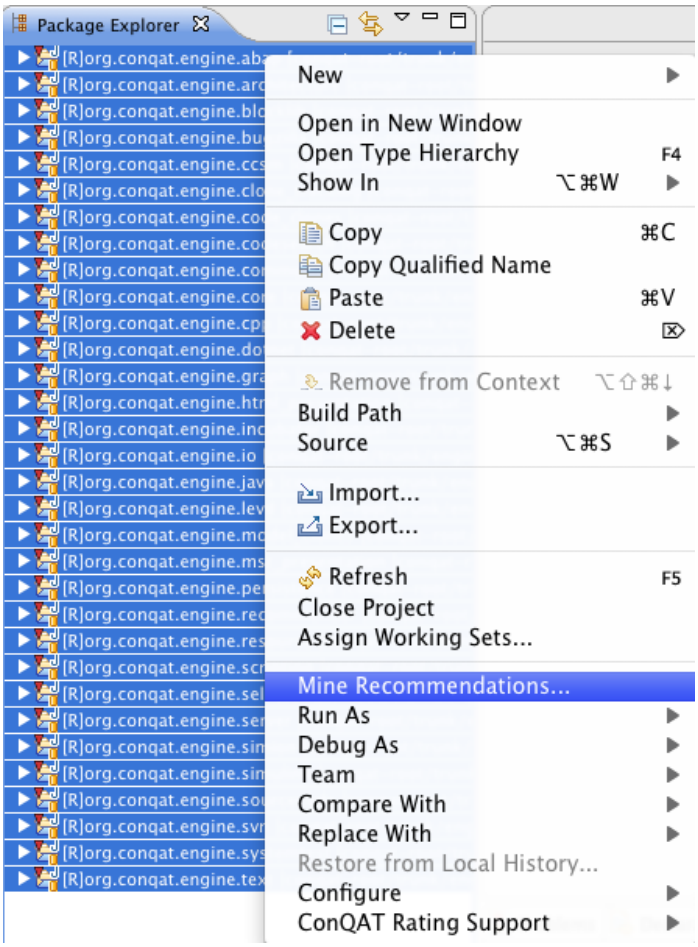

Figure 8.1: Initiating recommendation mining for selected projects

recommended method, the Javadoc view of Eclipse is linked with the recommendations view such that for the currently selected method the according documentation is displayed. Furthermore, a tooltip shows one or more usage examples of how the method can be employed in a context (these usage scenarios are those from the code analyzed in the mining process). Finally, a double-click on a recommended method navigates to the declaration of the method.

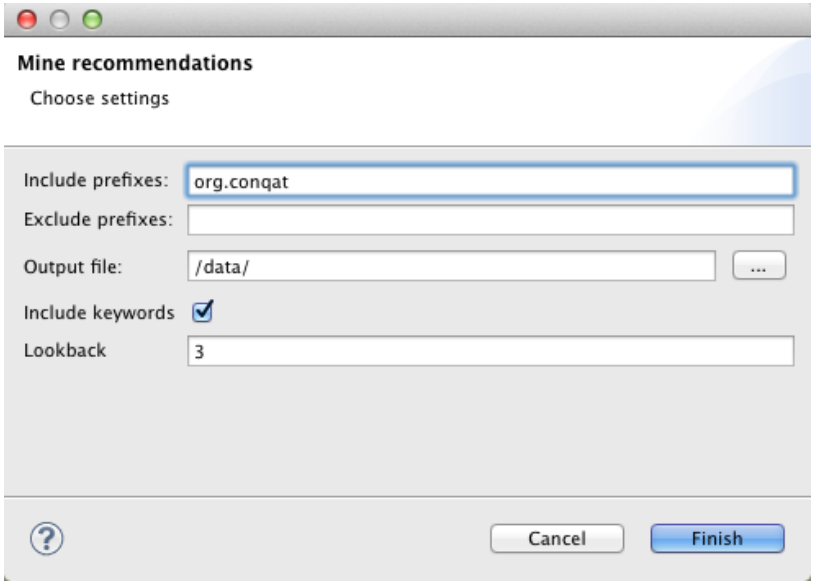

Figure 8.2: Setting parameters for recommendation mining

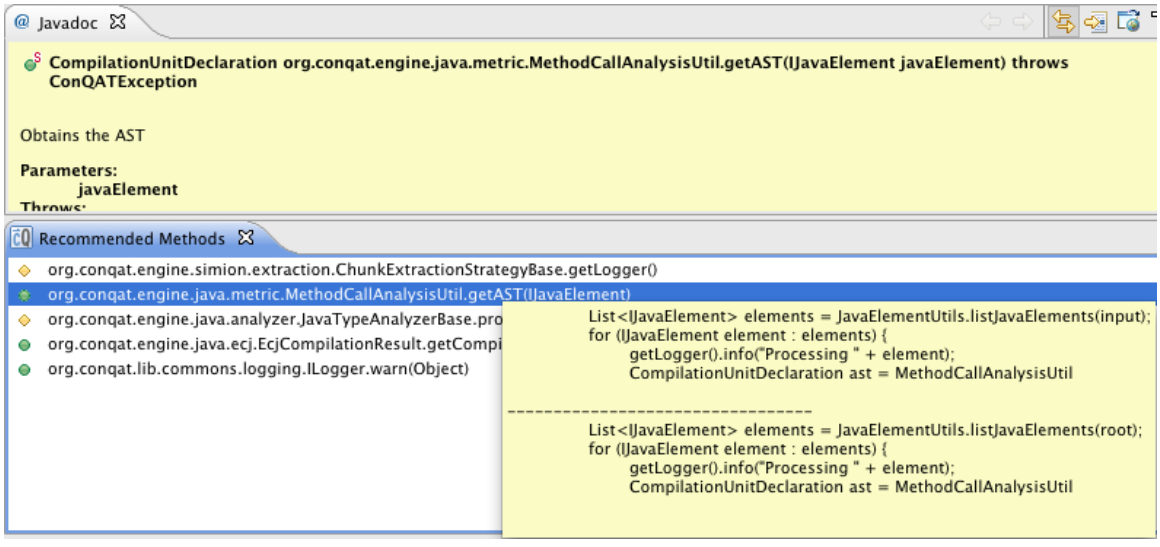

Figure 8.3: API Method Recommendation View

8.2 Model Recommendation

In Chapter 7, we developed a model recommendation system for Simulink models. Due to the proprietary nature of the Matlab/Simulink development environment, we could not build a user interface integration. To provide a proof-of-concept for the user interface part of a recommendation system for model-based development, we developed a recommendation system including a user interface integration for an open source modeling environment. For this, we used ConQAT (see Section 2.8) as a basis, which has a graphical model editor for quality assessment configurations.

ConQAT employs a graphical pipes-and-filters style modeling language to specify quality analyses. Quality analysis configurations are hierarchically composed of so-called *units*. Units can be either *processors*, ConQAT's atomic functional units, or *blocks*, which are composite units built of blocks and processors. As an illustration, Figure 8.4 shows the ConQAT Block Editor with a simple configuration that reads the source code of a system, counts the number of code lines for each file and produces a HTML table presenting the data.

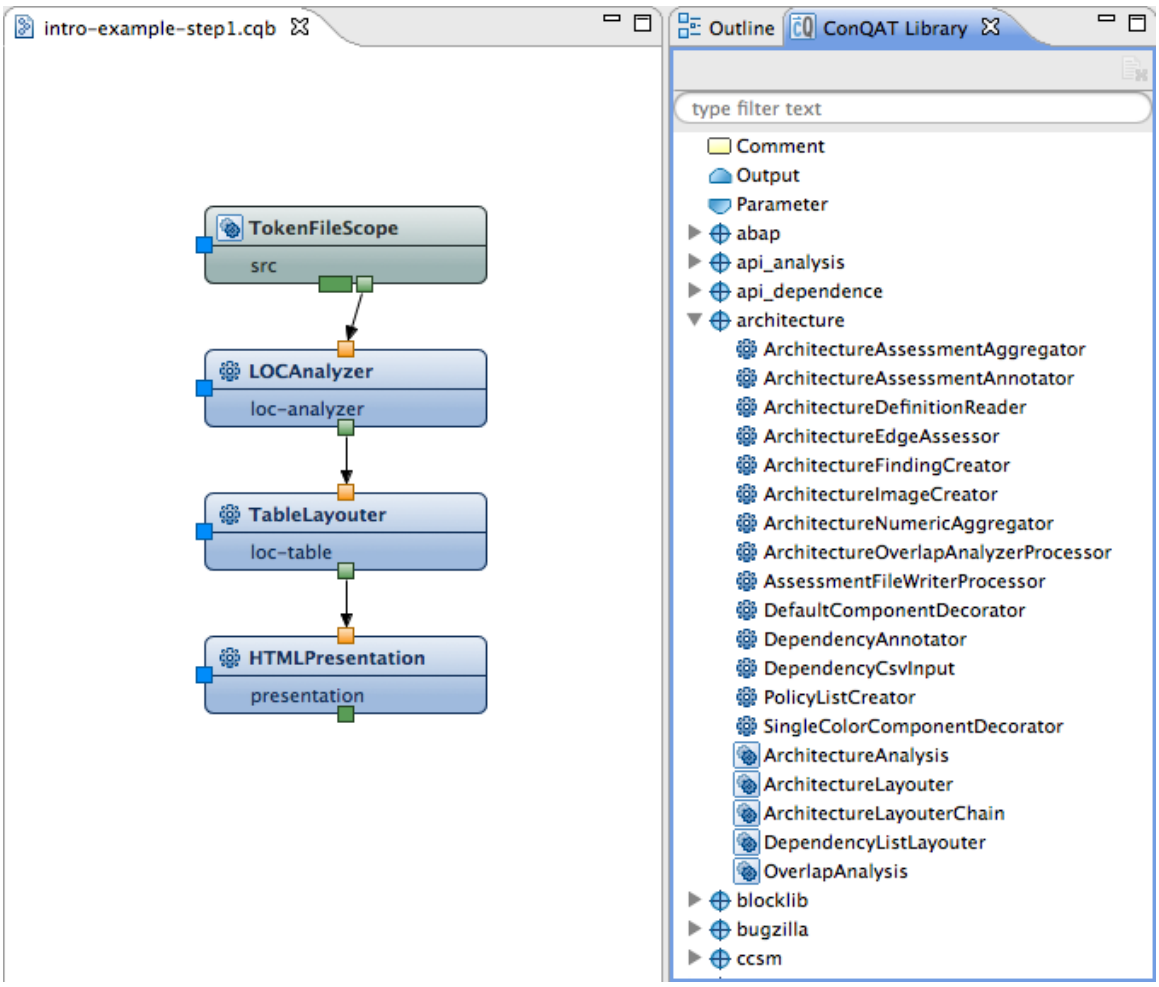

Figure 8.4: ConQAT Block Editor and Library Browser View

Realistic ConQAT configurations that analyze diverse quality characteristics can consist of hundreds of units. ConQAT provides a large variety of predefined quality analyses in the form of ready-to-use units. Therefore, the ConQAT block editor provides a library browser that presents the available library elements organized in 19 functional categories. Overall, several hundred library elements are available. Figure 8.4 (right) shows the ConQAT Library Browser view.

To assist the user in finding the right unit for a given task, the developed recommendation system plug-in provides a new view that gives context-specific recommendations for ConQAT library elements. Figure 8.5 illustrates the ConQAT block editor and the user interface of the model recommendation plug-in for ConQAT.

Depending on the block currently being edited, recommended units are displayed in a dedicated view. For each recommendation, a confidence level is shown in brackets. To further support the user, a tooltip shows the documentation for the recommended units. A recommended unit can directly be dragged and dropped from the recommendations view into the block editor. The view automatically updates the recommended elements in response to changes to the block being edited. The recommendation system is directly integrated in the development environment and thus does not interrupt the workflow of the user during modeling.

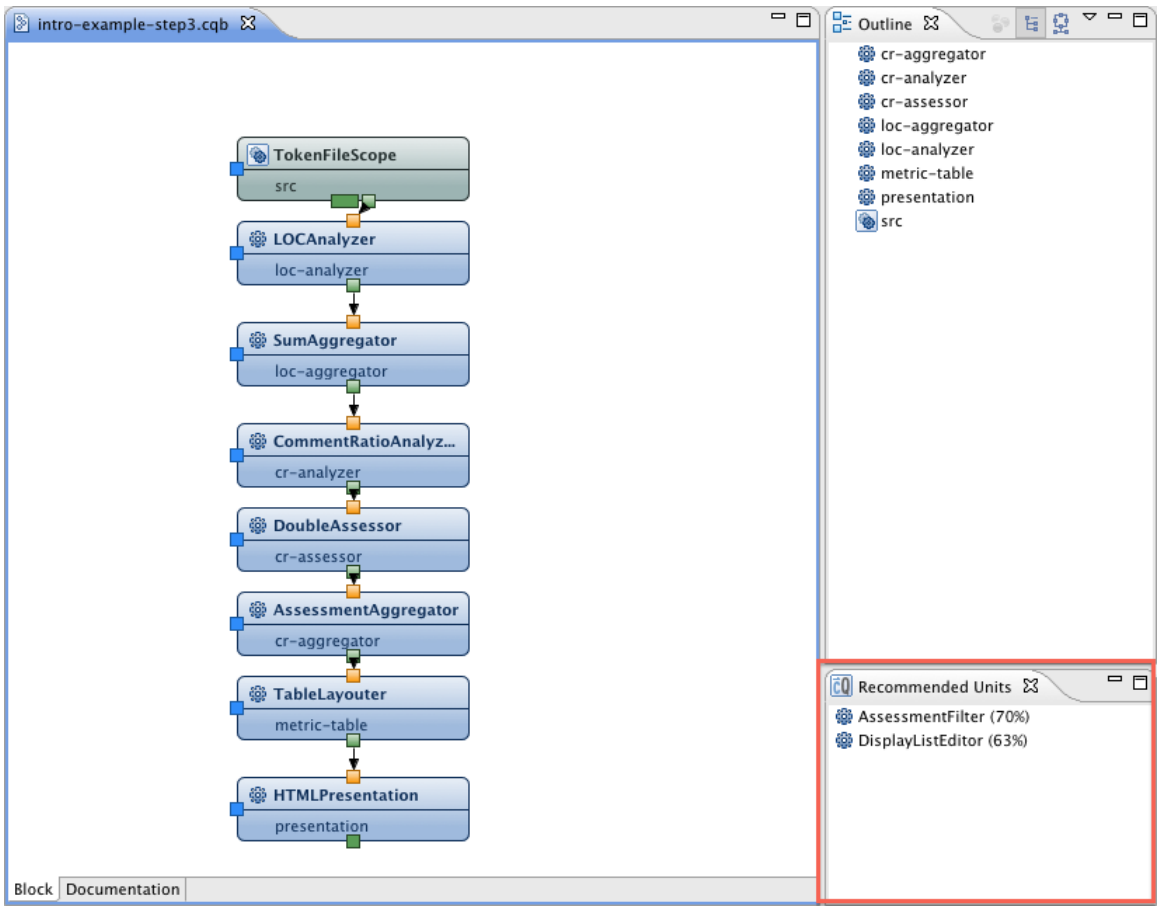

Figure 8.5: ConQAT Unit Recommendation System

9 Conclusion

This chapter presents the conclusions that can be drawn from the contributions in this thesis.

9.1 Third-Party Code Reuse in Practice

Software reuse, often called the *holy grail of software engineering*, has certainly not been found in the form of off-the-shelf reusable components that simply need to be plugged together to obtain a new software system. However, from our experience, reuse of third-party code is a common practice in many software projects, especially in projects using common programming languages like Java, where a large amount of free third-party code is available for reuse. Yet, we lacked detailed data about to what extent and in what forms software projects reuse third-party code.

To investigate the extent and nature of software reuse in open source Java development, we investigated four research questions:

RQ 1: *Do open source projects reuse software?*

Yes. We found reused code in 90% of the analyzed projects, thus indicating that reuse is a common practice in open source Java projects.

RQ 2: *What is the extent of white-box reuse?*

We detected white-box reuse in about half of the projects. However, the reuse rates for white-box reuse were only moderate and were all below 10%. Thus, white-box reuse is a fairly common practice, yet it occurs only to a limited extent. A possible explanation is the negative effect of copying&pasting source code on software maintenance as shown by clone detection research.

RQ 3: *What is the extent of black-box reuse?*

The study revealed considerable amounts of black-box reuse. 90% of the projects reused code in a black-box manner through integrating software libraries. The median reuse rate was about 45% and for about half of the projects the amount of reused code was larger than the amount of own code.

RQ 4: *What type of functionality is reused?*

The reused functionality was diverse among the projects. Most applications reused code from several functional categories and, overall, the usage of code from the different functional categories was fairly balanced.

Summary The study showed not only that reuse is common in almost all open source Java projects but also that significant amounts of software are reused: Of the analyzed 20 projects, 9 projects have reuse rates greater than 50%—even if reuse of the Java API is not considered. Fortunately, these reuse rates are, to a great extent, realized through black-box reuse of software libraries and not by copying&pasting (*i.e.*, cloning) source code.

We conclude that in the world of open-source Java development, high reuse rates are not a theoretical option but are actually achieved in practice. Especially, the availability of diverse reusable functionality, which is a necessary prerequisite for reuse to occur, is well-established for the Java platform.

9.2 Detection of Functionally Similar Code Fragments

Previous research proposed an approach for detecting functionally similar code fragments (*simions*) that splits the code of a software system into candidate code fragments, executes them on random input data and compares their output. A case study with the Linux kernel showed that such an approach yields high detection rates for C code. However, it was unknown if such an approach would produce similar detection results when adapted to object-oriented programs.

We transferred the basic idea of the approach to Java code and identified a number of challenges faced when attempting to detect functionally similar code fragments in five Java systems. The study investigated three research questions:

RQ 1: *How difficult is the simion detection problem?*

The study showed that the chunking strategy has a significant impact on the number of chunks extracted. While it is desirable to extract as many different chunks as possible in order to increase the probability that pairs of functionally similar code sections are found, a higher number of chunks results in higher computation time required for chunk execution. The most detailed chunking strategy extracted more than 200,000 chunks for a system with about 80 kLOC.

RQ 2: *How do technical challenges affect the detection?*

Object-oriented programs pose a number of unique challenges to simion detection. The study showed that input generation fails for a significant number of chunks, thus inhibiting simion detection for these candidate code fragments. Moreover, a considerable amount of the chunks required a specific environment, such as existing I/O, network, or UI resources.

RQ 3: *How effective is our approach in detecting simions?*

The detection pipeline detected only a small number of simions. The relative amount of chunks identified as simions was below 4% for all real software systems. Even these results still exhibited a considerable amount of false-positives (on average about 66%).

Summary Our study on the automated detection of functionally similar code fragments showed that the approach exhibits low detection results due to a number of challenges when employed for object-oriented software systems. The study results stress the importance of alternative constructive approaches that aim at *avoiding* the reimplementation of functionality already during the development of a system.

9.3 Developer Support for Library Reuse

Reuse recommendation systems can help developers using APIs more effectively thus fostering reuse of third-party libraries. Existing approaches are based on structural information in the code and are thus dependent on the usage of these elements. This thesis proposed an alternative approach that is solely based on the programer's intent embodied in the identifiers of a code context.

To assess the proposed approach, we investigated six research questions:

RQ 1: *How do the parameters of the approach impact the recommendation rate?*

The study showed that the similarity measure of the proposed approach has only a very limited effect on the recommendations. In contrast, the lookback has a notable effect. If it is set to only one identifier, the recommendation rate is significantly lower compared to considering more identifiers. Taking too many identifiers into account decreases the recommendation rate as well. The optimum value found in the experiments is four identifiers. Filtering stop-words has no notable effect on the recommendation rate while the inclusion of keywords increases the recommendation rate.

RQ 2: *How does ignoring well-known methods affect the recommendation rate?*

A user of an API method recommendation system may not want to receive recommendations for API methods that are already well-known to her. As these methods are employed frequently in the code, we expected the recommendation rate to drop. The study confirmed this expectation and showed that the recommendation rates decrease. However, for the majority of the study objects, the effect was only moderate. The maximum drop was about 11%.

RQ 3: *How does the approach perform for APIs other than the Java API?*

The study also showed that the results are transferable to other APIs by an experiment with the Eclipse API, where recommendation rates in the same order of magnitude compared to the Java API were measured.

RQ 4: *How does the approach compare to a method usage-based approach?*

We showed that the proposed approach outperforms an existing structure-based approach in a direct quantitative comparison.

RQ 5: *Can we use the similarity measure to derive a confidence level for recommendations?*

Yes, experiments indicated that the similarity measure of the approaches can successfully be used as a confidence level for the produced recommendations. For both approaches, this measure could be used to predict the correctness of a given recommendation.

RQ 6: *How are the recommendation rates for cross-project vs. intra-project recommendations?*

We can conclude that intra-project recommendation achieves better accuracy than cross-project recommendation.

Summary We showed that the identifiers in a program are a viable source for mining a knowledge base for recommending API methods by demonstrating the applicability of the approach to real-world software systems. Moreover, we verified that the proposed approach outperformed an existing structure-based approach.

9.4 Reuse Support for Model-Based Development

The model recommendation system introduced in this thesis produces recommendations for Simulink library blocks with satisfying precision and recall. Both employed variants (collaborative filtering and association rule mining) performed significantly better than a trivial baseline recommendation system. Parameters allow controlling the trade-off between precision and recall of the produced recommendations. The study showed that the variant based on collaborative filtering overall achieved the best results.

Summary We conclude that the transfer of recommendation system approaches to model-based development is a promising path to follow.

10 Future Work

This chapter gives directions for future work, derived from the insights gained throughout the studies presented in this thesis.

10.1 Analysis of Third-Party Code Reuse

Different Programming Ecosystems The study on code reuse presented in this thesis focussed on Java Open Source development. An interesting direction for future work is to extend the research on reuse to projects using different programming ecosystems. To this end, software systems developed in "legacy languages" like COBOL or PL/1 on the one hand and more recent languages like Python or Scala on the other hand could be analyzed. The hypothesis is that reuse rates of third-party code differ significantly for different programming ecosystems. This information could be an important aspect in the decision of an organization for the technology to be used for new development projects, as the availability of reusable third-party code has an influence on the expected productivity of the development.

Different Development Models A second path for continuative research regarding third-party code reuse in practice is to investigate how the results transfer to other development models. An open question here is, to what extent the open source development model facilitates reuse compared to commercial software development. As an example, legal constraints may inhibit reuse in commercial projects which is not the case for most open source projects. Quantitative empirical studies could give more insight in the degree of the influence of such factors.

Different Application Domains Another open question is whether the achievable reuse rates depend on the application domain of the developed software. Availability of reusable third-party code may be better established for more "common" domains as the organization creating the library can expect more users. Thus, the hypothesis is that higher reuse rates can be found in these areas.

Assessing the Adequacy of Code Reuse The study in this thesis focused on quantifying the extent of reuse in software projects and found that many of today's software systems build on a significant number of external libraries, *i.e.*, these systems consist to a considerable fraction of code from third parties. Consequently, external libraries have a significant impact on maintenance activities in the project. Unfortunately, reuse of third-party libraries can also impose risks, which may—if remaining unidentified—threaten the ability to effectively evolve the system in the future. It is an interesting path for future work to assess for a software project how adequate it employs external libraries.

10.2 Detection of Functionally Similar Code Fragments

Advanced Input Generation As we identified the input generation as one of the core weaknesses of the approach, the use of advanced test generation techniques is a promising path for future work. In general, white-box methods, where knowledge about the internal structure and logic of the code is exploited to create effective tests [82], is an interesting direction. Moreover, also recent approaches like feedback-directed random testing [81], which utilize the results of previous test executions to improve the generated inputs, may yield better inputs and thus improve the code coverage achieved during the execution. A major challenge here, compared to the basic random testing method, lies in the performance requirements of these more advanced approaches. However, parallelization of the detection pipeline could help to address this issue.

Notion of I/O Similarity There is reason to believe that similarities are missed by the detection approach due to the notion of I/O similarity. For instance, two code fragments may perform a semantically similar computation but use different data structures at their interfaces. Further research is required to quantify these issues as well as finding more suitable notions for functional similarity.

10.3 API Recommendation

Recommendation of Additional Libraries The API recommendation system presented in this thesis aims at enabling developers to use APIs more effectively. However, it remains open how the developer decides to reuse a certain library in the first place. An interesting direction for future work in this area are recommendation systems that recommend complete libraries for reuse in a project. These systems could use properties of a software project, such as the concepts used in artifacts (*e.g.*, specifications, code) to produce recommendations. The hypothesis is that projects referencing similar concepts in their artifacts may exhibit similar library usage. Thus, a machine learning approach could be used to extract this knowledge from existing projects in repositories and then recommend libraries for projects under development.

Personalization Since learning is an individual process, personalization of a recommendation system is an interesting direction for improvement regarding user acceptance. Therefore, the knowledge of an individual user of a recommendation system has to be taken into account. For instance, an API recommendation system could learn which API entities are used frequently by an individual developer and thus excluding them from the recommendations. The hypothesis is that this significantly improves the perceived usefulness of such a system.

Hybrid Approaches A promising path for further work is the development of a hybrid approach for API recommendation. Hybrid recommendation approaches have the goal to cope with the limitations of individual recommendation algorithms by combining their individual strengths [47]. A hybrid API recommendation system might achieve better overall recommendation results compared to a single individual approach.

10.4 Model Recommendation

Notion of Similarity The model recommendation system presented in this thesis used a highly abstract notion of similarity between partial models to derive recommendations. A possible direction for future research is to experiment with different notions of similarity for collaborative filtering. One option would be to take additional aspects of the model into account, such as the interconnections between the model elements.

Advanced Context Currently, the recommendations depend on *all* blocks employed in an unfinished model. Consequently, the recommendations are independent of the position where a new model element is to be inserted during editing. An alternative would be a more narrow context as given by a certain amount of predecessors of a given model element. It is an interesting open question if this could lead to a higher precision.

Different Modeling Languages It remains an open question how the proposed approach transfers to other modeling languages. To this end, it would be interesting to analyze if the recommendation system could be used for proprietary domain specific modeling languages in commercial software development environments.

Bibliography

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 1994.

[2] A. Alnusair, T. Zhao, and E. Bodden. Effective API Navigation and Reuse. In *Proceedings of the International Conference on Information Reuse and Integration (IRI)*, 2010.

[3] C. Anderson. *The long tail: Why the future of business is selling less of more*. Hyperion Books, 2008.

[4] J. Anvik, L. Hiew, and G. Murphy. Who should fix this bug? In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2006.

[5] S. Bajracharya, J. Ossher, and C. Lopes. Leveraging usage similarity for effective retrieval of examples in code repositories. In *Proceedings of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2010.

[6] V. Basili, L. Briand, and W. Melo. How reuse influences productivity in object-oriented systems. *Communications of the ACM*, 39(10):116, 1996.

[7] D. Batory. Feature models, grammars, and propositional formulas. *Software Product Lines*, pages 7–20, 2005.

[8] M. Bertran, F. Babot, and A. Climent. An input/output semantics for distributed program equivalence reasoning. *Electronic Notes in Theoretical Computer Science*, 137(1):25–46, 2005.

[9] B. Bislimovska, A. Bozzon, M. Brambilla, and P. Fraternali. Content-based search of model repositories with graph matching techniques. In *Proceedings of the International Workshop on Search-driven development: Users, Infrastructure, Tools and Evaluation (SUITE)*, 2011.

[10] B. Boehm. *Software engineering economics*. Prentice-Hall, 1981.

[11] M. Bramer. *Principles of data mining*. Springer, 2007.

[12] M. Bruch, M. Monperrus, and M. Mezini. Learning from examples to improve code completion systems. In *Proceedings of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2009.

[13] S. Chakrabarti. *Mining the Web: discovering knowledge from hypertext data*. Morgan Kaufmann, 2003.

[14] S. Chatterjee, S. Juvekar, and K. Sen. SNIFF: A Search Engine for Java Using Free-Form Queries. In *Proceedings of the International Conference on Formal Aspects in Software Engineering (FASE)*, 2009.

[15] J. Cleaveland. Building application generators. *IEEE Software*, 5(4):25–33, 1988.

[16] J. Dabney and T. Harman. *Mastering Simulink*. Pearson/Prentice Hall, 2004.

[17] F. Deissenboeck, L. Heinemann, B. Hummel, and S. Wagner. Challenges of the Dynamic Detection of Functionally Similar Code Fragments. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, 2012.

[18] F. Deissenboeck, B. Hummel, E. Jurgens, B. Schätz, S. Wagner, J. Girard, and S. Teuchert. Clone detection in automotive model-based development. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2008.

[19] R. Dijkman, M. Dumas, and L. García-Bañuelos. Graph matching algorithms for business process model similarity search. *Business Process Management*, pages 48–63, 2009.

[20] E. Duala-Ekoko and M. Robillard. Using structure-based recommendations to facilitate discoverability in APIs. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 2011.

[21] E. Duala-Ekoko and M. Robillard. Asking and Answering Questions about Unfamiliar APIs: An Exploratory Study. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2012.

[22] D. Eisenberg, J. Stylos, and B. Myers. Apatite: A new interface for exploring apis. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI)*, 2010.

[23] M. Fayad and D. Schmidt. Object-oriented application frameworks. *Communications of the ACM*, 40(10):32–38, 1997.

[24] B. Fischer and G. Snelting. Reuse by contract. In *Proceedings of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 1997.

[25] G. Fischer. Cognitive view of reuse and redesign. *IEEE Software*, pages 60–72, 1987.

[26] W. Frakes and K. Kang. Software reuse research: Status and future. *IEEE Transactions on Software Engineering*, 31(7):529–536, 2005.

[27] G. Furnas, T. Landauer, L. Gomez, and S. Dumais. The vocabulary problem in human-system communication. *Communications of the ACM*, 30(11):964–971, 1987.

[28] E. Gamma. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[29] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 1993.

[30] S. Haefliger, G. Von Krogh, and S. Spaeth. Code Reuse in Open Source Software. *Management Science*, 54(1):180–193, 2008.

[31] H. Happel and W. Maalej. Potentials and challenges of recommendation systems for software development. In *Proceedings of the International Workshop on Recommendation Systems for Software Engineering (RSSE)*, 2008.

[32] A. Hassan. The road ahead for mining software repositories. In *Proceedings of the Frontiers of Software Maintenance (FoSM)*, 2008.

[33] L. Heinemann. Facilitating Reuse in Model-Based Development with Context-Dependent Model Element Recommendations. In *Proceedings of the International Workshop on Recommendation Systems for Software Engineering (RSSE)*, 2012.

[34] L. Heinemann, V. Bauer, M. Herrmannsdoerfer, and B. Hummel. Identifier-Based Context-Dependent API Method Recommendation. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, 2012.

[35] L. Heinemann, F. Deissenboeck, M. Gleirscher, B. Hummel, and M. Irlbeck. On the Extent and Nature of Software Reuse in Open Source Java Projects. In *Proceedings of the International Conference on Software Reuse (ICSR)*, 2011.

[36] L. Heinemann and B. Hummel. Recommending API Methods Based on Identifier Contexts. In *Proceedings of the International Workshop on Search-driven development: Users, Infrastructure, Tools and Evaluation (SUITE)*, 2011.

[37] J. Herlocker, J. Konstan, L. Terveen, and J. Riedl. Evaluating collaborative filtering recommender systems. *ACM Transactions on Information Systems (TOIS)*, 22(1):5–53, 2004.

[38] E. Hill, L. Pollock, and K. Vijay-Shanker. Automatically capturing source code context of nl-queries for software maintenance and reuse. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2009.

[39] R. Hill and J. Rideout. Automatic Method Completion. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, 2004.

[40] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2005.

[41] R. Holmes and R. Walker. Informing Eclipse API production and consumption. In *Proceedings of the International Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2007.

[42] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt. Index-Based Code Clone Detection: Incremental, Distributed, Scalable. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, 2010.

[43] O. Hummel and C. Atkinson. Using the Web as a Reuse Repository. *Reuse of Off-the-Shelf Components*, pages 298–311, 2006.

[44] O. Hummel, W. Janjic, and C. Atkinson. Code Conjurer: Pulling reusable software out of thin air. *IEEE Software*, 25(5):45–52, 2008.

[45] I. Ianov. On the equivalence and transformation of program schemes. *Communications of the ACM*, 1(10):12, 1958.

[46] I. Jacobson, M. Griss, and P. Jonsson. *Software reuse: architecture, process and organization for business success*. Addison-Wesley, 1997.

[47] D. Jannach, M. Zanker, A. Felfernig, and G. Friedrich. *Recommender Systems: An Introduction*, volume 1. Cambridge University Press, 2010.

[48] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 96–105. IEEE Computer Society, 2007.

[49] L. Jiang and Z. Su. Automatic mining of functionally equivalent code fragments via random testing. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2009.

[50] E. Juergens, F. Deissenboeck, and B. Hummel. Clone detection beyond copy & paste. In *Proceedings of the International Workshop on Software Clones (IWSC)*, 2009.

[51] E. Juergens, F. Deissenboeck, and B. Hummel. CloneDetective – a workbench for clone detection research. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2009.

[52] E. Juergens, F. Deissenboeck, and B. Hummel. Code similarities beyond copy & paste. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, 2010.

[53] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2009.

[54] H. Kagdi, M. Collard, and J. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(2):77–131, 2007.

[55] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[56] E. Karlsson. *Software reuse: a holistic approach*. John Wiley & Sons, Inc., 1995.

[57] D. Kawrykow and M. Robillard. Improving API Usage through Automatic Detection of Redundant Code. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, 2009.

[58] H. Kim, Y. Jung, S. Kim, and K. Yi. Clone detection by comparing abstract memory states. Technical Memorandum ROSAEC-2010-008, Research On Software Analysis for Error-free Computing Center, Seoul National University, March 2010.

[59] R. Koschke. Survey of research on software clones. In *Duplication, Redundancy, and Similarity in Software*. Dagstuhl Seminar Proceedings, 2007.

[60] R. Koschke, R. Falke, and P. Frenzel. Clone detection using abstract syntax suffix trees. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, 2006.

[61] C. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–183, 1992.

[62] A. Kuhn, S. Ducasse, and T. Gírba. Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 49(3):230–243, 2007.

[63] R. Lämmel, E. Pek, and J. Starek. Large-scale, AST-based API-usage analysis of open-source Java projects. In *Proceedings of the ACM Symposium on Applied Computing*, 2011.

[64] R. Leach. *Software Reuse: Methods, Models and Costs*. McGraw-Hill, Inc., 1996.

[65] N. Lee and C. Litecky. An empirical study of software reuse with special attention to Ada. *IEEE Transactions on Software Engineering*, 23(9):537–549, 1997.

[66] W. Lim. Effects of reuse on quality, productivity, and economics. *IEEE Software*, 11(5):23–30, 2002.

[67] T. Lindholm and F. Yellin. *Java virtual machine specification*. Addison-Wesley, 1999.

[68] H. Liu, Z. Ma, L. Zhang, and W. Shao. Detecting duplications in sequence diagrams based on suffix trees. In *Proceedings of the Asia-Pacific Software Engineering Conference (APSEC)*, 2006.

[69] H. Ma, R. Amor, and E. Tempero. Usage Patterns of the Java Standard API. In *Proceedings of the Asia-Pacific Software Engineering Conference (APSEC)*, 2006.

[70] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the API jungle. *ACM SIGPLAN Notices*, 40(6):48–61, 2005.

[71] F. Mccarey, M. Cinnéide, and N. Kushmerick. Rascal: A recommender agent for agile reuse. *Artificial Intelligence Review*, 24:253–276, 2005.

[72] M. McIlroy, J. Buxton, P. Naur, and B. Randell. Mass produced software components. *Software Engineering Concepts and Techniques*, pages 88–98, 1969.

[73] C. McMillan, M. Grechanik, and D. Poshyvanyk. Detecting Similar Software Applications. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2012.

[74] R. Metzger and Z. Wen. *Automatic algorithm recognition and replacement: A new approach to program optimization*. The MIT Press, 2000.

[75] Y. Mileva, V. Dallmeier, and A. Zeller. Mining API Popularity. In *Testing - Practice and Research Techniques*, volume 6303 of *Lecture Notes in Computer Science*, pages 173–180. Springer, 2010.

[76] A. Mili, R. Mili, and R. Mittermeir. A survey of software reuse libraries. *Annals of Software Engineering*, 5(1):349–414, 1998.

[77] H. Mili, A. Mili, S. Yacoub, and E. Addy. *Reuse-Based Software Engineering: Techniques, Organizations, and Controls*. Wiley-Interscience, 2001.

[78] R. Milner. An algebraic definition of simulation between programs. In *Proceedings of the 2nd International Joint Conference on Artificial Intelligence*, pages 481–489. Morgan Kaufmann Publishers Inc., 1971.

[79] R. Mittermeir and H. Pozewaunig. Classifying Components by Behavioral Abstraction. In *Proceedings of the Joint Conference on Information Sciences (JCIS)*, 1998.

[80] A. Mockus. Large-scale code reuse in open source software. In *Proceedings of the International Workshop on Emerging Trends in FLOSS Research and Development*, 2007.

[81] C. Pacheco, S. Lahiri, M. Ernst, and T. Ball. Feedback-directed random test generation. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2007.

[82] W. Perry. *Effective methods for software testing*. John Wiley & Sons, Inc., 2006.

[83] A. Pitts. Operational semantics and program equivalence. *Lecture Notes in Computer Science*, 2395:378–412, 2002.

[84] A. Podgurski and L. Pierce. Retrieving reusable software by sampling behavior. *ACM Transactions on Software Engineering and Methodology*, 2(3):286–303, 1993.

[85] K. Pohl, G. Böckle, and F. Van Der Linden. *Software product line engineering: foundations, principles, and techniques*. Springer, 2005.

[86] J. Raemaekers, A. van Deursen, and J. Visser. An analysis of dependence on third-party libraries in open source and proprietary systems. In *Proceedings of the Sixth International Workshop on Software Quality and Maintainability (SQM)*, 2012.

[87] J. Raoult and J. Vuillemin. Operational and semantic equivalence between recursive programs. *Journal of the ACM (JACM)*, 27(4):796, 1980.

[88] T. Ravichandran and M. Rothenberger. Software reuse strategies and component markets. *Communications of the ACM*, 46(8):109–114, 2003.

[89] D. Reifer. *Practical software reuse*. John Wiley & Sons, Inc., 1997.

[90] S. Reiss. Semantics-based code search. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2009.

[91] R. Robbes and M. Lanza. Improving code completion with program history. *Automated Software Engineering*, 17(2):181–212, 2010.

[92] M. Robillard. What Makes APIs Hard to Learn? Answers from Developers. *IEEE Software*, pages 27–34, 2009.

[93] M. Robillard, R. Walker, and T. Zimmermann. Recommendation systems for software engineering. *IEEE Software*, 27(4):80–86, 2010.

[94] C. Roy, J. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009.

[95] C. K. Roy and J. R. Cordy. A survey on software clone detection research. Technical Report 2007-541, Queen's University, Kingston, Canada, 2007.

[96] J. Rutledge. On Ianov's program schemata. *Journal of the ACM (JACM)*, 11(1):9, 1964.

[97] T. Sager, A. Bernstein, M. Pinzger, and C. Kiefer. Detecting similar Java classes using tree algorithms. In *Proceedings of the International Workshop on Mining Software Repositories*, 2006.

[98] C. Sammut and G. Webb. *Encyclopedia of machine learning*. Springer, 2011.

[99] M. Sojer and J. Henkel. Code Reuse in Open Source Software Development: Quantitative Evidence, Drivers, and Impediments. *Journal of the Association for Information Systems*, 11(12):868–901, 2010.

[100] C. Spearman. The proof and measurement of association between two things. *The American journal of psychology*, 15(1):72–101, 1904.

[101] D. Spinellis and C. Szyperski. How is open source affecting software development? *IEEE Software*, 21(1):28–33, 2004.

[102] X. Su and T. Khoshgoftaar. A survey of collaborative filtering techniques. *Advances in Artificial Intelligence*, 2009:4, 2009.

[103] Sun Microsystems. Code Conventions for the Java Programming Language, Apr. 2011. http://www.oracle.com/technetwork/java/codeconv-138413.html.

[104] W. Takuya and H. Masuhara. A spontaneous code recommendation tool based on associative search. In *Proceedings of the International Workshop on Search-driven development: Users, Infrastructure, Tools and Evaluation (SUITE)*, 2011.

[105] P. Tan, M. Steinbach, V. Kumar, et al. *Introduction to data mining*. Addison Wesley, 2006.

[106] R. Terra, M. Valente, K. Czarnecki, and R. Bigonha. Recommending refactorings to reverse software architecture erosion. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, 2012.

[107] Text Fixer. List of common words, Apr. 2011. http://www.textfixer.com/resources/common-english-words.txt.

[108] S. Thummalapenta and T. Xie. Parseweb: a programmer assistant for reusing open source code on the web. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, 2007.

[109] M. Tsunoda, T. Kakimoto, N. Ohsugi, A. Monden, and K. Matsumoto. Javawock: A Java Class Recommender System Based on Collaborative Filtering. In *Proceedings of the International Conference on Software Engineering and Knowledge Engineering (SEKE)*, 2005.

[110] A. J. Viera and J. M. Garrett. Understanding interobserver agreement: the kappa statistic. *Family Medicine*, 37(5):360–363, 2005.

[111] G. von Krogh, S. Spaeth, and S. Haefliger. Knowledge Reuse in Open Source Software: An Exploratory Study of 15 Open Source Projects. In *Proceedings of the Hawaii International Conference on System Sciences (HICSS)*, 2005.

[112] C. Wohlin, P. Runeson, and M. Höst. *Experimentation in software engineering: An introduction*. Kluwer Academic, 2000.

[113] Y. Ye and G. Fischer. Information delivery in support of learning reusable software components on demand. In *Proceedings of the International Conference on Intelligent User Interfaces (IUI)*, 2002.

[114] V. Zakharov. To the functional equivalence of turing machines. In *Proceedings of the International Conference on Fundamentals of Computation Theory*, page 491. Springer-Verlag, 1987.

[115] V. Zakharov. The equivalence problem for computational models: decidable and undecidable cases. *Machines, Computations, and Universality*, pages 133–152, 2001.

[116] A. Zaremski and J. Wing. Signature matching: a tool for using software libraries. *ACM Transactions on Software Engineering and Methodology*, 4(2):146–170, 1995.

[117] C. Zhang, J. Yang, Y. Zhang, J. Fan, X. Zhang, J. Zhao, and P. Ou. Automatic Parameter Recommendation for Practical API Usage. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2012.

[118] W. Zheng, Q. Zhang, and M. Lyu. Cross-Library API Recommendation using Web Search Engines. In *Proceedings of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2011.

[119] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. MAPO: Mining and recommending API usage patterns. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 2009.

[120] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2004.