

Reuse-Based Test Recommendation in Software Engineering

Inauguraldissertation
zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften
der Universität Mannheim

vorgelegt von

Diplom-Wirtschaftsinformatiker Werner Janjic
aus Koblenz

Mannheim, 2014

Dekan: Prof. Dr. Heinz Jürgen Müller, Universität Mannheim
Referent: Prof. Dr. Colin Atkinson, Universität Mannheim
Korreferent: Prof. Dr. Reid Holmes, University of Waterloo, Canada

Tag der mündlichen Prüfung: 24.06.2014

for Silke...

Abstract

Still today, the development of effective and high-quality software tests is an expensive and very labor intensive process. It demands a high amount of problem awareness, domain knowledge and concentration from human software testers. Therefore, any technology that can help reduce the manual effort involved in the software testing process – while ensuring at least the same level of quality – has the potential to significantly reduce software development and maintenance costs. In this dissertation, we present a new idea for achieving this by reusing the knowledge bound up in existing tests. Over the last two decades, software reuse and code recommendation has received a wide variety of attention in academia and industry, but the research conducted in this area to date has focused on the reuse of application code rather than on the reuse of tests. By switching this focus, this thesis paves the way for the automated extraction of test data and knowledge from previous software projects. In particular, it presents a recommendation approach for software tests that leverages lessons learned from traditional software reuse to make test case reuse suggestions to software engineers while they are working. In contrast to most existing testing-assistance tools, which provide ex post assistance to test developers in the form of coverage assessments and test quality evaluations, our approach offers an automated, proactive, non-intrusive test recommendation system for efficient software test development.

Zusammenfassung

Auch heutzutage ist die Entwicklung qualitativ hochwertiger Software-Tests ein nicht zu vernachlässigender Kostentreiber in Software-Projekten. Die Mitglieder eines erfolgreichen Teams zur Software-Qualitätssicherung benötigen ein hohes Maß an Kenntnis über die projektspezifische Domäne, Konzentration und Problembewusstsein. Gleichzeitig steht das Testen von Software in einem Spannungsverhältnis zur Entwicklung neuer, für den Benutzer sichtbarer Funktionalität, und ist dadurch bei Entscheidungen im Entwicklungsprozess potentiell benachteiligt. Um diesem Missstand entgegenzuwirken haben Software-Ingenieure seit den frühen Zeiten des Software-Testens nach Möglichkeiten gesucht, den Prozess so weit wie möglich zu automatisieren und den manuellen Aufwand zu reduzieren. Dabei muss sichergestellt sein, dass die resultierende Test- und Software-Qualität vergleichbar oder besser ist, als beim manuellen Testen. Die vorliegende Dissertation beschäftigt sich mit der tool-gestützten Wiederverwendung des in bestehenden Tests enthaltenen Expertenwissens in zukünftigen Tests und in neuem Kontext. Dafür bedienen wir uns zuerst der allgemein bekannten Techniken für code-basierte Wiederverwendung und entwickeln darauf basierend eine sprachunabhängige Suchmaschine für Software-Tests. Weiterhin demonstrieren wir die Anwendbarkeit des entwickelten Ansatzes anhand einer prototypischen Implementierung als Eclipse Plug-In, welches auf Grundlage dieser Suchmaschine seinem Benutzer unaufdringlich und vorausschauend Vorschläge zum Software-Testen unterbreitet. Die potentiell wieder verwendbaren Tests werden im Hintergrund evaluiert und stehen dem Benutzer auf Tastendruck zur Verfügung. Um den Nutzen der vorgestellten Ideen zu erhöhen, stellt diese Arbeit auch einen Ansatz vor, mit dessen Hilfe potentiell falsche Ergebnisse automatisch erkannt und aus der Ergebnisliste gestrichen werden. Neben den grundlegenden Betrachtungen sorgen praktische Beispiele für ein besseres Verständnis der vorgestellten Ideen.

Contents

Abstract	vii
Zusammenfassung	ix
I Introduction	1
1 Introduction	3
1.1 Motivation	4
1.2 Research Objective	5
1.3 Contribution Of The Thesis	8
1.4 Scope of the Thesis	9
1.5 Structure of the Thesis	10
2 Software Testing	15
2.1 Software Testing Terms	15
2.2 Extracting knowledge from JUnit	23
2.3 Summary	28
II Search and Reuse	31
3 Software Search and Recommendation	33
3.1 Search Scenarios in Software Engineering	34
3.1.1 Speculative Searches	36
3.1.2 Definitive Searches	39
3.2 Software Search Engines	41
3.2.1 Agora	41
3.2.2 Merobase Component Finder	43

3.2.3	Sourcerer	45
3.2.4	S^6	47
3.3	Excursus: Recall and Precision	48
3.4	Test-Driven Reuse	49
3.5	Summary	53
4	Automated Interface Adaptation	55
4.1	Distributed Automated Adaptation System	56
4.2	Interface Adaptation	57
4.3	Improvements to Test-Driven Search	63
4.4	Summary	64
5	Reuse-Oriented Code Recommendation Systems	67
5.1	Recommendation Systems for Code Reuse	68
5.2	Software Reuse Process	69
5.3	State of the Art Systems	73
5.3.1	Code Finder	73
5.3.2	CodeBroker	75
5.3.3	Strathcona	78
5.3.4	Code Genie	81
5.3.5	PARSEWeb	84
5.3.6	Code Conjurer	87
5.4	Usage Scenarios	92
5.4.1	Component Reuse	92
5.4.2	Library Reuse	93
5.5	Characteristics of ROCRs	93
5.6	Summary	96
III	Reuse of Software Tests	99
6	Infrastructure for Test Reuse	101
6.1	Obtaining Reusable Test Cases	101
6.1.1	Potential of Open Source Repositories	103
6.2	Extracting Knowledge from Test Cases	105

6.2.1	A Meta-Model for Software Tests	106
6.3	Index Creation	115
6.3.1	Index Content	117
6.3.2	A File Parser for JUnit Tests	119
6.4	Summary	131
7	Reuse-Assisted Software Testing	133
7.1	Usage Scenarios for Test Search Engines	134
7.1.1	Analysis & Design	135
7.1.2	Implementation	135
7.1.3	Testing	137
7.2	Result Retrieval Techniques for Test Reuse	137
7.2.1	Interface-Based Searches	139
7.2.2	Value-Based Searches	149
7.2.3	Code-Based Searches	156
7.3	Retrieval of Exception Tests	158
7.4	Test Reuse Process	159
7.5	Implementation	161
7.6	Summary	162
8	Reuse-Assisted Test Recommendation	165
8.1	Characteristics	166
8.2	Process Outline	168
8.3	Implementation	170
8.3.1	Eclipse Plug-In	171
8.3.2	Continuous Speculative Testing	174
8.3.3	Exception Tests	176
8.3.4	Algorithmic Outline	177
8.4	Summary	178
9	Search-Enhanced Recommendation Improvement	181
9.1	Using Oracles in Software Testing	182
9.1.1	Excursus: The Knight and Leveson Experiment	184
9.2	Search-Enhanced Testing	187
9.3	Filtering False-Positives	199

9.3.1 Oracle-Based Filtering	201
9.3.2 Test Case Evaluation	203
9.4 Summary	206
IV Epilogue	211
10 Epilogue	213
10.1 Retrospective	213
10.2 Contributions	214
10.3 Future Work	216
10.4 Concluding Vision	218
Acknowledgements	221
Appendices	225
List of Figures	227
List of Tables	229
A Materials	231
A.1 Regular Expressions	233
A.1.1 Examples of Queries with Regular Expressions	234
A.2 Comparison of Retrieval Precision	235
B Listings	237
C Bibliography	247
D Index	263

Part I.

Introduction

“ Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.”

BRIAN KERNIGHAN
Computer Scientist

1

Introduction

Since the advent of the so-called *digital revolution* [Soc14], i.e., the transition from analog to digital technology, our lives have become more and more dependent on devices that are operated by software. While car drivers in the early 1970s were barely aware of any digital equipment in their vehicles, nowadays they are surrounded by a myriad of digital assistance tools to the extent that autonomously driving cars have become reality. In fact, every aspect of life has become more and more digitalized. Whereas in the 1980s only a small proportion of children had a games console let alone a computer, today they not only have their own personal computer at home, but also may possess pocket-size computers in the form of mobile phones or tablet PCs. Another big change occurred when the software industry started to move more and more functionality that was previously implemented in hardware to the driver software, as exemplified by so-called *softmodems*. Today, software is more important than ever and modern life is unimaginable without functioning, trustworthy software applications.

1.1. Motivation

The more dependent we become on software, the more shaken we are when it fails and does not behave as expected. Only recently, Apple's ominous *goto fail*¹ disaster and the even more severe *heartbleed*² bug demonstrated how vulnerable we have become and that even very small defects in a program can have significant impact. Hence, the creation of reliable and high quality software systems is a key goal and major challenge for software engineers. Although software testing has gained a lot of attention in software process theory over the last five decades [Roy70; Boe88; Jac+99] it is still a rather labor-intensive and tedious process. Developers are therefore often reluctant to sacrifice development time to test their system. Moreover, in addition to its time demands, software testing also requires a high degree of domain and expert knowledge, concentration and problem awareness from the testing staff.

Since the early days of software engineering, researchers have been struggling to find effective and efficient strategies to automate bug detection and reduce the manual effort involved in testing software [OB88; Vou90; EKR03; GO06; LT12]. In their introductory book on *software testing*, for instance, Ammann and Offutt address the need for more research in *test automation* and stress that one of the outstanding problems in this area is the automated generation of test data. However, they mainly consider the idea of using *genetic algorithms*, while arguing that these are more applicable at the system level than at the unit level [AO08, pp. 288].

Although numerous other techniques for acquiring test data have been proposed, such as *dynamic symbolic execution* [KS98], *concolic testing* [SA06] or *directed automated random testing* [GKS05], research in the area of automated test data generation has largely focused on algorithms that analyze program structure and utilize coverage criteria, program path inspection, etc.

With the emergence of dedicated software search engines in the 2000s, driven by the newly available open source software repositories [HA06], this situation

¹ <http://gotofail.com>

² <http://heartbleed.com/>

changed. These search engines supported a new generation of recommendation tools, especially reuse-oriented code recommendation systems, aimed at accelerating the software development process by obviating the continual need to “reinvent the wheel”. Prominent tools that emerged at that time are, for instance, *Strathcona* [HM05], *Code Conjurer* [HJA08] or the *Eclipse Code Recommenders* project [BMM09]. They share the common goal of improving the productivity of developers by automatically recommending generated or reusable code artifacts of different sizes (i.e., from method call statements to full sized Java classes) in software development projects. Essentially they reuse the historical knowledge wrapped up in existing repositories to accelerate the development of new applications. A typical example is to suggest previously written code for reuse or to indicate how classes from a class library should be used based on the way they were used in existing code.

Although intensive research has been conducted independently in the fields of automated software testing and software reuse, to date there has been no recognized attempt to bring both areas together and exploit their potential synergies. More specifically, no existing recommendation tool has attempted to use the information locked up in the vast number of tests stored in the many public and private corporate repositories to support the writing of new tests. Therefore, the goal of this thesis is to address that challenge and to present novel ideas that should help to leverage the reuse of previously created tests in future software projects.

1.2. Research Objective

The previously conducted research in the area of automation in software testing has created an impressive array of sophisticated algorithms and techniques to automatically generate test case values and to obtain appropriate expected results. These expected results can be compared to the outputs actually produced by programs under test when invoked using the same test case values. In the literature, a mechanism that inspects the output of the system under test and determines its correctness is usually called *test oracle* [Wey82]. However, the

biggest problem is obviously how to determine the *correct output* of a program. While the literature in earlier years expected the human tester to act as the test oracle, the automation of software testing shifts this burden to computer programs, which have to determine the expected behavior of the particular *system under test*.

One idea that represents a kind of intermediate approach between zero and full automation is the so-called approach of *back-to-back testing* [Vou90]. This technique utilizes a set of (manually created) functionally equivalent programs as *oracles* to help determine the correct output produced by a program for a set of test input data. Using this technique it is possible to automatically create lists of potentially interesting tests which need further inspection by human testers, but avoid the manual creation of tests for uncritical inputs. The back-to-back testing technique, however, requires the manual creation of several different implementations of the same functionality and therefore is itself labor intensive. Additionally, there is also the question of testing the set of independent implementations.

In this dissertation, we present a new approach of *reuse-oriented software testing* which aims to support (automated) software testing through the application of well-known software reuse techniques. By combining these two research areas, we aim to automatically acquire test data and expected outputs for components under test. In other words, we aim to extract the knowledge bound up in previously created software tests and make this information efficiently searchable in order to accelerate the creation of new tests. Although a dedicated search engine for software tests presents this information to the user, it actually does not create it. Usually the original source of the test data and expected results is still a human (i.e., a *human oracle*).

Despite a lot of work has been conducted on tools that support quality assurance in software testing, the current generation of IDE tools for testing generally focuses on increasing the quality of tests by analyzing them *ex post* using various criteria. Tools like Cobertura³ and JaCoCo⁴, for instance, help developers to

³ <http://cobertura.sourceforge.net>

⁴ <http://www.eclemma.org/jacoco>

identify parts of the system under development that are not yet reached and inspected by existing tests.

However, they still leave most of the labor intensive and time consuming work of fixing the identified weaknesses to the engineer. It is therefore necessary to run these tools over and over again and wait for their reports. On the other hand, approaches that try to generate tests based on formal specifications do not seem practical for mainstream development projects [San96] because developers usually do not want to learn another language or coding standard. More recently, there has been interesting work conducted on the tool-based, mutation-driven generation of test data [FZ12], but the problem of *equivalent mutants* still creates a significant amount of manual effort when using this technique.

With the novel *ex ante* approach for tool-supported software testing, which is presented in this dissertation, we provide the foundation for a new generation of tools that try to predict which tests developers are likely to write next, based on the information acquired from previously written tests. Leveraging the ideas of code search engines, we define the characteristics of reuse-assisted software testing tools and provide a prototypical implementation that fulfills the vision of “*a powerful integrated test environment which by itself, as a piece of software is [...] generating the most suitable test cases, executing them and finally issuing a test report*” [Ber07]. Using novel techniques like *speculative analysis* [Bru+10], we will show that it is possible to recommend reusable tests that have been evaluated even before they are considered for reuse, enabling test reuse tools to rank recommendations before they are presented to the user.

The aims of the work conducted in this dissertation are as follows:

1. to advance the current state-of-the-art in tool supported software testing, by developing a reuse-oriented approach for the creation of software tests,
2. to investigate the feasibility of using previously created software tests as test oracles and develop a model that allows the creation of language-independent, searchable repositories of software tests,

3. to define the requirements for reuse-assisted software testing in modern IDEs and create a tool that recommends contextually matching software tests during a software project,
4. to investigate the feasibility of performing an *ex ante* analysis of the application of reusable software tests in a newly developed application,
5. to employ the idea of back-to-back testing to reduce the number of false-positive results and therefore help preventing the recommendation of test data that is not suitable for the system under development.

1.3. Contribution Of The Thesis

The contributions of this thesis arising from our work are:

1. a definition of scenarios for test reuse in the software development lifecycle,
2. a meta-model, which allows the creation of models that capture software tests written in different frameworks,
3. a parser for JUnit test cases. Although developers are probably most familiar with the JUnit testing framework, we will identify weaknesses in its structure and the reusability of JUnit test cases which also affect the maintainability of large sets of test cases,
4. the SENTRE system – an internet-scale search engine and back-end for reusable software tests,
5. a set of value-based retrieval techniques for software tests that supplement name-dependent searches by using existing test data to identify reusable assets,
6. we provide a concise overview of the requirements and essential characteristics of reuse-oriented recommendation systems in software engineering, especially for reuse-oriented test recommendation,

7. a test reuse and recommendation plug-in for the Eclipse IDE and JUnit, which provides test recommendations on demand with minimum effort to reuse them. By autonomously inspecting reusable candidates and their impact on the test currently being written, the environment filters and ranks potentially reusable test cases and automatically integrates them into the user's development context,
8. a process model for tool-supported reuse of code and software tests,
9. a testing technique, known as Search-Enhanced Testing, and a prototype implementation based on the ideas of n-version programming and back-to-back testing, which utilizes reusable components as oracles,
10. an approach of automatically filtering false-positive recommendations from a list of potentially reusable tests.

1.4. Scope of the Thesis

The goal of this dissertation is to advance the state-of-the-art in software testing by leveraging the results of the software search and reuse community. Therefore, we will mainly focus on the utilization of tools and techniques from the area of software reuse in the context of software testing. This includes the development of a search engine for software tests and an accompanying test recommendation environment for the Eclipse IDE. As well as the ideas of reusing previously created tests, we will also present an approach that uses reusable software components as oracles to automatically generate test data.

Since we are bringing together two different research areas – namely, the area of software testing and the area of software search and reuse – our work could be regarded as a new application for reuse, as well as a new software testing technique. Nevertheless, given that we have chosen to view the problem from the search and reuse perspective, our work does not go deeper into the area of software testing than necessary. Although automated test generation represents a broad field in which a lot of work is being conducted, all of the currently

proposed approaches are different to the ideas developed in this thesis. Our technology does not try to understand what the developer wants to test and does not assume that it is smart enough to read the developer's mind, but rather relies on the human knowledge already embodied within previously hand-crafted software tests that are available for reuse.

1.5. Structure of the Thesis

This thesis is structured as follows:

Chapter 2 introduces the basic software testing terminology that is needed in the context of this thesis. In order to make the presented material as clear as possible, the terms introduced in this chapter are used consistently throughout the remaining thesis. Furthermore, we provide a short introduction to JUnit, which all readers should be familiar with. Based on an exemplary class under test, we show different possibilities of writing test cases that all test the same functionality although they differ in syntax.

Chapter 3 reviews the state-of-the-art in software search and reuse and surveys the corresponding literature. Beginning with an introduction to archetypal software search scenarios and their role in the software development lifecycle, we identify concrete applications for code search and group them into two main kinds, the so-called *speculative* and *definitive searches*. Subsequently, the chapter provides a historical overview of software search and reuse since the famous McIlroy paper [McI69] and introduces some of the most important milestones in software search over the last few decades. An introduction to the test-driven search for software components and its importance to software reuse concludes this chapter.

Chapter 4 presents our enhanced implementation of an automated adaptation system for component interfaces, which was initially envisaged for use in conjunction with test-driven reuse but is also necessary for the automated evaluation of reusable software tests. First, we present an overview of the problem and briefly explain the issues with incompatible interfaces. Subsequently, we

introduce the idea of parallelized distributed interface adaptation and describe an algorithm for non brute-force automated component adaptation and testing. Finally, we show how our implementation improves previously created systems and how the distribution of work among adaptation clients can help to speed up the process of software adaptation.

Chapter 5 discusses the application of the previously introduced ideas and software search engines in the context of modern software development environments. Since a goal of this thesis is to contribute to software testing from the point of view of the reuse community, it is necessary to survey existing reuse-oriented code recommendation systems. The chapter starts with an overview of the prerequisites for the development of reuse-oriented code recommendation systems and then introduces a micro-process of software reuse, which embeds the subsequently introduced systems. The presentation of the state-of-the-art of existing tools serves as the appropriate framework for the identification of their characteristics. These are then used to distill a general set of requirements for reuse-oriented recommendation tools.

Chapter 6 presents our work on the creation of an infrastructure for test reuse. After introducing the challenges presented by test reuse and an overview of possible sources of reusable assets, we discuss the possibilities of knowledge extraction from previously created software tests. To this end, we develop a meta-model for software tests, which captures all aspects of reusable test code in the context of potential test reuse, and embed our effort in the context of component based software development. The detailed description of the features captured by the model sets the scene for its concrete application in the context of reusable JUnit test cases. Subsequently, the chapter deals with the creation of a searchable index of JUnit test cases acquired from various open source software repositories and we introduce the heuristics which help us identify the classes under test in reusable software tests, as well as the classes that are necessary to extract the concrete test data contained in the file. In addition to describing our strategies for extracting test case values and expected results from previously existing JUnit test cases, the chapter also deals with exception tests, their recognition and extraction.

Chapter 7 presents SENTRE, a prototype search engine for reusable software tests. The chapter starts with a general overview of potential use cases for this kind of search engine and describes the applicability of test reuse in the context of the software development lifecycle. Since the search engine relies on an index that was created with the tools described in Chapter 6, we subsequently introduce a set of retrieval techniques for test reuse. Beside the well-known idea of interface-based searches, which has been implemented in the area of code reuse, the chapter describes a novel technique which uses test case values and expected results to obtain reusable tests. Furthermore, we enhance this technique with the capability to specify patterns instead of concrete values with the help of regular expression. This allows clients to improve recall compared to a search with concrete values, since the retrieval algorithm can consider a whole space of input values and expected results. The third concept discussed in this sequence, is the idea of code-based searches which do not rely on pure structural analysis but actually execute reusable tests in the context of the client's class under test. The chapter concludes with an overview of the implementation and architecture of the search engine.

Chapter 8 proceeds by linking the ideas and approaches presented in the preceding chapters. Based on SENTRE, it presents a reuse-oriented test recommendation system developed for the Eclipse IDE. Building upon the characteristics that we identified in Chapter 5, the chapter starts with an overview of the requirements that our system needs to fulfill. Furthermore, the chapter embeds the tool into a micro-process of tool-supported test reuse, which outlines the main activities involved in integrating reused software tests in a new development context. The description of our implementation includes some concrete examples of the capabilities of the system and describes its features, the underlying architecture and algorithms.

Chapter 9 addresses an issue that affects any kind of recommendation system – the presence of incorrect results in a list of recommendations. The more false-positive results are presented to the user, the less likely a recommendation tool is to be included in the standard workflow of a developer. Before we develop a strategy to avoid the recommendation of “wrong” tests (with respect to the

problem domain of the user), we introduce another approach which helps the automation of software testing. The *Search-Enhanced Testing* approach utilizes reusable software components as oracles and executes them with random test case values. We have created a tool which shows the discrepancies in tests executed on a set of oracles using so-called *execution profiles* and which indicates input values that have the potential to uncover problems in the system under development. Building on these ideas of Search-Enhanced Testing, the chapter introduces oracle-based result filtering, which uses a similar approach to identify reusable test cases that are not part of the domain covered by the client's system under test. These are automatically ruled out by the system in order to improve the precision of searches for reusable software tests. An initial evaluation of the underlying algorithm concludes the chapter.

Chapter 10 concludes the main body of the dissertation. Since this thesis comprises initial and foundational work in the area of test reuse, we present an outlook of promising future work that has the best potential to lead to new advances. Finally, we sum up the work conducted in this thesis with some final comments and remarks.

“ The worst I ever saw was a 500-instruction assembly language routine with an average of 2.2 bugs per instruction after syntax checking [...]. That person didn't belong in programming.”

Software Testing Techniques [Bei90]

BORIS BEIZER, Software Engineer

2

Software Testing

2.1. Software Testing Terms

Although testing is one of the most important activities in software engineering, it is not unusual for engineers talking about software testing to mean different things when using the same words. To reduce the resulting potential for confusion, we will introduce some definitions for the most important terms used in the context of this thesis. The definitions are mostly derived from those given in the books¹ “The Art of Software Testing” by Glenford Myers [Mye79; MS04] and “Introduction to Software Testing” by Paul Ammann and Jeff Offutt [AO08], accompanied by additional remarks to tailor them to the context of this thesis.

¹These books represent the standard literature in the area of Software Testing and cover different periods. Myers' book was initially published in 1979 (recently revised and updated), while object-oriented development was still in its infancy, whereas the book by Ammann and Offutt was published in 2008 and covers more recent trends and approaches in software development.

As we will see, however, not all of the terms are well-defined in the literature and sometimes authors use them ambiguously.

Test Case Values and Expected Results

A software system is generally regarded as an engine for transforming a given set of input data to (one or more) outputs through computational processing. The corresponding transformations are commonly referred to as an *IPO* (input-processing-output) model and constitute the basic starting point for our considerations. Software testing is essentially about checking the correctness of these transformations, and represents one way to use a system. In his seminal book on Software Engineering [Som10], Sommerville uses a drawing similar to Figure 2.1 to introduce and describe an input-output-model for software tests.

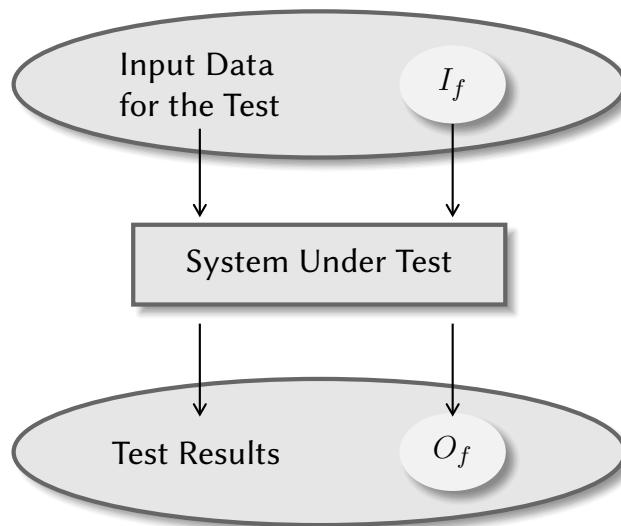


Figure 2.1.: Input-Processing-Output Model of Program Tests [Som10].

In this drawing, the I_f entity represents the input data that causes the system to fail, while O_f is the set of outputs produced by the system that indicates an error. Building on this IPO model, the essential ingredients for the construction of a software test are a description of the input data space, the software under

test, which is responsible for processing the input, and the expected outcome (i.e., output) of the execution of the software under test.

To describe a concrete test for the system under test, it is first necessary to pick a set of input data for the software from the input data space, the so-called *test case values*:

Definition 2.1 (Test Case Values). *Test case values are the input values necessary for the execution of an operation of the software under test.*

In terms of the IPO model, the *test case values* play the role of the input data / test data for the software under test, i.e., the values provided in the parameter list of an operation. In the remainder of this thesis they will be denoted by the Greek letter α . Thus, the test case values used for operation invocations in software tests can be denoted as *n-tuples*

$$(\alpha_{1,i}, \alpha_{2,i}, \dots, \alpha_{n,i})$$

thereby representing the n input parameters to the invocation of method i .

Since the examples and techniques used in this thesis primarily focus on Java and JUnit, we will regard the terms *class under test* and *component under test* (hereinafter referred to as *CUT*) as synonymous with *software under test*. An *invocation* or *execution* will usually mean the invocation of a *method* or an *operation*, respectively. Although we are aware of the perennial dispute about the question “*what is a component?*”, we will not investigate this issue in detail in the context of this thesis. For the interested reader, the literature provides plenty of discussions on the commonalities and differences between component models, such as those described in frameworks and methodologies like *EJB*, *DCOM*, *SOFA*, *PECOS* or *KobrA* (see, e.g., [Fal10]).

With the choice of the *Java* programming language and the *JUnit* testing framework as a vehicle for our considerations, the above definition has to be interpreted in conjunction with the abstract definition of the interface of a Java class, as depicted in Figure 2.2. The basic structure of a Java class interface is formed by the classname followed by a list of methods interfaces, which is enclosed in

brackets. A method interface itself consists of the method's name, the list of parameters it expects and the declaration of its return type. The parameters are expressed as types of the expected values.

$$C((m_n(\alpha_{x,n}^{0,i}) : \Gamma_n;)^{0,j})$$

C : classname

$\alpha_{x,n}$: parameter x of method m_n

m_n : name of method n

Γ_n : return value of method m_n

Potency ($y..z$) : multiplicity of occurrence

Figure 2.2.: Generic Structure of a Class Interface.

A method can expect an arbitrary number of comma-separated input parameters, while a method without input parameters is indicated by an empty pair of brackets (). The return type is mandatory unless there is no return value. This is indicated by the keyword void. Theoretically, a class may contain no method at all (beside those inherited from Object), which means that it is practically useless in our context.

The value returned by a software component in response to specific input is the essential cornerstone for software testing. Software tests observe a CUT's behavior, which is manifested by its output in response to given input. This "real" result obtained by the execution of the software under test is compared against the *expected result* of a test, which is defined as follows:

Definition 2.2 (Expected Result). *The **expected result** is the result that should be produced by the execution of the software under test if and only if it satisfies its intended behavior [AO08].*

Although it seems obvious and natural, in addition to this definition, Myers formulates a stronger requirement for expected results / outputs by requesting that they have to be defined prior to the execution of a program. However, it is an important question in the software testing literature, how to determine the

correct output of an invocation for a given set of test case values. The term that is commonly used to describe the source of information about expected results is the term *test oracle* [Wey82; Bei90; RAO92; SWH11], although it does not define *what* an oracle should be: it can be an algorithm, some equivalent implementation of the system under test (like in Search-Enhanced Testing [AHJ11]) or a human being, which is sometimes also called a “golden oracle” [Hum+06]:

Definition 2.3 (Test Oracle). *A **test oracle** determines the expected result of an invocation of the software under test for a particular set of test case values. The source of test data and the test oracle can be independent.*

Thus, it is clear that the term *expected result* should not be confused with the term *test result*. While the former is some kind of “virtual” value which has to be defined by an oracle, the latter is the concrete output of the software under test for a given set of test data. Beizer [Bei90] gives an overview on possible sources of oracles, amongst which he specifies *kiddie testing* (“run the test and see what comes out”) as a valid means to obtain test oracles if the tester has knowledge about intermediate variable values and shows high personal discipline. Further sources of oracles for software testing are, e.g., regression test suites (existing tests of the same project used during refactoring or maintenance) and existing programs (cf. Chapter 9.2).

Software Tests

Unfortunately, literature is not clear about the meaning of what a *test* itself represents. Although Ammann and Offutt give a large set of definitions for different terms in testing, they do not define the word *test* itself. Myers, however, defines the nature of testing and states that

“Testing is the process of executing a program with the intent of finding errors.” [MS04, p. 6]

Assuming that testing is a process that not only relies on the execution of a program, but also on the observation of its behavior, the above statement implies

that a test is the smallest element in the process of testing. Hence, we define a test as follows:

Definition 2.4 (Test). *A **test** is the atomic action in the process of software testing. It compares the actual test result, which is obtained by executing the software under test using test case values, to the expected result that is provided by an oracle.*

By saying that a test is “atomic” in the sense of the process of testing, we still acknowledge that it contains execution steps and comparison steps as its basic actions. If the test result differs from the expected result, one could automatically assume that a test has discovered an error in the software under test. It is, however, necessary to understand that the observation of an unexpected test result may also be caused by a bug in the test [Bei90, p. 20]. Therefore an unexpected test result should not only lead to the debugging of the software under test, but also to a review of the test itself.

In the JUnit framework, the test result (obtained from an invocation with test case values) and expected return value are subsumed in an *assertion* statement, which we will use synonymously with *test*. An assertion or test can be described as the assignment

$$\xi : (\alpha_1, \alpha_2, \dots, \alpha_n) \rightarrow \Gamma$$

meaning that the invocation of method ξ using the n -tuple of input parameters is expected to instruct the program to produce the result Γ . To conclude these considerations we can rephrase Myers’ initial statement and regard software testing as the process of finding errors by executing a program² using tests.

Test Cases and Test Suites

Usually it is necessary to define several assertions to effectively investigate a piece of software for the presence of bugs. Therefore software tests are usually contained in *test cases*, which combine tests with pre- and post-actions that

² this term applies to different levels of granularity, e.g., unit (i.e., class), component and system.

ensure that the required conditions are valid before the tests are executed, such as ensuring that the software under test is in a certain state. Based on the definition of Ammann and Offutt, a test case can be characterized according to the following properties:

Definition 2.5 (Test Case). *A **test case** is composed of tests (i.e., an operation invocation with test case values and a corresponding expected result), supplemented with set-up and tear down actions. Thus, test cases are usually used to evaluate an operation of the software under test.*

To keep software tests maintainable, one test case should test one single function or component of the system under test. Test cases for similar or related parts of the software under test are collected in *test suites* or test sets, which Ammann and Offutt simply define as a set of test cases [AO08].

Definition 2.6 (Test Suite). *A **test suite** is a collection of test cases. It orchestrates their execution and is intended to evaluate the system under test as a whole.*

These terms represent the fundamental vocabulary of software testing used throughout this dissertation. We have chosen not to rely on previous definitions from the literature, which are often ambiguous and sometimes contradictory, as we will see in the following subsection. Instead, we have aimed to stay very close to the most common definitions from the literature, but we have also clarified their meaning where this was necessary. This is essential for the following chapters, since we are going to define a meta-model for reusable software tests later in this thesis and this cannot be done without a consistent terminology for the considered domain.

Ambiguities

Despite the fact that software testing has to be performed in every development project, software testing still lacks a common definition of many of its terms. In the literature authors usually introduce different terms that have the same

de-facto meaning, or even worse, a single term that refers to different things. This deficiency is also mentioned in the book by Ammann and Offutt [AO08], who state that in order “to follow tradition” they sometimes use their definition of a test case (composed of test case values, expected results and so-called *prefix* and *postfix values*) in place of test case values, where this is clear from the context [AO08, p. 15].

Since imprecise and confusing terminology is a general malady in software engineering today, we want to avoid ambiguity wherever possible. For the sake of linguistic clarity, we give an overview of the introduced terms in Table 2.1 and list their corresponding element in the JUnit framework.

Software Testing	Java / JUnit
Test Case Values	Method Input Parameters
Expected Result	Expected Value (Assertion)
Test	Assertion
Test Case	Test Method
Test Suite	Test Case / Test Suite

Table 2.1.: Terms in Software Testing and Java / JUnit.

As one might have already observed from the definition of a test case and test suite, the general meaning of these terms is different to their usage in JUnit. We will discuss this issue in more detail later in this thesis, but it is nevertheless worth noting at this point, that the general definition of a test case is more similar to a test method contained in a JUnit test case than to the `TestCase` class defined in JUnit 3.

With the introduction of JUnit 4, the `TestCase` class is no longer used to define a set of test cases, but the usage of the `@Test` annotation before a test method is also misleading. Therefore, an annotation `@TestCase` might have been more precise in this context. Hence, our definition of a test case corresponds to a test method in a JUnit test class, while the term test suite describes both a test class as well as a test suite that orchestrates the execution of test classes.

2.2. Extracting knowledge from JUnit

Software testing is generally regarded as a labor intensive, challenging and expensive task that is nevertheless essential to the development of quality software. A prominent example of the huge impact of a single line of faulty code is the so-called *goto fail* bug, where a single line of code (i.e., a simple ‘`goto fail;`’ statement) corrupted the SSL stack of Apple’s OS X and iOS³.

For the remainder of this thesis, we have chosen to focus on tests written using the JUnit testing framework. Since it is a widely used tool for software testing, there is a broad range of literature available [BG14] and most readers should be familiar with JUnit’s testing model. The methodology and ideas developed in this thesis should, however, be easily applicable to other testing frameworks and languages as well. Based on the previous definition of software testing terms we are going to develop a generic model of software tests in Section 6.2.1 of this thesis.

The Class Under Test

Unfortunately, there is still no unambiguous standard methodology for software testing, and therefore it is no surprise that testers may write totally different tests for testing the same conditions for a given piece of software. As we will see, it is possible to describe functionally equivalent tests in syntactically very different ways. This is not surprising, since it is well known that two implementations of the same program will rarely look exactly alike. In this section, we consider a very simple scenario, where a tester wants to investigate whether a distance calculator performs the computation of the distance between two given points correctly.

Therefore we start with the little program in Listing 2.1, which serves as the class under test for the subsequently presented test cases.

³<https://github.com/landonf/Testability-CVE-2014-1266>, checked February, 24th 2014

Listing 2.1: Distance Calculator – Class Under Test.

```

1 public class Euclid {
2     public double dist(double x1,double y1,
3                         double x2,double y2) {
4         return Math.sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1));
5     }
6 }
```

For our purposes, we define the example class `Euclid` with the method `dist`, which calculates the distance of two points $p(x_1|y_1)$ and $q(x_2|y_2)$ using the well-known formula for the Euclidean distance:

$$d(p, q) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \quad (2.2.1)$$

A tester can arbitrarily choose any points, such as $p(4|2)$, $q(8|5)$ with a distance of $d = 5$, to test the program for bugs. According to our previous definitions, the general description of such a test

$$\text{dist: } (4, 2, 8, 5) \rightarrow 5$$

maps the given test case values $(4, 2, 8, 5)$ to the corresponding expected result of 5. It therefore states that the test is performed executing the `dist` operation of the system under test.

JUnit TestCase Example

Even though tests written for the code example from Listing 2.1 are intended to examine bugs in only one single calculation – which is performed in the return statement in line 3 – there are virtually unlimited ways of writing a test case for this class. The typical approach would be to write a test case that defines the correct outcome of the invocation of the distance calculator's `dist` method depending on an ordered quadruple of input parameters, like the one presented in Listing 2.2.

Listing 2.2: Test for the Distance Calculator with assertEquals.

```

1 public class EuclidTest {
2     @Test
3     public void testDistanceCalculation() {
4         Euclid calc = new Euclid();
5         assertEquals(5,calc.dist(4, 2, 8, 5));      // P, Q
6         assertEquals(5,calc.dist(0, 0, 3, 4));      // Origin
7         assertEquals(12.5,calc.dist(-3, -4, 4.5, 6)); // F1
8     }
9 }
```

This test adheres to the conventions of the IPO model: it takes, for instance, the input (0, 0, 3, 4) as test case values for the software under test and evaluates the test result against the expected result for equality. In this case, the developer also provides an expected value in the way defined in the JUnit documentation, i.e., as the first parameter of the assert statement. Hence, from this test case it is possible to extract the mapping of test case values to the corresponding expected results (0, 0, 3, 4) → 5 directly from the code.

Test Syntax vs. Test Semantics

Since JUnit classes are well-known POJOs⁴ there is, however, no automated check for the correct order of the parameters of the assertEquals statement. This fact imposes a first obstacle to the automated parsing of JUnit tests, since a tester may write line 5 in Listing 2.2 conversely as

```
assertEquals(calc.dist(4, 2, 8, 5), 5);
```

which would – strictly speaking – be represented by the following mapping of test case values to expected result: 5 → (4, 2, 8, 5). This would not be correct in the context of the class under test, although it represents a valid statement. But before addressing this issue, we want to investigate another way of writing a test for the same distance calculator class as presented in Listing 2.3.

⁴ POJO = plain-old Java object.

Listing 2.3: Test for the Distance Calculator with assertTrue.

```
1 public class EuclidTest {  
2     @Test  
3     public void testDistanceCalculation() {  
4         Euclid calc = new Euclid();  
5         int pqDist = calc.dist(4, 2, 8, 5);  
6         int originDist = calc.dist(0, 0, 3, 4);  
7         double floatingDist = calc.dist(-3, -4, 4.5, 6);  
8         assertTrue(pqDist == 5);  
9         assertTrue(originDist == 5);  
10        assertTrue(floatingDist == 12.5);  
11    }  
12 }
```

This test case is more challenging to an automated knowledge extractor, since there are several issues that make it difficult to create the mappings of input values to expected outputs. Although the use of `assertTrue` instead of the equality assertion can be resolved relatively easily, the definition of variables containing the test result and their usage in the assertion is somewhat challenging. Since the test is now split into two parts, where *a*) the CUT is invoked using test case values, *b*) the test result is assigned to a variable and *c*) this variable containing the test result is compared to the expected result the extraction of the tests from this test case demands more sophisticated algorithms.

Testing without using JUnit

For the sake of clarity this test case also makes it necessary to revisit the previous definition of the term “test”. Although this test case might appear to deviate from the definition in Section 2.4 this is, however, not the case. A test was defined as the execution of the software under test using test case values (e.g., line 5 in Listing 2.3) and comparing the test result to an expected result (e.g., line 8 of the same listing).

The last kind of *test case* we want to take a brief look at is not a JUnit test case in the narrow sense, but a test case expressed using plain Java code and a boolean value that indicates whether the returned result matches the expected result or

not. Nevertheless, the code presented in Listing 2.4 meets our definition of a test case and a test since it aims to reveal an unexpected result by executing the system under test using test case values.

Listing 2.4: Test for the Distance Calculator without JUnit.

```
1 public class DistCalcVerifier {  
2  
3     // this is true if the test is passed  
4     public boolean verifyDistanceCalculation() {  
5         Euclid calc = new Euclid();  
6         if (calc.dist(-5,5) != 0) {  
7             return false;  
8         }  
9         return true;  
10    }  
11  
12 }
```

The code in Listing 2.4 provides evidence that even before the advent of JUnit or other similar testing frameworks it was already possible to test Java classes and look for defects using so-called *plain old Java objects (POJO)*. A human can recognize the *Calculator* as the system under test, the tuple $(-5, 5)$ as test case values and 0 as the expected result.

However, for a parser such a piece of code does not contain enough structural information that would reveal it to be a test case containing a test for the above calculator. Hence, the problem of extracting the test case values and expected results out of such assets is out of scope of this thesis, since our approach envisages automatic knowledge extraction from test cases. In the following, we will focus on test cases which are automatically recognizable as such.

The following chapters will introduce software search engines and reuse-oriented recommendation systems along with the techniques applied in these field. Based on the foundations presented, we are going to create a search engine for reusable software tests and develop a reuse-oriented test-reuse environment, which enables users to reuse software tests directly from within their IDE.

2.3. Summary

In this chapter we have introduced the essential terminology of software testing used within this thesis. Furthermore, we have given an overview of the structure and usage of JUnit. We need the definitions and considerations from this chapter later in this thesis, when we define a data model for test reuse and explain the creation of a reuse-assisted test recommendation system. Additionally, we have identified issues arising from the fact that JUnit test cases are plain Java code, which offers developers and testers a large variety of possibilities to write a test case for a given program. The examples presented in this chapter show that semantically equivalent tests can be expressed using a different syntax, which makes it more difficult for a parser to automatically extract the information contained in test cases.

Since a detailed introduction to software testing and JUnit is out of the scope of this thesis, we refer the interested reader to literature available on these topics [Mye79; Bei90; Bec03; AO08; BG14]. Additionally, there is a large body of knowledge available on the internet.

Contribution of this chapter

- This chapter has given a definition of the essential terms for software testing. Tests execute one operation with a well-defined set of test case values and a corresponding expected result, which is compared to the result returned by the operation. Multiple tests for an operation are grouped in test cases. Test cases for several operations of a component are grouped into test suites.
- We have introduced a couple of examples of the use of JUnit that give a basic understanding of the framework's concepts and drawbacks.

Part II.

Search and Reuse

“ Computers are magnificent tools
for the realization of our dreams,
but no machine can replace the human spark
of spirit, compassion, love, and understanding.”

Louis Gerstner, Jr.
CEO of IBM 1993–2002

3

Software Search and Recommendation

As software becomes an omnipresent part of our environment and is embedded in ever more devices, the quantity and variety of source code written to support them grows steadily. Vast numbers of software artifacts have been made publicly available as part of the so-called *open-source revolution* or are stored in huge corporate repositories. Furthermore, the availability of high-bandwidth network connections has made these accessible at the touch of a button. During the early 2000s this triggered the emergence of numerous internet-scale code search engines, which aimed to leverage the success of document-based search engines to promote software reuse in general.

As earlier said, during that time, most of the available code search engines implemented the ideas of Google, Yahoo! or other “traditional” search engines and usually did not offer much more than simple keyword-based searches. Plain

text search is, however, not sophisticated enough to allow developers to find the components they need with reasonable time and effort, since it does not exploit the fact that code is executable – in contrast to a text document. The original research on software retrieval during the 1980s and 1990s did exploit the idiosyncrasies of software (such as operation signatures) but was unable to cope with more than a few thousand software components. The survey of Mili et al. [MMM98] provides a comprehensive overview of the state of the field in the late 1990s. Recent research has focused on providing more scalable and sophisticated retrieval approaches such as Component Rank [Ino+05] or Test-Driven Reuse [HJA08].

3.1. Search Scenarios in Software Engineering

“Software reuse is the process of creating software systems from existing software rather than building software systems from scratch.” – Charles W. Krueger

This simple idea expressed by Krueger in 1992 [Kru92] and first envisioned by McIlroy in the late 1960s [McI69] has been the general motivation for research into software search and reuse for over four decades. More detailed understandings of the different use cases for software search have only recently emerged through several studies and on-line surveys such as those described by Umarji et al. [USL08] in 2008. The basic goal of their survey was as to answer the question “what do developers search for?” and the most frequently given motivations for search are presented in Table 3.1.

From a total of 58 anecdotal descriptions of how developers used existing search engines, the authors of the survey identified nine archetypal motivations for code search (i.e., use cases). Eight of these nine use cases were motivated by the goal of reusing software, while one was motivated by the goal of fixing bugs. The eight reuse use cases were further divided into two groups of four – the first motivated by the desire to directly reuse code, of the size from small code snippets through class-size units to standalone (sub-)systems – and the second

	Description	Code for Reuse	Reference Example	Σ
Block	Wrappers, parsers, code excerpts	7	4	11
Subsystem	Code of algorithms and data structures, GUI widgets, Library, APIs	21	11	32
System	Stand-alone applications	6	2	8
Σ		34	17	51

Table 3.1.: Motivation for code search by target size [USL08].

motivated by the desire to find reference examples that provide ideas about how to implement a particular piece of functionality. A prominent example from this survey is the use of search engines to find guidance in the use of libraries – a topic that has received significant research attention (see, e.g., Holmes and Murphy [HM05]).

Summarizing the work by Umarji et al., we can group searches motivated by the goal of reusing code without modification into the following four categories:

- code snippets, wrappers or parsers,
- reusable data structures, algorithms and GUI widgets to be incorporated into an implementation,
- reusable libraries to be incorporated into an implementation,
- a reusable system to be used as a starting point for an implementation.

Furthermore, we can group searches motivated by finding reference examples into these four categories:

- a block of code to be used as an example,
- example of how to implement a data structure, algorithm or GUI widget,
- example of how to use a library,
- looking at similar systems for ideas.

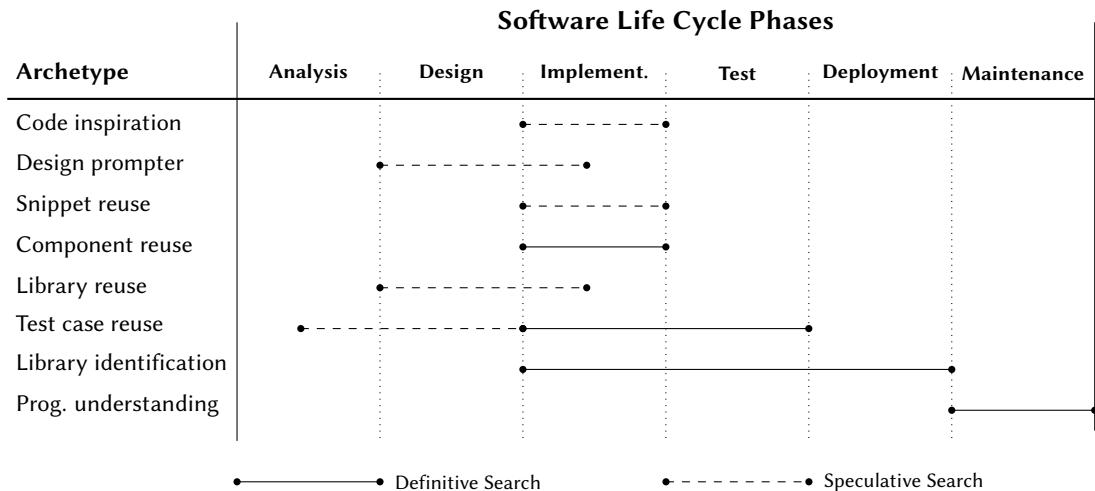


Figure 3.1.: Search Scenarios in Software Engineering [JHA10].

We have identified eight archetypal search scenarios in the context of the traditional software development life cycle, which are visualized in Figure 3.1. The searches are grouped into so-called speculative and definitive searches. The former are represented by dashed and the latter by solid lines [JHA10]. The figure shows that searches conducted early in the software development process (i.e., late in the analysis or early in the design phase) are rather speculative. At this point in time, developers are using software search engines to get an idea of what reusable material is available or to get some inspiration on how to solve a given task.

Later in the process (i.e., late in the design phase or during coding) searches can become much more definitive as typically a concrete specification of a required component is available. Since the focus of this thesis is test case reuse, which is a special case of source code recommendation, the tools presented in this chapter focus on the following four types of software search: *a*) Snippet Reuse, *b*) Component Reuse, *c*) Library Reuse and *d*) Test Case Reuse.

3.1.1. Speculative Searches

The most common usage scenarios for software search engines come under the umbrella of so-called *speculative searches*. These are searches performed

by developers with the goal of finding out what is available in a repository. Additionally it is a helpful strategy to get an initial idea of what might be a good design for a component that is going to be created. Typically users performing a search have a concrete description of a task and are looking for previous examples of how something similar has been implemented. They often start by searching the web for discussion threads, looking in particular for explanations of the pros and cons of different frameworks or for insights into how other developers have approached the problem before [JHA10; JHA14].

During this stage of software development neither a detailed syntactical nor a clear semantic description of the component in question is likely to be available. The developer uses the search engine to help design the component under development and identify a suitable architecture for the implementation. The approaches of tools such as the design prompter [HJA10] fall under the category of speculative search, as does the idea of drawing inspiration from open source and library searches described in [USL08]. Looking for reusable code snippets is the only minor exception in the context of speculative searches, since developers usually do this late in the development process when they are implementing the system in hand.

Even when all the technical difficulties for speculative searches (such as syntactical mismatches, etc.) have been solved, human factors such as company regulations, unsuitable open source licenses or the well known “not invented here syndrome” [FF95] often still deter developers from directly integrating reusable material into their code. Furthermore, common sense suggests that the earlier reuse can be performed in the software development lifecycle, the greater the benefit in terms of increased development efficiency and decreased development costs. This assumption is supported by Boehm’s observations, which show that the later changes occur in a software project, the more expensive they become [Boe81]. Furthermore, Crnkovic et al. found that reuse becomes more difficult the later it occurs in a system’s development cycle [CCL06].

Speculative searches are typically keyword-based searches, making little or no use of additional characteristics of the code – such as interface descriptions or more concrete functionality specifications – and usually occur during the

design phase of a software project or during the early implementation phase. Our investigations show that these keyword-based searches tend to be rather imprecise [HJA07], i.e., about four out of five results of a search are incorrect. This is not really a problem, however, since at this stage the developers usually have no clear picture of the component which has to be written and may investigate the available solutions.

These speculative searches can be performed in many ways. Two examples from our earlier publications are, for instance, 1) the so-called *design prompter* and 2) the *code inspiration* scenario. The latter is presumably the most frequently performed type of speculative searches [JHA10]. In the following paragraphs, we give a brief description and examples of how the above kinds of speculative searches can be used by developers.

Design Prompter Given the progress achieved in data mining and related areas in recent years, it makes sense to investigate the possibility of automatically generating design hints based on the knowledge wrapped in existing collections of software. Like the shopping systems of large online retailers a proactive design prompter system might, for example, suggest to a developer that “other developers that have created a stack component also assigned a push and a pop method to it” [HJA10]. Such a system needs to monitor the developers while they are designing or coding a system and then make recommendations based on the “mean value” of artifacts that other developers created in similar situations. This idea is not necessarily limited to the class level, it also seems feasible to extract helpful design or even architectural patterns from the contents of a software repository.

Code Inspiration When a task is assigned to developers, they usually have only a rough idea of how this could be implemented. A search engine can be helpful to get an idea of how other developers have solved a similar task, to better understand the problem domain, to get an idea of possible risks (like vaguely known APIs) and to create a draft outline of the code. Ye calls this approach *Glass-Box Reuse* where programmers do not directly reuse a software artifact but rather use it as an example for their own

implementation [Ye01]. This contributes to the quality and productivity of programmers by reducing their cognitive load [Nea96].

3.1.2. Definitive Searches

As soon as a software system's design has become concrete enough that the "contours" of its components are clear, the requirements for a search engine change significantly as the additional information provided by the system and component specifications can be used to define more focused searches. Although source code is text, the behavioral and structural information it contains can support much more sophisticated searches than merely text-based searches. For example, the process of compiling source code reveals a lot of information that can be exploited when searching for components and deciding which candidates to return within search results. In general, definitive searches can be categorized as into the following main groups:

Interface-based Searches Our previously published evaluations have shown that ordinary keyword-based (even pure signature-based) searches (see Mili et al. [MMM98] for detailed explanations) in internet-scale repositories do not deliver reusable material with a satisfactory level of precision [HJA07]. The main reason for this is the ambiguity that creates a large number of candidates that match a general keyword or signature [DR12]. Interface-based searches improve the precision by matching the complete interface of the sought after component (i.e., component name, method names and parameter profiles, etc.) against the interfaces of components in the search space. By including various stemming and relaxed name-matching techniques, significantly higher precision can be attained without sacrificing recall.

Test-Driven Reuse Although they improve on simple text-based searches, the precision of interface-based searches still leaves a lot to be desired. User's still have to evaluate the fitness-for-purpose of the results and may get frustrated by too many "false positives". Test-driven searches raise the precision even further by exploiting the fact that source code can be executed

and thus is behaviorally observable. In contrast with formal specifications, which also provide a description of the required functionality of components, test cases are frequently developed anyway within most mainstream development processes and thus lend themselves for use as queries for searches[Hum08; Rei09].

Discrepancy-Driven Testing The reuse of existing software components is not always an accepted tool in software development. Licensing issues or company regulations may dissuade developers from integrating reusable components into their software. However, it is possible to utilize such components for the purpose of software testing: the retrieved components serve as so-called test *oracles*, which can be executed alongside the component under development utilizing, e.g., a large amount of randomly created test input. The result values of these components can be compared with each other [Hum+06; AHJ11] and when a disagreement occurs between them an interesting test case worthy of human consideration has been discovered. We will elaborate on this in more detail in Section 9.2, where discrepancy-driven testing is utilized to improve the results of reuse-assisted software testing.

Test Reuse As software search engines not only contain components, but often also the test cases intended to test them, we can extract the knowledge stored within the test cases as well. The idea of reusing software tests to enhance newly created test cases is the main focus of this thesis.

Library Searches During the development of software systems, developers often need to incorporate additional external libraries into their projects. Software search engines can be used in a variety of ways to increase the productivity of developers in this context. When libraries are inadequately documented or the achievement of a certain result demands a possibly complex invocation chain that cannot easily be discovered by the developer, recommendation systems that provide suggestive code snippets can greatly reduce the learning time needed to understand and use the provided API [RWZ10].

3.2. Software Search Engines

Search engines specialized on code provide the backbone for modern code reuse recommendation tools. The most significant and widely recognized code search engines that were developed within scientific projects are *Agora* [SHW98], *Sourcerer* [Baj+06], *Merobase* [HJA08; Jan+13] and *S⁶* [Rei09]. All of the aforementioned focus primarily on the Java programming language and thus it is no surprise that the recommendation systems which rely on them are also primarily Java-oriented¹.

The timeline in Figure 3.2 summarizes the main scientific milestones of the last quarter of a century in the field of code search and recommendation. The white boxes in the figure represent important publications related to software reuse, the red ones the announcement of a search engine and the blue boxes represent the release of a recommendation system that is based on a search engine. It is important to note that although there have been industrial products for code search and reuse, none has yet “taken off” commercially. One of the commercially “biggest players” – *Google Codesearch* – shut down its service in January 2012 [Goo11].

3.2.1. Agora

From the visionary ideas in the late 1960s, it still took the scientific community more than twenty years to develop a widely-recognized reuse-recommendation system. The introduction of *CodeFinder* [FHR91] marked a milestone, although it took some more years until the emergence of Agora, a component search engine developed within the Software Engineering Institute at Carnegie Mellon University [Sea99]. Its purpose was to provide a search engine that supports searches for components based on the description of their interface properties.

Since its user interface was very similar to a “classic” web search engine’s interface, users had to explicitly open the browser and formulate search queries using

¹At the time writing only Merobase and S⁶ were still publicly available at their publicized address.

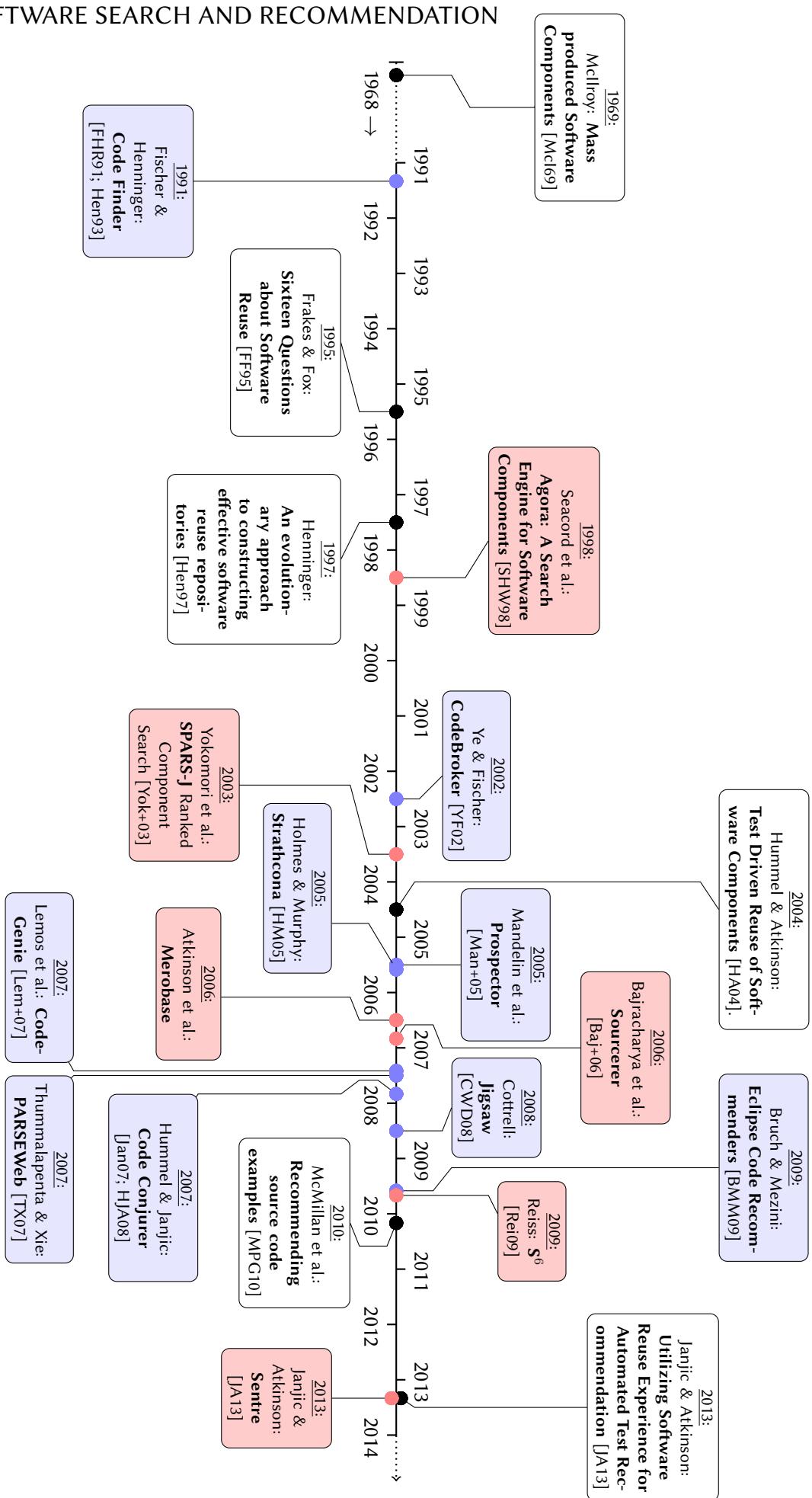


Figure 3.2.: Milestones in Source Code Search and Recommendation.

a dedicated query language. Searches with Agora were constructed using basic operators like ‘+’ or ‘-’, indicating whether a term should be required or prohibited. Furthermore more complex searches could be formulated using boolean operators AND, OR, NOT or NEAR. A search for a component that provides a scrollable drawing pane supporting color-draws would, for instance, require a query of this form:

```
+method:scroll +property:color +method:draw
```

Obviously, this is not a very convenient way to issue searches during software development since users must leave their development environment, open a browser window and – most importantly – invest additional time and effort to create an appropriate query. Therefore, they must have a fairly clear picture of what they are seeking, especially when using the specialized features of the query language like boolean operators. The need to learn the query language and to come up with a suitable query consumes time that diverts developers from their main task: the development of the software system. After the results have been presented, developers have to inspect them manually, adapt them to their dedicated context and integrate possibly suitable reuse candidates into their project. This can be a very time consuming and tedious task since it involves a lot of manual effort which requires developers to build and execute the system to try out each new component, regardless of whether or not it turns out to be suitable for the envisaged purpose.

3.2.2. Merobase Component Finder

Another half a decade passed before *SPARS-J* and the *Merobase Component Finder* were introduced in 2003 and 2006. At that time, scientific groups and companies began exploiting the possibilities to create software search engines. The commercially most famous were *Koders.com* (2005), *Google Codesearch* (2006) and *Krugle* (2006). It is necessary to mention that *Koders.com* was sold to Black Duck Software in 2008 and merged with *Ohloh Code* in 2012, which itself has already been sold twice and still merely supports keyword- and name-based searches.

Merobase was the first search-engine to support test-driven searches with server-side test execution. Its sandboxed execution environment enables it to exploit the power of distributed computing clusters and avoid the execution of possibly malicious source code on the developers' machines. Driven by the Lucene framework, Merobase is not only able to perform very fast keyword-, signature- and interface-driven searches, but also to provide accurate results to test-driven searches [Hum08]. When crawling for code, Merobase's analysis algorithm identifies the basic abstraction implemented by a module and stores it in a language-agnostic description format. The description's most important element is the abstraction's name, but other key features such as method names and parameter signatures are also stored within the search repository.

The above mentioned search strategies supported by Merobase can be used either via the official web site of Merobase or using a distinct web service API. In both cases, the most prominent query forms are those for interface-based searches and those for test-driven search. The *Merobase Query Language (MQL)* supports *function oriented* and *object-oriented queries*. To search for a component based on a function that has to be implemented by the results, the query contains the name of the function, a comma separated list of its input parameter types enclosed in braces followed by a colon and the type of its return value. The end of the query is indicated using a semicolon:

```
getDistance(float,float):float;
```

The above query is used to retrieve components that contain a method called `getDistance` with two input parameters of the type `float` and a return value of the type `float`. The MQL also supports empty parameter list and the return type (with colon) is optional. For object-oriented searches, the first element of the query is the component name followed by a list of function-oriented queries enclosed in curly braces:

```
Customer {  
    getAddress():String;  
    setAddress(String):void;  
}
```

The architecture of Merobase – as depicted in Figure 3.3 – was initially composed of three tiers: the backend with the search index and database, the system responsible for adaptation and evaluation of the candidates in test-driven searches and finally the web front-end that enables users to interact with the system.

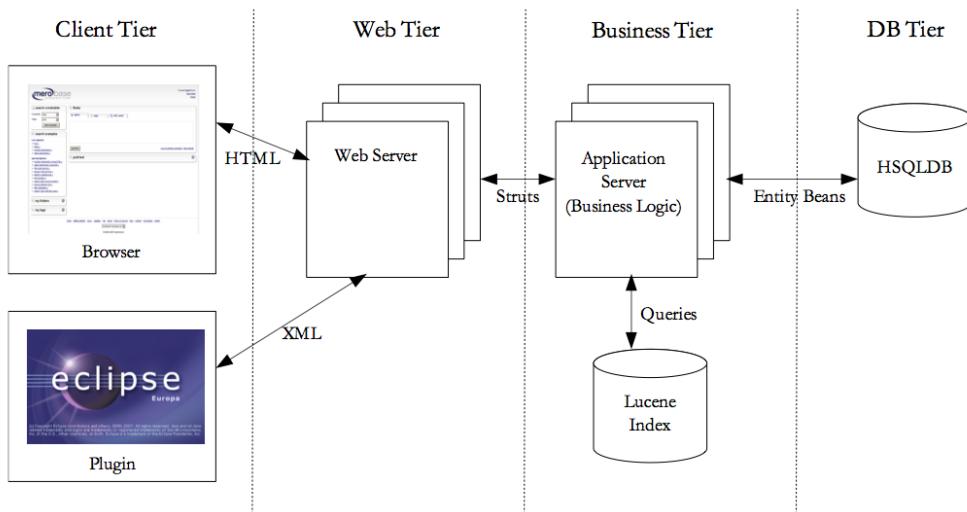


Figure 3.3.: Initial System Architecture of Merobase [Hum08].

Merobase also serves as back-end for our Code Conjurer Eclipse plug-in (cf. Section 5.3.6), which initially used a proprietary XML format to communicate with the web server. It enables users to communicate with the search engine directly from within their IDE, either pro-actively or via a background agent that looks for reusable artifacts relevant to the development context.

3.2.3. Sourcerer

The Sourcerer search engine, developed as a search engine for open-source code, relies on a relational database created by mapping basic source code elements and their relations to a relational model (see [Baj+06] for more details). It uses fingerprints, which are a vector-based representation of various attributes in the code, to provide the basis for structural searches. To improve the quality of

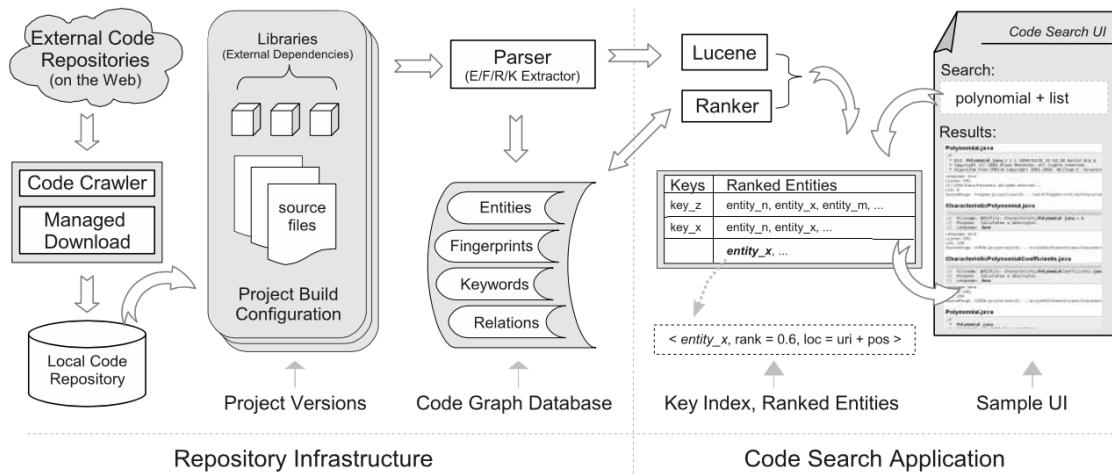


Figure 3.4.: The Architecture of the Sourcerer Infrastructure [Baj+06].

the results it incorporates a ranking technique called *CodeRank*, which is based on Google's *PageRank*. Sourcerer's index was built to exploit the fact that the retrieval of source code differs from traditional document retrieval and offers five different search functions:

1. Component Search
2. Component Usage Search
3. Function Search
4. Function Usage Search
5. Program Structure Search (Fingerprint Search)

The architectural organization of the Sourcerer infrastructure is depicted in Figure 3.4. It shows the main division into a repository infrastructure and a code search application. While the former is responsible for crawling, downloading and processing code from external repositories, the latter supports search for the retrieved artifacts and offers an appropriate user interface.

The Sourcerer code search engine serves as back-end for the *Code Genie* Eclipse plug-in [Lem+07]. In contrast to Merobase, Sourcerer does not support server-side test-driven searches, but shifts this technology to the developer's computer.

3.2.4. S⁶

The S⁶ system, another test-driven search engine, was made publicly available in 2009 [Rei09]. While the previously presented code search engines enabled their users to perform keyword-based queries or provided a query language to better describe the desired component, S⁶ combines keyword based queries with the specification of test information directly on its website, as seen on the screenshot of the search engine in Figure 3.5.

Although the search engine recommends Java classes, it does not make any use of the language's features in describing queries. Thus, its users have to invest significant effort in learning how to formulate queries before they are able to use the search engine. In addition to that, the usage of S⁶ is aggravated, since there is no recommendation system available that supports the search engine's functionality and makes its usage easier and more convenient.

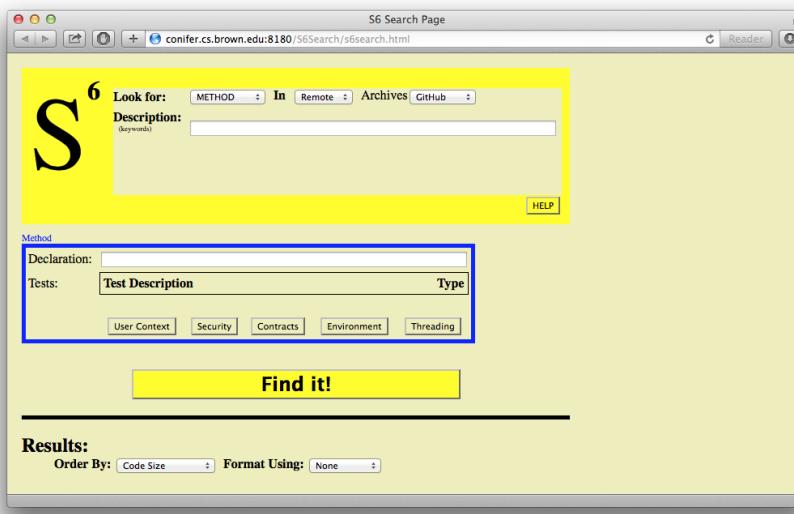


Figure 3.5.: The S⁶ Web Interface.

3.3. Excursus: Recall and Precision

Basic Measures in Information Retrieval

The information retrieval (IR) community has established two basic performance measures – *Precision* and *Recall* – to evaluate the efficiency of retrieval algorithms or systems (for a thorough introduction to information retrieval see for example the book from Baeza-Yates and Ribeiro-Neto [BR08]). Since there is still no other broadly accepted benchmark for software search engines on the horizon [HJ12], the software search and retrieval community still solely rely on these measures:

$$\text{Recall} = \frac{|\text{Relevant and Retrieved Documents}|}{|\text{Relevant Documents}|}$$

$$\text{Precision} = \frac{|\text{Relevant and Retrieved Documents}|}{|\text{Retrieved Documents}|}$$

To estimate the recall for a query, it is necessary to obtain detailed knowledge of all the documents in the repository [BR08]. Modern software repositories contain far too many components to perform a manual inspection and it seems infeasible to calculate a value for the recall of queries to modern software search engines.

In the context of this thesis, we therefore focus on precision, which is more important to the users of software reuse and recommendation systems than recall. Namely, the success of earlier published systems shows that developers obviously prefer to receive no results at all than to get a set with incorrect results that induce a significant amount of manual inspection.

3.4. Test-Driven Reuse

Past search-driven software reuse approaches failed to take off due to the lack of high precision results. Although the new code search engines that appeared since the 2000s indexed vast swashes of reusable code (thanks to the success of open source software) and offered better retrieval mechanisms with improved precision (see Table 3.2 and Table A.1), the ratio of suitable to non-suitable components in their search results was still relatively low. Our experiments presented in [HJA07] showed that Merobase's interface-based search approach was more precise than the searches of Google, Yahoo, Google Codesearch and Koders, but there is still room for improvement.

Query	Google	Yahoo	GCS	Koders	Merobase
Average Precision	12.2 %	17.9 %	29.5 %	5.9 %	53.7 %
Standard Deviation	13.3 %	18.9 %	26.5 %	7.8 %	22.4 %

Table 3.2.: Comparison of Code Search Engines [HJA07].

Another issue was that the effort involved in manually inspecting each result for suitability was relatively high since it involved the examination not only of their names and interfaces but also their behavior. Hence, there was also room for improvement to the cost-risk-effort-benefit balance in order to make effort invested in reuse more worthwhile. In his seminal work on *Semantic Component Retrieval in Software Engineering* [Hum08] Hummel describes a new approach that provides better results and reduces the effort involved in result inspection. In particular, he presents an approach called *Test-Driven Reuse*, which ultimately utilizes the fact that a software component is not only a simple textual document but an executable artifact with an observable behavior, as described by Mili et al. [MMM98].

Based on the ideas from *Extreme Programming*, where developers are encouraged to write test cases before the actual production code, test-driven reuse implements the philosophy of evaluating a software component's fitness for

purpose by checking whether it passes the test case(s) provided by the developer [HA04].

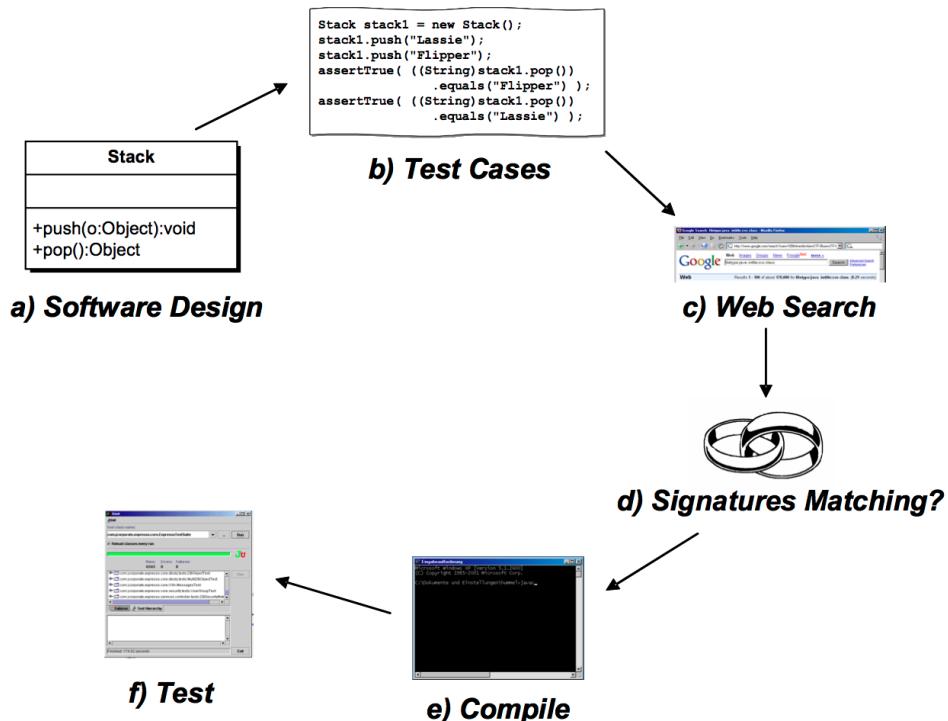


Figure 3.6.: Process of Test-Driven Reuse [HA04; Hum08]

The process overview depicted in Figure 3.6 outlines the process of test-driven reuse in its original form. Based on the design of the considered system, software developers create appropriate test cases whose required interface inherently contains a description of the interface of the desired software components in terms of call and use relationships. Suppose, for example, that a developer wants to create a payment system that accepts credit card payment and where Listing 3.1 is a fragment of a JUnit 3 test case for such a credit card component. This test can serve as an input query to a test-driven reuse system.

A search engine like *Merobase* is able to extract all necessary information from the test case – namely: 1. the required interface of the test case, which describes a credit card class that can parse a credit card number, return the vendor id

Listing 3.1: JUnit Query Snippet for a Credit Card.

```

1 public class CreditTest extends TestCase {
2     public void setUp () {
3         cc = new CreditCard();
4     }
5
6     public void testVendor1 () {
7         long number = cc.parseNumber("4111_1111_1111_1111");
8         long vendorId = cc.getVendorId(number);
9         assertEquals("Visa", cc.getIssuerName(vendorId));
10        assertNotSame("MasterCard", cc.getIssuerName(vendorId));
11        ...
12        long number = cc.parseNumber("12345678");
13        long vendorId = cc.getVendorId(number);
14        assertTrue(cc.getIssuerName(vendorId).
15                   toLowerCase().contains("error"));
16    }
17 }
```

and the issuer name, and 2. a very concrete description of the functionality of the required component by specifying test case values and expected results. Similar to the famous *caterpillar's fate* [Ker95], the short and simple interface description contained in the test case obviously requires some non-trivial code in the component in order to parse the credit card number or recognize the issuer and vendor of a card.

The extracted interface is used to issue a standard web search with an arbitrary web² or dedicated software search engine. Although it is theoretically possible to omit the name descriptors contained in the interface description during search (which then solely relies on the signature of the methods), this strategy is rather impractical (see, e.g., [Hum08]): although the number of potentially reusable results grows by several orders of magnitude, the precision drops to a relatively low level (e.g., from $\approx 21.7\%$ to $\approx 1.7\%$ for a Stack [Hum08]) because most candidates turn out to be unsuitable (i.e., higher recall is dearly bought with lower precision). Merobase therefore employs relaxed interface-based searches which has a higher precision. Table 3.3 compares the average precision of signature matching, text-based, name-based and interface-driven component retrieval

² Google, e.g., supports the search for Java files with the keyword “filetype:java”.

using the examples from Table A.1. To be efficient, a software search engine should rank the results based on the distance of their actually provided interface to the one used in the query. This allows for faster, incremental result delivery.

	Signature	Text-Based	Name-Based	Interface-Driven
Average Precision	0.9%	16.3%	17.2%	53.7%
Standard Deviation	1.8%	21.9%	19.3%	22.4%

Table 3.3.: Comparison of Retrieval Techniques [HJA07].

Once the initial interface-based search results are available, the test-driven search process looks for candidates that can be compiled without errors. Every time the provided test case can be compiled and executed successfully against the reuse candidate, the system regards it as a working implementation of the functionality specified by the developer's test case.

In addition to the idea of test-driven reuse depicted in Figure 3.6, as enhanced versions of test-driven search we already envisaged the creation of so-called software-reuse environments [HJ13; JHA14]. An initial implementation of such a software-reuse environment was realized through our *Code Conjurer* Eclipse plug-in for the Merobase Component Finder [Jan07]. The plug-in features a *background agent* which automatically creates and submits a query to the search engine and present a list of evaluated results within the IDE. Developers can then inspect the reuse candidates and choose the most appropriate one for their task at hand.

When two developers are assigned the task of implementing the same functionality it is unlikely that they will create classes that have equivalent interfaces. This issue also applies in test-driven search, where it is unlikely that a potentially

reusable component implements an interface that fully matches the requirements of a new project. Hence, it is necessary to develop additional technologies that fit the reusable assets into their new environment. In the following chapter we will address this issue in more detail, describe a possible solution and present our implementation of an automated adaptation system.

3.5. Summary

In this chapter, we have taken a closer look at the ideas of software search and reuse in general with a special focus on those technologies that are the basis for our work. After introducing typical software search scenarios, we gave an overview of the most prominent search engines for software reused developed in academia over the last decade. In addition, we briefly discussed the problem of interface mismatches in software reuse and presented the initial work from this area [Hum08; HA10].

Contribution of this chapter

- Literature review and introduction of typical software search scenarios.
- Review of software search research over the past four decades.
- An improved approach for the automated adaptation of software components, which is necessary for the automated evaluation of reusable software tests and their integration into new applications.

“ Programming today is a race between software engineers,
striving to build bigger and better idiot-proof programs,
and the Universe trying to produce bigger and better idiots.
So far, the Universe is winning.”

The Wizardry Compiled
Rick Cook, Sci-Fi author

4

Automated Interface Adaptation

The idea of test-driven reuse assumes that only the relevant results of a search for reusable assets are presented to developers. The relevance of a component is thereby defined by its ability to pass the test case defined by the developer. This strategy, however, obviously generates the problem that the execution of a component with an incompatible interface to the one required by the test case will fail and therefore it will be rejected, even if it de-facto provides the required functionality. An example for an *interface mismatch* is given in Table 4.1, which compares the required interface of the test case from Listing 3.1 to the provided interface of a search result.

Although the CreditCardTest is able to instantiate the CreditCard class (same constructor definition), it cannot test the methods responsible for parsing a credit card number, recognizing the vendor or issuer of the card. Nor is it able to format the number to a readable string (which usually divides the credit card number into groups of four). The component would be immediately rejected and never

Test Requires	Result Provides	Match
CreditCard()	CreditCard()	✓
parseNumber(String):long	parseDirtyLong(String):long	✗
getVendorId(long):long	recognizeVendor(long):long	✗
getIssuerName(long):String	issuerAsString(long):String	✗
formattedNumber(long):String	toPrettyString(long):String	✗

Table 4.1.: API Mismatch of Test and Candidate

considered as a possible search result, even if it provided the desired functionality. As long as the test and the components are written using meaningful names, however, a human would be able to map the meaning of the method definitions correctly and recognize that the component is a valid result. For a computer system, however, this is a challenging task.

4.1. Distributed Automated Adaptation System

Based on the adapter pattern initially presented by the so-called *Gang of Four* [Gam+94], Hummel described the idea of the automated adaptation of software components for reuse [Hum08; HA10], utilizing a rather naïve and inefficient algorithm. For our investigations on *Search-Enhanced Testing*, presented in Chapter 9, the automated adaptation of the provided interface of a software component to match the required interface of a test case is an essential ingredient. Therefore this section explains our improvements to automated adaptation in the context of the Merobase Component Finder. In particular, we introduce an enhanced version of the automated adaptation process, which is depicted in Figure 4.1. This process is more scalable than the previous approach and our implementation for Merobase is capable of performing the adaptation task much faster than the original system presented by Hummel [Hum08].

After the Merobase Component Finder has extracted the *required interface* of a developer's test case (as, e.g., presented in the first column of Table 4.1), the reuse candidates returned by the interface based search need to be evaluated for their fitness for purpose. In the original implementation of test driven search [Hum08]

this evaluation was performed on a separate test server where the candidate components were compiled and tested using the developer's original test case. In our improved approach, the execution environment is split into a client-server architecture that supports job distribution. The test server keeps a list of all candidate components, their current status – whether the result is *tested*, *not tested* or is *under test* – and the appropriate results.

Lightweight clients poll the server for a test / compilation task and if there are jobs in the queue, the server sends a candidate component to the client accompanied by the original test case. At this point it marks the job as being *under test*, associates it with the client and stores the current time. If the client does not return a result within a given time frame, the server considers the execution failed, increments the fail counter of this job and re-queues it. If a job fails more than three times, the server considers the corresponding class under test as potentially harmful, since it may have crashed the client's sandbox. The candidate is marked as a fail and therefore removed from the list of potential results.

On the client side the system investigates the required interface of the test case and the provided interface of the class under test. If there is an interface mismatch, the client creates new *in-between classes*, which map the calls from the test case to the appropriate methods of the candidate class. These in-between classes are also known as *adapters* [Gam+94; HA10], while the adapted class is the so-called *adaptee*.

4.2. Interface Adaptation

The easiest way to automatically generate the necessary glue code between the test and the reuse candidate would be using an algorithm that basically maps every method of the adaptee to every method in the test case. However, it is obvious that such a *brute-force* approach will not be very efficient, i.e., it will involve a lot of unnecessary executions resulting in increased resource consumption and longer duration. Hummel reported test-driven searches running more

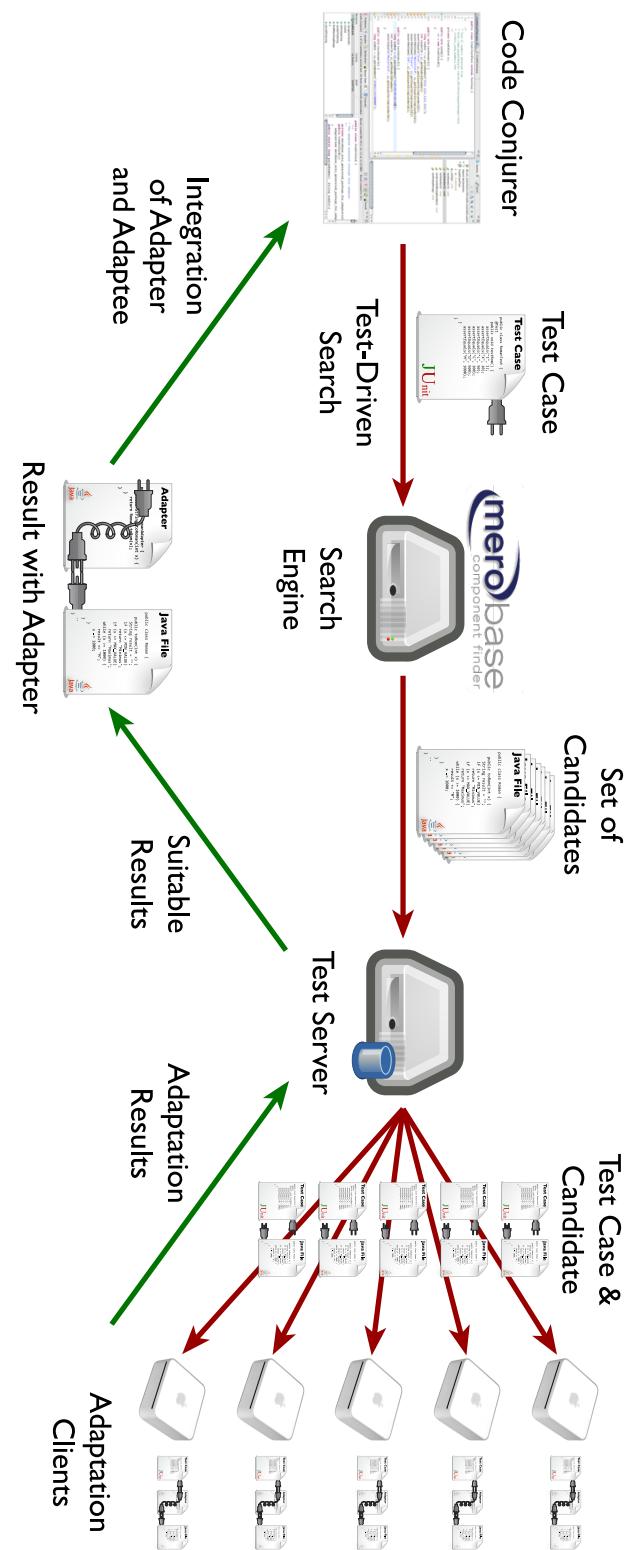


Figure 4.1.: Test-Driven Reuse with Distributed Adaptation.

than 24 hours [Hum08] using a naïve approach for automated adaptation of reuse candidates, which makes any kind of reuse-tools integrated into the IDE useless.

The input for our adaptation algorithm are two sets: M_r – one is the set of methods contained in the required interface of the test case and the other is the set of the methods contained in the adapter class and M_p , which is the set of methods provided by the adaptee’s interface. Their cardinality is defined by the number of methods contained in each particular interface:

$$n = |M_p| \quad (4.2.1)$$

$$k = |M_r| \quad (4.2.2)$$

Mapping n methods of an adaptee onto k methods of an adapter class is a combinatorial problem, which is known as *k-permutation* or partial permutation [TT10]. Since a partial permutation is defined as a bijection between two sets [Str83] the adaptation task is to assign exactly one element from (a subset of) M_p to one element from M_r . In the literature i -partial permutations are defined as originating from permutations of m . They are obtained by ignoring every element $j > i$ [Ehr73].

To estimate the overall number of possible associations between a test case and the appropriate reuse candidate, we need to calculate the k -permutation ${}^n P_k$ of the methods using the following formula:

$${}^n P_k := \frac{n!}{(n - k)!}, \text{ especially } {}^n P_n := n! \quad (4.2.3)$$

The special case results from the fact that $0!$ is defined as 1.

To give an impression of the complexity of the automated adaptation of reusable assets, we consider a class containing $n = 12$ methods¹ and a test case whose

¹ If, for example, developers adhere to the principle of *information hiding* and create getters and setters for all class attributes, four attributes would already result in eight methods.

required interface expects $k = 10$ methods. Therefore we calculate that there are

$${}^{12}P_{10} = \frac{12!}{(12 - 10)!} = \frac{479,001,600}{2} = 239,500,800 \quad (4.2.4)$$

possible combinations to map each method from the adaptee to the test case, and each of them has to be compiled, executed and evaluated. Even if the creation, compilation and execution of an adapter consumed only 1ms, this single task would consume more than 2½ days. To make things worse, the initial (name-and signature-based) search usually returns a large set of candidate components many of which require adaptation.

Obviously, this task is an enormous challenge even for modern computer systems and it is not possible to expect users to wait for the results such a long time. In the above case it is likely that developers will complete the implementation of the desired functionality well before the test-driven search results are available. Hence, to speed up the process we need a twofold strategy in which we: 1) reduce the number of considered candidates, and 2) distribute the load to a scalable number of clients as discussed above. Although the distribution to a virtually unlimited number of client systems is a theoretic option, in practice this is limited due to cost, management overhead on the server side, bandwidth limitations and ecologic constraints, etc. Thus we approach the problem from the software side and investigate possible optimizations. A smarter strategy for generating the appropriate adapters in a reasonable time with less overhead is outlined in Algorithm 4.1.

In order to prevent adaptation in situations where this is inappropriate, the adaptation algorithm has to ensure that it is only executed if the cardinality of $M_p = n$ is less or equal to the cardinality of $M_r = k$. Actually, it only makes sense to perform an adaptation in case where $k \leq n$ since in the case of $k > n$ it would be necessary to map one method from the adaptee to $m \geq 2$ adapter methods. In the context of our example, this would mean that the `getIssuerName` method of the test case would have to execute both – the `vendorToString` and the `toPrettyString` – methods. Moreover, it remains unclear how the return value of

Algorithm 4.1: Test and Adaptation Algorithm

```

AIF := adapter interface based on the required interface of the query test
case;
if candidate tested successful then
| return pass;
end
if  $k > n$  then
| return tooManyMethods;
else
|  $M_r :=$  methods in AIF;
|  $M_p :=$  methods in adaptee;
|  $L :=$  new list of type-compatible methods in adapter and adaptee;
| foreach  $m_i \in M_r$  do
| | foreach  $am_j \in M_p$  do
| | | if  $sig(am_j) \equiv sig(m_i)$  then
| | | | add  $m_i \rightarrow am_j$  to L;
| | | end
| | end
| end
| end
forall the entries in L do
| create adapter;
| if adapter tested successful then
| | return adapter and flag as passed;
| end
| end
end
return noMatch;

```

the adapter method should be obtained from those of the two adaptee methods. Obviously this case does not make sense in an automated environment and in such a case no adaptation is performed².

Hence, first and foremost the algorithm checks if the adapter's method count is less or equal to the adaptee's method count. Since we defined the mapping between adapter and adaptee as a bijection, the cardinality of the set of methods in the adapter and adaptee have to be equal. If the adaptee contains more

² Mathematics support this finding: ${}^5P_6 = \frac{5!}{(5-6)!} = \frac{5!}{(-1)!}$ has no solution, since the factorial is defined for non-negative integers only.

methods than the adapter, the resulting mapping will consider a subset of the adaptee's methods [Ehr73].

If there is a smaller or equal number of adapter methods to the number of adaptee methods, the algorithm iterates over all methods in the adapter and builds a list of adaptee methods that have a compatible method signature, i.e., where the types of the parameters and that of the return value of an adaptee method am_j are equivalent to those in the adapter's method m_i . Subsequently, the mappings of type-compatible methods are used to create all possible adapters. Once they have been generated, the adapters are executed and if the system finds an adapter that passes the test case, it stops further adaptation and marks the test of the candidate as successful. In the case of stateless tests, a further optimization strategy is to test the methods independently: if an adapter method has no adequate counterpart in the adaptee, the search stops and other methods remain unexamined. In the best case, the first adapter method fails to find a match. In the worst case, all adapter methods are executed before the adaptation attempt fails.

As we have shown in [JA12], this strategy effectively helps reduce the amount of unnecessarily compiled and tested classes (i.e., adapters). Our prototype implementation of the adaptation engine has been incorporated into Merobase, which is now capable of delivering the results of a test-driven search much faster than before. This is achieved by the aforementioned improvements to the adaptation engine (i.e., the upstream filtering of infeasible adaptations), as well as by the revised system architecture that facilitates load balancing and distribution of the work to several CPU cores and machines through client systems. A test-driven search for a credit card component, using a similar test to the one in Listing 3.1, adapted and tested 1.000 candidate components in less than a minute on a dual-core AMD Opteron processor with 2.6GHz and 8GB main memory.

The client returns the result of the test execution to the server, reporting whether it was necessary to adapt the candidate component or not. If adaptation was necessary, the client transfers the appropriate adapter. In the case of successful test execution / adaptation, the server stores the appropriate result and marks

the job as finished. In case the result reported by the client was negative, the server removes the candidate from the set of potentially reusable components.

If a client does not return after a specified timeout, the job is marked as open again and the fail counter is incremented by one. If an adaptation job fails more than three times, it is abandoned and considered as useless. Finally, those adapters that were successfully executed using the provided test case are returned to Merobase, which collects the list of working results and delivers them to Code Conjurer. There the results are presented to the developers who choose the desired component, suitable for the context of their development task.

4.3. Improvements to Test-Driven Search

We have carried out a number of initial experiments to determine whether the approach of automated-adaptation is feasible and if it delivers significantly more results than test-driven search without adaptation. In Table 4.2 we present some examples that illustrate the improvement of test-driven search with adaptation to test-driven search without adaptation. The first column of the Table outlines the required interface of the desired component defined by a test case, while the second column (TD) shows the number of reusable software components returned for a plain test-driven search. The third column (TDA) contains the number of results returned for test-driven searches with automated result adaptation. For this purpose, we looked at the first fifty candidates returned by Merobase.

These example searches demonstrate the benefits of enhancing test-driven reuse with automated adaptation. While the search for a Calculator, Stack and Prime class could have been achieved by a simple test-driven search without adaptation, the slightly more complex search for a Document fails completely without adaptation. With adaptation it returns 14 results which pass the user's test case. An efficient adaptation algorithm such as that implemented in Merobase therefore enables test-driven searches to return significantly more results in a given period of time.

Required Interface	TD	TDA
CreditCard(validate(long):boolean;)	0 / 50	2 / 50
Calculator(add(int,int):int; subtract(int,int):int;)	3 / 50	6 / 50
Stack(pop():Object; push(Object):void; isEmpty{}:boolean;)	7 / 50	11 / 50
Prime(isPrime(int):boolean;)	1 / 50	3 / 50
Document(Document(String,String,int); getAuthor():String; getTitle():String; getYear():int;)	0 / 50	14 / 50

Table 4.2.: Exemplary Searches With and Without Adaptation.

The execution of the examples from Table 4.2 could be performed in less than 10 seconds, making the overall process of code search by Merobase and result presentation by Code Conjurer sufficiently fast for on-demand component recommendation.

4.4. Summary

Based on the ideas presented in the literature, we have developed and presented an enhanced system for the automated adaptation of software components.

In order to develop an efficient test-reuse environment, which returns result in a reasonable time, it was necessary to enhance the existing approaches for automated adaptation. We described the enhanced process and sketched the idea of distributed adaptation on multiple adaptation clients in Figure 4.1 on page 58. Finally, the result of our investigation was a system for automated component adaptation, which is significantly faster than the initial implementation in Merobase [Hum08].

In the following chapter, we will discuss possible approaches for improving the way software search is integrated into developers' workflows. We will survey modern reuse-oriented code recommendation systems and distill the necessary properties for a recommendation system for software tests.

Contribution of this chapter

- In the context of the Merobase code search engine, we presented an improved approach for the automated adaptation of software components, which utilizes pre-adaptation filtering for efficiency improvement.
- We introduced a client-server architecture for automated adaptation that ensures high scalability for the automated adaptation of software component interfaces and supports better performance for test-driven searches.

“ The three most dangerous things in the world are a programmer with a soldering iron, a hardware type with a program patch and a user with an idea.”

The Wizardry Consulted

RICK COOK, SCI-FI AUTHOR

5

Reuse-Oriented Code Recommendation Systems

Most recommendation systems in software engineering aim to leverage data acquired by mining previous software projects and experience factories with a view to enhancing the decision making process of engineers and managers in new software projects. The decisions involved can include the creation of software implementations and tests, the development of requirements and designs, or indeed any other activity in software engineering. However, in most cases the artifacts from which the data were mined are not directly reused in new software projects, only the knowledge that was mined from them. Since we are going to develop a *reuse-oriented recommendation system* for software tests later in this thesis, this chapter presents the state of the art for reuse-oriented code recommendation (*ROCR*) systems [JHA14].

5.1. Recommendation Systems for Code Reuse

Reuse-oriented recommendation systems use data derived from the artifacts in software repositories and project archives with a view to pointing out opportunities for reusing the original artifacts themselves. A subcategory of reuse-oriented recommendation systems focuses on suggesting opportunities for reusing executable code. In a sense, ROCR systems are a special case of the more general form of recommendation engines. Their “mined” data is executable software. However, the artifacts recommended by such systems need not just be functional production code. All kinds of executable software used in the lifecycle of a project such as tests, prototypes, frameworks, environments etc. can be reused. Moreover, reuse can take many forms ranging from the direct inclusion of the artifact in the new software product to the use of the artifact to test the software product or provide oracle data to drive the testing process.

Before thinking of the creation of a recommendation system for test reuse, it is necessary to identify the requirements for such a system. Since the idea of test reuse represents a special case of software reuse in general, the same challenges and obstacles may apply to it and should be investigated in more detail. The development of an efficient test reuse and recommendation system thus needs to deal with the well-known challenges from traditional component reuse [HJA08; JHA14], which include

- the availability of a sufficiently large source of reusable artifacts (the so-called *repository problem* [FHR91; Sea99; Hum08]),
- the ability to effectively store and represent the reusable material (the so-called *representation problem* [FP94]),
- the ability to retrieve meaningful results from a repository of reusable artifacts (the so-called *retrieval problem* [MMM98; Hum08]),
- and the influence of the make-versus-reuse decision so that reuse is more cost-effective than the creation of artifacts from scratch.

A great deal of progress has been made in the mentioned areas over the last few years [HA06], and solutions to these problems have laid the foundation for the new generation of software search engines that appeared at the end of the first decade of the 2000s. However, while software search capabilities are an essential prerequisite for code recommendation tools, plain search engines cannot be regarded as code recommendation engines. They are necessary, but not sufficient. A true code recommendation system must also include a proactive agent that monitors the activities of a developer and unobtrusively issue queries in the background that can yield recommendations matching the context of the developer's code. Ideally, a code recommendation engine should also automate the process of evaluating / testing reuse candidates and of including them into the developer's new applications.

In this chapter we discuss the range of opportunities, challenges and techniques that reuse-oriented code recommendation engines can contribute to the software engineering process and survey the existing landscape of well-known academic tools. Building on the insights on software search technologies from Section 3 and the characteristics that state-of-the-art code search engines can provide today, we investigate how modern code recommendations systems are constructed. We continue by discussing the overall goal of code reuse in order to identify the requirements that a code recommendation tool should fulfill, which requirements have already been met, and which are yet to be considered. In the main part of the chapter we then discuss each of the existing research code recommendation tools in turn, describing their motivation, architecture and strengths / weaknesses. Finally, we conclude this chapter with a summary and some thoughts about where the future lies for the code recommendation technology.

5.2. Software Reuse Process

In the literature, there are numerous publications dealing with software reuse, its foundations and possible improvements. Almeida et al. [Alm+04], for instance, defined a comprehensive framework for software reuse, which cleanly described its key ingredients. Besides the need for a repository and search infrastructure,

they describe a generic software reuse process and various best practices for effective software reuse. The main obstacle to software reuse today is no longer the lack of components to reuse or the ability to retrieve them efficiently. The main obstacle is the balance between the effort needed to evaluate and incorporate components into new applications and the likely benefit (including the risk that a reuse candidate will turn out not to be suitable). This is where code recommendation tools come in. Their role is to non-intrusively and reliably find quality reusable code artifacts and to help developers integrate them into their systems with minimal effort.

Based on the lessons learned from the typical code search use cases described in the previous chapter, we can identify the most important characteristics that reuse-oriented code recommendation systems should provide and distinguish them from traditional code search engines. One of the major problems with code search engines is that developers have to leave their normal working environment to issue searches, which interrupts their development workflow. Moreover, since queries have to be executed in a different application (i.e., the web-browser) to the one where the system is developed (i.e., the developer's IDE) there is also the problem of media change.

Without access to the immediate context of the code under development, it is a demanding task for a developer to formulate queries that define his/her goal and find reusable assets that fit into the new application. In addition, developers have to fully understand how the search engines work to be able to formulate adequate queries that will deliver precise results. And, last but not least, developers have to invest a significant amount of effort to manually evaluate and integrate reusable assets into their new applications. To try out any of the recommendations, they have to switch between at least two windows, and may even lose track of their original work and ideas during the search process.

All of the aforementioned issues related to code reuse need to be reflected within a reuse-oriented code recommendation system, which ideally supports the full automation of the reuse process as well as the responses to developers' inputs.

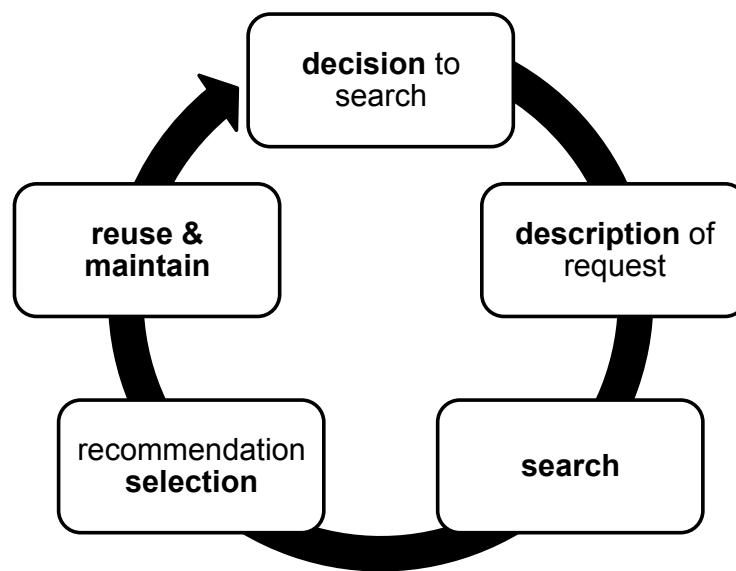


Figure 5.1.: Overview of the Micro-Process of Software Reuse [JHA14].

A simplified visualization of the process of software reuse is sketched in Figure 5.1. The process itself is generic and applies to manual as well as tool-supported software search and reuse. It comprises five major steps, each of which will be elaborated in more detail in the following description:

decision During a software project it is advisable to reuse existing software assets in order to save effort. As we already know from Section 3.1, there is a wide variety of potentially reusable artifacts and this list will be enhanced in Section 5.4. Hence, if the general decision for reuse has been made, it is necessary to decide in particular what kind of assets can actually be reused. Based on these general decisions, software engineers can decide during the development lifecycle to search for reusable existing artifacts.

description Once there was the decision to look for reusable software artifacts, an abstract description of *what* should be reused needs to be created. This specification should ideally comprise all required information that is necessary to find useful reusable assets.

search The description serves as the query to a search engine. Sophisticated algorithms should be able to automatically refine and adapt queries in order to filter out all useless artifacts and ensure that all useful ones are included.

This is almost impossible without tool support, as it would consume a lot of time to create a query, inspect the results, refine and re-issue the query ... This cycle may have to be repeated several times and is obviously not very efficient when done manually.

selection From the “raw” set of search results, the developer needs to choose whether any of the results are useful and if there are any candidates that fulfill the given criteria. If there is more than one, the developer has to select the best match which can be a very tedious task since it may involve the trial use of a large number of possible candidates. If this is carried out manually, it involves the copying of the code from the search engine, looking for necessary dependencies, possibly adapting the provided interface of a reused class and finally trying it out. This must be performed for every candidate in order to find the most suitable candidate.

reuse and maintain Once a candidate has been selected for reuse and integrated into the developer’s system, the micro-process of code reuse is completed. Nevertheless, the reused candidates are now part of the developer’s project development lifecycle and should be subject to all the same policies and processes as the other parts of the system like testing and maintenance.

Although the micro-process of reuse is complete, Figure 5.1 shows that reuse should not be a one-off event, in accordance with the ideas expressed in the literature (e.g., Krueger’s seminal work on software reuse [Kru92]). It should rather be continuously applied throughout the project development.

The following subsection provides an overview of state-of-the art recommendation systems for reusable code. In order to develop a reuse-oriented recommendation system for software tests, we will review them in the context of the aforementioned micro-process and identify their most outstanding characteristics. This will help us identify a general set of requirements for reuse-oriented code and test recommendation systems in software engineering.

5.3. State of the Art Systems

To provide a more detailed insight into the state-of-the-art in code recommendation we give an overview of a set of prominent research tools that emerged over the last two decades. Based on this review, we will give a general overview of the characteristics that need to be fulfilled by modern *Reuse-Oriented Code Recommendation Systems*.

In addition to Figure 3.2, which shows the scientific milestones of the last twenty years in code search and recommendation, Table 5.1 presents a chronological overview on ROCR systems, their main author, recommendation type and the year of their first or most influential publication. In the remainder of this section, we will briefly survey the listed tools and identify the most common and important properties of past recommendation systems. Based on these insights, we will identify and distill the most essential requirements of reuse-oriented code recommendation systems (cf. Section 5.5).

Name	Author ^a	Recommendation Type	Year ^b
Code Finder [FHR91]	HENNINGER	Software Objects	1991
CodeBroker [Ye01; YF02]	YE	Reusable Components	2001
Strathcona [HM05]	HOLMES	Source Code Examples	2005
Code Genie [Lem+07]	LEMOS	Reusable Components	2007
PARSEWeb [TX07]	THUMMALAPENTA	Invocation Sequences	2007
Code Conjurer [Jan07; HJA08]	JANJIC	Reusable Components	2007

^a Naming the main author of the particular project according to available publications.

^b Year of first recognized publication.

Table 5.1.: Code-Based Recommendation Systems.

5.3.1. Code Finder

Code Finder [FHR91; Hen93] was the first widely known interactive code recommendation tool available to developers. It was one of two influential products

from Gerhard Fischer's group at Boulder University (USA) along with CodeBroker which will be introduced in the subsequent subsection. Created as part of the PhD of Scott Henninger, Code Finder was presented in a research paper at the International Conference on Software Engineering in 1991, which elaborated on the whole chain of requirements to create an efficient code reuse system.

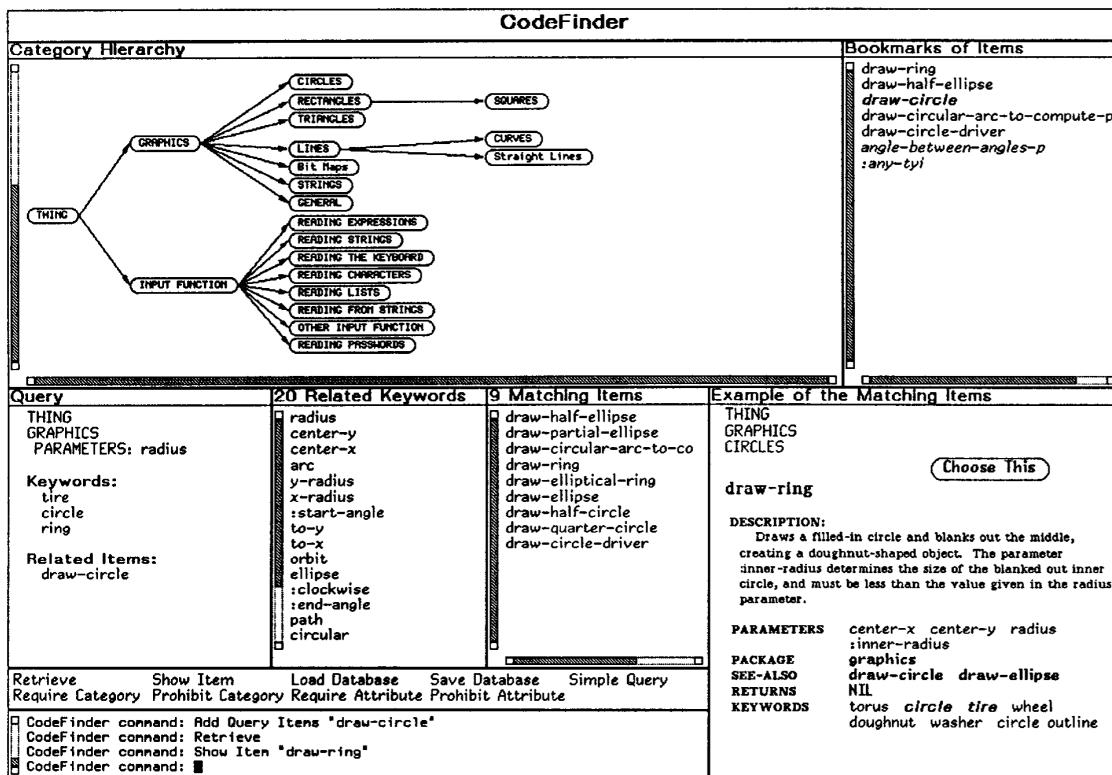


Figure 5.2.: The user interface of CodeFinder showing matching items to a user's query who wants to draw a circle. Taken from [FHR91].

Starting from the problems involved with the creation of a well structured repository, which is essential for the performance of a code reuse and recommendation system, Fischer et al. introduce their strategy for building the “backbone” for Code Finder and define the requirements of a user who wants to find something to reuse. At that time, in the early 1990s, none of the modern database and information retrieval system were available. Hence, the authors of Code Finder had to essentially create the underlying index manually. While this was possible

on a small set of reusable artifacts, it is impractical for today's large scale search engines.

Nevertheless, Code Finder represented an important milestone in the area of software search and reuse as it brought back attention to this field of research. As we have seen in the short overview depicted in Figure 3.2, during the following quarter of a century software search and reuse gained a lot of attention. Moreover, a lot of fruitful work has evolved from the ideas of Fischer and Henninger.

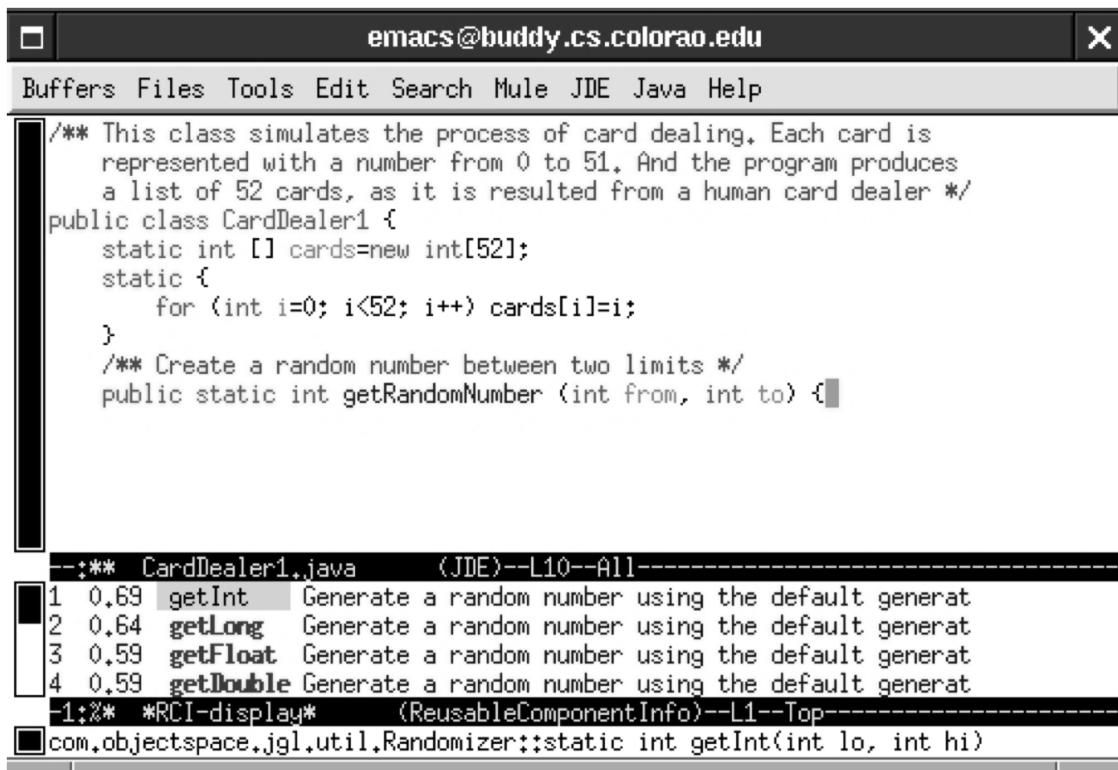
5.3.2. CodeBroker

Although the idea of reusing knowledge stored in existing components is not new, Yunwen Ye's *CodeBroker* was one of the first tools to explore this idea in the form of a proactive invocation service tightly integrated into the well-known Emacs editor [Ye01; YF02]. While developers work on their source code, CodeBroker offered coding suggestions based on information garnered from similar components in the repository. Ye identified two fundamentally distinct ways of getting this information from the repository:

- the classic *pull* or *reactive* approach, in which a user actively browses or searches for information, and
- the *push* or *proactive* approach, in which a tool monitors the user's activities and offers information it considers useful in a specific context.

CodeBroker basically has three components: *Listener* – a continuously running background agent which monitors the developer's input and automatically extracts queries from doc comments and signatures. To illustrate the functionality of the system, we consider a developer who wants to implement a card deck class for a game and writes the partial implementation shown in Figure 5.3.

Once the developer has written the declaration of the `getRandomNumber` method, Listener will extract an appropriate query from the source code. The retrieval process itself is carried out by the second part of the CodeBroker architecture – *Fetcher* – which queries the CodeBroker repository for matching components. Fetcher thereby makes use of the so-called Okapi technique [Wal+98] and



The screenshot shows a terminal window titled 'emacs@buddy.cs.colorao.edu' with a menu bar including 'Buffers', 'Files', 'Tools', 'Edit', 'Search', 'Mule', 'JDE', 'Java', and 'Help'. Below the menu is a Java source code editor containing:

```
/** This class simulates the process of card dealing. Each card is
 * represented with a number from 0 to 51. And the program produces
 * a list of 52 cards, as it is resulted from a human card dealer */
public class CardDealer1 {
    static int [] cards=new int[52];
    static {
        for (int i=0; i<52; i++) cards[i]=i;
    }
    /** Create a random number between two limits */
    public static int getRandomNumber (int from, int to) {
```

Below the code editor, a message indicates the file is being indexed:

```
--::** CardDealer1.java      (JDE)--L10--All-----
```

Following this, a list of method signatures is shown, each with a confidence score (e.g., 0.69, 0.64, 0.59, 0.59) and a brief description:

```
1 0.69 getInt  Generate a random number using the default generat
2 0.64 getLong Generate a random number using the default generat
3 0.59 getFloat Generate a random number using the default generat
4 0.59 getDouble Generate a random number using the default generat
:1:%% *RCI-display*      (ReusableComponentInfo)--L1--Top-----
```

At the bottom, the full path of the method is listed:

```
com.objectspace.jgl.util.Randomizer::static int getInt(int lo, int hi)
```

Figure 5.3.: CodeBroker's Presenter [YF02].

Latent-Semantic Indexing [LFL98] to compute the concept similarity value of each component in the repository to the query. With the standard settings, Fetcher returns the top 20 components identified as relevant. Subsequently, the *Presenter* shows the retrieved components within Emacs in the *RCI-display*. This is shown in the view located below the source code editor, which is depicted on the screenshot in Figure 5.3.

The *Presenter* not only shows results matching the tight context of the implemented class or method, but takes the larger context of the developer's application into consideration. This is necessary since programmers often do not comment their code very well. Therefore CodeBroker creates a *discourse model* and captures the developer's context. To be efficient, however, this discourse model needs further action from the developer to specify the task at hand more precisely. In addition to the programming context (i.e. the project / system under development), Ye also introduced a model of the user's knowledge of the reuse repository in CodeBroker.

Basically, in the context of his work on the CodeBroker system, Ye has identified four levels of knowledge about reusable software components:

1. Well-known components – these components are well-known to the developer and may be regularly reused. CodeBroker does not regard them as relevant recommendation results since they may repress components of which the user is not aware. Ye calls these components *reuse-by-memory* components.
2. Vaguely-known components – these components are reused rather seldom by the developer, who has a vague recall of them and has maybe used them from time-to-time but has not memorized them very well. Further investigation or approval is necessary before the developer uses such a component again.
3. Anticipated components – the user may have a certain belief about the component repository. The third level addresses this fact and incorporates the user's expectations about the repository. This may even mean that such components do not exist. At least they are hard to access due to the lack of concrete knowledge about them.
4. Unknown components – these components are totally unknown to the developer. Without a search engine and an assisting tool, it is nigh on impossible for the developer to find and reuse them without a lot of effort.

By incorporating this model into the logic of CodeBroker, the system is able to refine the result list in to include components about which the developer has rather vague or no knowledge. In this context, the recommendations are of a higher value to the developer than a list of results which are all well-known. In the latter case, the value added by the system would be rather low and would have nevertheless required inspection effort from the developer. Although CodeBroker was a very sophisticated tool when it was introduced, its repository never grew beyond a few hundred classes [Ye01] and thus it's performance is very hard to compare to modern code recommendation systems. Nevertheless, the lessons learned from this system inspired almost all developers in their creation of modern code recommendation systems: Google Scholar lists the

ICSE publication of CodeBroker as having 190 citations (as of February 2013). Unfortunately the tool is no longer available for download.

5.3.3. Strathcona

Strathcona is an *example recommendation tool* for source code within the *Eclipse* Java IDE, which was developed by Reid Holmes at the University of British Columbia [HM05]. Instead of following the established source code recommendation approaches that existed at the time, Strathcona tool focused on another problem: the lack of documentation accompanying the wide variety of frameworks and software libraries that were rapidly becoming an essential part of application development. This resulted in developers spending large amounts of time and frustration trying to find out how to solve a given task using such a framework. It is a tedious work for developers to navigate the huge number of libraries, understand how to use the provided classes and – especially – which sequence of method calls accomplishes a certain task or delivers the desired result. Without the assistance of a tool, developers would often spend hours figuring out how to write the “right five lines” of code.

The Strathcona example recommender assists users by retrieving usage examples that are relevant to the developer’s context. Therefore it does not create new hurdles for users – like the definition of a new query language that has to be learned – but extracts all necessary information from the code of the developer. The system basically consists of two parts: a server-side implementation of the tool, which holds the example repository and selects the relevant examples for a user’s query and an Eclipse client, which is the front-end to the developer and extracts the structural context of the code under development.

In their seminal work on Strathcona, Holmes and Murphy gave different example situations in which their system may help developers avoid getting stuck due to a lack of knowledge of how to solve a programming task with a framework. One of them is very familiar to Eclipse developers who try to create an abstract syntax tree (AST) from a piece of source code: a look into the API documentation leads to the `setSource` method of the `ASTParser` class, which seems to fulfill the

desired functionality. Nevertheless, even if the developers have identified this method, they may still not be aware of the three steps needed to complete the task in hand: (a) the parser needs to be created by using a factory method, (b) the parser needs to be made aware of the source code and (c) the AST has to be created. To trigger a search for an appropriate example with Strathcona, it is sufficient that the developer inserts the following seed statement into the code editor:

Listing 5.1: Strathcona Seed Example [HM05].

```
1 private void createASTFromSource(String source) {  
2     ASTParser.setSource(source.toCharArray());  
3 }
```

From this seed, the Strathcona Eclipse client will extract the structural context of the seed and identify the class, its parents, method calls and possibly existing field declarations to form a query. This is used to apply different matching heuristics at the server to find structurally matching code in the example repository. The server looks up possible example recommendations using PostgreSQL and returns the top 10 examples to the recommender client. As part of a small evaluation, in the same publication the authors presented the code from fragment in Listing 5.2 as a result selected by developers.

Listing 5.2: Example Recommendation for the ASTParser [HM05].

```
1 private CompilationUnit parseCompilationUnit  
2 (char[] source, String unitName, IJavaProject project) {  
3     ASTParser parser = ASTParser.newParser(AST.LEVEL_2_0);  
4     parser.setSource(source);  
5     parser.setUnitName(unitName);  
6     parser.setProject(project);  
7     parser.setResolveBindings(true);  
8     return (CompilationUnit) parser.createAST(null);  
9 }
```

The developers' work is obviously not complete when they are recommended an example code snippet. They also need to inspect the suggested components and take care of the following aspects:

Responsibility The developer has the responsibility to ensure that the recommended code snippet does not induce any (possibly malicious) unwanted behavior in the system under development.

Code Understanding Developers must understand what the code that they have added to their project actually does. This involves reading additional comments and documentation of the example in hand and, as a side effect, identifying possibly superfluous parts of the code.

Cleanup Since the recommended examples come from other contexts than the developer's one, the developer usually has to clean up the code. This may involve the removal of previously identified superfluous parts, where method invocations are performed which are not necessary for the part the developer was interested in, and changing the names of objects to fit into the developer's coding guidelines.

Most recommendation tools also leave these three major tasks to developers. The following example shows the severe side effects that can arise by simple copy and pasting recommended usage patterns into a new application. Figure 5.4 shows a screenshot of the Strathcona Eclipse plug-in, which calls a stub implementation of a method that should update the Eclipse status bar. The user has identified the responsibility of the `IStatusLineManager` class for executing this task and Strathcona recommends the following piece of code to get this done:

```
getViewSite().getActionBars().getStatusLineManager().setMessage(msg);
```

If users do not inspect this recommendation in more detail, they will not be aware of a fact that is unveiled by a look into the API documentation of the `getViewSite` call: this method may return a `null` value, which means that the above chain of invocations would lead to a `NullPointerException` at runtime and consequent system crashes. To avoid this, the user has to manually extend

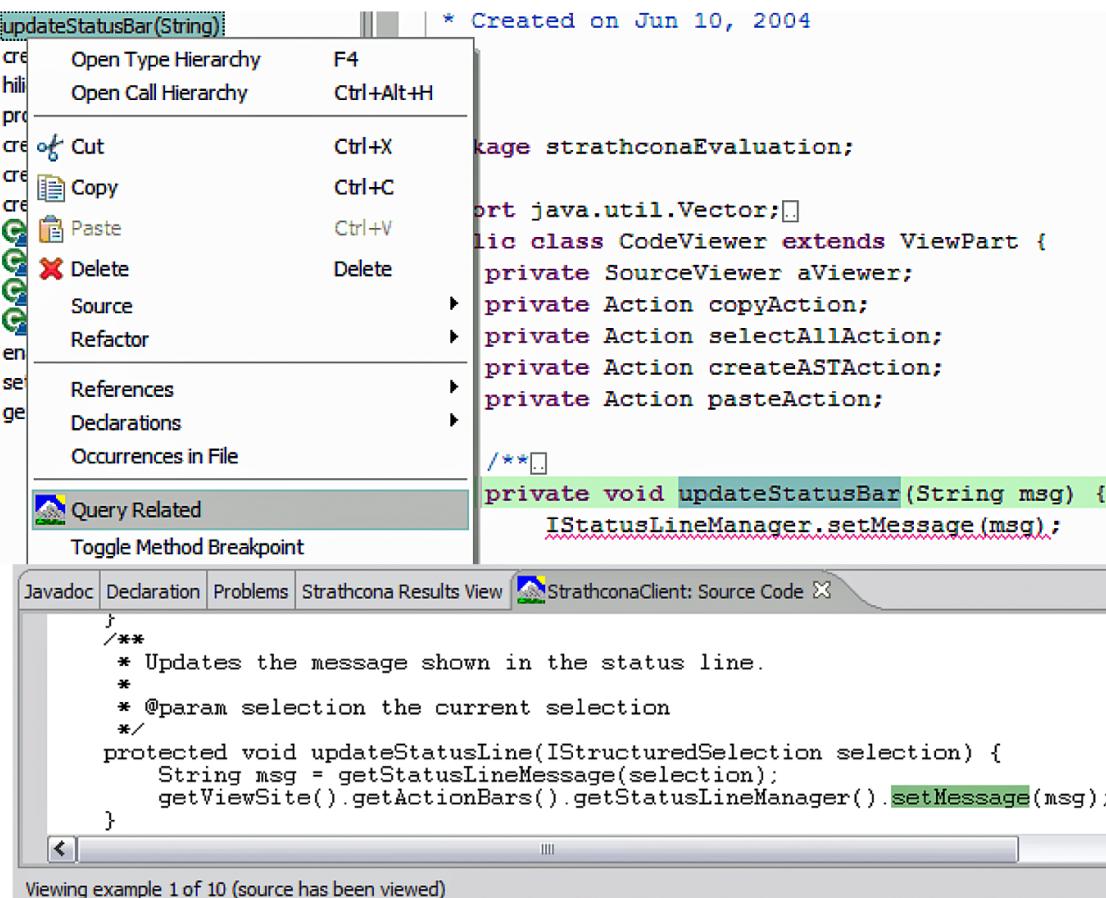


Figure 5.4.: Strathcona plug-in for Eclipse [HM05].

the code with an if-statement that ensures that none of the calls returns null. Although Strathcona was a major milestone in research on example-oriented code recommendation, it is unfortunately no longer available via its project's website. Thus this section solely relies on the examples given by the authors in the cited publications.

5.3.4. Code Genie

CodeGenie, developed at the University of California, Irvine, is a recommendation tool for reusable source code that followed the idea of testing the behavior of reusable artifacts. It allows developers to leverage the paradigm of *Test-Driven Development* (TDD) by first writing test cases before production code. Instead of implementing the desired functionality manually, the CodeGenie system enables

the developer to search for reusable assets that match the behavior specified by the test cases. The results are woven into the developer's project and the tests executed in this context.

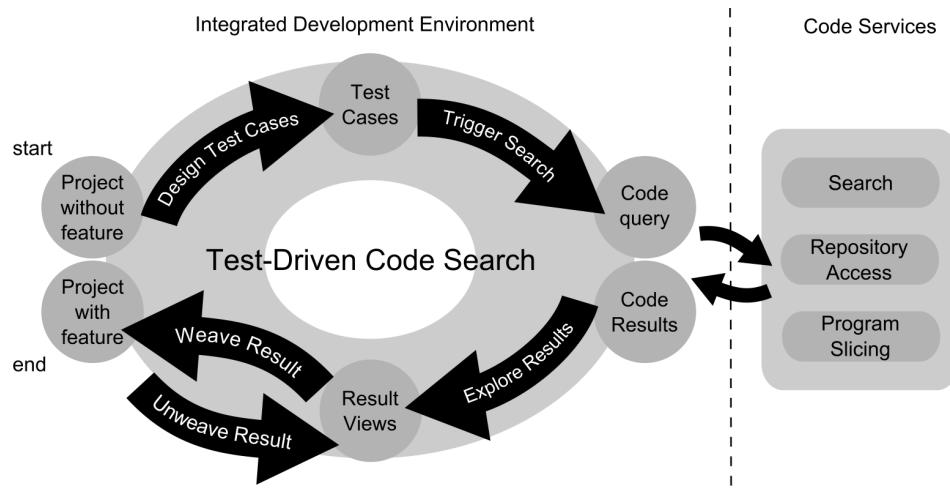


Figure 5.5.: CodeGenie Test-Driven Search Process [Lem+07].

CodeGenie relies on Sourcerer – an internet-scale search infrastructure for source code developed by the same group [Baj+06]. It uses the search engine for the retrieval of reuse candidates. Figure 5.5 depicts the process by which CodeGenie helps developers incorporate a new feature into their application:

1. A developer's project should be enhanced with a new feature.
2. The feature is described with a test case without implementing the feature itself.
3. Based on the test case a search for reusable code is triggered with CodeGenie. The plug-in issues a search via the Sourcerer search engine which returns possibly reusable artifacts.
4. On the client side the results are explored by the user and examined by weaving and local testing.
5. If a reuse candidate provides the desired feature and is recognized as being "fit-for-purpose", the new feature is incorporated into the project.

This simplified process description is, of course, based on the assumption that the desired feature can be found as a reusable artifact. In the case that Sourcerer and CodeGenie cannot find a reusable artifact, the developer will have to implement the feature manually.

Listing 5.3: Partial JUnit test for a number converter class [Laz+09].

```

1 public class RomanTest extends TestCase {
2     public void testRoman1 () {
3         assertEquals("I", Util.roman(1));
4     }
5     ...
6     public void testRoman6 () {
7         assertEquals("M", Util.roman(1000));
8     }
9 }
```

To understand the idea of *Test-Driven Searches (TDS)* in the context of CodeGenie, we take a small example from the literature¹. The screenshot in Figure 5.6 shows the CodeGenie Eclipse Plug-In returning possible results to the test case partially depicted in Listing 5.3. The RomanTest test case is the basis for CodeGenie's search and recommendation engine. The tool parses the test case that requires the following interface:



The information about the tested interface is then used to issue a search request to Sourcerer, which will return a set of ranked candidates. These can be inspected by the user and, as shown in the screenshot, tested by weaving them into the developer's project and executing the above test on the woven code slice.

¹ CodeGenie and Sourcerer are no longer available under the published URLs.

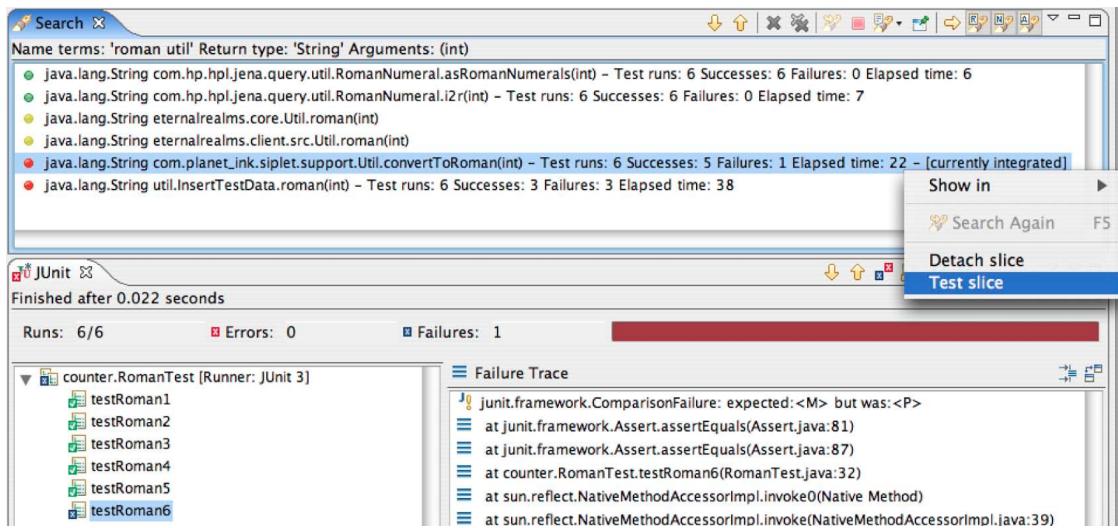


Figure 5.6.: Screenshot of CodeGenie for a Number Converter [Laz+09].

5.3.5. PARSEWeb

Strathcona was an early implementation of a code recommender that helps developers understand the usage of software libraries. It inspired many other projects like *PARSEWeb* [TX07] or, later, the *Code Recommenders Project*. Unlike the Strathcona recommender, which focuses on usage examples for concrete calls, PARSEWeb is a tool that is designed to support developers in using an unfamiliar API by specifying an object conversion task ($\alpha \rightarrow \gamma$).

To better understand the functionality of PARSEWeb, we can, for instance, refer back to the task in the Strathcona subsection. There the goal of the developer was to create an AST from a String containing source code: instead of specifying a pseudo-API call to the *ASTParser*, in PARSEWeb the developer needs to specify the *String* as the source object and the *CompilationUnit* as the target object. The tool is fully client oriented and works without any special server. To install it into Eclipse², however, the user needs to have Firefox installed with the FireBug plug-in, set a working path in the system and copy two files to that location.

As an example, suppose a programmer faces the task of using the Eclipse JDT API to create a *CompilationUnit* (γ) object from an *ICompilationUnit* (α) ob-

² Eclipse 3.5.0 is the last version officially supported by PARSEWeb. The manual is tailored to a Windows installation.

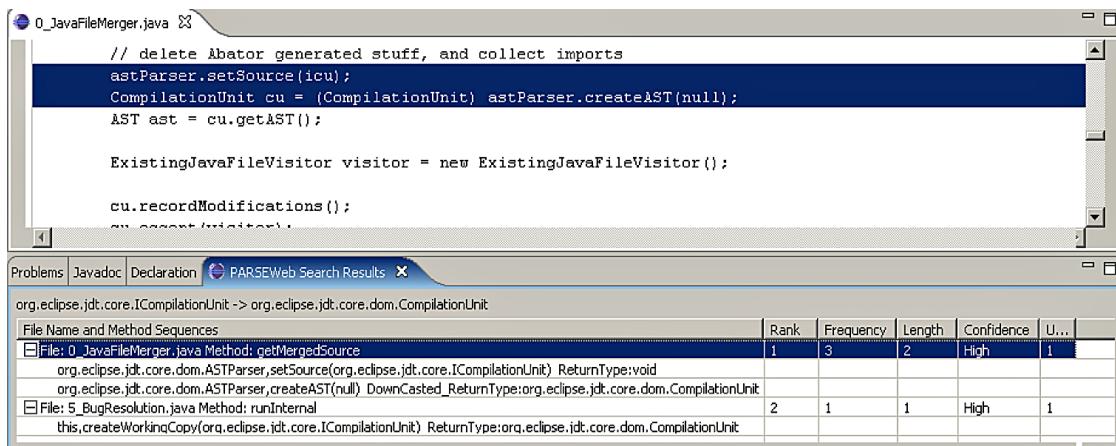


Figure 5.7.: PARSEWeb ICompilationUnit to CompilationUnit [TX07].

ject [TX07]. This problem may be very familiar to anyone who has tried to use the Java parser of the Eclipse project and usually involve significant time in API reading and unsuccessful trials before a working result is produced. In our example the user is now required to translate the given problem into a PARSEWeb query: $I\text{CompilationUnit} \rightarrow \text{CompilationUnit}$

PARSEWeb offers a query window for query formulation and after triggering a search, it interacts with a code search engine to find relevant code samples. In the primary publication for PARSEWeb, the authors found that their approach performed best with Google Codesearch [TX07]. This search engine, however, has been shut down [Goo11] and since there is no other information available, it is not clear whether the tool is still usable.

For the given query a search considers only those classes to relevant which have a usage relation with the object types specified in the query. To analyse the sources, an Abstract Syntax Tree is built as well as a Directed Acyclic Graph that is used to represent control-flow information in the artifact. This is traversed to generate a list of method calls that use the source object α as initial type and conclude with the destination type γ as end node. After finishing the process, PARSEWeb creates a method invocation result (cf. Figure 5.7).

The necessary sequence of method invocations is rather short in this case. It starts with a call to the *setSource* method of the *ASTParser* which takes as a parameter the source object α . The second invocation is a call to the *createAST*

method that returns the destination object γ after a downcast. Although the authors claimed in their publication on PARSEWeb that the tool performs better than other competing tools [TX07], the results still need inspection by the user and it is not clearly stated whether they reveal all possible / reasonable steps. Moreover, the context of the user's query is not examined.

A quick look at the JDT Documentation reveals that there are some more options which may be very interesting to a user performing the given task. For example the parser can be told which Java version should be used while performing the transformation. Therefore three more lines of code would be necessary and it is not clear how one could specify a query for them to PARSEWeb:

Listing 5.4: Relevant Invocations Missed by PARSEWeb.

```
1 Map options = JavaCore.getOptions();
2 JavaCore.setComplianceOptions(
3     JavaCore.VERSION_1_5, options);
4 parser.setCompilerOptions(options);
```

Although the tool focuses on Eclipse integration, we regard it as a recommendation system in the broad sense. The user's responsibilities mentioned in the section on Strathcona apply for this tool as well, but it hardly meets the requirements stated in Section 5.3 to be considered a full-featured recommendation tool (as mentioned, there is no context awareness for example).

Although the tool is still downloadable at

<http://research.csc.ncsu.edu/ase/projects/parseweb/>

the available version dates back to 2009 and the installation manual only considers a Windows installation on Eclipse 3.5.0. which is no longer provided by the Eclipse Foundation.

5.3.6. Code Conjurer

Code Conjurer is our own implementation of a ROCR system [Jan07; HJA08] and one of the rare systems considered in this chapter that are still available today. It incorporates the early ideas of a so-called *Software-Reuse Environment* [Gar+06]. The tool is driven by the *Merobase* component search engine to retrieve results to search queries for programming language units from various open source code repositories (such as SourceForge, the Eclipse project, JavaForge, or the Apache projects) as well as the open Web. It can be installed into Eclipse using Eclipse's built-in marketplace and requires no additional effort except setting a Merobase username and password.

Within the Eclipse IDE, Code Conjurer presents itself through a small icon showing a conjurer's hat and a *Reuse Recommendations View*, which is used to display and examine possible reuse candidates after a search. The tool supports two basic modes of operation, which themselves have different characteristics. The first mode is the so called interface-driven mode, in which Code Conjurer monitors a developer writing the implementation of a task. When the background agent is turned on, it reacts on any change to the interface to the current class under development.

To protect the intellectual property of the developer, the tool extracts only the necessary structural information of the class under development and sends it to the Merobase server for a search for reuse candidates. While the user is writing, results are retrieved and displayed in the Recommendations View. The user can click on any result and either inspect the whole component or expand the result tree and see a preview of any of the contained methods.

To reuse one or more of the artifacts, the user can insert the code using *drag and drop* from the Reuse View to the Java Editor of Eclipse or, if a class should be put in a different package than the one currently edited, the Package Explorer. Code Conjurer will automatically integrate the code into the developer's application and automatically attempt to resolve necessary imports required by the reused artifact.

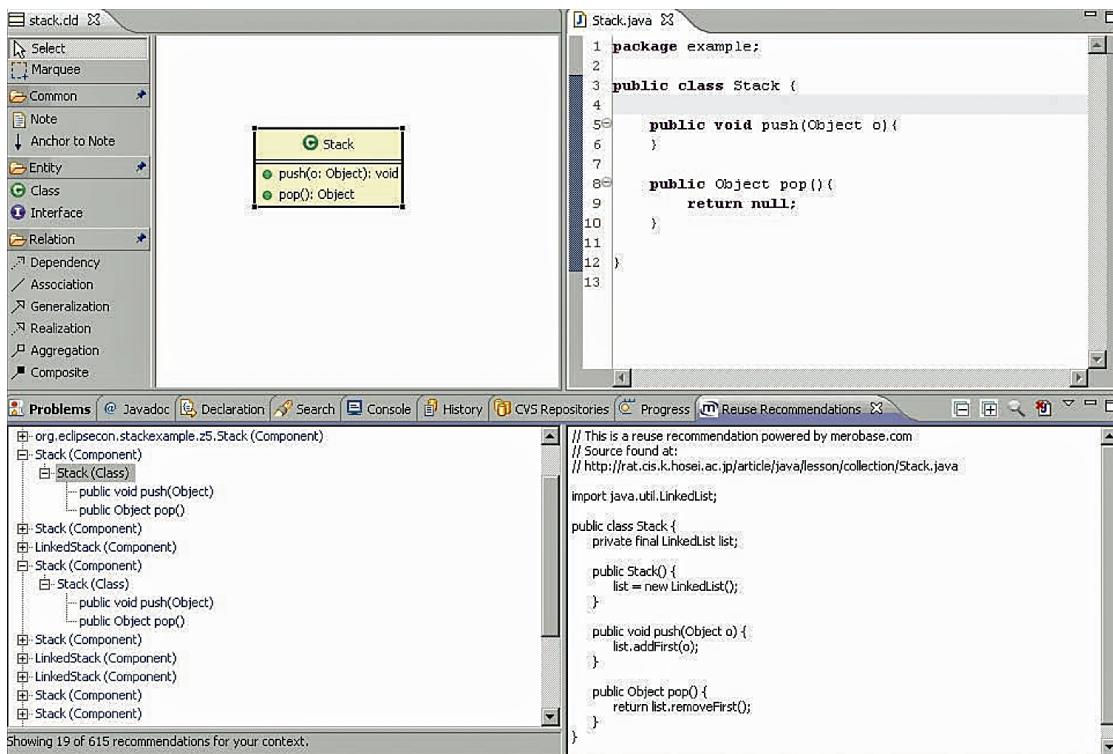


Figure 5.8.: Code Conjurer Recommendations Based on a UML [HJA08].

Consider a common scenario such as the one depicted in Figure 5.8 in which a developer defines the desired component API at the design stage. Code Conjurer can deliver implementation recommendations directly from the component's UML representation and, when set to proactive mode, it can issue new search requests each time the developer adds, removes, or changes an interface-defining part of the component. Code Conjurer then presents the retrieved components in the lower left Recommendation box. The user can explore any recommendation further by expanding its implementation in the lower right box.

Furthermore, it is also possible that the developer does not want to use one of the recommended components as is, but the information embedded in the recommended components is still useful for the development and improvement of the overall software design. Therefore, Code Conjurer not only returns a list of matching components but also analyzes them using various clustering techniques to create a characteristic design picture of the result set. Using this information, Code Conjurer can suggest the typical set of methods implemented

by components matching the partial interface defined by the user. In addition to the described capabilities, our tool is capable of assisting the developer with *quick fixes* when types within the developer's code cannot be resolved. Code Conjurer offers to search for the missing types using Merobase and to recommend them if available. Thus a developer implementing a Matrix which specifies a Vector type which is not yet implemented can look for one with the help of the recommendation tool.

Listing 5.5: JUnit Test Case Fragment for a Credit Card [JA12].

```

1 public class CreditTest extends TestCase {
2     public void setUp() {
3         cc = new CreditCard();
4     }
5     public void testVendor1() {
6         long number = cc.parseNumber("4111_1111_1111_1111");
7         long vendorId = cc.getVendorId(number);
8         assertEquals("Visa", cc.getIssuerName(vendorId));
9         assertNotSame("MasterCard",
10                     cc.getIssuerName(vendorId));
11    }
12    public void testErrorOnWrongNumber() {
13        long number = cc.parseNumber("12345678");
14        long vendorId = cc.getVendorId(number);
15        assertTrue(cc.getIssuerName(vendorId).
16                   contains("error"));
17    }
18 }
```

Nevertheless, the most prominent feature of Code Conjurer is its ability to perform *test-driven searches* in conjunction with automated adaptation of the reuse candidates [HJA08; JA12]. This approach was driven by the idea of a software search system, which is able to present only relevant results to its users. This means, that only those classes whose behavior matches the one specified in the test case (i.e., search query) are considered as relevant results and presented as recommendation to the user. To illustrate this approach, we refer to one of our publications, where we have shown the practicability of this idea using

Code Conjurer. In the given scenario, we consider a developer who wants to implement a credit card component [JA12] and who is writing a test case for such a component. The corresponding example for a credit card test (i.e., the search query) is shown in Listing 5.5.

Code Conjurer will use this test case to query Merobase for possible reuse candidates by examining the test's required interface, which describes the class under test (CUT). The set of candidates is then sent to an automated test and adaptation service which evaluates the result's fitness-for-purpose by trying to execute the provided tests on it. If the tests are not successful because of an interface mismatch, the automated adaptation engine will try to create "glue code" between the test and the candidate in order to map the calls from the test to the provided interface of the retrieved code artifact. We have discussed this process already in more detail in Section 4. After the search has finished, Code Conjurer will provide all reusable classes in the Recommendation view presented at the bottom of Figure 5.9.

The results can be inspected and reused in exactly the same way as the earlier described searches. One additional possibility, however, is the exploration of the adapter code, if it has been created. If it was necessary to adapt the reuse candidate's interface to the provided test case, Code Conjurer will deliver the additional code and when the user drags the result selected for reuse into the Package Explorer of Eclipse, the adapter will be automatically inserted along with the reuse candidate, the so-called adaptee. In this case no manual changes to the code are necessary and the developer can execute the primarily written test straight away and check that the reused component behaves correctly.

The possibility of reusing external components as "black boxes" whose implementation is unknown to the developer is a two-edged sword. It can bring great benefits but also great dangers. As the reuse of components becomes easier, the developer has a growing responsibility to be aware of the dangers and take appropriate counter measures (like code inspection). Developers must understand what they are reusing and what possible effects their use can have on their systems, especially if their use is just a "drag and drop" away.

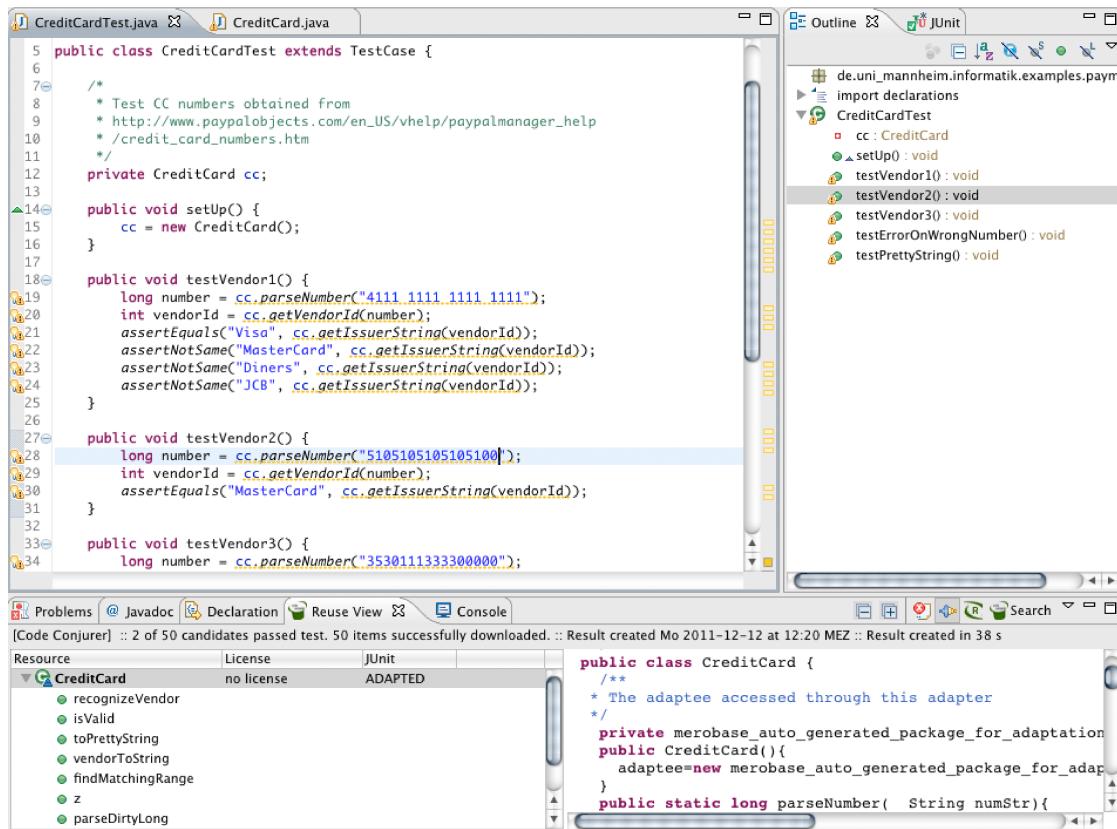


Figure 5.9.: Test-Driven Search for a Credit Card Component [JA12].

To summarize, since developers do not want to have their workflow disturbed by leaving their IDE to issue code searches (e.g., they do not want to open a web browser window and to enter a query in a search engine that requires them to manually transfer possibly matching results into their project) [Gar+06], Code Conjurer tries to integrate the reuse task seamlessly into the IDE and to recommend reuse candidates only when there is a high likelihood that they are useful to the developer.

The tool is available as open-source software hosted at SourceForge and has been continuously maintained and improved since 2007. Currently, the addition of an *ex ante* evaluation of the results based on the ideas of speculative analysis [Muş+12b] is under investigation. This measure should further improve the user experience when looking for reusable artifacts. More information about Code Conjurer can be obtained from codeconjurer.sourceforge.net.

5.4. Usage Scenarios

The term software reuse is usually associated with the integration of existing software (i.e., code) into a project under development. When using a (code) search engine, this is usually performed in a copy-and-paste approach [LM89], which is also known as *code scavenging* when contiguous blocks of source code are copied to the new system [Kru92]. The underlying goal of these techniques – which are known by different names and are subsumed under the term *pragmatic reuse* [HW07] – is to copy as much code as possible from already existing projects.

This is, however, not the only kind of reuse that is possible. There are many other forms of software reuse like *design scavenging*, where large blocks of code are reused and subject to major internal changes. This diversity in motivation for reuse leads to different varieties of reuse-oriented code recommendation systems. ROCR systems were designed to support other forms of reuse than just to copy pre-existing code. For example, some systems recommend automatically created code fragments by leveraging knowledge from pre-existing source code or other software artifacts. From our above survey on reuse-oriented code recommendation tools, we can identify different groups of scenarios for software reuse tools. Based on the above discussion, these systems can roughly be grouped into two camps – those whose goal is to provide advice to developers on how to use already identified code (e.g. frameworks and libraries) and those whose goal is to help developers find, evaluate and reuse as yet unknown code.

5.4.1. Component Reuse

The most obvious use-case for a ROCR system is to present previously written code assets to developers. These artifacts may have different levels of granularity ranging from code snippets, methods, classes up to whole subsystems and systems. A well known member of this family is Code Conjurer [HJA08] which offers developers the possibility to find reusable code artifacts from the Merobase source code repository [Jan+13]. When using this Eclipse plug-in in its pro-active

mode, developers are offered suggestions for reusable methods and classes which fit into their programming context and they can simply drag-and-drop the best match into their project. By offering the possibility of automatic dependency resolution, where classes are accompanied by those classes which they make use of (e.g., by instantiation or method invocation), Code Conjurer even offers the automated reuse of (smaller) systems, which we call components in the sense of component-based software development [Atk+08a].

5.4.2. Library Reuse

Especially within object-oriented development projects, developers constantly utilize pre-fabricated building blocks provided in the form of libraries by invoking some of their functionality. This is very convenient at first sight, since libraries form a cohesive piece of software that usually incorporates a lot of reusable objects with their dependencies. Although they can make the development of new software much easier, there are, however, numerous obstacles to their usage that every developer experiences on a regular basis.

Questions like “how is this library used”, “which objects do I need”, “how are they created” and “what sequence of calls do I have to make” arise almost every time a new framework, API or library is used. Tools like Strathcona [Hol04] or Prospector [Man+05] explicitly addressed this problem by recommending code snippets that show examples of how libraries can be used or which call sequence is necessary to transform an object from one type into another type (e.g., a *File* into an *AbstractSyntaxTree*).

5.5. Characteristics of ROCRs

From the observations made in our survey, we can now identify a few minimum requirements that have to be met by modern code recommendation tools to make code reuse more convenient.

The screenshot shows a code editor with three tabs at the top: Hello.java, HelloTest.java, and UML.diag. The Hello.java tab is active and displays the following Java code:

```

1 public class Hello {
2
3     /*
4      * Hello Recommendations Example
5      *
6      */
7
8     public void sayHello() {
9         |
10    | Complete stub
11    | Insert method
12    | Complete class
13    | System.out.println
14    | Insert JFrame
15
16
17

```

A tooltip window is open over the code editor, specifically over the line `String hello = "Hello" + "World!";` It contains five suggestions:

- Complete stub
- Insert method**
- Complete class
- System.out.println
- Insert JFrame

Figure 5.10.: IDE Auto-Complete Recommendation.

In other words, we have obtained the necessary insights for the creation of our own reuse-oriented recommendation system for software tests and identified the requirements that such a system needs to implement:

Pro-Activeness Reuse-oriented recommendation systems for code should constantly monitor the user's development progress and pro-actively decide when to trigger a search for recommendable artifacts. This should happen without any trigger by the user, who should not be disturbed by this action. Since it is the key feature of a recommendation system, this proactive behavior needs to be well designed and will play the biggest role in determining whether the recommendation system will be a success.

Context awareness An important driver for the proactive behavior of a recommendation engine is the ability to analyze context data to judge when to make recommendations. Depending on the kind of recommendation system, this may range from the immediate environment of the cursor to the source code of the whole project. Recommendation systems perform different types of assessment on context information and usually rank their results according to either fixed, automatically derived (this could be, e.g. the user's feedback on previous recommendations) or user-defined criteria.

IDE integration A Code-Reuse Environment should ensure full integration of the reuse process into the developer's IDE. This can only be achieved if a tool has access to the developer's workspace and project in order to understand the context of a class under development and, possibly, also the developer's behavior. It is also possible to filter out which classes and types are in the *reuse-by-memory* space of a developer and need not be recommended by the system. This ensures that the workflow of the user is not broken but reuse can be more easily integrated into the development life-cycle.

Autonomous Evaluation Ideally a code recommendation tool should predict the consequences of the inclusion of any of the suggested reuse candidates into the developer's project. Therefore, ideally a recommendation tool should perform Speculative Analysis [Bru+10] and autonomously apply possible recommendations in the background and evaluate their effect on the system's development state. Combined with a sophisticated ranking algorithm, this should significantly reduce the effort that developers have to put into evaluating recommendations.

Ready on-demand Developers do not want to wait for code recommendations. A recommendation system for code reuse must ensure that the reusable artifacts are available exactly when they are needed. If it takes more time for a system to find reusable artifacts and propagate them to the developer than it takes for him to implement a task himself, the reuse recommendation system will be useless.

We can imagine reuse-oriented code recommendation systems as advisers for software developers. They provide their users with an easy to use interface for sophisticated search engines and mining tools. Recommendations need to be unobtrusive and preferably appear immediately when demanded. However, they should not require high cognitive decisions or a significant amount of effort. Ideally, recommendations should also be presented in the context of their potential application. For example, reusable code can show up in the auto-complete feature of the code editor. An example for this editor integration is sketched in Figure 5.10.

5.6. Summary

These given examples illustrate that there are different forms of reuse-oriented code recommendation systems supporting different services and use cases, ranging from copy & paste reuse, library reuse and example recommendation up to the idea of this thesis to reuse the knowledge bound up in previously written software tests. Clearly, the given list of examples cannot be complete, nor can we today foresee what kinds of reuse-oriented code recommendation systems may arise in the future. Nevertheless, based on the generic definition of a recommendation system from Robillard et al. [RWZ10], it is possible to formulate the following definition of reuse-oriented code recommendation systems:

Definition 5.1. *A Reuse-Oriented Code Recommendation (ROCR) system is a tool that autonomously recommends code artifacts of any kind and size to developers in their particular development context.*

The last part of the definition concerning the development context leads to the observation that

Observation 5.1. *ROCR systems are assistant tools for developers, which are seamlessly integrated into the developers' software development process and environment.*

As we have learned earlier in this thesis, tools that are not integrated into the developers' software development environments are usually doomed to failure and many of the aforementioned systems were not able to acquire a wide user base. Due to the missing context information and constant disruption of their workflows, users still unfortunately seemed to have opted primarily for the "make" option in the traditional *make or reuse dilemma*.

The main obstacle to software reuse is no longer the lack of components to reuse or the ability to retrieve them efficiently. Many projects have shown that this is feasible with modern technology [Baj+06; HM05; HJA08; Rei09]. The main obstacle is rather the balance between the effort required to evaluate and incorporate components into new applications and the likely benefit (including

the risk that a reuse candidate will turn out to be unsuitable). This is where code recommendation tools come in. Their role is to non-intrusively and reliably find and recommend high quality code artifacts leveraging software reuse and to help developers integrate them into their systems with minimal effort.

Based on these observations we have identified a minimum set of requirements that have to be met by modern code recommendation tools to make code reuse more convenient. These “best practices” should be standard features of ROCR systems as they contribute to higher acceptance of such systems among users.

This chapter has provided a survey of the state-of-the-art in reuse-oriented code recommendation systems and the search engines that often lie behind them. The properties of reuse-oriented recommendation systems that we identified in this chapter represent a sound foundation for the following chapters, where we develop our approach for reuse-oriented test recommendation, especially a reuse-oriented test recommendation system for the Eclipse IDE (cf. Chapter 8).

Contribution of this chapter

- Survey on existing reuse-oriented code recommendation systems.
- A definition of the software reuse process.
- A set of general characteristics for code recommendation tools.

Part III.

Reuse of Software Tests

“ *Garbage-in equals garbage-out is no explanation for anything except our failure to test the system’s tolerance for bad data.”*

Software Testing Techniques
BORIS BEIZER, Software Engineer

6

Infrastructure for Test Reuse

6.1. Obtaining Reusable Test Cases

To provide an efficient system for (semi-)automated test reuse, it is necessary to build an appropriate infrastructure that allows for the analysis, indexation, storage and retrieval of existing test cases. In addition to the ever-present problems faced by the code reuse community, such as inefficient or imprecise retrieval techniques, test reuse imposes some new challenges that have to be overcome. One is the creation of an effective parsing mechanism for test cases, another is the question of retrieving suitable results.

In contrast to traditional software reuse, where usually either textual comments are made searchable [Ye01] or (parts of) the provided interface of a reusable artifact are extracted and stored [ZW95; Hum08], it is not possible to apply the same techniques directly to tests. Whereas object-oriented development is a widespread paradigm for production code, there is no similar paradigm on the

horizon for writing tests and test cases. Even worse, today there are very few limitations on the way tests have to be defined. Hence, there are uncountable different possibilities to test the same piece of functionality (cf. Section 2.2).

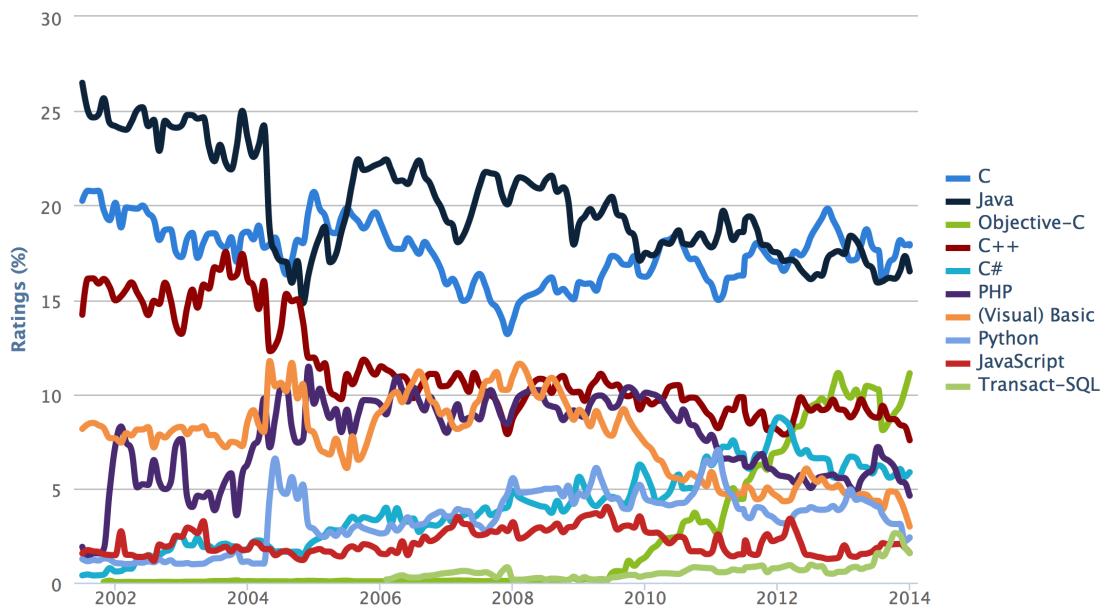


Figure 6.1.: The TIOBE Programming Community Index [TIO14]

In this chapter we will discuss the possibilities of leveraging the knowledge embedded in existing test cases and develop the necessary techniques and heuristics to extract the information they contain. For our purposes, the tests are collected via the internet from projects stored by open source hosting services. The size of the repositories that are created from these sources demonstrates that the technology developed in this thesis is applicable to internet-scale repositories, as this is common practice of the reuse approaches of the last decade [Baj+06; Hum08; Rei09]. Throughout the remainder of this thesis, we will focus on the JUnit testing framework for Java, since it is a widespread, commonly used testing framework for one of the most popular programming languages (as indicated, e.g., by the TIOBE programming community index [TIO14] or the RedMonk Programming Language Ranking [Red13]).

Although their methodologies to obtain the programming language usage statistics are questionable, these communities confirm practical experience such as the fact that Java is very popular among GitHub projects [Bar13] or that it is the main language in development of Android OS apps. After all, since the underlying ideas and methodologies developed and applied within this thesis are very general they should be applicable to other languages and testing frameworks as well.

6.1.1. Potential of Open Source Repositories

In the earlier parts of this thesis, we discussed the problems faced in software reuse and identified the fact that, today, far from having “too few” components (i.e., the *repository problem*), we now almost have “too many” available. The problem is not the shear number but the fact that reusable software is unstructured and scattered over multiple data stores around the world. The early literature in software reuse advocated centralized repositories which offered “the right combination of efficiency, accuracy, user-friendliness and generality to afford us a breakthrough in the practice of software reuse” [MMM98]. These ideas, however, were already challenged by Seacord at the end of the 1990s, who stated, that “problems with this approach include limited accessibility and scalability of the repository, exclusive control over cataloged components, oppressive bureaucracy, and poor economy of scale (few users, low per-user benefits, and high cost of repository mechanisms and operations)” [Sea99]. This statement has to be read in the context that at that time Seacord et al. published one of the first widely recognized code search engines [SHW98].

The purpose of *Agora*, as they called their system, was to provide a search engine that supports searches for components based on the description of their properties contained in their interfaces. Another argument in favor of the usage of specialized search engines over centralized directories was the fruitless investment in the creation of the UDDI Business Registry (UBR), which according to Hummel et al. did not contain a lot of usable material [HA06], and its shutdown in January 2006 affirmed these findings. Furthermore the timeline presented in

Section 3.2, where we gave an overview of the history of software search and reuse, shows that search engines have dominated the last decade of this research area, while centralized repositories neither gained public recognition nor were reported as being successful.

In the following section, we will investigate the feasibility of creating a searchable index of reusable tests from freely available code in open source repositories. In the preparative work for our search engine *SENTRE*, which supports the search for reusable software tests, we harvested projects from several open source hosting platforms, mainly from the widely used GitHub [Git14], SourceForge [Med14] and Bitbucket [Atl14] source code hosting services. While the sources obtained from GitHub and Bitbucket are relatively new (project downloads were performed in summer / fall 2013), our index additionally contains the sources from the Merobase dataset [Jan+13], which was built in 2006 and revised in 2010 with files from various sources such as SourceForge, apache.org and Google Code.

The Content of Open Source Repositories

To provide an impression of the amount of code available in open source hosting services, we provide some up-to-date statistics, which can be regarded in the context of the research of Hummel et al. [HA06; Hum08; HJA08], whose findings date back more than half a decade. GitHub is one of the fastest growing project hosting services, with an impressive growth rate. In April 2011 they published the number of 2,000,000 repositories in 1.1 million projects¹, while we counted approximately 6.5 million projects in July 2013 and the latest run of our scripts from January 2014 lists 16,143,093 projects. The rapid growth of GitHub may, however, also be related to the way in which developers have to use the service. In general, developers who want to contribute to a project need to fork the project, which results in a new project under their user account. This project fork contains the code that the developers can work on and where they contribute new features, translations, etc. If they feel that their work would be a contribution to the original project they have to send the original author a pull request. In

¹ <https://github.com/blog/841-those-are-some-big-numbers>

contrast, for example, SourceForge organizes project contributions by allowing developers to provide patches to existing projects and / or join projects.

Some facts about the major sources of code for our test search engine are presented in Table 6.1. We obtained the information either directly from the hosters website, if available, or by counting the code modules ourselves during the extraction processes (marked with a small black triangle ▲).

These numbers show that the availability of reusable artifacts is no longer an obstacle to the creation of viable repositories. Nevertheless, we will see that the success of reuse is still related to the quality of the reusable material in terms of programming quality and style. This applies in particular to the reusability of software tests.

Hoster	Projects Hosted	Forked Projects	Users
GitHub [Git14]	16,143,093▲	6,954,849▲	2,301,480▲
SourceForge [Med14]	> 324,000	n/a	3,400,000
Bitbucket [Atl14]	206,882▲	34,163▲	128,389▲

Table 6.1.: Open Source Hosters Facts.

6.2. Extracting Knowledge from Test Cases

As we have already seen in Section 2.2, knowledge extraction from JUnit test cases can be a relatively simple task, when the test cases are self-contained, when the CUT ideally does not rely on further dependencies and when the developer of a test adheres to the coding guidelines defined by the framework. However, our investigations show that this is the exception rather than the rule, and it is very likely that the tests analyzed during the creation of our search engine will be much more complex and ambiguous. Therefore, this section develops an overall structure of a generic meta-model for software tests that is consistent to the previously defined testing terminology. Hence, the primary goal of our work is to

identify and describe the information necessary for building a general purpose search engine for tests, whilst omitting concrete language- or framework-binding, which is not relevant for this particular task. Subsequently, we describe how this meta-model can be used in a derived model of the JUnit framework, to give a concrete usage example.

6.2.1. A Meta-Model for Software Tests

From the preceding chapters and sections, we have learned that the main goal of a test reuse system is to support efficient and effective searches for reusable assets. However, this requirement is very vague in terms of *what* should be the focus of a search. A valuable test reuse system should be able to cover a variety of search scenarios in order to assist its users in different situations. A system for reuse-assisted software testing should be able to process similar searches to those known from the “classic” reuse of production code. This means that in addition to the well-known keyword- or name-based searches, such a system also needs to support more sophisticated queries. We have already mentioned our work where we have shown that using interface descriptions as search queries, for example, leads to better results than the aforementioned approaches [HJA07] and it is no surprise that well known code search engines like *Merobase*, *Sourcerer* or *S⁶* support this form of query formulation.

The Test Model

In order to create a search engine for reusable tests, we need a data model that is capable of capturing all those facets of a software test, which are necessary to reconstruct the test at reuse time. In this section we are going to develop a meta-model for tests that can be instantiated for different test frameworks. In subsequent subsections we will give examples of its use for the extraction of information from tests using the JUnit framework. In order to keep the model manageable, we use the idea of component decomposition, i.e., we will decompose the software test model into smaller artifacts that can be addressed

separately. The particular artifact of interest in each UML class diagram will be annotated with the stereotype «*subject*».

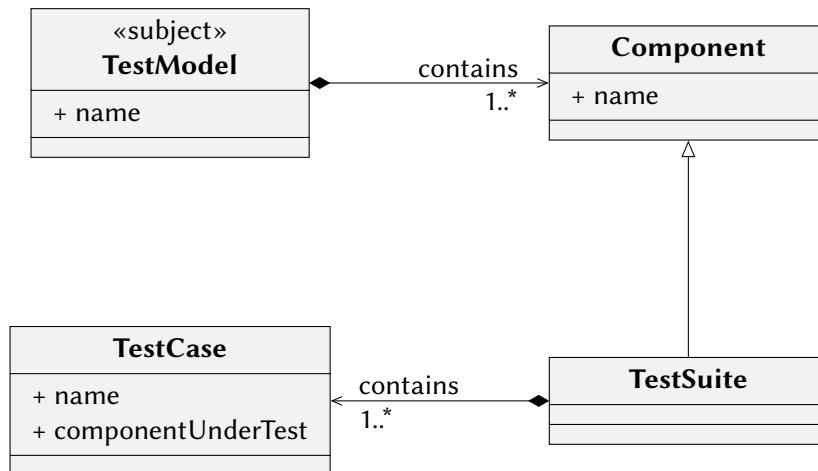


Figure 6.2.: The Test Model Contains Test Suites and Required Components.

The root element of our meta-model is the *TestModel*, which can be regarded as some kind of container that incorporates components and especially test suites. The diagram in Figure 6.2 depicts the structures related to a *TestModel*: a *Component* has a name attribute and a *TestSuite* is a specialization of a *Component* which inherits its characteristics. Later we discuss the structure of *Component* in more detail, as well as the decomposition of a *TestSuite* into *TestCases* and their related artifacts.

Naturally, the execution of a test suite involves the satisfaction of the test suite's required interface, which represents its dependencies to other components and needs to be captured in order to build a meaningful database of reusable tests. When we instantiate the model to JUnit, the *TestSuite* element corresponds to different concepts: recalling definition 2.5 – which defines a test case as a container for tests – it is clear that JUnit 3 is too coarse-grained when defining a whole test class as a test case². JUnit 4's introduction of the @Test annotation, which precedes every method containing tests, improved the situation since it

² e.g., public class MyTest extends TestCase { ... }.

maps the definition of a test case to the method level. Since it therefore allows a component to contain its own test cases expressed in methods, the concept of a test case in our model corresponds to JUnit's test method. On the other hand, the test class and test suite in JUnit are both an instance of the *TestSuite*. We have already given a comparison of the testing terminology and the naming conventions of JUnit on page 21 in Chapter 2.1.

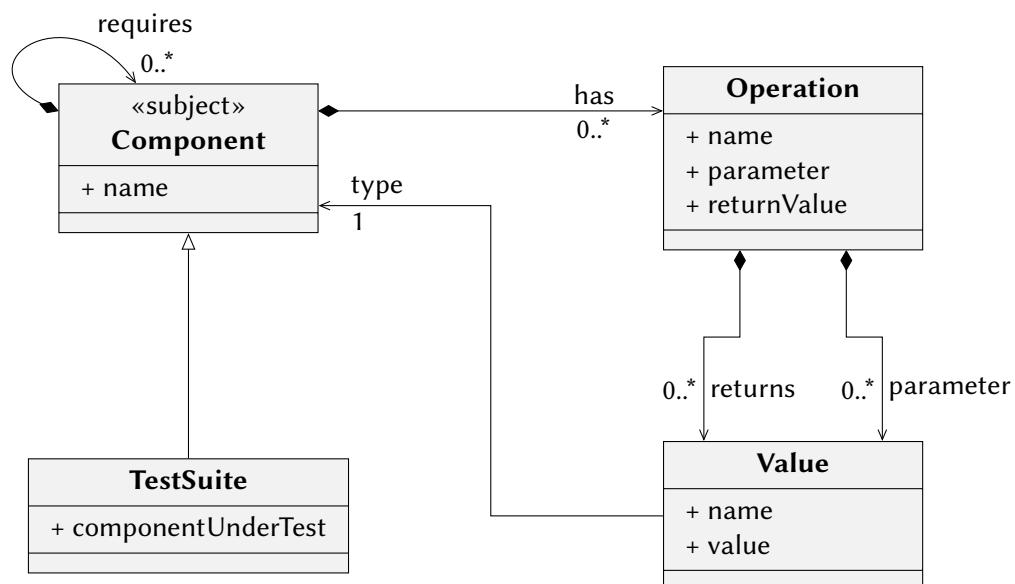


Figure 6.3.: The Decomposition of a Component.

Component Relations

The diagram in Figure 6.3 shows the structure of a *Component*. Inspired by Szyperski's discussion about the nature of components [Szy02] and driven by the goal to be as generic as possible, we define a composition relationship between *Component* and *Operation*. Actually, we are not strictly following his component model, since we are dealing with object-oriented tests that may be written, e.g., in Java and JUnit and therefore we need to be able to represent them adequately. In the Java programming language, attributes are used to represent

important information, given the language's principle that (almost) everything is an object and objects have an observable state exposed through their publicly visible instance attributes [Boo+98].

Although there is no specific definition of component attributes in our model that would allow for an observable state, our meta-model is still capable of handling Java / JUnit or other object-oriented languages and testing frameworks. Instead of directly accessing attributes, we assume them to be private and exploit the design principle of information hiding introduced by Booch et al. in their famous book on object-oriented analysis and design [Boo+98]. We therefore assume that so-called *getter* and *setter* operations are used to access private attributes in Java classes. Beside the general description of dependencies between components, our data model must be capable of handling dependency information about the components required for the execution of tests. Therefore, the conceptual class *TestSuite* inherits the self-association from *Component*, which captures information about all required components and operations that are used by the tests contained in the test suite. This association represents the so-called *required interface* of any component captured by the model. Hence, it is used to represent the contextual dependencies of a component. Naturally, a test case makes use of the *requires* association at least once – namely, to describe the class under test.

Referring to the example in Listing 2.2 (pp. 23–25), the required interface of the distance calculator's JUnit test case can be described with the help of the *Merobase Query Language* (cf. MQL on page 44) as

```
Euclid(dist(double):double);
```

which corresponds to the *provided interface* of the distance calculator class from Listing 2.1.

The information about the required interface also enables us to represent existing inheritance relations, for example, when a JUnit test case is defined as a child class of another test case. It is not unusual that developers of test cases define their own JUnit test cases that extend the *TestCase* class, in order to follow their

own naming conventions and for maintenance reasons. Therefore, an instance of our data model is able to capture test class definitions using multi-level inheritance relationships, as shown in the following:

```
public class MyTest extends MyOldTest {...}
```

Here, the `MyTest` class inherits from `MyOldTest`, which itself extends the `TestCase` class provided by JUnit 3 framework.

To fully describe the structure of a required interface, we refer again to Szyper-ski's concept of a software component [Szy02] and the findings of Hummel et al. [HJA07; Hum08]. The diagram depicted in Figure 6.3 describes the required interface of a *Component* by its name and the information exposed by its contained operations, i.e., their provided interface. The provided interface of an operation itself is composed of the operation's name, its return type and parameters. A cardinality of 0 on one of the associations between an operation and the value class corresponds to a void return type or an empty input parameter list, respectively.

Decomposition of the TestSuite class

One of the most important pieces of information for a test search engine is the association of *TestSuite* to the component under test. Since it is our goal to provide users the possibility to search for reusable tests using the interface declaration of their own components, it is necessary that the search system is able to compare their provided interface with the interface required by the potentially reusable tests.

Although there are obviously many more properties of software components, many of them are out of the scope of this thesis and are not required in this context. Thus, they do not need to be stored in a data model for software tests. A component's source code, for instance, can be obtained directly from the file system when necessary and does not need to be redundantly saved in a database. The information about a test suite's required components is, however, very useful.

Its main use is to enable users to search for reusable tests using the interface of the component under test and to enable our test parser to resolve return types of method invocations, which are declared in their corresponding classes and cannot be derived from an invocation unambiguously. In the following, we will investigate the concept of test suites and test cases in more detail and describe how the information contained within test suites is captured.

The structural view depicted in Figure 6.4 shows the containment hierarchy in our model starting from the *TestSuite*. It also helps to explain the structural decomposition of a *Test*, which is marked with the aforementioned *subject* stereotype. Since *TestSuite* is a specialization of *Component*, it inherits its concept of operations, which are preferably utilized to set up and prepare the test environment for the ensuing test cases and tests. An operation may also serve, however, as a container for tests. An instantiation of this meta-model for JUnit therefore maps the *TestSuite* class to the JUnit *TestCase*, because JUnit *TestCases* are Java classes which encapsulate tests (assertions) in test methods which correspond to the *Operation* class contained in *Component*. Although JUnit documentation demands developers to implement a method for each test³, this scenario tends to be rather the exception than the rule. Hence, we have to create a corresponding mapping in our meta-model.

Now we can take a closer look at the decomposition of *Test*. It contains three classes, which represent the main concepts associated with tests: 1. an invocation of the class under test, 2. the definition of an expected result, and 3. an unspecified number of statements. Naturally, a test only makes sense if there is something to be tested. Hence, there has to be (at least) one invocation of the class under test. Figure 6.4 annotates the association between a test and a corresponding invocation of the *CUT* with a multiplicity of 1 to emphasize and reflect that a test is intended to inspect the behavior of one invocation of the component under test for particularly defined input values (i.e., the *test case values*). This apparent constraint, however, does not stop the model from capturing more complex tests. Therefore, a test is also associated with the *Statement* class, which

³ see, e.g., <http://junit.sourceforge.net/junit3.8.1/javadoc/junit/framework/TestCase.html>: “For each test implement a method [...]”

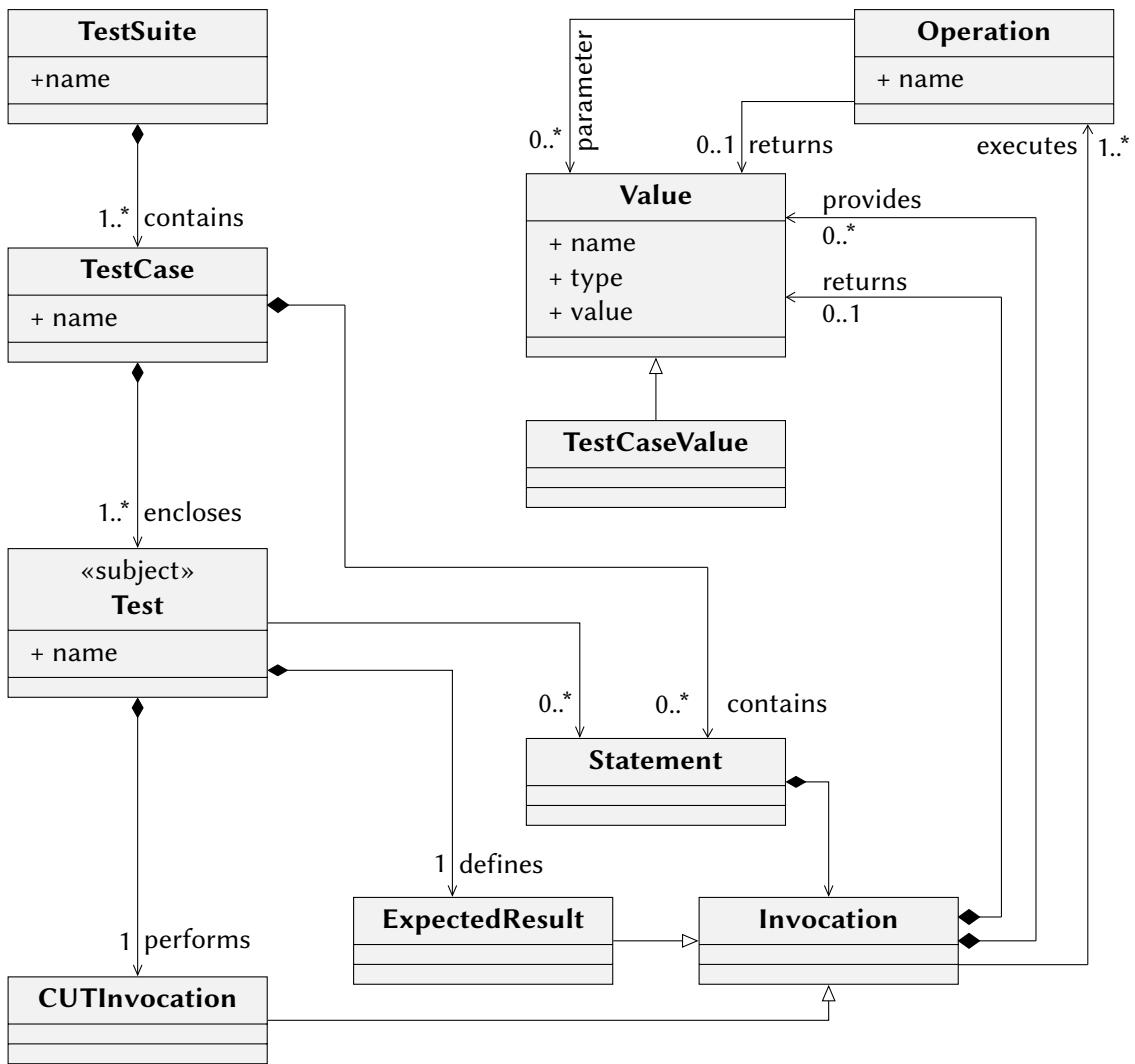


Figure 6.4.: The Decomposition of a Test Suite.

can be instantiated to any type of statement like assignment- or if-statements. The introduction of this concept allows more sophisticated tests to be captured such as, for example, conditional tests or tests depending on the state of the system under test.

Although the execution of statements may be a requirement of a test, it is also possible that a test case contains only one invocation of the class under test (i.e., one test) in its code. Nevertheless, at runtime this test code may result in a large number of tests being executed, if the test statement is for instance contained in a loop. Thus, we have decided to contain the *Statement* class within the *TestCase*

instead of within the *Test*. In addition to that, we define a link between *Test* and *Statement* in order to capture the statements relevant to the context of the *CUTInvocation*. The setup- and tear-down-operations of a test case are covered by the inheritance relation between the *TestSuite* and the *Component* class, i.e., they are described using the *Operation* class of a Component.

A test cannot be complete without the declaration of an expected value, which has to be compared to the actual result created by the CUT's invocation using test case values. The structural view of our model depicted in Figure 6.4 defines an inheritance relation between the *Invocation* and the node representing the expected value. While such a relationship is quite natural for the invocation of the CUT, it is not so obvious why we declare this also in the case of the expected value. To explain this we refer to the example in Listing 6.1, which originates from a JUnit test case for a distance calculator obtained from a GitHub project⁴.

Listing 6.1: Example of a Calculated Expected Value.

```
1 assertEquals(java.lang.Math.pow(2*2*2 + 1*1*1, 1.0/3.0),  
2             d1.distance(v1, v2), 0.0001);
```

If our model only captured literals as expected values, it would not be possible to reflect situations like these in which the expected value is the result of the execution of an operation. The above example uses the static `pow` operation of the `Math` class from the Java standard toolkit in order to obtain the expected value. One reason why the author of the test may have done this is to make it maintainable, and the calculation using concrete values is much more self-explanatory than providing the plain result of the calculation. We can also imagine a test that reads the expected value from an input file or that obtains the required information from a web-service. Since our model has to consider such situations we therefore define the *ExpectedResult* as a child class of *Invocation*.

⁴ <https://github.com/Zephyr-Trail/KDD-lingpipe/blob/master/xinghuangxu.lingpipe/src/com/aliasi/test/unit/matrix/MinkowskiDistanceTest.java>

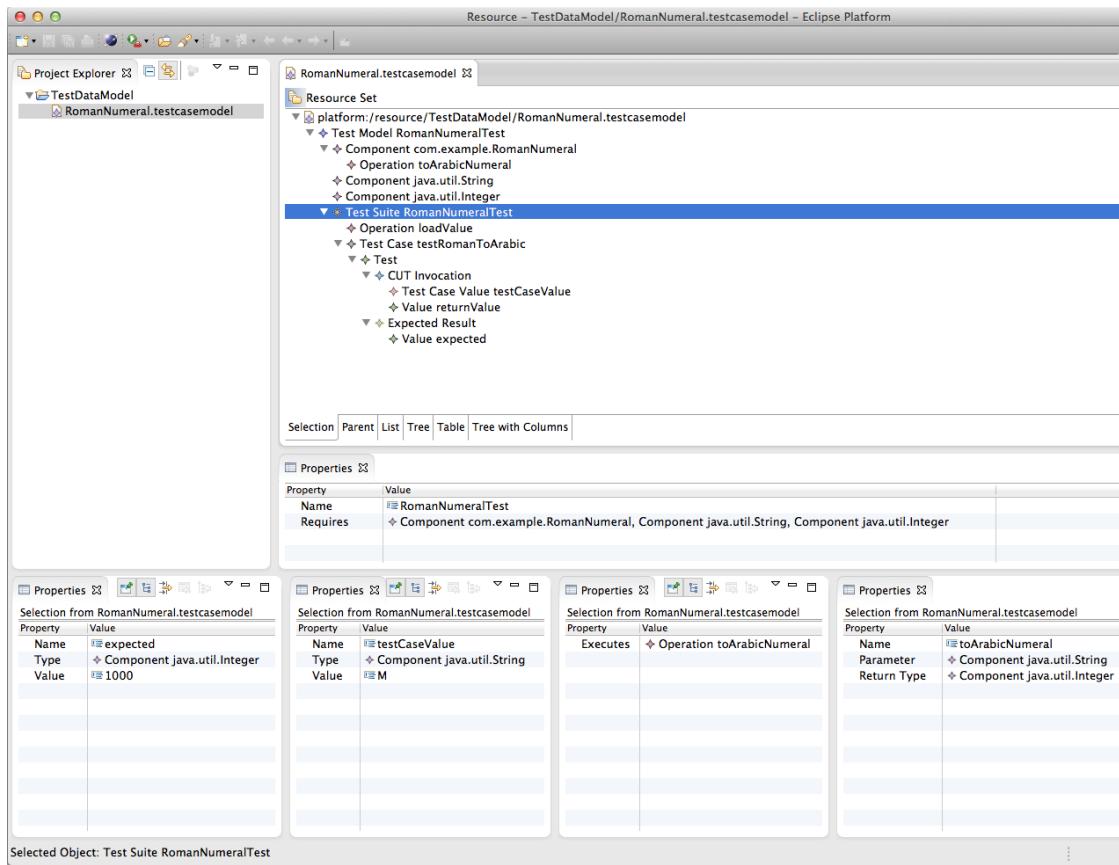


Figure 6.5.: Exemplary Meta-Model Instantiation.

An *Invocation* node subsumes the execution of an operation and the associated values. For a *CUT/Invocation*, it is necessary that the invocation provides the *test case values* to the tested operation and that it returns the value obtained by the execution of the CUT's operation. The node *TestCaseValue* is therefore introduced for two reasons: for convenience, in order to better reflect the terms defined in this thesis within our meta-model and in order to make test case values a separately searchable criterion within the database.

For the expected result there are two possible scenarios that are covered by the invocation. The “traditional” scenario, where the expected result is a literal, which can be covered by declaring the called operation to be responsible for an instantiation of the particular type of the expected value. Hence, for a JUnit test case where the expected value is the integer value ‘5’, the invocation contains the *Integer()* constructor as operation, no provided parameter and ‘5’

as the return value. On the other hand, an expected result derived from a concrete invocation like the one in Listing 6.1, would result in the node storing the Math.pow invocation as operation, the provided input values and the result delivered by the operation.

Summary

In order to evaluate the introduced model, we implemented it in Eclipse ECore model and successfully validated it for consistency. The screenshot in Figure 6.5 shows an Eclipse model editor derived from our meta-model which can be used to model JUnit test cases. In this particular case, we show an example of a single test for a Roman numeral calculator contained within an enclosing JUnit test case.

Naturally, neither our meta-model nor its instantiations are intended to check whether a test suite and the contained tests actually make sense. It exclusively serves to capture the necessary information in existing tests and allows us to represent them adequately in a database. In the following section, we will discuss the creation of a search index for JUnit test cases which drives the *SENTRE* search engine for reusable tests.

6.3. Index Creation

Based on the ideas presented in Section 3.2 and the lessons learned from other software search engines this section describes the creation of a search index for reusable software tests. This index will be the backend driver for *SENTRE*, storing the information captured by our data model and enabling users to efficiently search for reusable tests. The database software we use is *MongoDB*⁵, which is a document-oriented, so-called NoSQL database. It enables us to represent our models as JSON documents. JSON is a format that is human-readable, yet easy to generate and parse by machines [JSO14]. The MongoDB backend

⁵ <http://www.mongodb.org/>

is capable of storing the large amount of data we acquired from the GitHub repositories and makes it efficiently searchable. Not only the data is stored as JSON documents, but also all the database-operations and queries are formulated using this format.

Database optimization is a crucial factor for the retrieval efficiency of a search engine. The usage of indexes in MongoDB allows SENTRE to respond to search requests quickly and efficient. For our purposes, we create indexes on the fields affected by our query language, i.e., for the class names, method names and their parameters and return value respectively. Nevertheless, the usage of indexes cannot speed up result retrieval in all situations. Since MongoDB uses *B-trees* during index creation, searches with leading wildcards are slower than queries with trailing ones, i.e., in the case worst they take as long as if there was no index created at all.

Listing 6.2: JSON Representation of a Component.

```
1 {
2   _id : <ComponentId>,
3   name : "Euclid",
4   operations : [
5     name : "dist",
6     parameter :
7       [
8         { type:double name:x1 }, { type:double ,name:y1 },
9         { type:double ,name:x2 }, { type:double ,name:y2 }
10       ],
11     returnValue : { type : double }
12   ]
13 }
```

The JSON-style storage format of MongoDB simplifies the translation of information captured in the previously introduced data model into database entries. The classes from the model are represented as so-called *collections*, while the concrete entries are stored as *documents* within them. To provide a better understanding of how a test can be captured, we recall the JUnit test from Listing 2.2 which

we partially translate to MongoDB's JSON format. Due to the implementation of MongoDB, which makes searches in nested arrays rather complicated, we stick to storing the information from our conceptual classes in separate entries. Furthermore, in order to reflect the associations between the classes, we reference them by a unique id wherever this is appropriate. This allows the relevant information contained in the referenced test code to be represented as MongoDB entries according to Listing B.9 in the appendix. Listing 6.2 shows an example representation of a CUT and its operation.

6.3.1. Index Content

Now that we have defined an information model and defined the technology used to store the information represented using this model, we refer to the discussion from Section 6.1 and describe the building of the SENTRE search index. Subsequently, we describe *what* we actually have stored in our backend. While other examples from the literature used to find reusable assets with the help of tools like *nutch*, we eschew a web crawler and instead draw our reusable assets from the major open source software hosters. In particular, we utilize a set of Linux shell scripts, to retrieve all available projects from *GitHub*. This was carried out in three main steps: 1. using the GitHub web API, we created a list of projects available on GitHub, 2. to reduce network traffic we only downloaded the master branch of each project without history, and 3. after download we inspected every project for Java files.

GitHub offers a web API to access its services programmatically⁶. With a little shell script, we were able to retrieve the provided information about all projects publicly hosted on GitHub, including project id, creator, fork information, html url and repository url. Although GitHub provides information about the language used in a project, its automatic recognition seems to be rather imprecise. For example, it appears that Java web application projects that are accompanied with files written in HTML, JavaScript, etc. are not reliably recognized as Java

⁶ Accessible via <https://api.github.com/<command>>.

Source	Java Files	JUnit Files	TFR	Size
GitHub	98,767,946	13,380,472	13.5 %	14.5 TB
Merobase [Jan+13]	2,427,029	121,988	5.0 %	49 GB
Bitbucket	1,368,830	259,798	19.0 %	147 GB
Σ	102,563,805	13,762,258	13.4 %	14.7 TB

Table 6.2.: Repository Content of SENTRE.

projects. Hence, we have to download the files of each project and scan them for Java files.

In contrast to earlier search engines like *Merobase*, which was able to checkout the full repositories of SourceForge, the content provided by GitHub is far too large to download and store using limited academic resources. Since we are not (primarily) interested in the evolution of the files to be parsed and do not collect information about their evolution over different versions, we download only the latest version of the master branch of each project and do not clone each project's Git repository. Nevertheless, to create *SENTRE* we still had to download over 100TB of data. Each downloaded zip file was inspected for Java files and only those projects were extracted to our local repository storage which contained at least one Java file. Although these measures reduced the amount of data we actually store, the whole *SENTRE* code repository still consumes \approx 14.5TB of storage space. As a comparison, the repository of the research project Merobase.com consumes just about \approx 49GB.

The whole *SENTRE* repository is composed of data from different sources, which are listed in Table 6.2. While the projects acquired from GitHub form the major part of the repository, we have also incorporated the repository of Merobase and projects from BitBucket. As described in the literature [Hum08; Jan+13] the Merobase repository itself is mainly composed of projects from SourceForge and the Apache projects. In total we inspected more than 100 million files, amongst which we found more than 13.5 million JUnit test cases. The third column of the table above shows the test-file-ratio (TFR), while the size of all projects is listed in the last column.

In addition to that, we have listed some code metrics based on a per line analysis⁷ in Table 6.3. Although the lines of code are usually regarded as a weak metric, they allow us to compare SENTRE to commercial systems like Ohloh.com, which claims on its home page to contain more than 21 billion lines of code in 102 programming languages⁸, as of April 2014. They do not state, however, whether these are “productive” code lines or whether this number includes comments and blank lines, for example, as well. For comparison, SENTRE contains more than 17.5 billion lines of code in Java files, among which we counted approximately 7 billion commented or blank lines.

Source	LOC	CLOC	Blank	Total
GitHub	10,419,125,210	4,498,517,183	2,351,307,353	17,268,949,746
Merobase	212,947,748	107,680,663	49,253,236	370,881,648
Bitbucket	109,208,265	44,587,998	23,522,939	177,319,202
Σ	10,741,281,223	4,650,785,844	2,424,083,528	17,816,150,595

Table 6.3.: Lines of Java Code in the SENTRE Repository Grouped by Source.

The following subsection describes how we extract the relevant information from the retrieved files to store them in the SENTRE search index, before we discuss the effective and efficient retrieval of results from this index.

6.3.2. A File Parser for JUnit Tests

In Section 2.2 we have discussed some of the ways in which a Java class can be tested, and we demonstrated that there are many pitfalls to be considered when trying to automatically extract relevant test information. The technology we utilize to extract a test suite’s features is the Java Compiler API, which offers a straightforward technology to access the *abstract syntax tree (AST)* of a class, which is a tree representation of the syntactic structure of a particular source

⁷ Data obtained using the open source utility CLOC from <http://cloc.sourceforge.net>

⁸ cf. <https://www.ohloh.net/languages>

code asset. Furthermore, the Java Compiler API allows us to efficiently visit the AST's tree nodes and derive the relevant test information.

The parsing process itself is split into three phases: 1. find all Java files on the file system and recognize the declared classes, 2. resolve dependencies, and 3. parse the recognized test cases, i.e., extract and store the contained information. First, the system searches for all Java source files on the file system and subsequently extracts some basic structural information. An overview of the features stored is provided in Table 6.4. We have chosen to use a relational database for this step, since the information schema is fixed and predictable. In addition to the path of the class on the file system, we store its canonical name (e.g., com.example.MyClass), an MD5 hash of the class body, the canonical name of the super-type and whether the class is recognized as a JUnit test case.

Descriptor	Content
path	file system path
fqdn	canonical name
md5	MD5 hash of code
supertype	class super-type (fqdn)
isTestCase	true if class is a JUnit test case, false otherwise

Table 6.4.: Java File Table.

During the initial file system scan it is not possible to reliably recognize all JUnit test cases, since their Java nature allows inheritance hierarchies of test cases to exist, as well as to mix different versions of JUnit. Thus, only classes, that directly inherit from the class `junit.framework.TestCase`, can be recognized as JUnit 3 tests during the first pass, as well as all those that contain the `@Test` annotation from JUnit 4. Hence, we need a second pass to recognize JUnit test cases in deeper hierarchy levels and during this second step, our system inspects the entries of our Java file table and looks for unresolved super-types.

The code snippet in Listing 6.3 shows an excerpt of a JUnit test case that cannot be identified as such in the initial run:

Listing 6.3: Unrecognized JUnit Test Case

```
1 import com.example.special.MySpecialTest;
2 public class MyVerySpecialTest extends MySpecialTest {
3     public void testMyVerySpecialTestMethod() {
4         ...
5     }
6 }
```

The code of `MyVerySpecialTest` neither contains a reference to the `TestCase` class from the JUnit 3 framework, nor does it contain an annotation that would reveal it to be a JUnit 4 test case. Hence, we need to inspect the inheritance structure of `MySpecialTest` (cf. Listing 6.4), which we find inherits from `MyTest` and can therefore also not be recognized as a JUnit test case.

Listing 6.4: Superclass of MyVerySpecialTest

```
1 public class MySpecialTest extends MyTest {
2     public void tearDown() {
3         ...
4     }
5 }
```

As we can see in Listing 6.5, the class `MyTest` inherits from the `TestCase` class, which is contained in the JUnit 3 framework, and therefore it is recognized as a test case during the initial file inspection. Now, in the next run of the parser, we scan the Java file table, for classes with unresolved super-types – in the example case this is `MyVerySpecialTest` and `MySpecialTest`. These are inspected recursively, i.e., the system walks through the whole inheritance tree until it arrives at `TestCase` or the basic Java type `Object`. Using this strategy, our algorithm recognizes both example classes as a successor of `TestCase` which is reflected by the “true” value of the `isTestCase` flag (see Table 6.4).

Listing 6.5: Class Inherits from junit.framework.TestCase

```
1 import junit.framework.TestCase;
2 public class MyTest extends TestCase {
3     public void setUp() {
4         ...
5     }
6 }
```

Subsequently, the parser processes each test case file in order to extract the test information captured by the previously introduced data model. As already discussed before, JUnit test cases are usually not written in accordance with the recommendations of the developers of the framework. Thus the following subsection discusses heuristics that help to maximize the information yield from each test case.

Information Extraction from JUnit Test Cases

Earlier in this thesis we have discussed some of the many ways of writing JUnit test cases (see Section 2.2). Since JUnit neither provides language elements that help to describe the nature of a test case, nor does it contain any meta-information, the development of a parser for JUnit test cases involves the implementation of different heuristics in order to extract the appropriate information from the test case. To be able to build a database of reusable tests, first and foremost it is necessary to “understand” *what* a test actually tests, i.e., the parser needs to unambiguously recognize the *class under test* (CUT).

By definition, JUnit does not make the class under test any kind of “first class citizen” among the dependencies of the test class (i.e., the JUnit test case). Before we consider more sophisticated approaches for recognizing the class under test, we first assume adherence to the guidelines in the literature [BG14]: a JUnit test case should reflect the name of the class under test in its own name. Consequently, the examples from the JUnit documentation shown in Listing 6.6 call the test case of a Money class MoneyTest.

Listing 6.6: Example from the JUnit Documentation [BG14].

```
1 class Money {
2     private int fAmount;
3     private String fCurrency;
4     public Money(int amount, String currency) {
5         fAmount= amount;
6         fCurrency= currency;
7     }
8     public int amount() {
9         return fAmount;
10    }
11    public String currency() {
12        return fCurrency;
13    }
14    public Money add(Money m) {
15        return new Money(amount()+m.amount(), currency());
16    }
17 }
18
19 public class MoneyTest extends TestCase {
20 // ...
21     public void testSimpleAdd() {
22         Money m12CHF = new Money(12, "CHF");           // (1)
23         Money m14CHF = new Money(14, "CHF");
24         Money expect = new Money(26, "CHF");
25         Money result = m12CHF.add(m14CHF);           // (2)
26         Assert.assertTrue(expect.equals(result)); // (3)
27     }
28 }
```

The documentation example also shows that a JUnit test mainly consists of three parts: (1) the testing context, which is also called a test's *fixture*, (2) an execution of the objects created in the fixture, and (3) a result verification. It is also necessary to note that this is only a very basic definition, since the authors do not consider static references to the class under test, for example. Thus, if the code of the JUnit test case contains a reference to a type with a name that is equal to the test case's class name without the *Test* extension, the parser can assume that the class under test has been discovered. Nevertheless, it is recommended to apply additional inspection to confirm this assumption and combine the individual indications into a unified result.

Another useful piece of information for CUT detection are the tests contained in the test case, i.e., the information contained within the assertion statements. The JUnit framework defines a set of *assert* statements which expect a given set of parameters for result evaluation. The most important ones⁹ are listed in Table 6.5 on page 124. Additionally, Figure 6.6 depicts the percentage distribution of the JUnit assert statements contained in the SENTRE repository sources acquired from GitHub. We counted a total number of 157,368,390 assert statements, among which the *assertEquals* statements represent the largest group with a total of 93,861,398 appearances. Together with the *assertTrue* statements, they represent more than 80% of the assert statements contained in our sources, while the remaining six assert-types form a less significant portion of the repository.

Statement	Parameters	Explanation
<code>assertTrue</code>	<code>boolean condition</code>	Inspects whether the condition parameter is <i>true</i> .
<code>assertFalse</code>	<code>boolean condition</code>	Inspects whether the condition parameter is <i>false</i> .
<code>fail</code>		Fails a test.
<code>assertEquals</code>	<code>Object expected, Object actual</code>	Compares whether two objects are equal. Defined also for primitives and arrays.
<code>assertNotNull</code>	<code>Object object</code>	Tests whether object is <i>not null</i> .
<code>assertNull</code>	<code>Object object</code>	Tests whether object is <i>null</i> .
<code>assertNotSame</code>	<code>Object expected, Object actual</code>	Tests whether two objects do <i>not</i> refer to the same object.
<code>assertSame</code>	<code>Object expected, Object actual</code>	Tests whether two objects do refer to the same object.

Table 6.5.: Digest of the JUnit assert Statements.

In the code example from Listing 6.6, line 26 contains an assert statement whose inspection reveals that it references two objects created by instantiating the Money class. Therefore our CUT detection algorithm is able to identify Money as the class under test for this JUnit test case.

⁹ obtained from <http://junit.sourceforge.net/javadoc/org/junit/Assert.html>

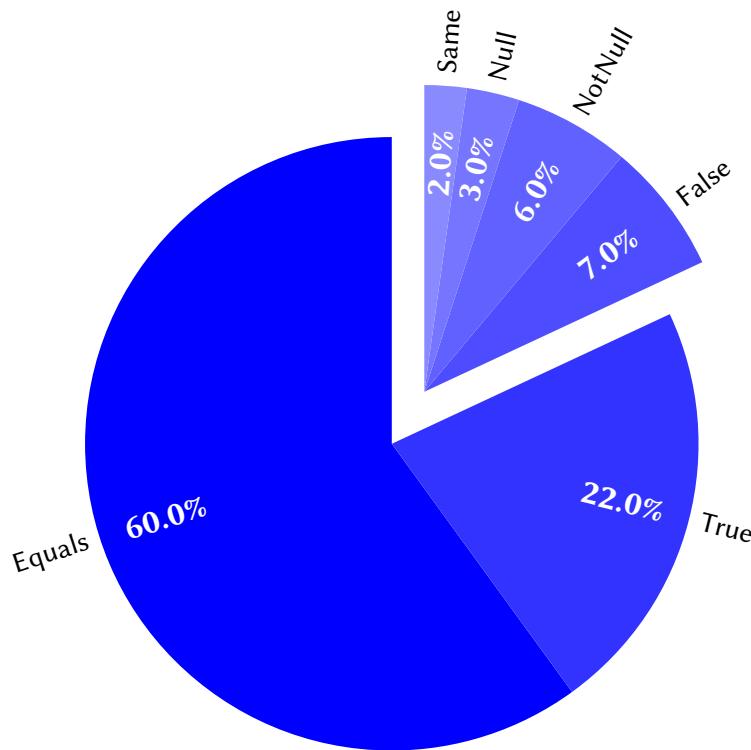


Figure 6.6.: Distribution of assert Statements.

With the help of the previously created AST the JUnit test parser is able to inspect the assert statements and to analyze their parameters. In line 26 of our example, where the call of `assertTrue` expects one boolean parameter, it becomes obvious that inspection of the assert statement alone may not be enough. In our case, the boolean value is obtained by a call to the `equals` method of the expect object, which is of the type `Money`. This method is, however, not defined by the class `Money`, but inherited from `Object`. Hence, the parser needs to inspect the full AST of the test case in order to track all objects that are created and, in order to support stateful tests, it must also track the (order of) changes to these objects (i.e., the operations which they are involved in). The data model from Section 6.2.1 reflects this requirement with the introduction of statements that are contained in test cases and associated with tests.

Finally, Listing 6.7 shows an example of a rather “bad test” from the point of view of an automated code parser. First, the test class is called `OrderTest` although there is no `Order` class referenced in its source code. Additionally, the assert

statement compares two integers, although the intention of the author was not to test the correctness of this primitive data-type, but rather to find bugs in the CollectionUtil class from line 8, whose static *sort* method is called with an integer array as parameter. To make the situation even worse, the assert statement is utilized with wrong parameter order, since the first parameter should be the expected value, while the second parameter should be the test result.

Listing 6.7: JUnit Test for Sort

```
1 import static org.junit.Assert.assertEquals;
2 import org.junit.Test;
3
4 public class OrderTest {
5     @Test
6     public void testOrder() {
7         int[] collection = new int[] { 4, 2, 5, 1, 3 };
8         CollectionUtil.sort(collection);
9         for (int i = 0; i < 5; i++) {
10             assertEquals(collection[i], i + 1);
11         }
12     }
13 }
```

In such a situation neither the strategy of name comparison, nor the information obtained from the assert statement helps to identify the class under test. Nevertheless, we are still able to deal with this issue by implementing a context-aware heuristic that checks the origin of the objects and values, which are referenced by the assert statement. If they are included in the standard Java libraries, it is very likely that they are not what the author of the test case wanted to inspect. Hence, we have to look back in the abstract syntax tree and investigate where they originate from, e.g., whether they are a parameter or the result of a method invocation.

In this case, the system finds that the int array collection is a parameter to the sort method of the CollectionUtil class, and since there is no other reference

to it, chances are good that the class under test has been discovered. Obviously this strategy does not work when the test case under consideration creates multiple objects from different self-defined complex types and two or more of them manipulate the object inspected by the assertion. Even a human examiner would have difficulties understanding *what* is actually tested so such a test case has to be considered to be of bad quality and should be abandoned.

Although it is very convenient – especially for developers – that JUnit simply allows standard Java to be used to test Java classes, the drawbacks arising from this freedom should not be underestimated. As our investigations show, only a small fraction of developers and testers actually adhere to the recommended conventions for test cases outlined by the authors of the JUnit framework. The freedom to use Java can therefore be both blessing and curse, and there is a need to either enhance the framework with structures that embody meta-information or tools like Test-Sheets [Atk+08b] need to become mainstream in testing.

Information Extraction from Software Tests

When the class under test has been identified from a test case, the extraction of other relevant information stored in the test case is a straightforward task. However, there is one fundamental constraint related to the nature of object-oriented programming: software tests may test the functionality of a program that does not rely on states or they may test software that shows different behavior in different states. While the former is usually found in algorithms that perform some sort of calculation (e.g., a Roman numeral converter or a distance calculator), the latter are related to more complex *business objects* (e.g., a stack, a shopping cart or a customer).

Nevertheless, in principle the process of information extraction from software tests is not affected by this distinction. The consideration of the state-awareness is more relevant and important during the retrieval of reusable tests, when new tests are composed from existing ones. More specifically, for stateless tests the order of the invocations performed on methods of the class under test is not important, i.e., the order of the tests is commutative. Nevertheless, even for

Listing 6.8: Example for Stateless Tests.

```

1  public class RomanNumeralTest extends TestCase {
2
3      RomanNumeral r = new RomanNumeral();
4      String ten = "X";
5      int hundred = r.fromRoman("C");
6
7      public void testThere() {
8          assertEquals(10, r.fromRoman(ten));
9          assertEquals(100, hundred);
10         assertEquals(1000, r.fromRoman("M"));
11     }
12
13    public void testBackAgain() {
14        assertEquals(1000, r.fromRoman("M"));
15        assertEquals(100, hundred);
16        assertEquals(10, r.fromRoman(ten));
17    }
18
19 }
```

these tests it is necessary to consider eventually present set-up and tear-down operations. State-aware tests, however, strongly depend on the execution order which must not be transposed.

An example of a stateless test is given in Listing 6.8, where both test methods `testThere` and `testBackAgain` induce the same behavior of the class under test and both show equal behavior (i.e., discover the same bugs if present). In the following we use this example to describe how information is extracted from a test case. The principal information extracted by the parser is the name of the class under test. In our case, the system recognizes an instance of a `RomanNumeral` class, which is also defined by the name of the test case. Further analysis shows that no other objects are instantiated and therefore the algorithm unambiguously recognizes the `RomanNumeral` as the class under test. Subsequently, the system inspects all statements of the test case and stores any call to a method of the CUT in an ordered list (cf. Table 6.6).

Since the assertions do not necessarily contain all calls to the CUT, it is necessary to store all calls performed in the test case. Furthermore it is clear that it is

No.	Method	Parameters	Result
1	init		RomanNumeral
2	fromRoman	"C"	100
3	fromRoman	"X"	10
4	fromRoman	"M"	1000
5	fromRoman	"M"	1000
6	fromRoman	"X"	10

Table 6.6.: List of Calls to the CUT in Listing 6.8.

necessary to inspect any assignment that is related to the CUT. In the given example the integer variable hundred is assigned the return value of the fromRoman method when it is called with the string parameter "C". This is very important if we want to recognize all tests contained in the test case, since the assert statements in line 9 and 15, respectively, do not compare the return value of a call to the class under test with an expected value. Instead, they compare the literal value 100 to the value of the integer variable hundred and if the parser was not aware of the fact that hundred contains the return value of the CUT's fromRoman method, it would miss the test. As a consequence, based on the invocation order of the CUT operations, the tests recognized by our parser are as follows:

fromRoman: (C) → 100;
 fromRoman: (X) → 10;
 fromRoman: (M) → 1000;

In order to support the reuse of tests for state-aware business objects, the database must not only contain the mappings of test case values to expected results, but also all state-related information derived from the test case. This information has to be stored along with the corresponding test and delivered by the search engine upon request. When the tests are stored in an appropriate order, it is clear that any test n in the chain requires the execution of test $n - 1$ before it is executed, while $n - 1$ requires $n - 2$, etc. up to the first test. This ensures the reconstruction of the appropriate state of the tested object for every test in the set.

Due to the given reasons, the reuse of state-aware tests can so far only be performed on a *per test case* level, i.e., at the current state of the art it seems

hard to recommend single tests for state-aware objects. This is a highly complex problem which seems hard to solve since the automated assembly of test cases for stateful objects from independent test cases requires a deeper understanding of the domain of the CUT and of the evolution of any arbitrary state through method calls. Nevertheless, with our technology it is still possible to recommend complete test cases. This is as valid for test reuse as the recommendation of classes instead of single methods is for traditional software reuse.

Handling Exception Tests

Before we conclude this chapter, we want to introduce another important aspect in the way information is extracted from test cases in JUnit. The Java programming language supports the concept of throwing exceptions upon the malfunction of a program. These exceptions are not necessarily thrown at the level of the Java Virtual Machine (JVM), but can also be declared in the code of a Java program and indicate an invalid input, for example. The JUnit framework supports testing Java code for exceptions, and therefore it is a mandatory requirement for the creation of a searchable index of JUnit software tests that these exception tests are also covered. Exception tests present a very valuable source of information for reuse-assisted software testing. Since software testing is some kind of “destructive” activity in the software development process (testers are happy when they find a bug), we are especially interested in such test case values that are capable to cause the system under test to fail.

The code snippet in Listing 6.9 contains an example of a JUnit test case that actually verifies whether the RomanNumeral class throws a RomanNumeralException if its fromRoman method is called with an invalid input value. Therefore the test code calls the given method with an illegal input string and, if the program continues or throws any other exception type than the self-defined RomanNumeralException, it calls the fail method of JUnit. After the parser has again initially recognized the class under test, it examines the try-catch block contained in the test method and observes that it contains an invocation of the class under test.

Listing 6.9: Test Expects an Exception.

```
1 public class RomanNumeralTest extends TestCase {  
2  
3     RomanNumeral r = new RomanNumeral();  
4  
5     public void testBilbo() {  
6         try {  
7             r.fromRoman("Bilbo_Baggins");  
8             fail();  
9         } catch (RomanNumeralException rne) {  
10             System.out.println("Test_Passed!");  
11         } catch (Exception e) {  
12             fail();  
13         }  
14     }  
15  
16 }
```

Consequently, it stores the exception test in the SENTRE database for retrieval upon a search, e.g., for a RomanNumeral.

6.4. Summary

In this chapter, we have introduced the challenges presented by the creation of a repository of reusable test cases, discussed how to obtain reusable assets and we investigated the potential of mining open source software repositories for test cases. The inspection of some of the most prominent open source hosting platforms revealed that a large number of JUnit test cases is present in modern open source projects, and supported our initial findings from [JHA10]. We introduced the structure of JUnit test cases and discussed problems arising from the many ways in which they can be written. Subsequently we introduced a generic meta model and described how it is instantiated to a concrete data model for JUnit test cases. Finally, we gave an overview of how the index of the SENTRE search engine was created and which strategies and heuristics a file parser needs to apply to recognize the class under test.

Based on our experiences during the creation of a parser for JUnit test cases, we recommend future research in the area of test representation. Although the JUnit framework has gained a lot of popularity it was obviously not developed with reuse in mind. Since the main attraction of JUnit originates from the fact that tests can be expressed in the same language as the code being tested, one option would be to enhance JUnit with such things as new annotations that specify the class under test and add more meta-information for the tests of state-aware objects.

Contribution of this chapter

- We have defined a meta-model for test reuse, which captures the necessary aspects of object-oriented software-tests. To this end we have modeled the domain of software tests according to the definitions from Chapter 2.
- We have pointed out the issues involved in instantiating the meta-model for Java / JUnit and instantiated the meta-model for JUnit using ECore in Eclipse and demonstrated its feasibility with an example test case.
- We described how test-related knowledge can be extracted from JUnit test cases and presented a set of heuristics for test parsers to identify the class under test in an existing software test.
- For the recognized class under test, we have presented inspection techniques used by our parser in order to extract the appropriate test case values and corresponding expected results.
- We have explained how exception tests are recognised and extracted from JUnit test cases.
- An investigation of the GitHub source code showed, that – with a share of 60 % – the assertEquals statement is the most often utilized assert statement from the JUnit framework.
- The work conducted in this chapter has unveiled some problems arising from the high degree of freedom that is offered by JUnit, which allows its users to write test cases using plain Java code.

“

First law: The pesticide paradox.

Every method you use to prevent or find bugs leaves a residue of subtler bugs against which those methods are ineffective.”

BORIS BEIZER

Software-Engineer

7

Reuse-Assisted Software Testing

In the preceding chapters we have explained that software search engines have received a tremendous boost through the availability of large-scale repositories of reusable source code and have observed that these repositories also contain large numbers of potentially reusable test cases. Having provided the conceptual foundations for exploiting the knowledge bound up in existing (JUnit) test cases, in this chapter we look at possible usage scenarios for test search engines and describe the implementation of a search engine that benefits from our earlier findings. The search engine *SENTRE*¹ was developed as part of this thesis and serves as a proof-of-concept implementation that enables its users to find reusable assets stored in a code repository. In this chapter we describe the search and retrieval mechanisms implemented in our search engine. We take a look at the principles underlying each of the techniques, explain their usage in the context of SENTRE and describe how the retrieved results can be ranked appropriately in order to maximize their value for users.

¹Search-ENhanced Testing with REuse.

7.1. Usage Scenarios for Test Search Engines

Beside the question about the benefit, the two most important aspects of any newly developed approach in the area of software engineering are the questions about “*who* is going to use the newly introduced technology” and “*when* can it be applied” in the software development process. We have partially touched upon these issues in Section 3.1 (see pp. 34) while discussing *search scenarios* in software engineering. In this section, we identify *usage scenarios* for the reuse of software tests and describe how developers and testers can benefit from them.

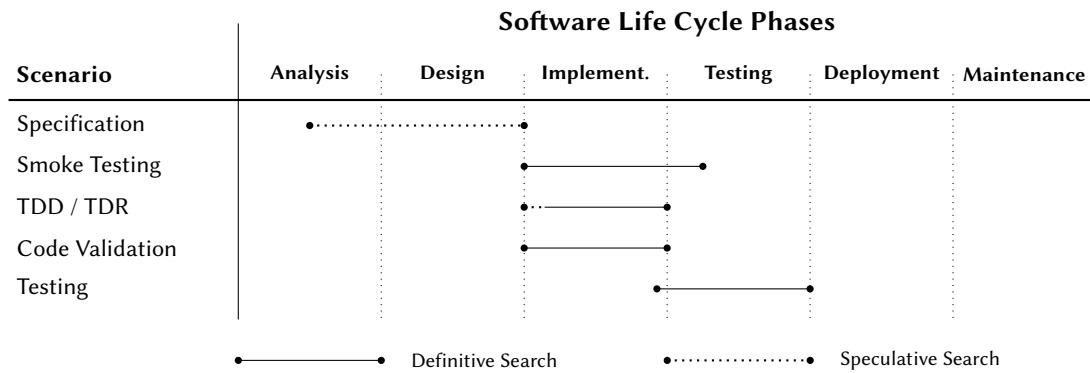


Figure 7.1.: Test Search & Reuse Scenarios in Software Engineering.

To illustrate when it may be appropriate to apply our tools and techniques, we have outlined usage scenarios for test search and reuse in Figure 7.1 similarly to those we defined for software search and reuse. Again we divide the possible scenarios into the two groups of *definitive* and *speculative searches*, where the former rely on precise specification of what is desired while the latter provide a way of exploring “what is around” in the repository. Obviously, the reuse of software tests makes most sense during the main development and testing phase, but it can be of some assistance in the early phases of a software project. It does not play a significant role during the deployment and maintenance phases, however, since all tests should have been written by then. At most, new tests may be added to the search repository for reuse in future projects.

7.1.1. Analysis & Design

For the specification of a software project, reusable tests that have been written for a similar scenario and domain are most likely to help to better understand how a similar previously created system works. This understanding bears the potential to help software engineers to write better specifications for a newly developed system. The participants in the software project can use previously created tests as a common basis of understanding how the system should work and create a better specification of the system. Reusable test cases for similar systems can also help to define acceptance tests and lower the likelihood that a software is being delivered that does not match the customer's requirements.

During the design phase, the reusable tests can similarly assist the system architects as traditional code reuse does, where they can obtain design inspirations from previously created systems. The application of test reuse in this scenario can, for instance, help to improve a system's design due to functional- and non-functional-requirements. The former may be directly expressed in unit tests while the latter, e.g., can be transported in test suites that are developed for the performance of stress tests.

7.1.2. Implementation

Since the development of a new application rarely starts with a green field, legacy systems often provide the basis for requirements definition and design. Such systems also provide a potential source of test cases that can be reused during implementation and the early testing phase. The reuse of tests from legacy systems can be very valuable to developers and testers and is a potentially convenient way of supporting *smoke testing*.

Test reuse can also be interesting during implementation when applied in conjunction with *test-driven development* or *test-driven reuse*. In both approaches, tests are written *before* the actual production code is written. In the latter case, developers can use existing tests to define better (i.e., more precise) queries for test-driven code search engines in less time. While searches for reusable tests

are assumed to be more speculative in character in the early stage of development (when there is little concrete information available), they are expected to become more concrete (definitive) relatively quickly and therefore more valuable to developers.

By reusing the knowledge contained in existing tests, developers can benefit from the knowledge that other persons – presumably domain experts – have invested in developing them. These tests can help to gain better insights into how a particular kind of component should behave. A natural implication of this idea also represents an enhancement to test-driven search: developers can inspect test cases similar to their own and decide whether the semantic description of the class under test suits their application. If it does, they may decide to reuse the test case and class under test at the same time. Although this idea seems appealing, it introduces once again all the issues related to “traditional” software reuse such as the *not-invented-here syndrome*.

The literature strongly recommends that software developers should not test their own code (i.e., testers and developers should be different persons, as stated by Myers [Mye79]). This is, however, a rather expensive and inconvenient approach in the (early) phases of a project where developers may try out different ideas to solve a given task.

Test reuse can be a means to bridge the gap between testers and developers, by offering the latter a tool to perform testing at development time without having to write the tests themselves. In the later phases of implementation, this tool support can help to address the “standard” tests for the class under development, allowing testers to concentrate on the “hard stuff”, i.e., those parts of the code that are not amenable to reusable tests and those assets for which no reusable tests could be found.

Reuse-assisted code recommendation during the development of production code is also the main scenario targeted by this thesis: developers should be enabled to test their code without needing to be domain experts. In other words, the expert knowledge bound up in existing test cases can become an important cornerstone for the improvement of code quality during implementation.

7.1.3. Testing

The previously suggested idea that smoke testing can be performed by developers using previously written test cases may also apply in the early phases of testing. Before testers start writing more thorough tests, they might want to analyze the delivered software for general correctness without investing a lot of effort. Reusable test cases are a simple tool that developers can use to identify software that does not perform the basic functionality that it was intended for. Although definitive searches for reusable tests may still occur later in testing, we do not expect reuse-assisted test recommendation to be the tool of choice for testing experts. Nor do we expect it to be the tool of choice for project management. It is important to remember that testing experts are paid to evaluate software based on their specifications and there are more aspects of testing than unit testing – like stress testing, penetration testing, testing non-functional requirements, ... It would be dangerous to (solely) rely on reused tests in any of these cases.

During the deployment and maintenance phases of a system's lifecycle, the potential for test reuse is rather low. One could imagine customers that reuse tests from a legacy system to validate the functionality of a new system during acceptance testing, but we have not performed further investigations in this area. The activities performed during the maintenance phase are likely to create new input to a test repository rather than to benefit from test reuse. If and when a bug is discovered within the software, the corresponding tests need to be persisted in the repository. The tests created during deployment and maintenance can further enrich the test repository of the test search engine and serve future developers for their purposes and help them to reuse tests that contain the knowledge obtained from earlier software failures.

7.2. Result Retrieval Techniques for Test Reuse

Earlier in this thesis we already mentioned the highly influential survey on “the storage and retrieval of reusable assets” by Mili et al., who describe a classification system of retrieval methods for software reuse [MMM98]. Based on their insights,

in this section, we will introduce a couple of retrieval mechanisms for test reuse systems and investigate their strengths and weaknesses. Furthermore we describe how the identified and developed retrieval techniques have been implemented in the SENTRE reuse system for software tests and give an overview of how users of the system can create appropriate search queries. Finally, we will also discuss the possibilities of result ranking. This is necessary in order to provide users with better and more valuable results with regards to their particular context.

Before we go into more detail, we identify the following three search strategies for the retrieval of reusable software tests:

Interface-Based Searches The required interface of the user's test suite is used as a search query for reusable tests.

Value-Based Searches The search for reusable software tests is based on the mappings of test case values to their corresponding expected results, both of which are extracted from the user's test suite.

Code-Driven Searches The developer's production code serves as the input query. Candidates for test reuse are evaluated against the developer's CUT and grouped by their execution profile.

Although the latter variant looks like the opposite of test-driven reuse, the strategy behind this technique is quite different. The evaluation of potentially reusable test cases and software tests involves more than answering the simple binary question of whether a search result is suitable or not. The results obtained by applying a reused test case to the developer's class under test need to be examined in a more differentiated way as we will discuss later in this section.

At the beginning of this section, we examine the “traditional” techniques used for the retrieval of reusable code – the interface-based search. We will describe query formulation issues, introduce query refinement techniques and discuss how to rank the retrieved results. Subsequently, we focus on a new form of search whose goal is to identify reusable tests based on the mapping of test-case values to the corresponding expected result.

7.2.1. Interface-Based Searches

Over the last decade, the rise of modern code search engines helped to establish a set of common retrieval techniques for reusable software. One of them is the idea of interface-based searches, where components are usually retrieved based on a textual description of their (publicly visible) interface. The SENTRE test search engine, which is developed as a part of this thesis, also supports interface-based searches and therefore provides an easy and self-explanatory query format. However, in contrast to code search engines that rely on the provided interface of a component, the SENTRE query is a representation of the required interface of reusable software tests. This difference between test search engines and search engines using the provided interface of a component also means, that a system for test reuse actually relies on information contained in the code and that the required interface describes the *uses- and calls relations* to a component under test, i.e., a test case contains, *inter alia*, a subset of the provided interface of the corresponding component under test.

According to the classification scheme of Mili et al. [MMM98], interface-based search can be classified as a member of the *denotational semantics methods*. This category subsumes retrieval techniques that are based on formal specifications, as well as those that are based on signature descriptions. We have chosen to integrate this technique in SENTRE, since the interface description of a software component does not involve any additional effort during development and maintenance, as already stated by Meyer in his seminal work on the application of the principle of *design by contract* [Mey92].

Following these arguments, the provided interface is amenable to automatic extraction as an abstract part of the code. Therefore, it can easily be extracted by a parser and stored in the search engine's index as a descriptive element for the tests contained in the underlying repository. Earlier in this thesis we described our data model for a test search engine and already presented some examples of interface-based queries. Now we are going to look at query formulation in more detail, explain the process of result retrieval, automated query refinement and the ranking of search results.

Query Formulation

As an example we assume that a developer needs to write a Java class to convert Roman numerals to Arabic numerals. We also assume that the chosen approach to achieve this is test-driven development. Thus the developer might write an initial JUnit test case something like the one presented in Listing 7.1.

Listing 7.1: Excerpt of a Test Case for a Roman Numeral Converter.

```

1 public class RomanNumeralTest {
2     @Test
3     public void testOne() {
4         assertEquals(1, RomanNumeral.toInt("I"));
5     }
6 }
```



Figure 7.2.: Provided and Required Interface of the Test in Listing 7.1.

The Java code of this test uses a `RomanNumeral` class, or more precisely, it invokes the static `toInt` method with the string parameter "I". The `assertEquals` statement expects the CUT to return the integer value 1. From this information the system infers that the developer potentially wants to search for tests that inspect a class exposing the interface depicted in the UML class diagram in Figure 7.2. To describe the CUT's provided interface, we use a format similar to the Merobase Query Language MQL, which allows us to use the following query to describe the test's required interface

`RomanNumeral(toInt(String):int;)`

and which triggers the search for the appropriate tests.

However, this is not the only scenario envisaged during the creation of our search infrastructure. Especially for *speculative searches* it is not realistic to expect users to provide completely formulated queries. It is more likely that they only have a limited idea of the interface and the method signatures of the component they are looking for. Therefore, it is necessary to provide an additional degree of freedom in query formulation, which is accomplished with the provision of a *wildcard* symbol for queries.

If a user is not sure about the classname for a roman numeral converter, it is possible to use the following query, where the dollar sign serves as a wildcard:

```
$ (toRoman(int):String;)
```

This query searches for any test of any arbitrary class that contains a `toRoman` method, which requires an integer value as parameter and returns a string object.

Result Retrieval

Internally, SENTRE converts the query into an appropriate JSON-style representation that serves as a request to the database infrastructure. Before the search is actually performed, the system checks whether the request represents a well-formed query. If it does, the query is then decomposed into its structural elements (i.e., the classname, method names and signature information) and a search query is generated similar to the MongoDB query presented in the subsequent Listing 7.2.

Line 2 of the listing contains the description of the classname. Since we have chosen to append a wildcard to the word *Roman*, the query parser translates the query using a regular expression, and the same strategy is applied for the method name. The method parameter is provided without any degree of freedom, as well as the return value of the query.

Listing 7.2: MongoDB Query in JSON-style Format.

```
1  {
2      "name" : { $regex : "roman.*" },
3  {
4      "operations.name" : { $regex : ".*" },
5      "operations.parameter" : "String",
6      "operations.returnValue" : "Integer"
7  }
8 }
```

Query Refinement

Naturally, it is convenient for the users of a search engine if they can put all available information from their task in hand into one search query. Nevertheless, this imposes the problem that the addition of (non-redundant) information to search queries may reduce the size of the retrieved result set. Even worse, a well specified and detailed query may ultimately lead to an empty result set. Hence, it is necessary to apply a strategy for query reformulation and refinement, which is able to deal with over-specified queries and stops when the retrieved result set contains a minimal number of elements or no further refinement is possible.

To avoid a situation in which the user has to perform the query reformulation, we have implemented an automated relaxation algorithm that handles the refinement of the initial query. When a pre-defined minimum number of results is discovered, the algorithm stops and the query is not relaxed any further. If there is an insufficient number of results, the system will relax the query in the way shown in Algorithm 7.1. The relaxation strategy applied in SENTRE is based on the following steps:

1. Search for exact matches of the query (no relaxation).
2. Add wildcards to the method names.
3. Search for the classname and the method signatures.
4. Search for exact classname matches.
5. Add wildcards to the classname.

We have chosen to apply relaxation first to the methods and then to the class-name due to the fact that method names may differ a little but still belong to the same context. This addresses, for example, the scenario where the index contains test cases for two distinct RomanNumeral classes, where one contains a fromRomanToInt method, while the other one contains a toInteger method.

Algorithm 7.1: Smart-Search Query Relaxation.

```

Data:  $q \leftarrow$  query
Data:  $S \leftarrow$  relaxation subject
Data:  $n \leftarrow$  minimum number of results
foreach  $s_i \in S$  do
     $q_c \leftarrow$  relax query  $q$  with  $s_i$  ;
    search with query  $q_c$  ;
    foreach search result  $r$  do
        | add  $r$  to global result set  $R$  ;
    end
    if  $|R| \geq n$  then
        | exit loop ;
    end
end
Return  $R$  ;

```

When the search is performed SENTRE presents an overview of all matching test cases. Each test case result shows the name of the class under test and a table that contains the corresponding tests and method invocations.

These *invocation tables* are inspired by the *test-sheet* metaphor [Atk+08b], which envisages the presentation of test cases in a spreadsheet like format. Figure 7.3 shows the results of an example search using SENTRE, where the first column of the test sheet contains the name of the CUT's method which is part of the required interface of the original test case. The second column shows the expected result of the test and the following columns contain the test case values of the method invocation.

Typically, a search result is composed of a method invocation using test case values and the appropriate expected result. In addition to that, our SENTRE search engine offers the possibility to deliver so-called *exception tests* for the class under test. In Java, programs may throw an exception when they face an unexpected condition such as an invalid input.

With JUnit it is possible to test whether a program throws an exception for given test case values and our parser is able to recognize these tests. Consequently, our search engine delivers exception tests in a separate result table, next to the result test-sheet. Since exception tests tend to be more complex than plain value mappings, we will take a closer look at them in Section 8.3, when we also discuss the usage and potential of exception tests in the context of a test-reuse environment in the Eclipse IDE.

The screenshot shows a web-based application window titled "SENTERE ::: Search-Enhanced Testing with REuse". The URL is "sentre.se-testing.info/alpha/QueryServlet". The interface has a toolbar with icons for back, forward, search, and save. The main content area is divided into two sections:

- CLASS**: Displays the Java code for the `RomanNumerals` class, specifically the `toRoman(Integer):String;` method.
- TEST-SHEET**: A table titled "RomanNumerals" showing the mapping between Roman numerals and their corresponding integer values.

Method Name	Return Value	Parameter #1
toRoman	I	1
toRoman	II	2
toRoman	III	3
toRoman	IV	4
toRoman	V	5
toRoman	VI	6
toRoman	IX	9
toRoman	X	10
toRoman	XX	20
toRoman	XXIV	24
toRoman	L	50
toRoman	XL	40
toRoman	XLII	42
toRoman	C	100
toRoman	XC	90
toRoman	CVIII	108

Figure 7.3.: Screenshot of a SENTRE Result Table.

Result Ranking

To provide users with better results, it is convenient to reorder the retrieved results by their relevance. In order to define a relevance function it is necessary to first identify relevance criteria. For searches based on the required interface of reusable tests we can obviously utilize the degree to which the provided interface of the CUT matches the required interface of the tests.

Since the interface of a class is basically formed by the classname and the contained methods, we have decided to split the description into its parts and base the calculation of result relevance on them.

In order to define a relevance function for the n results returned by a query and a weighted set of criteria c_1, \dots, c_k , we utilize the well-known formula for the weighted sum: the relevance R of the n^{th} result is therefore defined as

$$R_n = \sum_{i=1}^k w_i \cdot c_i \quad (7.2.1)$$

where w_i represents the corresponding weight of each criterion c_i . Each criterion is defined as a value between 0 and 1. This value indicates how well the search result matches the query with respect to the particular criterion. For the relevance of a full test case, the upper part of Table 7.1 gives an overview of the ranking criteria and their corresponding weight.

No.	Criterion	Weight
1	classname	6
2	method interface	4
No.	Criterion	Weight
2.1	method name	5
2.2	method signature	5

Table 7.1.: Criteria and Weights for Result Ranking.

SENTRE calculates the relevance of a result in two steps. First, it calculates the relevance of each method. A method is only included in the calculation of the relevance of a result if it has a non-zero relevance itself. The distance between each method and the query is therefore determined with the help of the weights from Table 7.2. Class- and method names are assigned a value between 0 and 1, depending on the similarity between the result and the query.

If the name of the result is exactly the same as the corresponding name in the query, we assign it the value of 1. If only a leading or trailing wildcard needs to be appended to the query's name to match the result name the value of .75 is assigned. The ranking algorithm also examines the names for camel case and extracts the words contained. If all words in a query's class- or method-name match the words in the name of the CUT, this is assigned the value .5, while a match of a subset of the words contained in the query result in the value .25. If the classname of the query and the CUT do not match at all, the classname is assigned the value 0. This happens as well if no method of the CUT matches any method declaration from the query.

Criterion	Match	Value
Name	Exact Match	1
	Trailing / Leading Wildcard	.75
	All Words Match	.5
	Subset of Words	.25
	Nothing Matches	0
Signature	Exact Match	1
	Only Parameter Types Match	.5
	Only Return Type Matches	.5
	No Match	0

Table 7.2.: Distance Weights for Methods and Queries.

After the method names of the CUT and the query have been examined, the algorithm compares the signatures of the methods that were assigned a value of .25 or greater. If the signature is an exact match, it is assigned the value 1. If only

the parameter types or only the return types match, the signature is assigned the value .5.

If neither the parameter types nor the return type match, the signature is considered a mismatch and assigned the value zero. Subsequently, these values allow the algorithm to calculate the relevance of each method declared in the query and to sum them up to the value of the *method interface* in the upper part of Table 7.1, which accounts for 40 % of the total result relevance. The comparison of the required classname of the result and the classname defined by the query is also performed with the help of Table 7.2 and contributes 60 % to R_n .

Wrappers for Primitive Data Types

Although Java is generally associated with the idiom that “*everything is an object*”, this is not entirely true. The Java language specification defines eight primitive data types that are not considered to be objects in the narrow sense of the word. While Java objects are created through the instantiation of a class using the keyword new, primitive data types are usually assigned a literal value. The Java Language Specification (JLS), however, introduces additional *wrapper classes* for the primitive data types, which are listed in Table 7.3 along with instantiation examples.

Since Java 1.5, the so-called *autoboxing* feature implements the automatic conversion between primitives and their wrapper classes. Hence, a distinction between primitives and wrappers is no longer necessary and the parser always stores the wrapper class when a primitive data type is found in a method signature.

We have chose to disregard the differences between the primitive data types and their wrappers and the underlying index of our SENTRE test search engine only makes use of the wrapper classes regardless of whether the original search interface was specified using primitive data or their corresponding wrapper type. Although we are well aware of the difference between primitive data types and their wrapper classes, this does not play a role for the results returned by the search engine, as they are in any case expressed as literals.

Primitive	Wrapper		Instantiation
byte	Byte	byte b = 2;	Byte b = new Byte("2");
short	Short	short s = 5;	Short s = new Short("5");
int	Integer	int x = 2;	Integer x = new Integer(2);
long	Long	long l = 5L;	Long l = new Long(5);
float	Float	float f = 2f;	Float f = new Float(2);
double	Double	double d = 5d;	Double d = new Double(5);
char	Character	char c = 'c';	Character c = new Character('c');
boolean	Boolean	boolean b = true;	Boolean b = new Boolean(true);

Table 7.3.: Java Primitive Data Types and Wrapper Classes.

Evaluation of Interface-Based Searches

As we have shown in earlier publications, interface-based searches are an improvement over plain keyword- or signature-based searches [HJA07]. Nevertheless, it is obvious that they have some drawbacks as well. One of their major weaknesses is their dependency on the names chosen by developers and searchers when looking for reusable artifacts. This problem is not new to the information retrieval community and in the literature there are many suggestions to tackle this such as through the application of similarity thesauri [BR08]. In his seminal work in semantic component retrieval, Hummel provided an overview of these technologies and describes the implications of these drawbacks on the area of software reuse [Hum08].

Naturally, the technology applied in test-driven reuse cannot be directly transferred to the reuse of software tests. Although the evaluation of the developer's CUT against reusable test cases seems feasible, the outcome does not suite our needs. We are interested in the scenario in which a developer searches for tests by providing the CUT from the project under development, in the same way when a test case is provided as a search query in test-driven reuse. Based on such a query, the search engine finds candidate reusable tests and applies them to the CUT in question. The CUT may pass all the tests without failing or may fail one or more of the tests from the test case. In test-driven reuse, the successfully passed test case would consider the CUT to be a suitable reusable component,

but test reuse has a different focus: since the goal of software testers is to find bugs, they regard failed tests as successful goal achievement.

Nevertheless, before we discuss possible improvements to the precision of search results, we will introduce other search and retrieval techniques that help us find potentially reusable assets. In the following, we will discuss a new way to express search criteria, which neither solely relies on class- or method names nor does it only consider type information from method signatures. Instead, it uses the test case values and expected results of the execution of the class under test to find relevant reuse candidates. This strategy shifts the focus from structural to semantics-based searches for reusable software tests.

7.2.2. Value-Based Searches

Although we have seen that interface-based search can be more precise than keyword- or signature-based searches [HJA07], there is still room for improvement. Therefore we introduce a new kind of search which is not based on structure, but on the behavioral information provided by a test. Earlier in this thesis, we discussed the nature of software tests and that they can basically be regarded as comparisons of test results against expected values. The test result is obtained by executing an operation of the system under test, while the expected result is provided by some kind of oracle. Based on this observation, we introduce another form of query for reusable tests which relies on test case values and expected results instead of structural information. Hence, in contrast to interface-based searches, these *value-based searches* utilize the behavioral information contained within the test cases and the tests respectively.

The earlier referenced classification scheme of Mili et al. [MMM98] does not directly mention a retrieval method corresponding to value-based searches. However, the authors mention so-called *operational semantics methods* for asset retrieval, which take into account the fact that software components are different from textual documents. As well as the structural difference, they emphasize that software components are executable and therefore *operational semantics methods* utilize the executability of components to select reusable assets. Although our

approach of *value-based searches* does not fully accommodate the notion of asset execution, we characterize this approach as a *specification-based operational semantics method*. While it does not involve the execution of the retrieved assets, the criterion for asset selection is the behavior of the class under test.

Query Formulation

Just as we did earlier, we start by providing an exemplary JUnit code snippet that is used to demonstrate the usage of value-based queries in the SENTRE search engine. Listing 7.3 shows a test case for a roman numeral converter, which tests the conversion of four Roman numerals (I, X, C, M) into their corresponding Arabic numeral counterpart (1, 10, 100, 1000).

Listing 7.3: Excerpt of a Test Case for a Roman Numeral Converter.

```

1 public class RomanNumeralTest {
2     @Test
3     public void testToInt() {
4         assertEquals(1, RomanNumeral.toInt("I"));
5         assertEquals(10, RomanNumeral.toInt("X"));
6         assertEquals(100, RomanNumeral.toInt("C"));
7         assertEquals(1000, RomanNumeral.toInt("M"));
8     }
9 }
```

Our goal is to make the search engine as easy and intuitive to use as possible. This applies to query formulation as well, since the formulation of a query is the first way in which a user interacts with the search engine. To create our value-based query language we draw upon the definitions from Section 2.1, where we described a test as an invocation

$$\xi : (\alpha_1, \alpha_2, \dots, \alpha_n) \rightarrow \Gamma$$

of method ξ with the test case values $\alpha_1, \dots, \alpha_n$ that leads to the expected result Γ . Since this notation transports all necessary information in a very minimalistic

and intuitive form, we can take the above form as a basis for the value-based query language. Hence, the example code from Listing 7.3 can be translated to the following query description

```
(I)->1;  
(X)->10;  
(C)->100;  
(M)->1000;
```

where the semi-colon marks the end of a test. Thus, the search could have been written in one line as well. The query is, however, not yet complete, since it does not unambiguously indicate whether the user wants to find test cases that contain any of the above tests or whether the search results have to contain all of the tests. To offer users the possibility to search for test cases that contain a subset of the specified tests, but at least one, we introduce the search operator `v:` that is prepended to the query. To enforce strict adherence of the query, the operator `vs:` ensures that all tests from the query are contained in any of the results retrieved by the search engine. A query that uses the strict value-based search operator is depicted in Figure 7.4, together with the first two results and a result test-sheet.

If we take a look at the results that are returned by SENTRE in response to the above query, we find that the second search result requires the following interface from the class under test:

RomanNumbers
+ <code>toString(Integer) : String</code>
+ <code>valueOf(String) : Integer</code>

Thus, an interface-based search such as the one presented on page 141 would not have found this test case, as it requires a method name that contains the

The screenshot shows the SENTRE Laboratory version interface. At the top, the SENTRE logo is displayed with the tagline "Search-Enhanced Testing(with():Reuse;)". Below the logo is a search input field containing the query string "vs: (1)->l; (10)->X; (100)->C; (1000)->M;". A "Sentre Search" button is located below the input field. The search results are displayed in a list format:

- 1 of 6 TextUtilTest.java (1 CUT – 82 TESTS – 0 EXCEPTION TESTS – 1 METHODS) *****
- CLASS**
TextUtil(
 toRomanNumberString(Integer):String;
)
- TEST-SHEET**
...
- 2 of 6 RomanNumbersTest.java (1 CUT – 41 TESTS – 0 EXCEPTION TESTS – 2 METHODS) *****
- CLASS**
RomanNumbers(
 toString(Integer):String;
 valueOf(String):Integer;
)
- TEST-SHEET**
- RomanNumbers**

Method Name	Return Value	Parameter #1
toString	I	1
toString	V	5
toString	X	10
toString	L	50
toString	C	100
toString	D	500
toString	M	1000
toString	XVI	16
toString	XIV	14

Figure 7.4.: Strict Value-Based Search with Result List.

word *roman*. This is not the case with the above interface, however. Value-based searches therefore represent a useful enhancement to pure interface-based queries. When applied jointly, the approaches complement each other and help to improve the recall of search results.

Regular Expressions

Although the previously introduced value-based searches offer a convenient way to specify behavioral characteristics of the search results, pure reliance on fixed values might not be the best way to achieve better recall. Given the search from above, our search algorithm might miss those test cases where the authors have chosen other test case values than those specified in the query. In software testing there are no *first-class citizens* for test case values, meaning that no test is better than another.

Though we are aware of the many testing techniques that have become established in software testing. Boundary value testing techniques and the like, which require testers to define a fix set of tests at the boundaries of equivalence classes, can certainly improve the effectiveness of a test suite, but it is almost like a law of nature that none of these techniques can guarantee the discovery of all possible bugs. As Fred Brooks stated in his famous book [Bro87], there is “no silver bullet”.

Since errors in program code may have many different origins, from developer negligence to purposely inserted malicious code, each test case value in an equivalence class has the same potential to discover a bug in a program and therefore the rule *the more we test the better is our test case* is certainly not without virtue. If the designer of a test decides to use different values to those specified in a searcher’s value-based search, our search engine will miss these results. We tackle this serious problem by an enhancement of our query format with a more formal way of specifying test case values and expected results. Instead of only allowing users to enter concrete values in their queries, the operator rex: indicates the usage of *regular expressions* in the query. Regular expressions are a well-known and widely adopted technique for pattern matching which are

often used by input validation algorithms². They allow us to specify input values in a more general way. In the appendix we have enclosed a short overview of the most important operands and example queries. More general information on regular expressions can be found in the literature, e.g., in Friedl's book on *Mastering Regular Expressions* [Fri02].

Value-based queries with regular expressions allow general rules to be specified rather than mappings between specific test case values and expected results. To clarify this we will show how to formulate a general rule for our Roman numerals example:

A Roman numeral is made up of a combination of the Latin letters I, V, X, L, C, D and M. A test of a translation of a Roman numeral to Arabic numeral therefore maps a Roman numeral to the corresponding Arabic numeral, which is composed of at least one digit between 0 and 9.

To translate this general rule to a value-based test with regular expressions we can exploit the fact that SENTRE supports the whole set of rules, delimiters and meta-characters for regular expressions implemented in Java. The query for our Roman numerals example can thus be written as follows:

```
rex:(^|[IVXDCLM]+$)->"\d+";
```

The prefixed rex operator tells the system to switch to value-based search with regular expressions, which allows the regular expressions to be embedded within quotation marks. Besides, the query structure is the same as for pure value-based searches – including the semi-colon at the end of each test. It is therefore possible to unify multiple criteria under the umbrella of one value-based search with regular expressions and even mix them with pure value-based searches.

Just as with the v: and vs: operator, a search enhanced with regular expressions can require a result to contain at least one of the search criteria or all of them. The former effect is achieved using the operator rex:, while the latter kind of search is performed with a prepending rex: operator.

² SENTRE uses regular expressions itself to identify malformed search queries.

Result Ranking

Obviously, the relaxed searches, where not all search criteria have to be met by a reuse candidate, may discover reusable test cases that contain only one of the specified tests from the query. Even worse, this test case may be intended to test a completely different abstraction and therefore contain other tests that have nothing to do with the user's class under test. In Chapter 9.3 we present an approach to eliminate such false-positive results and do not consider them further in our current discussion.

Nevertheless, it is necessary to rank the results obtained using value-based searching³ by some criteria. The most obvious criterion is the degree of overlap between the tests defined in the search and those contained by a reuse candidate. Those test cases that contain all specified mappings of test case values to expected results are ranked first, while the remaining are listed in descending order depending on the number of query mappings contained.

Although this strategy seems appealing at first sight, a closer look tells us that it is not appropriate. More specifically there appears to be no reason to rank results higher just because they contain the tests specified in the search query, as the user has obviously already discovered the specified tests. Remembering our arguments from above that there are no *first-class citizens* in software tests, we cannot ignore the fact that test cases with less query overlap can still contain tests that are as good as those with full query overlap. Hence, we need some other criterion for result ranking.

As we have seen, the search and retrieval of software tests can be improved by the application of value-based searches. Especially the fact that the dependence on names can be omitted is a major improvement. Nevertheless, the approach also relies on historic execution data and can therefore be classified as a hybrid of structural and behavioral searches.

The following section introduces *code-based searches*, which extends the previously introduced search strategies with dynamic execution to searches.

³ In the current context we apply the term *value-based searching* to both kinds of searches, i.e., with and without regular expressions.

7.2.3. Code-Based Searches

Until the end of the 1990s the software reuse community developed a large set of methods for the retrieval of reusable software components [MMM98]. However, researchers in this area were not satisfied with the rather low precision of the existing search techniques and engines [HJA07]. Over the last decade, one of the most promising new approaches for improving precision was based on the idea of executing reuse candidates and evaluating their fitness for purpose using software tests [Lem+07; HJA08; Rei09]. Interface-based searches for software tests can be regarded as equivalent to the same kind of searches in code reuse. The introduction of value-based searches for software tests represents a potential improvement to this approach, but we have also seen that this technology still relies on static information.

Therefore, the idea of *code-based search* for test reuse, which represents a kind of symmetric technology to *test-driven search* in classic software reuse, is to exploit the executability of software tests to improve the quality of the result set. While test-driven search uses test cases as queries and evaluates the retrieved reuse candidates by executing the test cases, we are going to use the actually existing implementation of a developer to evaluate potentially reusable test cases.

In contrast to test-driven reuse, in code-based searches the test search engine regards the class under test as a supplementary part of the search process. After an interface- and / or value-based search has been performed, the potentially reusable tests are evaluated against the class under test. Since the CUT is an untested potentially buggy piece of software, the results of this evaluation have to be treated only as an indication of the test's fitness for purpose; they are by no means an evidence for it. In this discussion, we consider the following possible outcomes of a CUT-based result candidate evaluation:

1. The CUT passes none of the tests of the candidate test case.
2. The CUT passes a subset of the tests of the candidate test case.
3. The CUT passes all tests of the candidate test case.

In case that a CUT fails on all tests contained in a test case, it is very likely that the CUT's problem domain is different to the test case's domain. This is, however, only an assumption and at first sight one might challenge it and state that the failing tests reveal serious problems in the CUT like, for instance, a problem at the very start of the program. Nevertheless, if the search returns a couple of test cases and all others contain at least some passing tests, the assumption of a totally broken CUT becomes weaker. Hence, we consider the first criterion to be a "rule-out" criterion for reuse candidates, although this may lead to so-called "false negatives" under some circumstances.

While users of test-driven search engines consider themselves fortunate when a reuse candidate passes all the tests of the query's test case, in our case this is not the most desirable situation. Although it might be satisfying for developers to see that their code passes other people's tests, the goal of tests is to discover faults and not to prove that a program is free of bugs [Mye79].

Therefore, the second category contains the most interesting candidates. A significant number of successful tests is a good indicator that the candidate test case fits in to domain of the CUT, while the failing tests are those that can add new value to a developer's test cases. When a test from a reusable test case fails during execution, the chances are high that it has discovered a bug in the class under test. Thus, the affected code needs to be inspected by the developers who have to revise the potentially faulty code and fix it accordingly.

Naturally, it may also happen that a failing test is faulty itself and that the class under test has returned a correct result. However, if we assume that the tests contained in the repository are created by professional testers and domain experts, this should be the exception rather than the rule. Moreover, in this case the reuse of software tests offers the opportunity to improve the quality of existing projects: a faulty test should be corrected upon discovery and the new version propagated to other projects using this test. Finally, the faulty test also needs to be updated in the search engine's repository so that it is not returned in future searches.

7.3. Retrieval of Exception Tests

When describing how JUnit test cases are analyzed, we explained how exception tests are recognized and relevant information is extracted from them. Since the exception tests are strongly related to their original class under test, they are included in the results of any of the previously introduced retrieval techniques. Therefore it is not necessary to discuss any special kind of additional retrieval mechanism, but there is one important aspect that needs to be addressed. As we have seen earlier in this thesis, an exception test verifies whether the call of a method of the class under test results in the program throwing a pre-defined exception.

The example in Listing 6.9 on page 131 explicitly required the CUT to throw a RomanNumeralException. Although the specified exception is usually contained in the source code repository from which the reusable test was obtained, it is doubtful whether developers want to incorporate a potentially large number of foreign exception types instead of their own programs, since they might have already written their own exception types for their class under test. Hence, we consider it more reasonable to deliver exception tests at a higher level of abstraction. Instead of testing for a specific exception type, we refactor the test and require it to expect the generic Exception type, which is inherited by any individual exception class.



```

EXCEPTION TEST-SHEET

RomanToArabicNumeralsTranslator

block           exception1
{
    assertEquals(1, translator.translate("ih"));
    fail();
}
  
```

Figure 7.5.: An Exception Test in SENTRE.

Nevertheless, it is not necessary to hide the identity of the original exception type from users. If they want to deploy the foreign exception type within their

project, they should still be able to reuse the exception test in its original form. Figure 7.5 shows a screenshot of SENTRE presenting an exception test for a Roman numeral converter that verifies that invalid input is rejected properly.

7.4. Test Reuse Process

In Chapter 5, we introduced and described a process for software reuse. This process outlines the major steps involved in software reuse in general. At this point, we want to review the steps from the decision to search and the description of the subject of the search up to the selection of a reusable asset in the context of the reuse of software tests. Therefore, the fundamental process of acquiring a test recommendation is outlined in Figure 7.6.

Basically, the process begins once the developer has written a class and starts to write tests (1) that, as well as implicitly describing how to test the class, also describe its intended behavior (i.e., the developer provides a syntactic and a semantic description of the system under development). Subsequently the developer performs a reverse search using the test search engine, whereas “reverse” means that it does not look for components providing a specific interface, but returns those tests that require a similar interface to the one declared in the query, i.e., provided by the class under test (2).

The system judges their fitness for purpose (3) and those tests that pass this step represent possible test recommendations which are subsequently ranked (4) and delivered to the user. If necessary, the involved interfaces can be dynamically adapted using the technology we introduced in earlier (cf. Section 4, pp. 55). Finally, the user inspects the recommended set of tests and decides which test(s) should be reused in the current project (5). By reusing a test and abandoning other recommendations, developers provide valuable feedback to the system, which can be automatically analyzed and used to improve the backend’s evaluation algorithms and influence future result ranking.

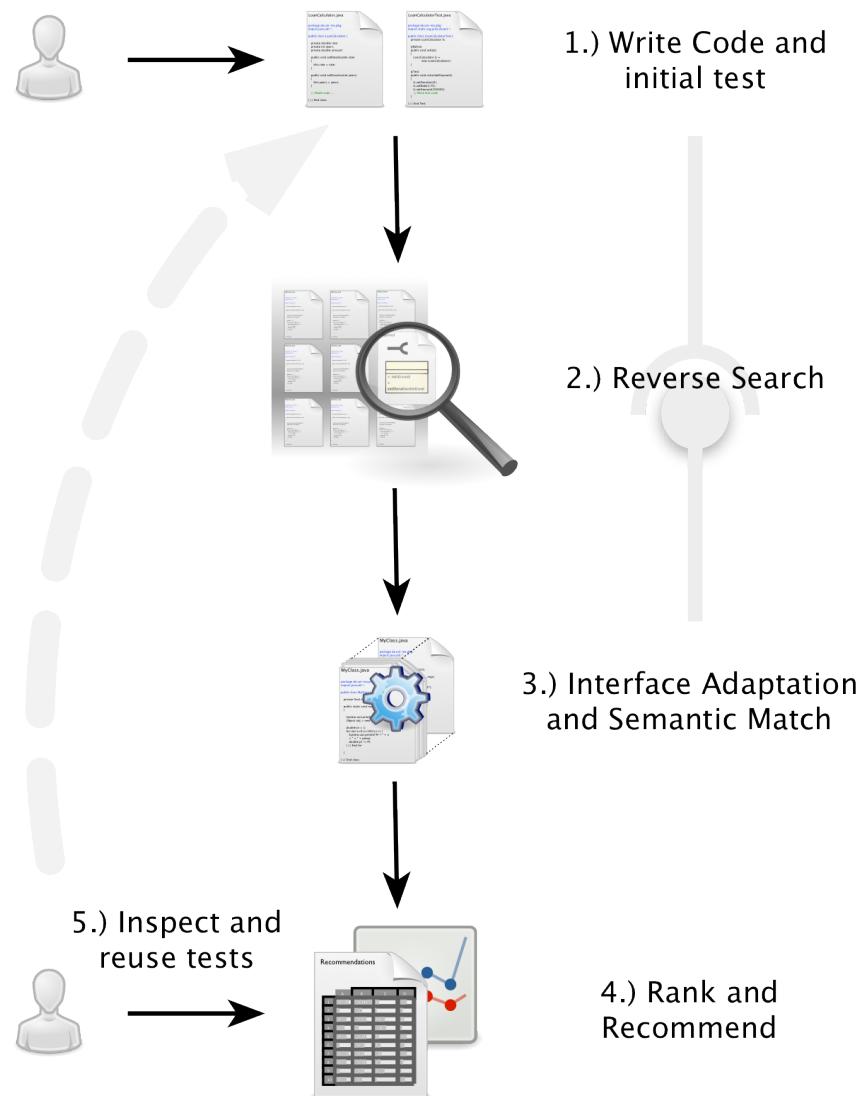


Figure 7.6.: Micro-Process of Test Reuse.

7.5. Implementation

Throughout this thesis we have referred to SENTRE which embodies the results of our theoretical considerations and practical experiences from research on the reuse of previously created software tests. Although we have given insights on its implementation at various points, it is necessary to briefly describe the architectural design of SENTRE and mention some facts about the current implementation. Naturally, the development of such a sophisticated tool in a completely new area of research is always a challenge and demands a lot of endurance and patience during its emergence. From the initial investigation of technologies and APIs, appropriate database implementations and finally tool-assisted test reuse in the IDE a lot of difficulties had to be overcome. Not only the information extraction from JUnit tests sometimes ended with disappointing results, but also the server-side implementation contained many pitfalls like the problems arising during the search for reusable assets within nested arrays in a MongoDB collection. Having said that, we do not want to go into all details and pitfalls of the implementation, but rather present “the big picture”.

Some technical information about SENTRE should make it easier for future researchers, who want to follow-up on our work, to build similar and enhanced systems. For the current parser, we have chosen to utilize the Java Compiler API⁴ which provides a set of tools to access the abstract syntax tree (AST) of Java files. With its TreeVisitor it provides an easy way to inspect a given Java file, based on the concept of the visitor pattern [Gam+94]. The information extracted by the test parser is stored in a MongoDB collection, which provides a convenient way to store documents with dynamic structures and offers fast index-based searches. Furthermore it supports searches with regular expressions, which made the translation of search queries much easier.

The SENTRE search engine is implemented using the Java Enterprise Edition and deployed on a JBoss 7.1 application server running on a Linux operating system. The underlying hardware (as of April 2014) comprises an Intel Core i7-2600 CPU

⁴ <http://docs.oracle.com/javase/7/docs/jdk/api/javac/tree/com/sun/source/util/package-summary.html>

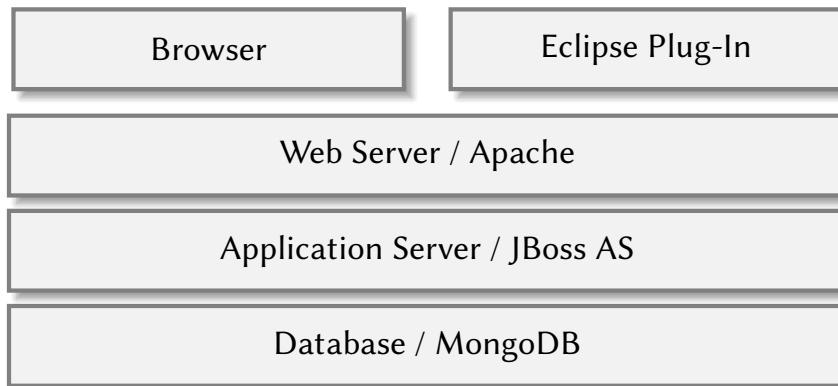


Figure 7.7.: System Architecture Sketch of SENTRE.

with 3.40 GHz, 16GB of RAM available and approx. 20TB harddisk space for the system and the code repository. To communicate with external client software, we have implemented a web service using the capabilities of the JBoss AS. All access to the application server is performed via an Apache server using the AJP module. SENTRE is therefore based on a multi-tier architecture, where the services are separated and encapsulated.

The architecture of the reuse-assisted test recommendation system is visualized in Figure 8.3, which shows the different tiers of the system. The client tier, containing the Eclipse plug-in, will be the subject of the following chapter. Although SENTRE offers a browser UI, we have already discussed in Chapter 5 that an appropriate IDE integration is essential for effective utilization and adoption of software search engines by developers.

7.6. Summary

At the beginning of this section we discussed and identified typical scenarios for using dedicated test search engines.. We identified potential use cases for these systems and discussed them in the context of the traditional phases of the software development lifecycle. Subsequently, we introduced a set of complementary retrieval techniques that can be used to search for previously written test cases, based on the well-known idea of using interface descriptions of the class under test. Furthermore, we introduced the idea of value-based

searches that rely on a user's specification that describes the mapping of test case values to their corresponding expected results. With the introduction of regular expressions as an extension to value-based searches, more general queries can be specified and the values can be described as patterns.

While the preceding retrieval techniques rely on static content analysis, the description of code-based searches introduced dynamic evaluation of reuse candidates to test search engines. Based on the application of potentially reusable test cases to the user's class under test, the system can identify and rank valuable tests. After describing the retrieval of exception test, the chapter concluded with an overview of the architecture and current implementation of SENTRE, our search engine for reusable software tests.

In the following chapter we are going to describe the implementation of an Eclipse plug-in for test reuse that automates the search for reusable software tests, integrates it into the developer's IDE and makes the whole process more user friendly.

Contribution of this chapter

- We have discussed and identified a set of archetypal usage scenarios for dedicated search engines for software tests.
- We have presented interface-based searches for test reuse and an approach to automatic query refinement and ranking.
- We have discussed searches for reusable software tests based on mappings of test case values to expected results.
- In addition to value-based searches, we have enhanced this retrieval technique with a pattern-based value specification using regular expressions. This strategy potentially improves the recall compared to plain value-based searches, where the test case values and expected result must be a perfect match.
- We have introduced the idea of code-based searches for software tests that can be used for dynamic result evaluation of reusable software tests.

- Based on our considerations of a software reuse process, we have introduced a micro-process for the reuse of software tests.
- We have described the system architecture and implementation of SENTRE, our search engine for reusable test data.

“ So is it with programming and bugs:
I have them, you have them, we all have them –
and the point is to do what we can to prevent
them and to discover them as early as possible.”

Software Testing Techniques [Bei90]

BORIS BEIZER, Software Engineer

8

Reuse-Assisted Test Recommendation

Software search engines are only as good as the underlying technologies used to drive their repositories, the parsers used to build their indexes and the retrieval algorithms used to search for results. However, in Chapters 3 and 5 we learned that although good solutions to these challenges are necessary for a successful search engine, they are not sufficient. A successful search engine must also provide a user friendly interface and environment that automates the process of reuse to the greatest extent possible. Over the past decade a lot of research has been conducted in the area of software search, but only a minor proportion has focused on the requirements for reuse-oriented recommendation systems in software engineering.

Based on our work conducted on *Code Conjurer* [Jan07; HJA08] and on the characterization of similar tools [JHA14], in this chapter we introduce a reuse-

oriented test recommendation system for Eclipse. The plug-in acts as a client to SENTRE and utilizes its capabilities to provide on-demand test recommendations [JA13; Erl13]. First, we discuss the requirements for a reuse-oriented test recommendation system and outline the general process of tool-supported test reuse. Subsequently, we describe the design and implementation of the tool, before we present some usage examples of our prototype implementation accompanied with a couple of screenshots.

8.1. Characteristics

Earlier in this thesis we discussed the general characteristics of so-called reuse-oriented code recommendation systems and presented our findings also in a chapter of the book on *Recommendation Systems in Software Engineering* [JHA14]. This chapter will therefore focus on adapting such systems for test reuse. Our goal is to focus on their implementation and how the idea of a reuse-oriented test recommendation system can actually be realized.

As already mentioned, the implementation of a recommendation system for reusable software tests needs to be seamlessly integrated into the development environment of its users. In order to minimize the barriers to its use, the system needs to “feel” familiar to developers and needs to be non-intrusive. If a recommendation system for software tests disturbs the workflow of the user by continuously demanding attention users may quickly become annoyed and deactivate or remove the system. Hence, it is necessary that the recommendations offered by the system are well integrated into the IDE and are ready *on demand* whenever the user desires them.

Therefore, like the reuse-oriented code recommendation system Code Conjuror [Jan07; HJA08], a recommendation system for software tests needs to implement an autonomous background agent which continuously monitors the developer’s actions. More specifically, it needs to inspect the test cases associated with the class under development and unobtrusively spring into action, when

the user edits them. Autonomous background agents are therefore a central part of our test-reuse environment that we will introduce below (cf. Section 8.3).

Based on our findings in Chapter 5, we emphasize that a recommendation system for reusable software tests needs to fulfill the following requirements:

Proactive service The test-reuse environment needs to constantly monitors the users' testing activities and becomes active when a test case is opened in the editor. The system autonomously decides when to trigger a search for potentially reusable assets, i.e., when to start the process depicted in Figure 8.1. The user should not be aware of the system's activity and should not be diverted from their normal workflow in any way.

Context awareness In order to work appropriately, a reuse-oriented test recommendation system needs to be aware of the developer's context. For example, since the JUnit assertions in a developer's tests usually contain just method invocations or literals as expected values, the type of the expected result is not obvious. Nevertheless, this information can be obtained from the class under test via its interface declaration.

IDE integration The test-reuse environment should provide seamless IDE integration. In other words, the users should not need to learn any new concepts to learn it and should become familiar with the system with as little effort as possible. For example, a single assert statement can be recommended using the auto-complete feature of the IDE.

Candidate evaluation Recommendation systems are only useful, when they provide valuable results to their users. This applies even more to reuse-oriented systems, since they need to tip the *make or reuse* dilemma in favor of reuse. Hence, the test-reuse environment needs to evaluate the potentially reusable tests and create a ranked list with the potentially most-useful recommendations on top. As a failing test potentially represents the discovery of a bug in the CUT, these tests should be ranked first, but nevertheless also inspected carefully.

Ready-on-demand The evaluation of tests can potentially be time consuming. In order to be useful for developers, a test-reuse environment must be re-

sponsive and provide results as quickly as possible. Therefore we utilize the idea of speculative analysis [Bru+10] and perform background evaluation of potential results even before the user requests them.

Feedback evaluation The system needs to examine whether the user was satisfied with its recommendations. Since recommendations are usually ranked using a weight function, the system can adjust the corresponding weights for future result evaluation and ranking.

8.2. Process Outline

Before we come to the actual implementation of our reuse-oriented test recommendation system for the Eclipse IDE, we define the process for tool-supported test reuse based on the outline depicted in Figure 8.1 (i.e., we identify the necessary actions and components, as well as the moment when they have to be executed).

The process of tool-supported test reuse comprises eight steps, from which the main portion is carried out automatically in background. At the beginning of this process, we consider a developer who is writing code and/or tests in the IDE, as well as a recommendation system that constantly monitors the editor content (1) and triggers either a search for reusable software tests or performs a re-evaluation of the results of a previous search. The autonomous decision to search is mainly driven by

- the change of an interface-defining part of the class under test,
- the change of the order of the tests in the editor, or
- the addition of a new test or removal of an existing test.

Thereby, the last aspect also covers the modification of an already existing test in the developer's project, since a change can be achieved through a removal and an addition.

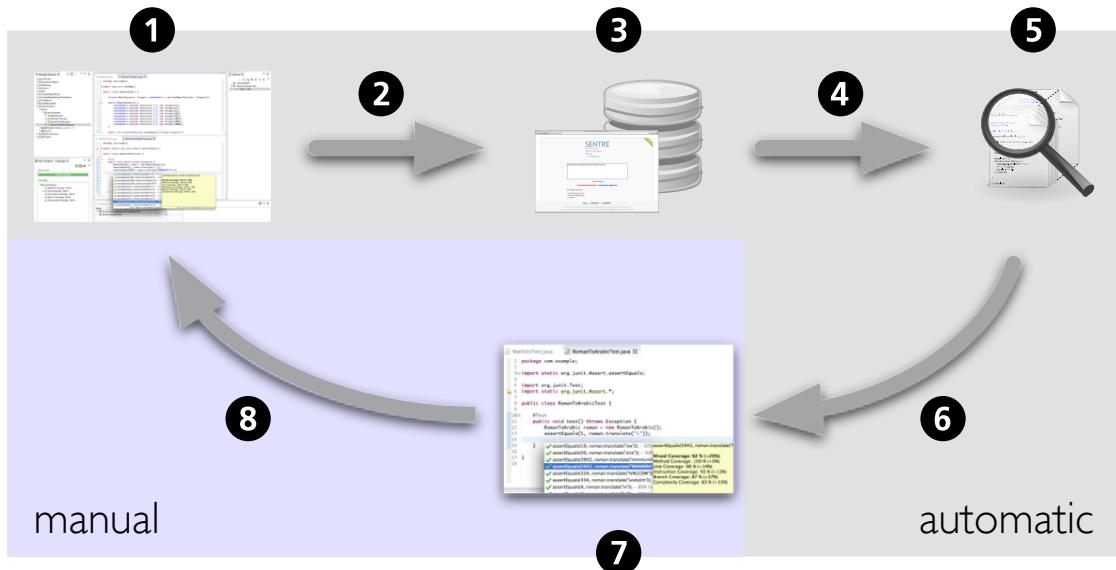


Figure 8.1.: Process of Tool-Supported Test Reuse. The parts of this process, which are carried out automatically (1–6), are highlighted with a light gray background color, while the manually performed actions (7 and 8) are blue-shaded.

When the interface of the class under test changes, a new search might reveal new reusable test cases and therefore it needs to be triggered by the system. If the order of the tests changes, this may merely be related to a different state of the tested object so is not necessary to perform a new search (if the required interface and utilized values are the same). It is only necessary to re-evaluate the previously retrieved results in the context of this possibly new state. When the system triggers a search for reusable tests, it formulates appropriate queries and sends them to the test search engine (2). Subsequently, the search engine performs a search and returns a set of potentially reusable tests (3).

After an initial server-side evaluation of the results, they are retrieved by the recommendation system for further processing (4). Usually, code recommendation systems tend to provide lists of possibly reusable assets to developers, from which they have to choose the right one for their task in hand. This is a very tedious duty since it involves the manual inspection of the provided results and a judgement for each of them, whether they add value to the actual project, i.e., support the developer's work. In the case of test reuse, this inspection corresponds to the investigation of whether a test improves the effectiveness and

quality of the test case under development by, for example, discovering a bug or at least increasing the level of test coverage.

To enhance the value provided by recommended systems for reusable software tests, our system automatically evaluates the candidates' fitness for purpose using a technique introduced by Brun et al., which is called *speculative analysis* [Muş+12a; Muş+12b]. The approach envisages to utilize unused computation power to examine future development states of a system in background in order to support the user's decision process, which alternative is appropriate in a particular situation. In our system, a background service adds the retrieved reusable tests to the test suite of the project under development and examines the outcome of their execution (5). Based on the results of these "dry runs", the search results have to be ranked by their context relevance, which is influenced by the following aspects: a) whether a test fails and potentially discovers a bug, b) how it contributes to coverage metrics, and c) the kind of test, i.e., whether it is the mapping of test case values to an expected result or an exception test. The background process prepares all this information for the test recommendation system (6), which displays the ranked and reusable tests to the user upon request (7). After a user has requested the recommendations, he or she inspects the recommended tests and chooses the most appropriate for the given context. Based on the user's choice, the recommendation system integrates the selected test (8) into the developer's project, including any adapters that were necessary to execute the test on the software under test. If the user has chosen a result other than the topmost in the list of recommendations, the system should investigate whether other values for the weights used in the ranking algorithm would have assessed this result as the most appropriate. This investigation enables the system to adjust to the user's preferences and therefore to create better and more user-centric recommendations in the future.

8.3. Implementation

Driven by the ideas developed for recommending code for reuse, our goal is to suggest useful test cases to developers and help them write better tests for the

software they are developing. This is only helpful, of course, if the application of such an approach requires less effort and time than the original approach. Therefore, as with traditional code recommendation systems, it is important that such a system does not require developers to significantly change their traditional behavior while creating software. It should avoid generating further overhead by demanding developers to write any additional specifications or learn any new query languages. Thus we extract all necessary information from the context of the code under development – including the main functional software that will be part of the final product – and the test cases that will be used to test it.

8.3.1. Eclipse Plug-In

In this section we introduce our Eclipse plug-in for the reuse of JUnit software tests, which realizes the previously mentioned requirements for reuse-oriented code recommendation systems in general, as well as those identified specifically for the reuse of software tests. The system relies on the software test search engine SENTRE, which we described earlier in this thesis. The screenshot in Figure 8.2 shows an example of our plug-in, while it is recommending reusable test cases acquired from the SENTRE search engine. As the screenshot of the plug-in shows, the tool seamlessly integrates into the Eclipse development environment and although it adds two additional views to the IDE, the recommendations are non-intrusively integrated using the auto-complete feature of the Eclipse editor. This approach makes access to the recommendations as easy as a keystroke. Moreover, they are presented using a familiar layout and users can discard them very easily, since the recommendations disappear while the users continue to type in their own code.

The plug-in architecture, as depicted in Figure 8.3, basically consists of a *Background Agent*, a *Communication Stack* and an *Analyzer*. The Background agent is aware of the project context of the test under development and uses different heuristics to determine the identity of the class under test. These heuristics are similar to those that we described earlier in Chapter 6.2 and were required

to develop a parser for test cases. Hence, our system is capable of performing name-based searches by attempting to match the name of the test to the name of a class included in the same package as the test.

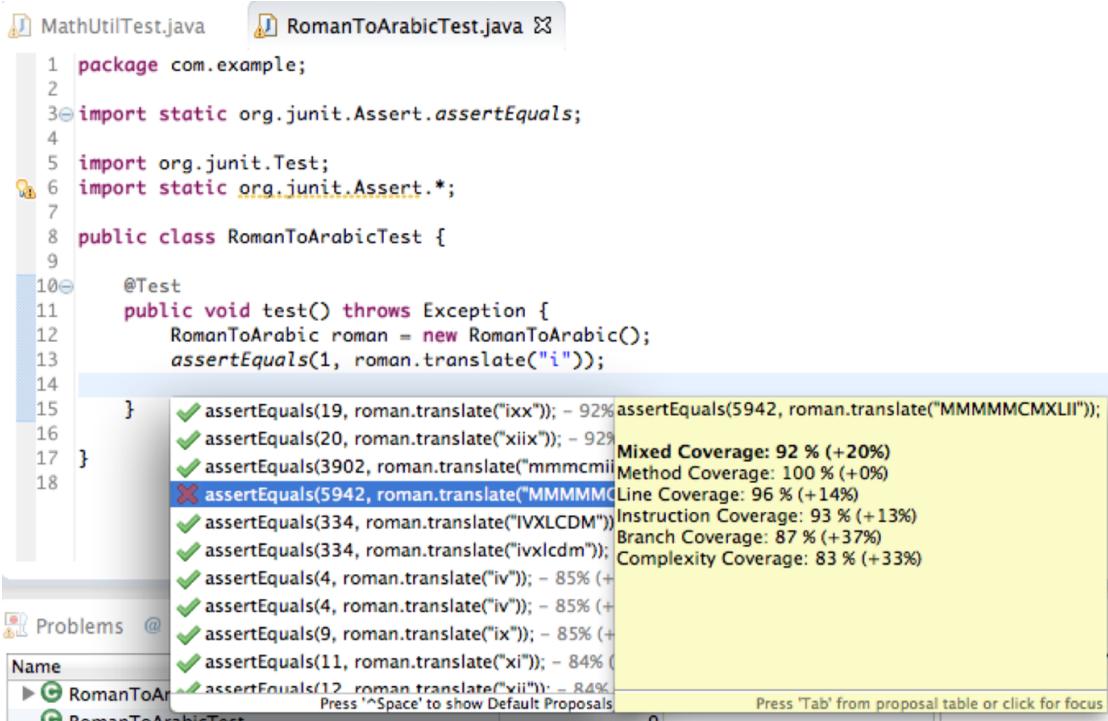


Figure 8.2.: IDE Auto-Completion for Testing Single Operations. Upon request, the plug-in displays a list of test reuse recommendations and additionally presents the corresponding coverage contribution of the selected result.

Preliminary experiments on our initial data set containing 65,003 test files revealed successful name matching for 56,930 cases (i.e., $\approx 87.5\%$ of the time). In cases where this is not successful, the system eliminates all standard imports from the JDK (since we assume that developers usually do not try to test the standard toolkit or, e.g., Object) and tries to identify the class under test amongst the remaining artifacts in the project.

If the class under development is recognized the system displays it in a small field. If not, the user has the possibility to intervene and guide the process.

During the development of a test, the background service constantly monitors the code input by the user, extracting all method invocations on the class under test and storing the invocation parameter tuples $(\alpha_1, \dots, \alpha_n)$ along with the return value Γ .

Based on the gathered information the Background Agent triggers a search via the Communication Stack in order to retrieve reusable test cases. The Communication Stack is basically a web service client, which incorporates all the logic needed to communicate with SENTRE and to use the retrieval techniques described earlier in Chapter 3 of this thesis.

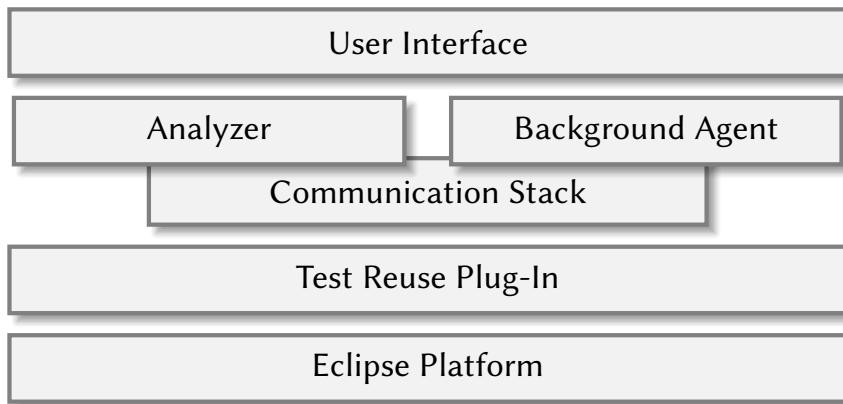


Figure 8.3.: Layered Architecture Schema of the Eclipse Plug-In.

Upon result retrieval, the Analyzer performs a speculative analysis in which it evaluates the potentially reusable tests. Therefore it appends each test to the test currently written by the developer and executes the newly created test case in the context of the whole project. Basically, there are three possible outcomes according to which the list of recommendations is created to provide as much value as possible. The two main factors involved in evaluating a result's fitness for purpose are (a) whether its is able to discover a potential bug and (b) how it contributes to the overall test coverage.

The tests retrieved and evaluated by our Eclipse plug-in also contain exception tests, and these are recommended to the developer where appropriate. The

ranking of the recommendations is performed in the Analyzer, which utilizes the JaCoCo¹ framework to determine a reusable test's contribution to the future development state of the test under development. This is done by evaluating various coverage measures, such as line and branch coverage. To create a user-centric ranking of the recommendations, users are provided with a preference page that allows the weights of the applied coverage criteria to be adjusted. If the user wishes, however, it is also important that the system can re-adjust these values automatically based on the chosen recommendation to improve future rankings. Therefore the results need to be re-evaluated using other weights and the calculated ranking has to be compared with the actual decision of the user.

8.3.2. Continuous Speculative Testing

Using speculative analysis, we have developed a novel approach for software testing. This approach goes beyond the traditional *ex post evaluation* usually applied in test development. It leverages in-background evaluation of reusable software tests by automatically applying them in the context of the user's test under development. This *ex ante* evaluation consequently foresees, that the system "knows" the contribution of available reusable tests to the quality and effectiveness of the developer's test case even before any of them is considered by the user.

With our approach, we enhance the idea of *continuous testing* [SE05; AO08], which originally envisaged the continuous execution of developed tests in the background to see if a performed change in the code has broken the system. We have combined continuous testing with speculative analysis and merged them in our approach to *continuous speculative testing* of reusable software tests.

Coverage Calculation for Recommended Tests

As already mentioned, with our Eclipse plug-in it is not only possible to receive test recommendations instantly, but with the help of continuous speculative

¹ a standard framework for coverage calculation: <http://www.eclemma.org/jacoco/>

testing, the system ranks and accompanies each result with additional information. This information illustrates the impact the application of the test would have to the overall future coverage quality of the developer's test case. The screenshot in Figure 8.2 shows this information appearing in a yellow box next to the result list. The box displays each kind of coverage supported by the system, the new coverage rate for each of these and the improvement achieved with the application of the selected recommendation.

If the user selects and applies a recommendation from the list, the background agent validates the test case and the system updates the coverage information view in Eclipse, similar to the screenshot in Figure 8.4.

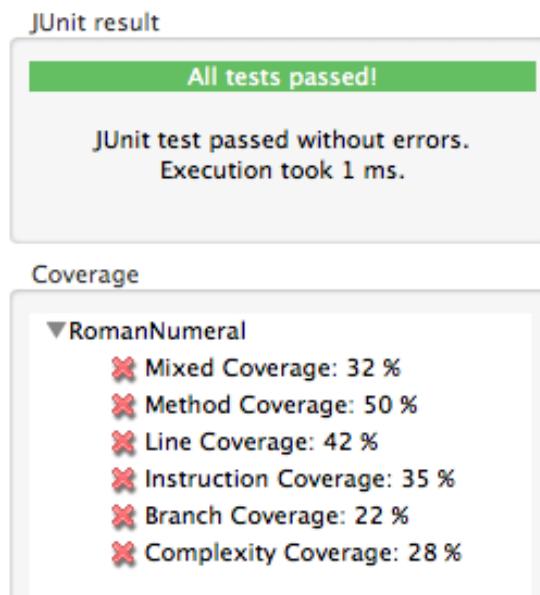


Figure 8.4.: View for Continuous Testing.

Thereby, the information in the coverage view is calculated based on the test case under development and is updated whenever a new test is added, even those manually written by the user. If a test case reaches a 100% coverage value for any of the given criteria, the symbol in front of the appropriate coverage method turns green. Additionally, with the information from the coverage view, developers do not need to repeatedly execute their test cases manually, since they are continuously informed about the test results by the system.

8.3.3. Exception Tests

As we have already seen, the SENTRE search engine supports the retrieval of exception tests from previously created tests, i.e., it offers the possibility to find reusable test data on which a system should fail with an exception. Naturally, our Eclipse plug-in incorporates this capability as well and offers exception tests in the list of recommendations. An example of such an exception test recommendation is shown in Figure 8.5, where the system has decided to recommend that a test be introduced to check that the system under development throws an exception on invalid input.

The screenshot shows an Eclipse code editor with Java code. A tooltip is displayed over the code at line 17, suggesting the creation of an exception test method. The tooltip contains the following text:

```
Creates an exception test method with the following body:
try
{
    converter.getArabic("e");
    fail("should raise a NumberFormatException");
}
catch (NumberFormatException e0)
{}
```

The code editor shows the following Java code:

```
1 package com.example;
2
3+ import static org.junit.Assert.assertEquals;...
4
5 public class RomanToArabicTest {
6
7     @Test
8     public void test() throws Exception {
9         RomanToArabic roman = new RomanToArabic();
10        assertEquals(1, roman.translate("i"));
11        assertEquals(5942, roman.translate("MMMMMCMXLI"));
12
13        assertEquals(0, roman.translate("VI")); - 92% (+)
14        assertEquals(67, roman.translate("LXVII")); - 92%
15        assertEquals(7, roman.translate("vii")); - 92% (-)
16        assertEquals(8, roman.translate("viii")); - 92% (-)
17        assertEquals(9, roman.translate("ix")); - 92% (+)
18    }
19
20 }
```

The tooltip also lists several other suggestion options related to exception testing.

Figure 8.5.: Exception Test in Eclipse.

As we can see, reusing exception tests is as simple as reusing existing tests that are expressed as assert statements. When the user requests auto-completion, the test reuse system shows retrieved exception test recommendations in the auto-complete list and displays a fragment of the original test case in the tooltip box. When the user decide to integrate the exception test into his own test case,

the test is integrated at the current cursor position, while the test's required interface is adapted to that of the test under development.

8.3.4. Algorithmic Outline

So far, we have discussed all relevant aspects of the implementation of our test-reuse environment (i.e., our Eclipse plug-in for reuse-oriented recommendation of software tests). To sum up our discussion, the general steps in the recommendation process are outlined in Algorithm 8.1.

Algorithm 8.1: Outline of Object-Oriented Test Recommendation.

```

 $prov \leftarrow$  provided interface of class under test ;
 $cand \leftarrow$  set of potentially reusable test cases ;
forall the test cases in cand do
  forall the object  $o$  in test case do
     $req \leftarrow$  required interface of test case for object ;
    if  $req$  matches  $prov$  then
      forall the method invocation on  $req$  do
        store mapping  $(\alpha_1, \dots, \alpha_n) \rightarrow \Gamma$  ;
        /*  $\alpha_1, \dots, \alpha_n$  is the in-parameter vector */ 
        /*  $\Gamma$  is the return value of the invocation */
      end
    end
  end
end

 $R \leftarrow$  result set from search for matching test cases ;
forall the  $r \in R$  do
  do speculative analysis for  $r$  ;
  rank result with respect to user-preferred weights ;
  store result ;
end

show recommendations in order of ranking ;
store user's choice and readjust ranking weights ;

```

Beginning with the provided interface of the class under test, the system triggers a search and obtains a set of potentially reusable test cases and inspects their required interface (i.e., it looks for instantiations of the class under test). If a matching object is recognized, all invocations to the CUT are stored along with the corresponding test case values and expected results. Subsequently, during speculative analysis the set of matching tests is applied to the class under test and the results are ranked accordingly.

Finally, the user chooses tests from the result set and integrates them into the test currently being written. With their integration into the new test, the process starts again from the beginning and the user can either manually add more tests or choose to integrate the next potentially valuable recommendation.

8.4. Summary

In this chapter we have introduced and described our implementation of a reuse-oriented test recommendation system, which is integrated into the Eclipse IDE. To define the requirements that such a test-recommendation tool should meet, we adapted our previous considerations about the requirements of reuse-oriented code recommendation systems in general to the context test reuse.

In our considerations, we have stressed that failing tests are the kind of reusable artifact users might be most interested in. Nevertheless, it is also clear that a failing test does not necessarily mean that a bug in the system under development has been discovered. It is also possible that the test itself is erroneous. There is, however, an even worse scenario for a reuse-oriented test recommendation system. It is possible that some of the recommended tests fail because they were intended to test code from a completely different domain. Hence, we need to find a mechanism that reduces false positive results to a minimum and therefore increase the quality of the recommendations presented to the user.

In the following chapter we introduce the idea of oracle-based filtering, which relies on the so-called approach of Search-Enhanced Testing. This approach is an enhancement to the ideas of *n-version programming* and *back-to-back testing*,

supported by *test-driven reuse*. With the help of automatically obtained test oracles, we will present an approach that bears the potential to effectively remove false positives from the list of search results.

Contribution of this chapter

- This chapter presented a definition of the main characteristics of a test-reuse environment.
- We have introduced a micro-process of tool-supported test reuse.
- We have described the application of dedicated evaluation and ranking mechanisms in a test-reuse environment, utilizing the ideas of continuous testing and especially focusing on the benefits of speculative analysis.
- Based on our earlier considerations, we have presented a working proof-of-concept implementation of a test-reuse and recommendation system as an Eclipse plug-in.

“ In summary, the claims that our critics did not get our results are unsupported and appear to be based more on wishful thinking than scientific analysis.”

A Reply to the Criticisms of the Knight & Leveson Experiment

JOHN C. KNIGHT & NANCY G. LEVESON

9

Search-Enhanced Recommendation Improvement

The occurrence of false-positive results during a search for reusable tests is a potential weakness that should not be underestimated. Although it is not possible to generally provide a quantification of how often false positives may occur, as the results of a search are the product of many variables, they are responsible for at least two undesirable scenarios: *a*) developers are forced to inspect recommendations that are totally useless for them losing time they could have spent in the development of own tests; *b*) unsuitable tests are incorporated into the quality assurance lifecycle of the system under development and may lead to wrong behavior of a test suite (e.g., falsely indicating defect components).

It is therefore an essential requirement that software testers and developers reusing tests always show a high level of responsibility and inspect what they reuse. For the success of a recommendation system, however, the benefits of

using it, have to outweigh the “costs”. In this chapter we consider possibilities for automatically removing *false positive results* by discovering discrepancies between expected results described in search results and “real” test results provided by test oracles. The filtering mechanism exploits the approach of *Search-Enhanced Testing* [AHJ11], abbreviated as *SET*, which we introduce in the following section.

9.1. Using Oracles in Software Testing

In the preceding chapters, we have discussed some of the main issues in software testing and explained how it is possible to benefit from earlier investment in software tests by reusing them.. Thus it is only natural to apply software testing as well to try to resolve the issue of false positive results appearing in a set of recommended tests. In this context we recall that software testing has always had the basic problem of finding an “oracle” that is able to define the expected result of a test [Wey82]. Despite the fact that there are techniques for automatically generating partial oracle information from formal specifications of systems (e.g., the pre- and post conditions of an operation) [AO08], this information is unfortunately often limited and the development of the required kinds of specifications is expensive. The reuse of already existing tests is also an inappropriate solution, as we would end up by trying to create a *perpetual motion machine* where reusable tests would be inspected by reusable tests recursively.

To address this situation we take a look at software engineering approaches that involve the building of numerous functionally equivalent implementations of the same software. This is mainly carried out in the domain of mission and safety critical domains, where redundant implementations can serve as fully automated oracles for a system and their use can help systems to recover from malfunction [CA78]. The strategy behind this idea is called *n-version programming* [Avi95] or *NVP*, and involves the implementation of a decision algorithm to produce a consensus result from the results delivered by the $n \geq 2$ versions implemented in a system. Incidentally, this approach has been the subject of an extensive academic dispute between Avižienis vs. Knight and

Leveson [KL86; KL90]. This dispute, however, does not affect the forthcoming considerations and we refer the interested reader to the original sources. For the sake of completeness, we give a short summary of the Knight & Leveson experiment at the end of this section.

In addition to Avižienis work on NVP, Vouk also describes the creation of $n \geq 2$ functionally equivalent versions of a program as a potentially cost-effective way of evaluating a software system's fitness for purpose [Vou90]. He calls this approach *back-to-back testing* which, as the “testing” in its name implies, is not a strategy to be applied at runtime like NVP, but serves as a means to discover faults during development time.

In the case of automatically assisted test reuse, it is not feasible to expect developers or testers to provide multiple implementations of the system under test, which would serve as comprehensive oracles for the inspection of reusable test candidates. After all, test reuse is about lowering the workload of developers and testers – it is not meant to impose additional effort. However, instead of expecting developers to create the multiple versions required, the idea behind Search-Enhanced Testing is to harvest oracles with next to zero effort from open source repositories, as described in our work published at ICSE and SUITE¹ in 2011 [AHJ11; Jan+11]. Although we do not need these oracles to validate the system under test (which can be done by using SET), we utilize the approach to leverage the reuse of tests by improving the quality of test recommendations and filtering unsuitable reuse candidates.

¹ ICSE workshop on Search Users Infrastructure, Tools and Evaluation (<http://resuite.org>)

9.1.1. Excursus: The Knight and Leveson Experiment

In this excursus, we give a short overview of the outcome of *Knight and Leveson Experiment* by summarizing [KL86]. For the experiment, which was performed jointly by the University of California, Irvine (UCI) and the University of Virginia (UV), 27 students with different backgrounds were told to program a simple anti-missile system. Their result should have been 27 different programs which behave equally to the same input.

The programs were tested by executing one million tests on them and the outputs (241 in total for every program) were compared to a 28th so-called “gold” program which was used to automatically determine the only correct answer to an input. According to definition of statistical independence

$$P(X|Y) = P(X) \quad (9.1.1)$$

the probability that there would be no program failures on a given test case would be

$$P(f = 0) = P_0 = \prod_{n=1}^{27} 1 - p_n \quad (9.1.2)$$

where f is the number of programs producing a fault on the same test input data and p_n the probability of failure for the n -th version. The probability that there was exactly one program failing is described by the following equation:

$$P(f = 1) = P_1 = \sum_{n=1}^{27} \frac{P_0 p_n}{1 - p_n} \quad (9.1.3)$$

The overall goal of the experiment was to show that there are faults that occur in two or more equivalent implementations of a program simultaneously. More precisely, Knight and Leveson were interested in the probability of more than one fault per test input. Using the following equation, they describe this kind of probability:

$$P(f \geq 2) = P_{more} = 1 - P_0 - P_1 \quad (9.1.4)$$

For one million test inputs, it happened 1255 times that more than one of the programs failed on the same input data. Although Knight and Leveson do not describe the nature of all of these faults, they state that it is more important *that* a failure occurred than *why* it occurred.

If f was the number of failing programs on the same test input data and n the number of overall test runs, under the hypothesis of independent failures, there would be a binomial distribution for f with parameter P_{more} and since the set of executed tests was large enough, they could use a normal approximation to the binomial distribution. As a consequence they state that

$$z = \frac{f - n \cdot P_{more}}{\sqrt{f \cdot P_{more} \cdot (1 - P_{more})}} \quad (9.1.5)$$

has a distribution that is closely approximated by the standard normal distribution. With the data derived from the experiment ($f = 1255$ and $n = 10^6$; P_{more} was not published), they calculated a z -value of 100.51, which is far more than 2.33 that represents the 99% value in the standard normal distribution. Thus they rejected the null hypothesis that the above model was correct with a confidence level of 99% meaning that the model was wrong due to the assumption of independence. Therefore the assumption of independence had to be rejected.

In the context of this thesis, the findings of Knight and Leveson therefore mean that Search-Enhanced Testing is a technique that supports testers but still demands manual effort, i.e., developers and testers should not solely rely on automatically derived or reused software tests.

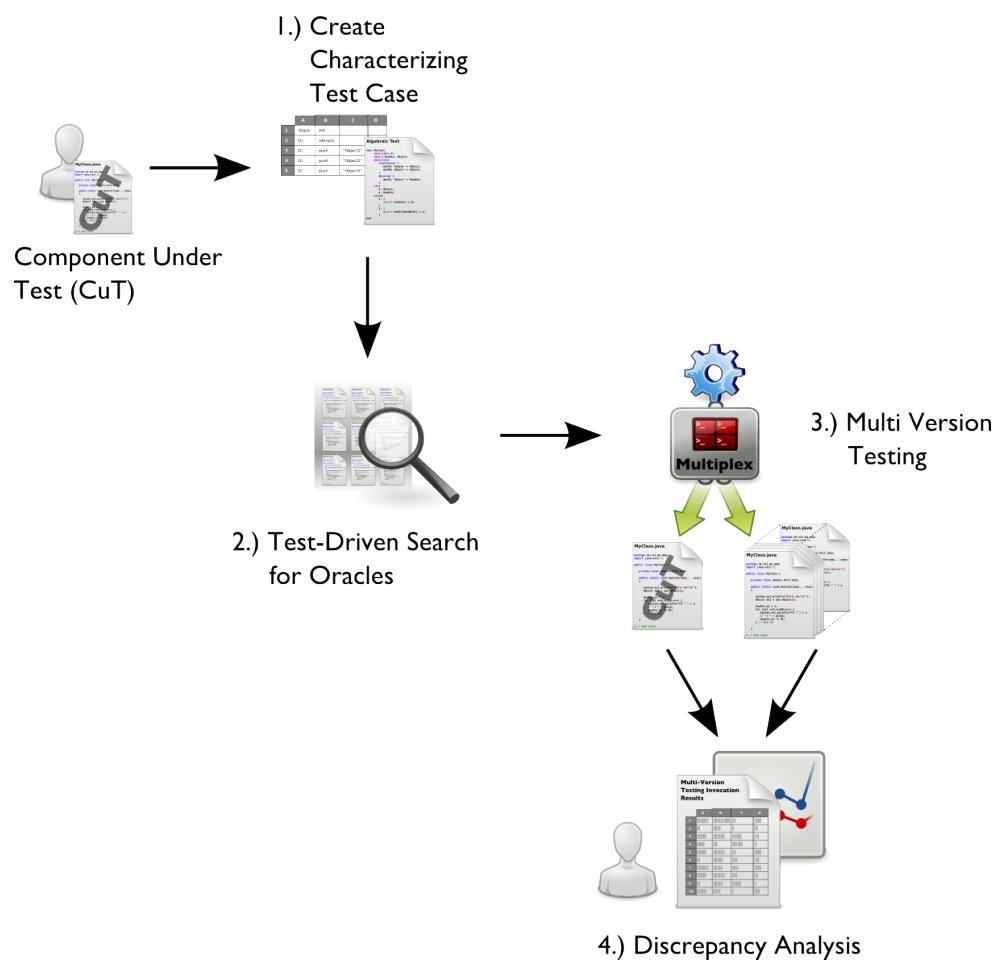


Figure 9.1.: Process of Search-Enhanced Testing.

9.2. Search-Enhanced Testing

One of the key obstacles to NVP and back-to-back testing are the high costs involved in creating multiple equivalent implementations of functionally equivalent (critical parts of a) system. Nevertheless, the advent of specialized, internet-scale code search engines and their ability to find reusable assets based on the specification provided by test cases has changed the situation. These search engines are the basis for test-driven search, which is the key enabling technology for Search-Enhanced Testing and the process depicted in Figure 9.1. The particular actions performed during each step can vary slightly, depending on the desired balance of automated versus manual effort in the test evaluation processes. In the following subsections, we will explain the outlined steps in greater detail, before we switch our attention to the application of SET in reuse-assisted software testing. A copy of our poster, which was presented at ICSE 2011 and which describes the process of SET, is included as Figure A.1 in appendix A of this thesis.

Characterizing Test Case

As Search-Enhanced Testing depends on *test-driven reuse*, which by definition expects a test case as input to its process of searching for reusable software components² (as depicted in Figure 3.6), the initial step for Search-Enhanced Testing is to define a test case that “characterizes” the component under test – we will call this a *characterizing test case*³. In accordance with the definition of TDR in Section 3.4 this test is used to find virtually all classes with the desired functionality, yet discard those with similar interfaces but different functionality. With regard to the premise implied by TDR that the characterizing test is not meant to be a tool for bug detection but a behavioral description of the CUT, the effort involved in writing it should be far less than the effort involved in writing and composing normal tests using traditional testing approaches.

² By “component” we mean any cohesive and compact unit of functionality with a well defined interface. For Java this will typically be a class.

³ For consistency reasons and according to the definitions from Section 2.1 we will not use the terms used in the aforementioned publications, where it is called *characterizing test*.

Since the technology of test-driven search engines is still in its infancy, there have been few, if any, evaluations that clarify the criteria for how a test case should be written to define the “minimal” characterizing test case. A programming language agnostic method to do so would be to use test sheets [Atk+08b] or the use of algebraic specifications (see, e.g., [Som10]) as the basis for defining such tests. Algebraic specifications are a well known technique for writing relatively comprehensive, yet compact, black-box definitions of a component or system’s behavior and seem to satisfy the aforementioned requirements. The example in Listing 9.1 is an algebraic specification of a classic stack component, which fully specifies the externally visible properties of this data structure in terms of method relationships.

Listing 9.1: Algebraic Specification of a Stack.

```

TYPE Stack;
IMPORTS Boolean;
FUNCTIONS
    Stack : → Stack × void;
    push : Stack × Object → Stack × void;
    pop : Stack → Stack × Object;
    isEmpty : Stack → boolean;
AXIOMS
    ∀ s:Stack, o:Object
        (A) pop(push(s,o).state).retval = o;
        (B) pop(push(s,o).state).state = s;
        (C) pop(Stack().state).retval
            → ArrayIndexOutOfBoundsException;
        (D) isEmpty(Stack().state).retval = true;
        (E) isEmpty(push(s,o)) = false;
        (F) isEmpty(pop(push(Stack().state,o).state).state) = true;

```

The section named FUNCTIONS describes the signatures of the component’s operations in terms of their input and output types. The AXIOMS section defines the expected behavior for each operation in relation to another one. The first axiom (A), for example, defines the relationship between the pop and push function such that one pop is the inverse of one directly preceding push. A transformation

of this specification to JUnit can be performed with little effort and the resulting test case is illustrated in Listing 9.2.

Listing 9.2: Algebraic Specification as JUnit Test Case.

```
1 public class StackTest extends TestCase {
2     Stack s = null; Object obj = null;
3     public void setUp() {
4         s = Util.getStack();
5         obj = Util.getRandomObject(); }
6     public void testAxiomA() {
7         s.push(obj);
8         assertEquals(obj, s.pop()); }
9     public void testAxiomB() {
10        s = new Stack();
11        assertEquals(null, s.pop()); }
12    public void testAxiomC() {
13        s = new Stack();
14        assertTrue(s.isEmpty()); }
15    public void testAxiomD() {
16        s.push(obj);
17        assertFalse(s.isEmpty()); }
18    public void testAxiomE() {
19        s = new Stack();
20        s.push(obj); s.pop();
21        assertTrue(s.isEmpty()); }
22    public void testAxiomF() {
23        int before = s.size();
24        s.push(obj); s.pop();
25        assertEquals(s.size(), before); }
26 }
```

A test case like this would naturally be very limited in its capability to detect defects in a class under test and it would hardly meet the coverage criteria normally applied in practice as the *for all* quantifier actually used within the axioms of the algebraic specification is not transformed into *brute-force* tests that invoke the CUT with all possible (combinations of) inputs. Instead, they are each transformed into one single test. However, the results achieved by test-driven search indicate that for the purpose of finding multiple versions of a system such a test case is sufficient [HJA07; Hum08]. Furthermore, for the purpose of finding

multiple versions of a system that can be used for back-to-back style testing, it is not essential that every version used in the process be a perfect replica because the voting system ensures a reasonable result is obtained if some of the versions return spurious results from time to time. This can even be used to manually double-check those “suspicious” invocations that have caused dissent amongst the oracles and to ensure that the CUT executes them correctly as desired. We will address this issue in the subsequent sections on multi-version testing and result inspection.

Test-Driven Search and Multi-Version Testing

Once the characterizing test case is defined, a test-driven search enabled code search engine is used to find suitable oracles for *multi-version testing* (MVT). In our experimental work on Search-Enhanced Testing, we utilize the Merobase Component Finder [Jan+13] due to its capabilities in test-driven search, which include the search for reusable assets, automated interface adaptation and delivery of the adapter and adaptee class. Especially the delivery of necessary adapters is useful for search-enhance testing, since in many cases the characterizing test requires some other interface than the one provided by a semantically matching class. Since the process in Figure 9.1 envisages the human testers to be involved only during the specification of the characterizing test case and on result inspection, the adaptation of the oracles has to be done automatically.

The results obtained from a search for a stack, driven by the above test case derived from algebraic specification, are summarized in the first row of Table 9.1. This shows that from a total of 25,000 candidates found in the index (i.e., executable components with matching methods profiles), 656 functioning versions of a stack that passed the characterizing test were found.

In the second row the table shows how many functionally equivalent components are returned when a more elaborate test is used to drive the search. As expected, the number of results is lower because fewer components were able to pass the more stringent tests. However, the manual effort that is required to define such a test is much greater.

Technique	Candidates	Equivalent Versions
Algebraic Test Suite	25.000	656
Traditional Test Suite	25.000	454

Table 9.1.: Results of a Test-Driven Search for a Stack.

Multi-Version Testing with Random Test Case Values

In the introductory publication, the idea of discrepancy-driven testing⁴ envisaged different usage scenarios for the components “harvested” through test-driven search. One of these scenarios was to use the harvested components as a unified “body of knowledge” that together form an oracle that determines the “correct results” for randomly generated test case values. At the same time, the component under test is tested using the same test case values and, as depicted in Figure 9.2, its test results are compared to those delivered by this *pseudo-oracle* [Hum+06].

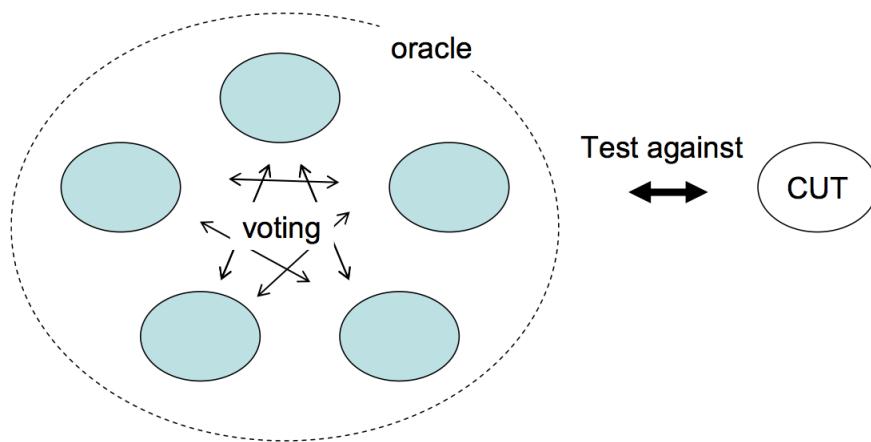


Figure 9.2.: Harvested Components as Oracle [Hum+06].

During the introduction of the technique, the authors of the original publication on discrepancy-driven testing mention that with large numbers of components from internet-scale repositories, it should be easy to improve on the findings

⁴ The terms *discrepancy-driven testing* and *multi-version testing* are used synonymously.

of the Knight-Leveson experiment [Hum+06]. Later experimentation with test-driven reuse [Hum08], however, showed that we cannot generally expect very large numbers of effectively reusable oracles. This leads us to conclude that the findings from Knight and Leveson can not be discarded lightly. Therefore in Search-Enhanced Testing, we do not aggregate the voting of the components into such a pseudo-oracle, but instead we record so-called voting profiles, which we will introduce in the subsequent section.

In general, the multi-version testing step has three sub-steps depending on the level of manual involvement that is desired in the final analysis phase of the process: *a*) the selection of a subset of the list of oracles retrieved with test-driven search, *b*) execution of discrepancy-driven testing, and *c*) the elimination of outliers and eventual replacement with unused oracles. The level of manual effort in the final analysis phase can range from virtually zero, when all judgments about results are made automatically, to relatively high, when all detected discrepancies are evaluated manually. Nevertheless, Hummel et al. already emphasized that it is impossible to completely trust the results generated during the voting process, which makes a certain amount of human inspection inevitable [Hum+06].

To illustrate the idea and to demonstrate our implementation of Search-Enhanced Testing, we refer to the illustrative example of a converter class that transforms an Arabic numeral into a Roman numeral. With the help of test-driven search the Search-Enhanced Testing environment is able to retrieve a set of suitable replicas (see, e.g., appendix B, Listings B.2 – B.7 for those used hereinafter) of the class under test (Listing B.1) and Search-Enhanced Testing continues with multi-version testing, i.e., the behavior / test results of the CUT and those of the n harvested replicas (i.e., oracles) are compared using techniques inspired by back-to-back testing.

Since Search-Enhanced Testing demands a high level of automation, we have implemented a system to demonstrate the viability of the approach using the technologies available today. Figure 9.3 shows a screenshot of our *Multi-Version Testing Environment* (MVTE), which enables users to match a class under test against a set of oracles. The system involves the automatic generation of a *test*

broadcaster service, which mediates calls from a test driver object to the test oracles and the CUT and logs the observed behavior (i.e., the test results returned by the test oracles and the CUT) for later inspection. In order to provide an interface which conforms to that of the component written by the developer (and also to the oracles), the broadcaster class has to have the same name and contain the same operations as the CUT, which can be achieved with automated interface adaptation (cf. Section 4). Any object that makes calls to the CUT and invokes its methods will consequently be able to perform the same invocations on the broadcaster, and such an object is called an *execution driver* of the broadcaster.

Invocation	Prof.	Ret...	CUT	O1	O2	O3	O4	O5	O6	O7
toRoman(1469)	FA	String	MCDLXIX							
toRoman(6877)	MIN	String	MMMMMMDC...	ERROR	MMMMMMDC...	ERROR	ERROR	ERROR	ERROR	MMMMMMDC...
toRoman(5850)	MIN	String	MMMMMDCCCL	ERROR	MMMMMDCCCL	ERROR	ERROR	ERROR	ERROR	MMMMMDCCCL
toRoman(2132)	FA	String	MMCXXXII							
toRoman(-1857)	MIN	String	String	ERROR	String	ERROR	ERROR	ERROR	ERROR	ERROR
toRoman(-2203)	MIN	String	String	ERROR	String	ERROR	ERROR	ERROR	ERROR	ERROR
toRoman(1044)	FA	String	MXLIV							
toRoman(4742)	MIN	String	MMMDCCXLII	ERROR	MMMDCCXLII	ERROR	ERROR	ERROR	ERROR	MMMDCCXLII
toRoman(-1881)	MIN	String	String	ERROR	String	ERROR	ERROR	ERROR	ERROR	ERROR
toRoman(6916)	MIN	String	MMMMMMCM...	ERROR	MMMMMMCM...	ERROR	ERROR	ERROR	ERROR	MMMMMMCM...
toRoman(3491)	MAJ	String	MMMCDXCI							
toRoman(4803)	MIN	String	MMMDCCCIII	ERROR	MMMDCCCIII	ERROR	ERROR	ERROR	ERROR	MMMDCCCIII
toRoman(908)	FA	String	CMVIII							
toRoman(5944)	MIN	String	MMMMMCMX...	ERROR	MMMMMCMX...	ERROR	ERROR	ERROR	ERROR	MMMMMCMX...
toRoman(4491)	MIN	String	MMMMCDXCII	ERROR	MMMMCDXCII	ERROR	ERROR	ERROR	ERROR	MMMMCDXCII
toRoman(3259)	MAJ	String	MMMCCLIX							
toRoman(4053)	MIN	String	MMMLIII	ERROR	MMMLIII	ERROR	ERROR	ERROR	ERROR	MMMLIII
toRoman(3255)	MAJ	String	MMMCCLV							
toRoman(3397)	MAJ	String	MMMCXXCVII							
toRoman(-1168)	MIN	String	String	ERROR	String	ERROR	ERROR	ERROR	ERROR	ERROR
toRoman(2097)	FA	String	MMXCVII							
toRoman(2382)	FA	String	MMCCCLXXXII							
toRoman(3752)	MAJ	String	MMMDCLII							
toRoman(2520)	FA	String	MMDXX							
toRoman(-2195)	MIN	String	String	ERROR	String	ERROR	ERROR	ERROR	ERROR	ERROR
toRoman(2374)	FA	String	MMCCCLXXIV							
toRoman(7132)	MIN	String	MMMMMMMC...	ERROR	MMMMMMMC...	ERROR	ERROR	ERROR	ERROR	MMMMMMMC...
toRoman(-1017)	MIN	String	String	ERROR	String	ERROR	ERROR	ERROR	ERROR	ERROR
toRoman(320)	FA	String	CCCXX							
toRoman(667)	FA	String	DCLXVII							
toRoman(5991)	MIN	String	MMMMMCMXCI	ERROR	MMMMMCMXCI	ERROR	ERROR	ERROR	ERROR	MMMMMCMXCI
toRoman(2718)	FA	String	MMDCCXVIII							
toRoman(-817)	MIN	String	String	ERROR	String	ERROR	ERROR	ERROR	ERROR	ERROR
toRoman(1604)	FA	String	MDCIV							
toRoman(4169)	MIN	String	MMMCCLIX	ERROR	MMMCCLIX	ERROR	ERROR	ERROR	ERROR	MMMCCLIX
toRoman(3890)	MAJ	String	MMMDCCXC							

Figure 9.3.: Multi-Version Testing of a Roman Number Converter.

The creation of the broadcaster involves several automated code generation steps. First the system has to integrate the CUT and the harvested test oracles into a package structure such that the CUT is stored in package p_1 and the n test oracles in the subsequent packages $p_2 \dots p_{n+1}$. The test oracles have to be available as binaries so that the broadcaster can instantiate them. However, to

be compliant with the interface of the CUT, the test oracles may need some adaptation before they can be used and compiled.

Our previously introduced implementation of an automated adaptation service [JA12] can be utilized in order to perform this task. After the test oracles and the CUT have been prepared, the MVTE generates a broadcaster class which maps any method invocations to the corresponding methods of the test oracles and the class under test. During this process of multi-version testing the returned values are collected and held in the test results logger. The complete source code of this broadcaster class can be found in Listing B.8 in the appendix.

Subsequently an *execution driver* starts making calls to the multiplexer, which forwards these calls to the oracles and the CUT. The execution driver should ideally contain a large number of (randomly generated) invocations to cover a variety of scenarios. Since the broadcaster handles distinct instances of the oracles and the CUT, their state is preserved and the execution driver can also perform a state-aware comparison of their behavior. This is especially useful for the inspection of stateful components, which are not commutative in terms of the order of their method invocations. An in memory integer-adder, for example, delivers different intermediate results on the calls (3, 5, 2) and (2, 3, 5): the former returns 8 after the first two inputs, while the latter returns 5 despite the fact that the third call leads both to the result of 10.

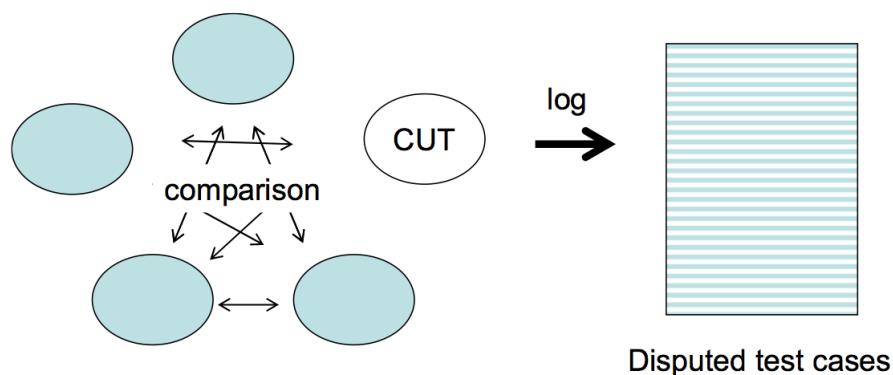


Figure 9.4.: Discrepancy-Driven Testing [Hum+06].

Obviously our approach makes it possible for the developer to not only execute the multiplexer from a specially written test, but it may also be deployed within the system for which the CUT is developed, so that Search-Enhanced Testing can be run in a *real-world* setting. With every invocation that the execution driver class performs on the broadcaster, a new row of data is created in the *invocation table* (see, e.g., Figure 9.3) that contains the returned values of the oracles and the CUT for the input parameters provided by the execution driver. These return values are stored in an XML file which can be processed by the multi-version testing environment to create a *discrepancy table* of the form shown in Table 9.3. The MVTE performs an analysis of the table and looks for discrepancies between the results returned by the oracles and the CUT. The discrepancies are classified by their discrepancy category, which allows a ranking of the discrepancies in the discrepancy table. As depicted in Figure 9.4, it is the disputed test cases which are logged for further human inspection and thus the multi-version testing environment highlights them.

Result Analysis

To obtain usable results from the “en masse” invocation of the previously described functionally equivalent components, the resulting discrepancy table needs to be analyzed. The results of any invocation are classified by an invocation profile. We will use the Roman numerals example to introduce these groups in Table 9.2:

#	CUT	Oracles					Name
1	M	M	M	M	M	M	Full Agreement
2	M	M	X	M	M	M	CUT with majority
3	M	X	X	X	M	X	CUT with minority
4	M	X	X	X	M	M	Draw
5	M	X	X	X	X	X	CUT alone
6	M	M	X	C	X	C	Disagreement
7	M	X	C	V	C	X	Disagreement & CUT alone

Table 9.2.: Invocation Profile Categories.

If all implementations, including the self-developed CUT, deliver the same result for a specific method invocation (this group is called “Full Agreement” in the table) there is obviously no discrepancy. This result is uncritical except in the exceptional case that all implementations are faulty and return the same false value to the same given test case values. Nevertheless, this situation is by definition out of scope of discrepancy-driven testing and not further discussed. Therefore it is, however, clear that Search-Enhanced Testing has to be applied in conjunction with other testing techniques, which help uncover such (inscrutable) bugs that spread among a large variety of implementations and may represent a common misunderstanding of the domain.

The other groups from Table 9.2 are all subject to further investigation, either (semi-) automated or manual, and have to be treated differently. Taking group 2, for instance, it seems rather uncritical if only one or a few (up until a certain threshold) of the components disagree from the majority and the CUT is with the majority. Nevertheless, there is no guarantee that the majority of the implementations returns the correct result and it is obviously sensible not to consider a single invocation separately from the whole set. Hence, it is best to perform a context-aware assessment of the test results: if there is, for example, one outlier that votes against the other implementations most of the time, it is very likely that it is an implementation of some other functionality that passed the test-driven search by chance. Therefore it might be advisable to remove this component from the set of oracles.

At the other end of the spectrum, with group 6 and 7, we find more interesting cases as well. It can be assumed that if several oracles in the pool deliver scattered results, the discrepancy-driven testing system has discovered a critical set of input values. Obviously, a number of programmers have come to different conclusions about how they should be processed and there is consequently a high risk that the implementation of the CUT may be faulty as well. Or in other words, it is certainly worth the effort of a human engineer to determine the correct outcome for this input and to check whether this has been delivered by the CUT as well.

Furthermore, it is theoretically possible that this invocation profile appears throughout the process, which would consequently lead to a manual investigation of every single invocation, which is not feasible. Even worse, this undesired situation could indicate that there has been a significant problem during the oracle selection process rather than the fact that the programmers of the harvested oracles have not understood the problem domain for most of the possible inputs. If it turns out that this is the case, the characterizing test case needs to be revised and the whole process has to be repeated.

Another special case is certainly the one where only the CUT has voted against all oracles (group 5, CUT alone). This can basically imply two things: either that the CUT contains a serious error and needs to be corrected or – and this is probably the more problematic case – that the CUT is correct, but the harvested test oracles are wrong, which leads to the same consequence as discussed before – it is necessary to revise the characterizing test case. It is, however, advisable to apply the same technique as mentioned before and automatically investigate the voting history of the CUT and the oracles, to determine whether this voting profile occurs repeatedly. If this is not the case, the result should simply be put into the discrepancy table for analysis by a human tester.

After the automatic process of analyzing the invocation table has finished and the identified discrepancies have been evaluated and ranked, the MVTE creates a discrepancy table which lists those invocations that have lead to discrepancies among the test oracles. In our example, the analysis reveals that the CUT is not implemented correctly. For the number 4000, which does not adhere to the rule that Roman numerals must not contain the same letter more than three times in a row, the CUT returns MMMM, although it should return an error (category 4 – CUT alone).

Therefore the system will flag this as a discrepancy so that the developer can inspect the results obtained for this component. The other interesting case is the one for –1492. Since the other oracles not only disagree with the CUT but also with each other as well, this is obviously an interesting test which has to be examined by the human tester. The problem relates to the fact that the negative numbers were introduced as early as the 7th century by the Indian

mathematician and astronomer *Brahmagupta*, which is more than a century later than the fall of the Roman Empire in AD476 [SO03].

When the process of discrepancy analysis has finished, the developer sees a table similar to Table 9.3 that shows potentially interesting tests that the developer can analyze further by hand. The invocations with the integers 1890 and 2320 are recognized as a vote from the first invocation profile group where all oracles agree. These are therefore regarded as less interesting test cases and would be removed from the discrepancy table presented to the human. The final step of the Search-Enhanced Testing process is the analysis of the information gathered in the discrepancy table by a human tester, when a final verdict on the outcome of the tests is decided. The human tester simply has to analyze the discrepancies and make a judgment about the results provided by the CUT.

Invocation	CUT	O ₁	O ₂	O ₃	.	O _n	#
toRoman(1890)	MDCCCXC	MDCCCXC	MDCCCXC	MDCCCXC	.	MDCCCXC	1
toRoman(2320)	MMCCCXX	MMCCCXX	MMCCCXX	MMCCCXX	.	MMCCCXX	1
...	
toRoman(-1492)	MCDXCII	ERROR	-1492	NaN	.	ERROR	3
...	
toRoman(4000)	MMMM	ERROR	ERROR	ERROR	.	ERROR	4
...	

Table 9.3.: Discrepancy Table for the Roman Numerals Example.

In other words, the testers act as final arbiters (i.e., the golden oracle) in cases where there is any doubt about the results. Our assumption is that this judgment process takes less effort than the writing of test cases using a traditional testing approach, or alternatively, is more effective at uncovering faults. This is because the effort of the human tester is focused on analyzing tests that have actually caused discrepancies rather than on trying to identify “high-quality” test cases using the “hit-and-miss” heuristics provided by *traditional* testing techniques.

Benefits for Test Reuse

A key feature of Search-Enhanced Testing is that the test data used to drive the testing process is generated automatically using random values and that no human intervention is needed. However, this also means that this approach is not able to use domain knowledge to select appropriate test case values but follows a kind of *brute force* philosophy. Nevertheless, the idea of Search-Enhanced Testing is valuable to the reuse of software tests, when it comes to the filtering of so-called *false positive* search results. In the next section we will therefore look at how to transfer the idea of executing multiple implementations of the same functionality and use it for the evaluation of previously written tests, i.e., to inspect their “fitness-for-purpose” to help developers improve their software and testers to find bugs.

9.3. Filtering False-Positives

Although the threats arising with test reuse are different to those related to the reuse of production code, it is nevertheless important to be aware of the pitfalls and issues related to it. Developers in “classic” reuse scenarios who attempt to exploit previously written components must ensure that they do not add malicious code to their system under development (which, while remaining undetected, might be critical in releases of the system), that the code does not introduce unwanted side-effects to the system under development (e.g., instabilities) and that the resource consumption at runtime is efficient.

Although all of these are naturally inherent to test reuse, an efficient system for test recommendations must primarily ensure that false positive recommendations are eliminated to the greatest possible extent. This not only helps build up confidence in systems for test reuse it also helps to reduce the effort involved in result inspection. The latter, in particular, helps to make the process less time consuming and tips the balance in favor of the “make” choice rather than “buy”.

To give an example of the occurrence of false positive recommendations, we refer again to the Roman numerals examples. Suppose a developer is using the test recommendation system within the Eclipse IDE and the reuse recommendation algorithm starts searching for reusable artifacts immediately after the first available test. In this case we have defined a test that checks whether the integer 10 is correctly converted to the Roman numeral X. When triggering a search for reusable tests, the system will deliver 19 results, from which 18 are reusable test sets for Roman numeral converters. There is, however, also one test case out of 19 which was intended to test a class for a board game and was just retrieved due to the fact that the search query contains the one single mapping

(10)->X; (9.1.)

The retrieved false positive result for this query is shown in Listing 9.3. It shows a test for a board game which just happens to map the value of 10 to an ‘X’ and therefore was retrieved as a reuse candidate.

Listing 9.3: False Positive Original Test Code.

```

1 @Test
2 public void shouldCopyItself() throws Exception {
3     board.populate('X', 10);
4     Board newBoard = board.copy();
5     assertEquals('X', newBoard.charAt(10));
6     board.populate('O', 11);
7     assertEquals('O', newBoard.charAt(11));
8 }
```

Without a filtering mechanism for this kind of results, a user of our Eclipse plug-in would be recommended to use a test that expects the value of ‘O’ for the test case value of ‘11’ (i.e., with the application of interface adaptation the following recommendation would appear):

```
assertEquals('O', roman.convert(11));
```

This is obviously a wrong recommendation and a system offering such poor suggestions would be of low benefit for its users.

Naturally, a refinement of the above query could improve the search results and prevent this particular result from appearing in the list of candidates. However, this involves a restriction of the search space and the exclusion of potentially useful results which by chance do not adhere to the more narrow query. Hence, it is desirable to apply a kind of “hybrid” approach, where the query should contain as few restrictions as possible, yet as many as necessary, and the results are evaluated against concrete implementations.

9.3.1. Oracle-Based Filtering

Naturally, situations in which the system recommends unsuitable tests cannot be entirely excluded. Nevertheless, to improve the user experience, in this section we are going to develop a filter mechanism that is built around the original ideas of Search-Enhanced Testing. Instead of using a characterizing test case to find replicas of the class under test, we can use the CUTs of the recommended tests for this purpose.

A reusable test case is usually associated with a corresponding class which it was originally written to test. In the case of our Roman numerals test cases, where 19 results were retrieved, we have 19 test cases τ_i with 19 CUTs ζ_j . Our goal is to retrieve the CUTs of these test cases and use them as oracles θ_k ($i, j, k = 1..19$). Algorithm 9.1 shows the general way in which the inspection and evaluation of potentially reusable tests works. At first sight, this algorithm can be interpreted as some kind of inverse variant of test-driven search – although it is not used to find reusable assets, but to identify misbehaving test cases. The test case under examination executes the oracles (i.e., it executes the CUTs of the other test cases in the result set) and marks those on which it does not execute successfully by storing them in a list of mismatching oracles. If the size of this list is larger than a previously defined mismatch threshold, the test case has to be removed from the set of possible reuse recommendations. The mismatch threshold t should not be too high (and naturally smaller than the number of oracles available), since a higher number makes it more likely that a false positive is delivered to users which passes fewer oracles. With a threshold of 1, as depicted in Figure 9.5,

a candidate is abandoned if it fails on all oracles (except its own CUT). This is, however, still quite restrictive, as the chances are high that a single test in the test case under examination fails on most or all oracles and thus the whole test case is rejected.

Algorithm 9.1: Oracle-Based Inspection for One Test Case.

```

Data: test case  $\tau_n$ , CUT  $\zeta_n$  of  $\tau_n$ ,  $n$  fixed.
Data: set of  $m$  oracles  $\theta_{1..m}$ ,  $x = 0$  counter,  $t \geq 1$  mismatch threshold.
Result:  $L =$  List of oracles not passing  $\tau_n$ 
while  $x < m$  do
     $x = x + 1;$ 
    if  $\theta_x \neq \zeta_n$  then
        if interface of  $\theta_x$  does not match  $\tau_n$  then
            | adapt provided interface of  $\theta_x$ ;
        end
        run  $\tau_n$  against  $\theta_x$ ;
        if  $\tau_n$  runs with errors then
            | add  $\theta_x$  to  $L$ ;
        end
    end
end
if  $|L| == m - t$  then
    | remove  $\tau_n$  from recommendations;
end

```

In the subsequent section we will discuss in more detail how the evaluation of test cases can be differentiated. In general, we can say that if there are enough “good” results in the set of reusable tests, false positives can almost certainly be excluded (an exception to this rule is the unlikely case that most or even all results of a search for reusable tests are testing the same wrong implementation). There may also be some false-negatives, however (i.e., test cases that would have fit in the users’ context but caused an issue with at least one of the oracles).

Obviously, the former is a desired effect, which outweighs the loss of a small fraction of reusable assets that, by chance, caused some issue with one or more of the oracles.

9.3.2. Test Case Evaluation

The previously described evaluation of tests based on the CUTs associated with the retrieved results is very general and restrictive (a test case is expelled when its execution yields an error), as the only variable influencing the rejection of a candidate is the given threshold (i.e., the number of oracles making the test case fail). In this subsection we will review the rejection process of a test case in a more concrete way, i.e., we will define what the criteria for rejection are.

For demonstration purposes, we stick to our Roman numerals example and utilize the following value-based query

$$("^\text{IVXDCML}"]\$") \rightarrow "[1-9]+\d*"; (\text{I}) \rightarrow 1; \quad (9.2.)$$

which looks for the mapping of a Roman numeral (composed of the allowed characters) to an Arabic numeral. For sanity reasons the query ensures that the Arabic numeral does not start with a 0 and, additionally, there is a check that ‘1’ is mapped to ‘I’, which is expected to appear in any reasonable test for Roman numerals. With this query we obtained eight results from SENTRE on which we applied our challenge algorithm. Each of the test cases ($\tau_i, i \in [1; 8]$) was executed against the CUTs of all other tests ($\zeta_n, n \in [1; 8], n \neq i$), while counting the successful tests. The results presented in Table 9.4 and in Table 9.5 clearly show that this approach helps to identify outliers like τ_7 .

In Table 9.5 we have visualized the results obtained from the execution of the test cases against the CUTs $\zeta_1 \dots \zeta_8$ and the colors of the particular cells visualize to what extent the test cases were successfully executed on the CUTs. The “red cross” spanning over row τ_7 and column ζ_7 impressively marks the useless test case τ_7 , which was just retrieved by chance and the two light red fields at (τ_7, ζ_1) and (τ_7, ζ_8) are just lucky hits in which the mapping of an irregular input

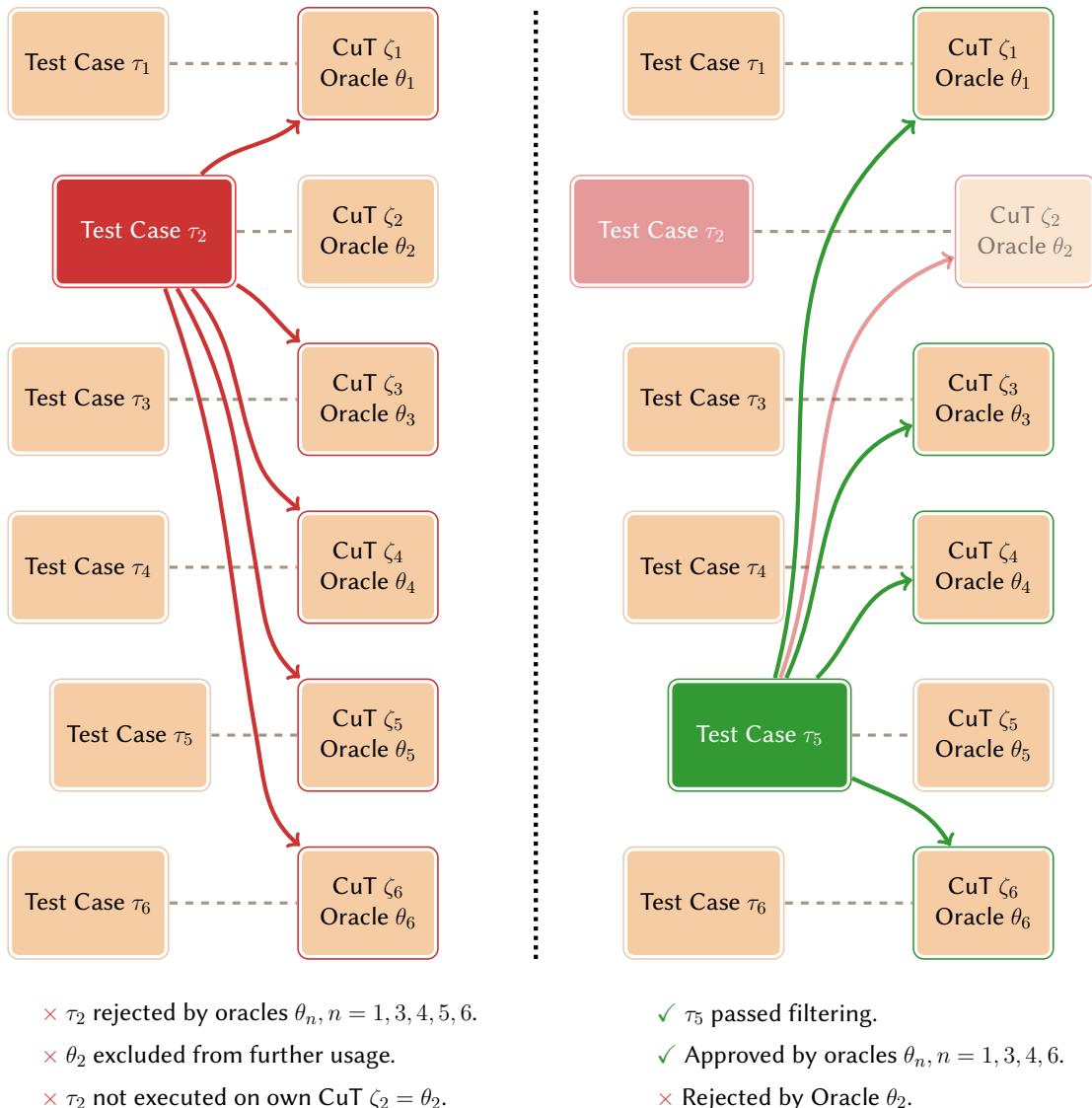


Figure 9.5.: Schema of Oracle-Based Filtering for Test Reuse.

Candidate	ζ_1	ζ_2	ζ_3	ζ_4	ζ_5	ζ_6	ζ_7	ζ_8
τ_1	—	29/29	29/29	29/29	29/29	29/29	0/29	29/29
τ_2	10/46	—	10/46	10/46	10/46	10/46	0/46	10/46
τ_3	25/25	25/25	—	25/25	25/25	25/25	0/25	25/25
τ_4	7/7	6/7	7/7	—	7/7	7/7	0/7	6/7
τ_5	14/14	14/14	14/14	14/14	—	14/14	0/14	14/14
τ_6	12/12	12/12	12/12	12/12	12/12	—	0/12	12/12
τ_7	1/11	0/11	0/11	0/11	0/11	0/11	—	1/11
τ_8	5/6	4/6	4/6	4/6	4/6	4/6	0/6	—

Table 9.4.: Evaluation of the Results for Query 9.2.

value ('A') to the 0-value matched the two CUTs' behavior. The fields over the diagonal are intentionally left white since in all of these cases the test case fully matched it's own class under test. Although it might be interesting to further investigate how to deal with test cases that reveal defects in their own CUTs (or are themselves erroneous), we leave this to future research as this is out of the scope of this thesis. To us, it is much more important that for the given example, the four reusable test cases τ_1, τ_3, τ_5 and τ_6 seem to be "fit for purpose" and do not need any further intervention from the user.

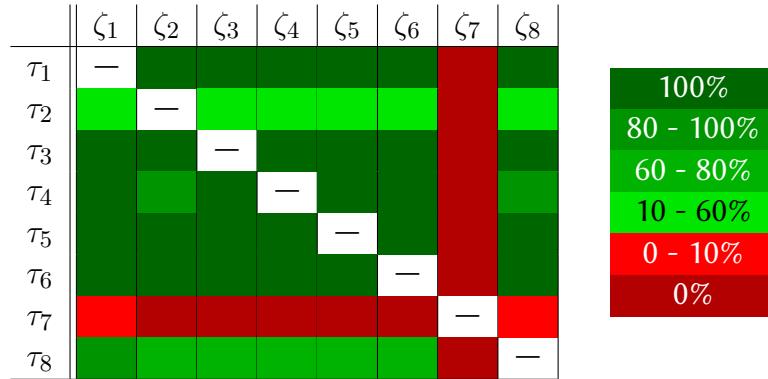


Table 9.5.: Visualization of the Evaluation Results.

Manual inspection of the retrieved test cases shows that our algorithm works correctly for the aforementioned scenario and that the four test cases marked

with a green color contain valid and correct tests for a Roman numeral class. Finally, Table 9.6 shows the tests contained in the test cases retrieved by SENTRE which have not been rejected by our algorithm. If the search was performed using our Eclipse plug-in, these results are executed in the developer's context against the class under test and subsequently ranked during speculative analysis.

α_1	Γ	α_1	Γ	α_1	Γ
I	1	II	2	III	3
IV	4	V	5	VI	6
VII	7	VIII	8	IX	9
X	10	XIV	14	XIX	19
XX	20	XXIV	24	XXXIII	33
XXXIV	34	XXXIX	39	XL	40
XLV	45	XLIX	49	L	50
LI	51	LXVII	67	LXXXIX	89
XC	90	XCIX	99	C	100
CXLIX	149	CLXXXIX	189	IVXLCDM	334
CCCXLIX	349	CD	400	CDLVI	456
D	500	DXLV	545	DCIV	604
DCCCXLIX	849	DCCCLXXXIX	889	CM	900
CMIV	904	CMXLIV	944	CMXCIX	999
M	1000	MVII	1007	MCCLIX	1259
MDCLXVII	1667	MCMLXXII	1972	MCMXCIX	1999
MMVIII	2008	MMXI	2011	MMCXXIV	2124
MMDCCCVI	2806	MMCMXCIX	2999	MMMMCMXCIX	4999
MMMMMMCMXLII	5942				

Table 9.6.: Roman Numerals Tests for Query 9.2.

9.4. Summary

In this chapter we have tackled the problem of providing better results to the users of our search engine SENTRE. To convince users of the value of a new

technology it is necessary to provide a sufficiently high level of reliability to them (i.e., the technology should reliably save them effort when performing a specific task). The major threat for any search engine, regardless of whether it provides reusable production code or reusable test cases, is the time required for result inspection. If the system presents a list of n potentially reusable assets, it is unlikely that users will inspect more than a couple of them, especially if the first ones do not provide any value or are useless in the user's context. Hence, we have introduced the idea of oracle-based filtering which helps to eliminate false-positive results from a search and delivers only valuable results to the user. The introduced technology relies on the ideas presented in our earlier work on Search-Enhanced Testing, which utilizes multiple versions of a program as oracles in order to identify possible defects in an implementation.

Our technology retrieves the appropriate CUTs with the test cases identified as reuse candidate. This means, that n reusable test cases deliver n oracles. Each potentially reusable test case is executed $n - 1$ times (i.e., once against each oracle except its own). If a test case fails on the majority (or all) of the oracles, it is classified as an outlier and therefore discarded from the list of results. Although this technique may lead to the exclusion of a small number of potentially appropriate test cases, it is better to present less more trustworthy test cases than more test cases that are less trustworthy.

In the following chapter we conclude our contribution before briefly discussing open issues and presenting a roadmap for future work in the reuse of software tests.

Contribution of this chapter

- We have presented the ideas behind Search-Enhanced Testing and gave an overview on the underlying techniques.
- We have shown an approach to combine test-driven search with multi-version testing using random test case values.
- Furthermore we have introduced a set of discrepancy tables and voting profiles for multi-version software testing. Based on our considerations,

we have shown a working tool for Search-Enhanced Testing, which automatically generates discrepancy tables for given code artifacts and characterizing tests.

- Based on the ideas of Search-Enhanced Testing, we have developed and described an approach for oracle-based evaluation of reusable tests and the determination of their fitness for purpose. With this approach we can effectively remove wrong results from the list of recommended tests in a reuse-oriented test recommendation system.

Part IV.

Epilogue

“ Ever tried, ever failed, no matter!
Try again, fail again, fail better!”

Worstward Ho

SAMUEL BECKETT

Winner of the 1969 Nobel Prize in Literature.

10

Epilogue

We started this thesis with a quote by Brian Kernighan who stated that debugging is twice as hard as writing the code itself and he concludes that this means if developers put all their intellect into writing their code, they are by definition not smart enough to debug it. Guided by this finding, this dissertation has focused on tackling this problem with a new approach based on the idea of reusing the knowledge of somebody else. The goal is that collectively, this knowledge will be smart enough to help users of reuse-oriented test recommendation tools to find and remove bugs in their code.

10.1. Retrospective

The central goal of the work conducted in this thesis was to develop a solution for reuse-assisted software testing – that is an approach that enhances (automated) software testing by leveraging software reuse techniques. We started with a

motivation of our idea and explained its *raison d'être*. Our literature survey showed that the most important underlying motivation of the software testing community is driven by the goal to reduce manual effort for testers and to find effective and efficient strategies to automatically generate test data. Although there have been many approaches for tackling this problem, none has strictly focused on the reusability of previously written software tests.

Within the scope of this thesis, we therefore developed this idea and presented initial considerations on the creation of an internet-scale repository of software tests, discussed the representation of these in a searchable database and identified techniques for extracting the knowledge contained in existing JUnit test cases. Our findings and solutions were also accompanied with a description of the challenges we faced in our work and how we were able to solve them. With the presentation of the SENTRE search engine for software tests, we demonstrated the practical results of our work in the aforementioned area and also introduced a set of retrieval techniques that were either tailored to or newly developed for the search of reusable software tests.

As well as creating the SENTRE search engine, we have also developed a dedicated client application that seamlessly integrates into the well-known Eclipse development environment. With the help of the plug-in, developers can reuse previously written tests by simply using the auto-complete feature included in the Eclipse editor. Finally, we concluded our work with the presentation of a multi-oracle testing approach that significantly enhances the trustworthiness of search results by automatically excluding “false positives” (i.e. wrong reuse candidates) from the list of reusable tests.

10.2. Contributions

In the first chapter, we introduced our research objective and presented a list of contributions we wanted to accomplish with this thesis. As evidenced by the results presented in the preceding chapters we can conclude that we have

fulfilled the identified goals. More specifically, the concrete results of the work carried out during this PhD research are as follows:

1. We have conducted a survey on reuse-oriented code recommendation systems and presented their characteristics in order to identify a general set of characteristics for future systems of this kind.
2. We have created a meta-model that captures the relevant information needed to support the reuse of knowledge contained in previously written software tests.
3. Based on this meta-model, we created a parser for JUnit test cases that stores the extracted information in an efficiently searchable database. Furthermore, we identified unfortunate drawbacks of JUnit which make it particularly hard to automatically recognize the class under test from a given test case.
4. By defining typical usage scenarios for our approach in the software development lifecycle, we have also identified various potential applications of the approach in future software projects.
5. We presented SENTRE – a sophisticated search engine for reusable software tests. Since it is based on our generic test meta-model, the data available through SENTRE is language independent, i.e., the knowledge extracted from a Java project can easily be reused for software written in other languages, such as a newly created C# project, for instance.
6. Based on the findings that interface-based searches are handicapped by their dependence on names, we have introduced a new set of retrieval techniques that are value- and pattern based. Furthermore we presented the possibility to reverse the ideas from test-driven search and use production code to find corresponding reusable software tests.
7. We presented a test reuse environment implemented as an Eclipse plug-in which fulfills the characteristics for modern reuse-oriented recommendation systems identified earlier in this thesis.

8. We have enhanced the usefulness of the approach by providing a model of the micro-process for the reuse of code and software tests. This also shows the relationship between the “traditional” reuse of code and the reuse of software tests.
9. Based on the ideas of n-version programming and back-to-back testing, we developed the ideas of Search-Enhanced Testing and a tool that utilizes reusable software components as oracles to generate discrepancy tables. We described the approach and how it can help to reduce the manual effort in software testing.
10. With the help of Search-Enhanced Testing, we finally presented an approach that filters wrong results from the list of reusable tests (also known as *false positives*). We have shown the feasibility of our approach and presented an algorithm that describes the strategy behind it.

10.3. Future Work

Like all research that tries out new ideas, the work we presented in this dissertation is just the first stepping stone towards fully automated recommendations of reusable software tests. We have identified and described several challenges that need to be resolved by future researchers to make this vision a reality in mainstream software engineering projects. The SENTRE search engine and our test recommendation plug-in for Eclipse are successful proof-of-concept implementations that demonstrate the feasibility of effective and efficient searches for reusable software tests and their preparation for use in new projects. Based on our findings, the next step is to scale the approach to larger entities. While we have successfully applied our system on unit tests, further investigations need to explore the possibilities of reusing more coarse-grained software tests and of using them to test larger components.

Our initial implementation relies on a large repository harvested from the internet. Although this offers a large data set for further research in the area of test

reuse, we strongly encourage the evaluation of our approach in an industry setting, where professional testers and domain experts create large sets of reusable tests. We envisage an experiment, where two groups of developers are asked to write tests for different sets of programs in a given time frame in order to discover as many (purposely seeded) defects in the code as possible. Afterwards, they are introduced to the reuse-oriented test recommendation tool. Thereby, the time necessary for the introduction needs to be taken into account, when the overall performance is measured. Subsequently, the groups' program sets are switched and the developers are asked once again to find as many defects as possible in the code using the test recommendation system. Initially, we propose the following two hypothesis:

1. A developer using a reuse-oriented test recommendation system will find more bugs, B , in a program, P , within a given time frame, T , than a developer who writes the tests using best practices:

$$B_{tool} \geq B_{manual}, T_{tool} = T_{manual}.$$

2. In program P , a developer using a reuse-oriented test recommendation system will discover a fixed number of known bugs, B , more quickly than a developer who writes the tests using best practices:

$$T_{tool} \leq T_{manual}, B_{tool} = B_{manual}.$$

To explore the potential of test reuse, another experiment that needs to be carried out is the effectiveness of software tests that exist in software projects compared to such tests that are created using automated test recommendation. In particular such an experiment should investigate whether a test case that is generated by reusing existing software tests outperforms those tests that were particularly written for distinct components. A possible approach to seed bugs in the existing code is the utilization of mutant generators, while the mutant kill rate is a valid measure for test suite effectiveness [ABL05].

Finally, the example from Listing 6.7 on page 126 has shown that the JUnit testing framework is not very reuse-friendly. Since the framework does not include techniques and conventions that support the reuse of test cases, it is a challenging task to make the information bound up in them available to developers. Future

work on a reuse-friendly testing framework needs to enhance the reusability of JUnit test code whilst retaining JUnit's simplicity and familiarity to developers. It is especially important to clearly identify the class under test in the code of the test case, which could be accomplished either by the introduction of a new annotation or a convention that the first variable declaration in the class always identifies the CUT.

10.4. Concluding Vision

Along with other techniques for enhancing the quality of software, testing is undoubtedly one of the most effective and important instruments in the arsenal of quality assurance engineers. No mainstream software development process should underestimate the importance of effectively testing the system under development. Nevertheless, it still remains a very labor intensive and tedious task, which has to be carried out manually to a large extent. In this dissertation we have discussed a new approach that is a first step towards an integrated test reuse environment. Beside the pure reuse of software tests, we envisage a more extensive speculative analysis mechanism that is able to evaluate the fitness for purpose of reusable artifacts and to learn from users' choices. Furthermore, such a test recommendation system should not represent a single, isolated island of functionality in a developers environment, but should play a proactive part in the evolution of the repository of reusable information. In particular, when a developer discovers a wrong test and corrects it, the system should propagate the new version to the repository so that other users can benefit from the improvement.

The idea of combining software reuse with software testing can also be a first step towards a more integrated approach to software development and software testing, where the development of a system is influenced by previously created tests. Similar to the ideas from test-driven development, a newly created class can be developed according to the behavioral description of an already existing test suite. Even further, the tests contained in a repository can be used as an additional search criterion for reusable components to support the search for

previously developed programs. Through the work described in this dissertation we have therefore delivered an initial tool set that opens up a whole set of future research perspectives, not only in the area of test reuse, but also in the further automation of software testing and software reuse in general.

Was ich noch zu sagen hätte...

When I started as a research assistant in the group of Colin Atkinson, one of the first things he told me was that writing a PhD is different to writing a diploma thesis. He said that – in contrast to the diploma thesis – there is nobody anymore who tells one what to do, but that it is now time to conduct my own research work, supported by him as my supervisor. And that's what I did!

The ideas for the topic of this thesis already emerged in the last months of the year 2009 and they were first mentioned in our SUITE publication [JHA10] presented in Cape Town, South Africa during my first ICSE. Since then, a lot of work has been conducted and I would have never been able to carry all this load on my own. Therefore, it is time to say thank you to all the people, who made it possible for me to conduct the exciting research on this topic and to finally finish the work on this thesis. The next page is for them!

Finally, my main motivation for this thesis was, to explore the possibilities to extract the large and precious amount of knowledge contained in previously created test cases and make this knowledge available to software developers. I considered it a waste of resources to write test cases and only use them in the context of one dedicated software system and project. Therefore I conducted the work presented in this thesis, which is yet foundational research and there is still a lot of space left to future researchers... I hope they will enjoy it as much as I did!

Bad Dürkheim, May 2014

Werner Janjic

Thank you...

Colin Atkinson, who gave me the opportunity to do this work.

Oliver Hummel, for many fruitful discussions, his support and faith in this idea.

Reid Holmes, for coming from overseas to report on this work.

Christoph Giess, for supplying me with BitBucket sources, providing momentum and keeping me running.

The former and current staff of the Software-Engineering Group – Gabi, Dietmar, Marcus, Olli, Ralph, and all colleagues and friends – for the open and friendly atmosphere.

Giovanni, for introducing me into the world of software architecture and a great time during the joint Navy project.

All my students, for their commitment.

My father Zvonimir and my stepmother Yvonne, for all their support, faith and trust in me. A further thank to my grandmother Theresia, who passed away just before I could defend this thesis, leaving us in sadness.

And finally

Silke, my beloved wife and friend, who made many sacrifices to enable me to conduct the work on this dissertation. Thank you for being by my side, supporting me and my work through all these years.

The biggest emotion in creation is the bridge to optimism.

BRIAN MAY, musician and astrophysicist.

Appendices

List of Figures

2.1	Input-Processing-Output Model of Program Tests [Som10].	16
2.2	Generic Structure of a Class Interface.	18
3.1	Search Scenarios in Software Engineering [JHA10].	36
3.2	Milestones in Source Code Search and Recommendation.	42
3.3	Initial System Architecture of Merobase [Hum08].	45
3.4	The Architecture of the Sourcerer Infrastructure [Baj+06].	46
3.5	The S ⁶ Web Interface.	47
3.6	Process of Test-Driven Reuse [HA04; Hum08]	50
4.1	Test-Driven Reuse with Distributed Adaptation.	58
5.1	Overview of the Micro-Process of Software Reuse [JHA14].	71
5.2	The user interface of CodeFinder showing matching items to a user's query who wants to draw a circle. Taken from [FHR91]. . .	74
5.3	CodeBroker's Presenter [YF02].	76
5.4	Strathcona plug-in for Eclipse [HM05].	81
5.5	CodeGenie Test-Driven Search Process [Lem+07].	82
5.6	Screenshot of CodeGenie for a Number Converter [Laz+09]. . .	84
5.7	PARSEWeb.	85
5.8	Code Conjurer Recommendations	88
5.9	Test-Driven Search for a Credit Card Component [JA12].	91
5.10	IDE Auto-Complete Recommendation.	94
6.1	The TIOBE Programming Community Index [TIO14]	102

6.2	The Test Model Contains Test Suites and Required Components. . .	107
6.3	The Decomposition of a Component.	108
6.4	The Decomposition of a Test Suite.	112
6.5	Exemplary Meta-Model Instantiation.	114
6.6	Distribution of assert Statements.	125
7.1	Test Search & Reuse Scenarios in Software Engineering.	134
7.2	Provided and Required Interface of the Test in Listing 7.1.	140
7.3	Screenshot of a SENTRE Result Table.	144
7.4	Strict Value-Based Search with Result List.	152
7.5	An Exception Test in SENTRE.	158
7.6	Micro-Process of Test Reuse.	160
7.7	System Architecture Sketch of SENTRE.	162
8.1	Process of Tool-Supported Test Reuse.	169
8.2	IDE Auto-Completion for Testing Single Operations.	172
8.3	Layered Architecture Schema of the Eclipse Plug-In.	173
8.4	View for Continuous Testing.	175
8.5	Exception Test in Eclipse.	176
9.1	Process of Search-Enhanced Testing.	186
9.2	Harvested Components as Oracle [Hum+06].	191
9.3	Multi-Version Testing of a Roman Number Converter.	193
9.4	Discrepancy-Driven Testing [Hum+06].	194
9.5	Schema of Oracle-Based Filtering for Test Reuse.	204
A.1	Poster about Search-Enhanced Testing. Presented at ICSE 2011. .	232

List of Tables

2.1 Terms in Software Testing and Java / JUnit.	22
3.1 Motivation for code search by target size [USL08].	35
3.2 Comparison of Code Search Engines [HJA07].	49
3.3 Comparison of Retrieval Techniques [HJA07].	52
4.1 API Mismatch of Test and Candidate	56
4.2 Exemplary Searches With and Without Adaptation.	64
5.1 Code-Based Recommendation Systems.	73
6.1 Open Source Hosters Facts.	105
6.2 Repository Content of SENTRE.	118
6.3 Lines of Java Code in the SENTRE Repository Grouped by Source.	119
6.4 Java File Table.	120
6.5 Digest of the JUnit assert Statements.	124
6.6 List of Calls to the CUT in Listing 6.8.	129
7.1 Criteria and Weights for Result Ranking.	145
7.2 Distance Weights for Methods and Queries.	146
7.3 Java Primitive Data Types and Wrapper Classes.	148
9.1 Results of a Test-Driven Search for a Stack.	191
9.2 Invocation Profile Categories.	195
9.3 Discrepancy Table for the Roman Numerals Example.	198
9.4 Evaluation of the Results for Query 9.2.	205

9.5 Visualization of the Evaluation Results.	205
9.6 Roman Numerals Tests for Query 9.2.	206
A.1 Comparison of the Precision of Search Engines (relevant / candidates) [HJA07; Hum08].	235

A

Materials

This appendix contains accompanying material referenced in this thesis, which was too voluminous to be contained within the original text. The following sections contain figures, text and tables.

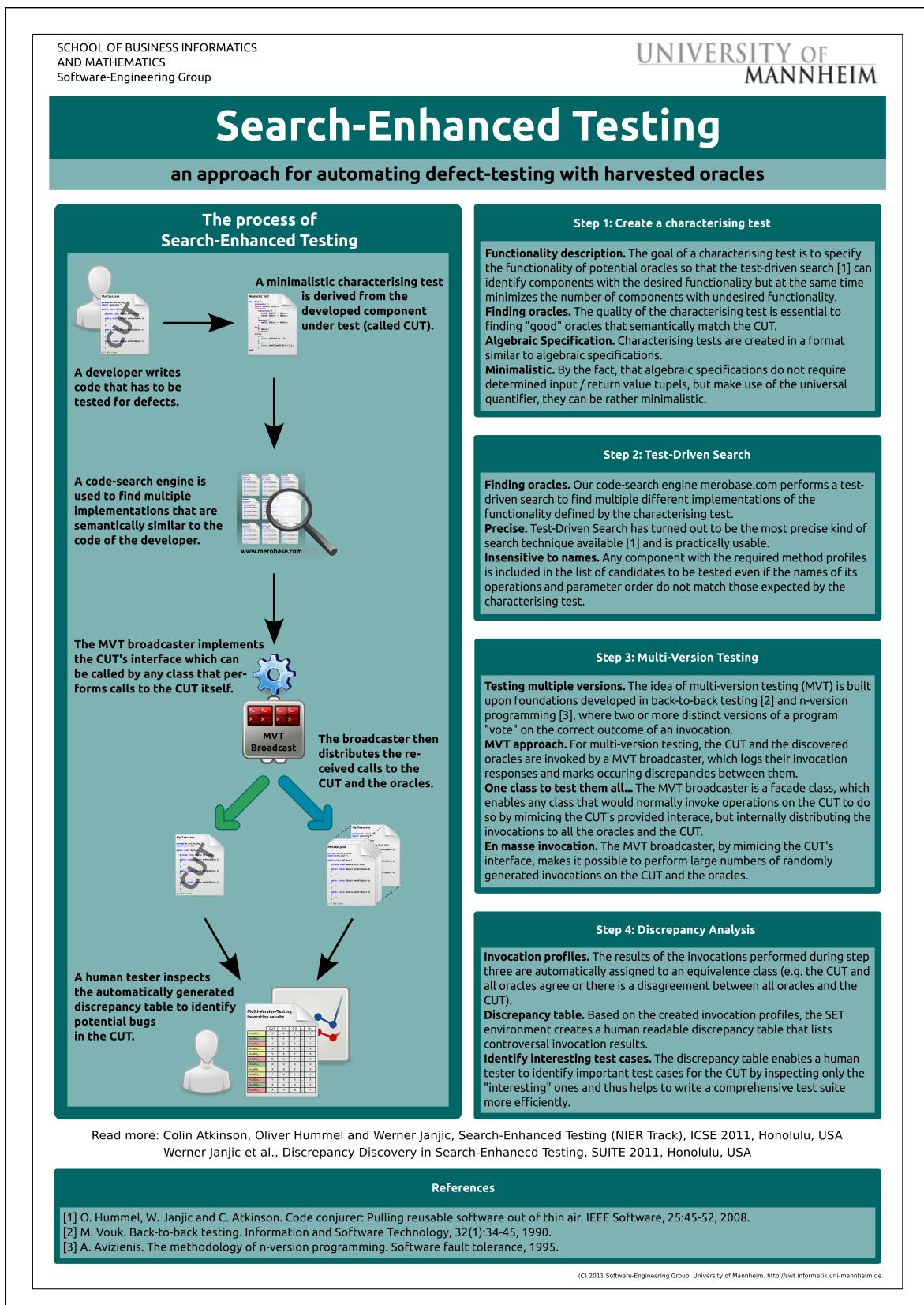


Figure A.1.: Poster about Search-Enhanced Testing. Presented at ICSE 2011.

A.1. Regular Expressions

Regular Expressions are a powerful tool that allow its users to describe and parse text using almost a kind of mini programming language [Fri02]. The SENTRE search engine for software tests enables its users to use this tool in search queries to describe test case values and expected results in an abstract and more formal way.

This section gives a short overview about the most common and relevant elements of regular expressions that enables the readers of this thesis to create some basic queries themselves.

- ^ The “hat” symbolizes the beginning of a text element. The following expression cannot be preceded by any other set of characters.
- \$ The “dollar” sign is the counterpart of the hat. It symbolizes the end of a text element, i.e., no character can follow after this.
- + , * The + and * quantifiers can be used to express that the preceding element has to appear one or more times (+) or zero or more times (*), respectively.
- {n} The number n between braces describes that the preceding element has to appear n times.
- \d The meta-character \d substitutes the digits 0 – 9.
- \w Represents any alpha-numeric character and the underscore _ symbol.
- [A-Z] The bracket expression matches any single character that is contained within the squared brackets.

A.1.1. Examples of Queries with Regular Expressions

To provide the reader an impression of value-based queries with regular expressions, we list a set of queries together with corresponding assert-statements from JUnit test cases.

When test case values or expected results are defined using regular expressions, the asserts serve as representatives for any statements that match the query.

```
rexx:(2000)->true;(2100)->false;
assertEquals(true, isLeapYear(2000));
assertEquals(false, isLeapYear(2100));

rexx:(\d{4})->boolean;
assertEquals(true, isLeapYear(2004));
assertEquals(false, isLeapYear(1999));

rexx:^[IVXDCLM]+\$)->"\d{2}";
assertEquals(10, valueOf("X"));
assertEquals(51, valueOf("LI"));
assertEquals(99, valueOf("XCIX"));

rexx:^[A-Za-z]*\$)->"\d+";
assertEquals(0, length(""));
assertEquals(10, valueOf("HelloWorld"));
```

A.2. Comparison of Retrieval Precision

Query	Google	Yahoo	Google Code- search	Koders	Merobase
copyFile(String, String):void	1 / 25	2 / 25	7 / 25	0 / 25	18 / 25
gcd(int,int):int	10 / 25	7 / 25	12 / 25	2 / 25	17 / 25
isLeapYear(int):boolean	8 / 25	12 / 25	3 / 25	2 / 25	14 / 25
md5(String):String	0 / 25	0 / 25	4 / 22	0 / 25	12 / 25
isPrime(int):int	6 / 25	15 / 25	7 / 25	4 / 25	5 / 25
randomNumber(int,int):int	0 / 25	3 / 25	2 / 7	0 / 7	14 / 25
randomString(int):String	4 / 25	2 / 25	6 / 25	4 / 16	5 / 25
replace(String,String, String):String	2 / 25	8 / 25	14 / 25	3 / 25	22 / 25
reverseArray(int[]):int[]	1 / 10	3 / 23	1 / 1	0 / 4	5 / 7
sort(int[]):int[]	0 / 25	0 / 25	5 / 20	0 / 25	20 / 25
sqrt(double):double	5 / 25	4 / 25	4 / 25	1 / 25	11 / 25
getMinMax(int[]):int[]	0 / 15	0 / 22	0 / 0	0 / 25	2 / 4
Stack(push(Object):void, pop():Object, size():int)	1 / 25	2 / 25	0 / 0	1 / 25	6 / 25
Average Precision	12.2%	17.9%	29.5%	5.9%	53.7%
Standard Deviation	13.3%	18.9%	26.5%	7.8%	22.4%

Table A.1.: Comparison of the Precision of Search Engines (relevant / candidates) [HJA07; Hum08].

“ Beware of bugs in the above code;
I have only proved it correct, not tried it. ”

DONALD KNUTH
Computer Scientist, author of \TeX .

B

Listings

The following pages contain the listings referenced in the text of this thesis. If a listing contains code that was harvested from the internet, the first line contains the origin. For convenience and readability, all parts which are not essential for understanding or are dead-code, i.e., if they are not called or executed at runtime, were removed from these listings. You may, however, still refer to the original URL. Furthermore, the source code of externally acquired classes may have been altered and adapted by algorithms like those in test-driven search, which adjust the provided interface of reuse candidates to the one required by the search query.

Listing B.1: Excerpt of the Class Under Test

```
1 // Origin: http://musicbrainzws2-java.googlecode.com/svn-history/r34/
   ↳ mc2java/src/org/mc2/Roman.java
2 // @author Ben Clifford
3 public class RomanNumber {
4     public class SymTab {
5         /** Roman symbol */
6         char symbol;
7         /** Numerical value */
8         int value;
9         public SymTab(char s, int v) {
10            this.symbol = s;
11            this.value = v;
12        }
13    };
14
15    public RomanNumber.SymTab syms[] = {
16        new SymTab('M', 1000), new SymTab('D', 500),
17        new SymTab('C', 100), new SymTab('L', 50),
18        new SymTab('X', 10), new SymTab('V', 5),
19        new SymTab('I', 1) };
20
21    public String toRoman(int n) {
22        int i;
23        String s;
24        s = "";
25        while (n > 0) {
26            for (i = 0; i < syms.length; i++) {
27                if (syms[i].value <= n) {
28                    int shift = i + (i % 2);
29                    if (i > 0 && shift < syms.length
30                        && (syms[i - 1].value - syms[shift].value) <= n) {
31                        s = s + syms[shift].symbol + syms[i - 1].symbol;
32                        n = n - syms[i - 1].value + syms[shift].value;
33                        i = -1;
34                    } else {
35                        s += syms[i].symbol;
36                        n -= syms[i].value;
37                        i = -1;
38                    } } } }
39        return s;
40    }
41 }
```

Listing B.2: Excerpt of Oracle 1

```

1 // https://svn.apache.org/repos/asf/jena/Import/Jena-SVN/ARQ/tags/ARQ-2.0-
   ↵ beta/src/com/hp/hpl/jena/sparql/util/RomanNumeral.java
2 public class RomanNumber {
3     int intValue;
4     public static String i2r(int i) {
5         if (i <= 0)
6             throw new NumberFormatException();
7         if (i > 3999)
8             throw new NumberFormatException();
9         StringBuffer sbuff = new StringBuffer();
10        i = i2r(sbuff, i, "M", 1000, "CM", 900, "D", 500, "CD", 400);
11        i = i2r(sbuff, i, "C", 100, "XC", 90, "L", 50, "XL", 40);
12        i = i2r(sbuff, i, "X", 10, "IX", 9, "V", 5, "IV", 4);
13        while (i >= 1) {
14            sbuff.append("I");
15            i -= 1;
16        }
17        return sbuff.toString();
18    }
19    public String toRoman(int value) {
20        try {
21            return i2r(intValue);
22        } catch (Exception e) {
23            return "ERROR";
24        }
25        static class RValue {
26            static RValue[] table = new RValue[] {
27                new RValue('M', 1000),
28                new RValue('D', 500), new RValue('C', 100),
29                new RValue('L', 50), new RValue('X', 10),
30                new RValue('V', 5), new RValue('I', 1)};
31            char lex; int val;
32            RValue(char s, int v) {
33                lex = s; val = v;
34            }
35        }
36    }

```

Listing B.3: Excerpt of Oracle 2

```

1 // http://wiki.hsr.ch/SimpleCode/files/RomanNumber.java
2 public class RomanNumber {
3     private final static char[] ROMANNUMBERS = { 'I', 'V', 'X', 'L', 'C',
4                                                 'D', 'M' };
5     private final static int[] DECNUMBERS = { 1, 5, 10, 50, 100, 500, 1000
6                                             };
7     private final static int POS_CHANGE = 2;
8     private final static int[] SPECIALDEC = { 4, 5, 9 };
9     public String toRoman(int inputNumber) {
10        StringBuffer returnValue = new StringBuffer();
11        if (inputNumber == 0) return returnValue.toString();
12        int restThousand = inputNumber % DECNUMBERS[6];
13        int thousand = inputNumber / DECNUMBERS[6];
14        int dividend = DECNUMBERS[2]; int oldDividend = 1;
15        for (int index = 0; index <= DECNUMBERS.length - POS_CHANGE; index
16             += POS_CHANGE) {
17            int restAkt = restThousand % dividend;
18            int rest = restAkt; int aktPos = 0;
19            if (rest == SPECIALDEC[2] * oldDividend) {
20                char[] addChars = { ROMANNUMBERS[index],
21                                    ROMANNUMBERS[index + POS_CHANGE] };
22                returnValue.insert(aktPos, addChars);
23                aktPos += 2;
24                rest -= SPECIALDEC[2] * oldDividend;
25            } else {
26                if (rest == SPECIALDEC[0] * oldDividend) {
27                    char[] addChars = { ROMANNUMBERS[index],
28                                        ROMANNUMBERS[index + 1] };
29                    returnValue.insert(aktPos, addChars);
30                    aktPos += 2;
31                    rest -= SPECIALDEC[0] * oldDividend;
32                } else if (rest >= SPECIALDEC[1] * oldDividend) {
33                    returnValue.insert(aktPos, ROMANNUMBERS[index + 1]);
34                    rest -= SPECIALDEC[1] * oldDividend; aktPos++;
35                }
36            for (; rest > 0; rest -= oldDividend)
37                returnValue.insert(aktPos, ROMANNUMBERS[index]);
38            oldDividend = dividend;
39            dividend = dividend * DECNUMBERS[2];
40            restThousand -= restAkt; }
41        for (int i = 0; i < thousand; i++)
42            returnValue.insert(0, ROMANNUMBERS[ROMANNUMBERS.length - 1]);
43        return returnValue.toString();
44    }
}

```

Listing B.4: Excerpt of Oracle 3

```
1 // http://www.dcs.bbk.ac.uk/~roman/sp1/java/RomanNumber.java
2 public class RomanNumber {
3     public String convert(int n) {
4         if ((n >= 1) && (n <= 3999)) {
5             int thousand = (n % 10000) / 1000;
6             int hundred = (n % 1000) / 100;
7             int ten = (n % 100) / 10;
8             int unit = (n % 10);
9             String roman = "";
10
11             // Convert the thousandth number into a Roman numeral
12             if (thousand == 1) {
13                 roman += "M";
14             } else if (thousand == 2) {
15                 roman += "MM";
16             } else if (thousand == 3) {
17                 roman += "MMM";
18             }
19             // end of if for thousand
20             // [...]
21             if (unit == 1) {
22                 roman += "I";
23             } else if (unit == 2) {
24                 roman += "II";
25             } else if (unit == 8) {
26                 roman += "VIII";
27             } else if (unit == 9) {
28                 roman += "IX";
29             }
30             // end of if for unit
31             return roman;
32         } // end of valid range
33     else {
34         return "ERROR";
35     }
36 }
37
38 // Adapter method
39 public String toRoman(int value) {
40     RomanNumber rn = new RomanNumber();
41     return rn.convert(value);
42 }
43 }
```

Listing B.5: Excerpt of Oracle 4

```
1 // https://code.google.com/a/eclipselabs.org/p/jcinetheque/source/browse/
   ↵ trunk/JCinetheque/src/main/java/utils/RomanNumber.java?r=41
2 public class RomanNumber {
3     private final static String[] BASIC_ROMAN_NUMBERS = { "M", "CM", "D",
   ↵ "CD", "C", "XC", "L", "XL", "X", "IX", "V", "IV", "I" };
4     private final static int[] BASIC_VALUES = { 1000, 900, 500, 400, 100,
   ↵ 90, 50, 40, 10, 9, 5, 4, 1 };
5     private int value;
6     private String romanString;
7
8     public String toRomanValue() {
9         if (this.romanString == null) {
10             this.romanString = "";
11             int remainder = this.value;
12             for (int i = 0; i < BASIC_VALUES.length; i++) {
13                 while (remainder >= BASIC_VALUES[i]) {
14                     this.romanString += BASIC_ROMAN_NUMBERS[i];
15                     remainder -= BASIC_VALUES[i];
16                 }
17             }
18         }
19         return this.romanString;
20     }
21
22     public Integer getValue() {
23         return this.value;
24     }
25
26     public String toRoman(int value) {
27         if (1 <= value && value <= 3999) {
28             RomanNumber rn = new RomanNumber(value);
29             return rn.toRomanValue();
30         } else {
31             return "ERROR";
32         }
33     }
34 }
```

Listing B.6: Excerpt of Oracle 5

```

1 // https://github.com/froderik/roman_numeral_katas/blob/master/java/
   ↵ RomanNumber.java
2 public class RomanNumber {
3     private int number;
4     public String toString(int number) {
5         this.number = number;
6         String result = "";
7         int thousands = this.number / 1000;
8         result += times(thousands, "M");
9         int hundreds = this.number / 100 % 10;
10        result += times(hundreds, "C", "D", "M");
11        int tens = this.number / 10 % 10;
12        result += times(tens, "X", "L", "C");
13        int ones = this.number % 10;
14        result += times(ones, "I", "V", "X");
15        if (value > 2999 || result.contains("ERROR")) {
16            return "ERROR";
17        }
18        return result;
19    }
20    private String times(int number, String character) {
21        String result = "";
22        for (int i = 0; i < number; i++) {
23            result += character;
24        }
25        return result;
26    }
27    private String times(int number, String onesChar, String fivesChar,
   ↵ String tensChar) {
28        switch (number) {
29            case 0 : return "";
30            case 1 :
31            case 2 :
32            case 3 : return times(number, onesChar);
33            case 4 : return onesChar + fivesChar;
34            case 5 :
35            case 6 :
36            case 7 :
37            case 8 : return fivesChar + times(number - 5, onesChar);
38            case 9 : return onesChar + tensChar;
39            default: return "ERROR";
40        }
41    }
42 }
```

Listing B.7: Excerpt of Oracle 7

```
1 // Original: http://grepcode.com/file_/repo1.maven.org/maven2/org.jodd/jodd/
   ↳ jodd/3.2.5/jodd/format/RomanNumber.java/?v=source
2 // Copyright (c) 2003-2011, Jodd Team (jodd.org). All Rights Reserved.
3 public class RomanNumber {
4     public static final int[] VALUES = new int[] { 1000, 900, 500, 400,
   ↳ 100, 90, 50, 40, 10, 9, 5, 4, 1 };
5     public static final String[] LETTERS = new String[] { "M", "CM", "D",
   ↳ "CD", "C", "XC", "L", "XL", "X", "IX", "V", "IV", "I" };
6
7 /**
8 * Converts to roman number.
9 */
10 public String toRoman(int value) {
11     try {
12         StringBuilder roman = new StringBuilder();
13         int n = value;
14         for (int i = 0; i < LETTERS.length; i++) {
15             while (n >= VALUES[i]) {
16                 roman.append(LETTERS[i]);
17                 n -= VALUES[i];
18             }
19         }
20         return roman.toString();
21     } catch (Exception e) {
22         return "ERROR";
23     }
24 }
25 }
```

Listing B.8: Broadcaster Class for Discrepancy-Driven Testing

```

1  public class RomanNumber {
2      public static ArrayList<RomanNumberInterface> components =
3          new ArrayList<RomanNumberInterface>();
4      public static Long start;
5      public static Long end;
6
7      public RomanNumber() {
8          components.add(new oracles.cut.RomanNumber());
9          components.add(new oracles.o1.RomanNumber());
10         components.add(new oracles.o2.RomanNumber());
11         components.add(new oracles.o3.RomanNumber());
12         components.add(new oracles.o4.RomanNumber());
13         components.add(new oracles.o5.RomanNumber());
14         components.add(new oracles.o6.RomanNumber());
15         components.add(new oracles.o7.RomanNumber());
16     }
17
18     public String toRoman(int n) {
19         ArrayList<Long> durations = new ArrayList<Long>();
20         ArrayList<String> callSource = new ArrayList<String>();
21         ArrayList<String> answers = new ArrayList<String>();
22
23         for (RomanNumberInterface c : components) {
24             callSource.add(c.getClass().getName());
25             start = System.nanoTime();
26
27             String answer =
28                 ((interfc.RomanNumberInterface) c).toRoman(n);
29             end = System.nanoTime();
30             answers.add(answer);
31             durations.add(end - start);
32         }
33
34         Object[] params = new Object[] { n };
35         TestResultsLogger.log("RomanNumber", 1, "toRoman(int n)",
36             "String", new String[][] {
37                 { "n", "int", "" + n }
38             }, "toRoman(" + n + ")", params,
39             answers, durations, callSource);
40
41         return answers.get(0);
42     }
43 }
```

Listing B.9: Excerpt of the JSON Representation of a JUnit Test.

```
1 { _id : <TestModelId>,
2   name : "EuclidTest",
3   components : [ <ComponentId> ],
4   tests : [ <TestSuiteId1> ]
5 } {
6   _id : <ComponentId>,
7   name : "Euclid",
8   operations : [ <OperationId> ]
9 } { _id : <OperationID>,
10   name : "dist",
11   parameter :
12     [ { type : double, name : x1 }, { type : double, name : y1, },
13       { type : double, name : x2 }, { type : double, name : y2 } ],
14   returnValue : { type : double }
15 } { _id : <TestSuiteId>,
16   name : "EuclidTest",
17   componentUnderTest_id : <ComponentId>,
18   testCases : [ <TestCaseId1>, <TestCaseId2>, ... ]
19 } { _id : <TestCaseId>,
20   name : "testDistanceCalculation",
21   tests : [ <TestId> ],
22   statements : [ <StatementId1> ],
23 } { _id : <TestId>,
24   preStatements : [ <StatementId1> ],
25   postStatements : [ ],
26   cutInvocation : <CUTInvocationId>,
27   expectedResult : <ExpectedResultId>
28 } { _id : <CUTInvocationId>,
29   operation : [ <OperationId> ],
30   provides : [ <ValueId1>, <ValueId2>, <ValueId3>, <ValueId4> ],
31   returns : <ValueId5>
32 } { _id : <ValueId1>,
33   name : "x1", type : double,
34   value : "4"
35 } { _id : <ValueId2>,
36   name : "y1", type : double,
37   value : "2"
38 } { _id : <ValueId3>,
39   name : "x2", type : double,
40   value : "8"
41 } { _id : <ValueId4>,
42   name : "y2", type : double,
43   value : "5"
44 } { _id : <ValueId5>,
45   type : double,
46   value : 5
47 }
```

C

Bibliography

References

- [Alm+04] Eduardo Santana de Almeida et al. “RiSE project: towards a robust framework for software reuse”. In: *Proceedings of the 2004 IEEE International Conference on Information Reuse and Integration, 2004. IRI 2004*. 2004, pp. 48–53. doi: 10.1109/IRI.2004.1431435.
- [AO08] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2008.
- [ABL05] J.H. Andrews, L.C. Briand, and Y. Labiche. “Is mutation an appropriate tool for testing experiments?” In: *Proceedings of the 27th International Conference on Software Engineering, 2005. ICSE 2005*. May 2005, pp. 402–411. doi: 10.1109/ICSE.2005.1553583.

- [AHJ11] Colin Atkinson, Oliver Hummel, and Werner Janjic. “Search-enhanced testing (NIER Track)”. In: *Proceedings of the 33rd International Conference on Software Engineering*. ICSE ’11. Waikiki, Honolulu, HI, USA: ACM, 2011, pp. 880–883. ISBN: 978-1-4503-0445-0. doi: 10.1145/1985793.1985932. URL: <http://doi.acm.org/10.1145/1985793.1985932>.
- [Atk+08a] Colin Atkinson et al. “Modeling Components and Component-Based Systems in KobrA”. In: *The Common Component Modeling Example*. Ed. by Andreas Rausch et al. Vol. 5153. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, pp. 54–84. ISBN: 978-3-540-85288-9. doi: 10.1007/978-3-540-85289-6_4. URL: http://dx.doi.org/10.1007/978-3-540-85289-6_4.
- [Atk+08b] Colin Atkinson et al. “Specifying high-assurance services”. In: *IEEE Computer* 41.8 (2008), pp. 64–71.
- [Atl14] Atlassian. *Bitbucket*. Jan. 2014. URL: <http://www.bitbucket.org> (visited on 01/22/2014).
- [Avi95] Algirdas Avižienis. “The methodology of n-version programming”. In: *Software fault tolerance* 3 (1995), pp. 23–46.
- [BR08] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. 2nd. USA: Addison-Wesley Publishing Company, 2008. ISBN: 0321416910, 9780321416919.
- [Baj+06] Sushil Bajracharya et al. “Sourcerer: a search engine for open source code supporting structure-based search”. In: *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. OOPSLA ’06. Portland, Oregon, USA: ACM, 2006, pp. 681–682. ISBN: 1-59593-491-X. doi: 10.1145/1176617.1176671. URL: <http://doi.acm.org/10.1145/1176617.1176671>.
- [Bar13] Adam Bard. *Top Github Languages for 2013 (so far)*. Aug. 2013. URL: <http://adambard.com/blog/top-github-languages-for-2013-so-far/> (visited on 08/30/2013).
- [Bec03] Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

- [BG14] Kent Beck and Erich Gamma. *JUnit Test Infected: Programmers Love Writing Tests*. Mar. 2014. URL: <http://junit.sourceforge.net/doc/testinfected/testing.htm>.
- [Bei90] Boris Beizer. *Software Testing Techniques*. 2nd edition. New York, NY, USA: Van Nostrand Reinhold, 1990.
- [Ber07] Antonia Bertolino. “Software testing research: Achievements, challenges, dreams”. In: *2007 Future of Software Engineering*. IEEE Computer Society. 2007, pp. 85–103.
- [Boe81] Barry W. Boehm. *Software engineering economics*. Prentice-Hall advances in computing science and technology series. Prentice-Hall, 1981. ISBN: 9780138221225. URL: <http://books.google.de/books?id=VphQAAAAMAAJ>.
- [Boe88] Barry W Boehm. “A spiral model of software development and enhancement”. In: *Computer* 21.5 (1988), pp. 61–72.
- [Boo+98] Grady Booch et al. *Object-oriented analysis and design with applications*. 2nd edition. Addison-Wesley, 1998.
- [Bro87] Frederick P. Brooks. “No Silver Bullet – Essence and Accidents of Software Engineering”. In: *IEEE Computer* 20.4 (1987), pp. 10–19.
- [BMM09] Marcel Bruch, Martin Monperrus, and Mira Mezini. “Learning from examples to improve code completion systems”. In: *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ESEC/FSE ’09. Amsterdam, The Netherlands: ACM, 2009, pp. 213–222. ISBN: 978-1-60558-001-2. DOI: 10.1145/1595696.1595728. URL: <http://doi.acm.org/10.1145/1595696.1595728>.
- [Bru+10] Yuriy Brun et al. “Speculative analysis: exploring future development states of software”. In: *Proceedings of the FSE/SDP workshop on Future of software engineering research*. FoSER ’10. Santa Fe, New Mexico, USA: ACM, 2010, pp. 59–64. ISBN: 978-1-4503-0427-6. DOI: 10.1145/1882362.1882375. URL: <http://doi.acm.org/10.1145/1882362.1882375>.

- [CA78] Liming Chen and Algirdas Avižienis. “N-version programming: A fault-tolerance approach to reliability of software operation”. In: *Proceedings of the 8th IEEE International Symposium on Fault-Tolerant Computing (FTCS-8)*. 1978, pp. 3–9.
- [CWD08] Rylan Cottrell, Robert J. Walker, and Jörg Denzinger. “Semi-automating small-scale source code reuse via structural correspondence”. In: *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. SIGSOFT ’08/FSE-16. Atlanta, Georgia: ACM, 2008, pp. 214–225. ISBN: 978-1-59593-995-1. doi: 10.1145/1453101.1453130. url: <http://doi.acm.org/10.1145/1453101.1453130>.
- [CCL06] Ivica Crnkovic, Michel Chaudron, and Stig Larsson. “Component-Based Development Process and Component Lifecycle”. In: *Software Engineering Advances, International Conference on*. Oct. 2006, p. 44. doi: 10.1109/ICSEA.2006.261300.
- [DR12] B. Dagenais and M.P. Robillard. “Recovering traceability links between an API and its learning resources”. In: *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, 2012. June 2012, pp. 47–57. doi: 10.1109/ICSE.2012.6227207.
- [Ehr73] Gideon Ehrlich. “Loopless algorithms for generating permutations, combinations, and other combinatorial configurations”. In: *Journal of the ACM (JACM)* 20.3 (1973), pp. 500–513.
- [EKR03] Sebastian Elbaum, Srikanth Karre, and Gregg Rothermel. “Improving web application testing with user session data”. In: *Proceedings of the 25th International Conference on Software Engineering*. IEEE Computer Society. 2003, pp. 49–59.
- [Erl13] Oliver Erlenkämper. “Realizing Automated Test Recommendations in Software Development Environments”. Diploma Thesis. Chair of Software Engineering, University of Mannheim, Germany, June 2013.

- [Fal10] Giovanni Falcone. *Hierarchy-Aware Software Metrics in Component Composition Hierarchies*. Logos Verlag Berlin GmbH, 2010.
- [FHR91] Gerhard Fischer, Scott Henninger, and David Redmiles. “Cognitive tools for locating and comprehending software objects for reuse”. In: *Proceedings of the 13th international conference on Software engineering*. ICSE ’91. Austin, Texas, United States: IEEE Computer Society Press, 1991, pp. 318–328. ISBN: 0-89791-391-4. URL: <http://dl.acm.org/citation.cfm?id=256664.256813>.
- [FF95] William B. Frakes and Christopher J. Fox. “Sixteen questions about software reuse”. In: *Communications of the ACM* 38.6 (June 1995), 75–ff. ISSN: 0001-0782. doi: 10.1145/203241.203260. url: <http://doi.acm.org/10.1145/203241.203260>.
- [FP94] William B. Frakes and Thomas Pole. “An empirical study of representation methods for reusable software components”. In: *IEEE Transactions on Software Engineering* 20.8 (Aug. 1994), pp. 617–630. ISSN: 0098-5589. doi: 10.1109/32.310671.
- [FZ12] Gordon Fraser and Andreas Zeller. “Mutation-driven generation of unit tests and oracles”. In: *Software Engineering, IEEE Transactions on* 38.2 (2012), pp. 278–292.
- [Fri02] Jeffrey E. F. Friedl. *Mastering Regular Expressions*. Ed. by Andy Oram. 2nd ed. Sebastopol, CA, USA: O’Reilly & Associates, Inc., 2002. ISBN: 0596002890.
- [GO06] Leonard Gallagher and Jeff Offutt. “Automatically testing interacting software components”. In: *Proceedings of the 2006 international workshop on Automation of software test*. ACM. 2006, pp. 57–63.
- [Gam+94] Erich Gamma et al. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.
- [Gar+06] Vinicius C. Garcia et al. “Toward a Code Search Engine Based on the State-of-Art and Practice”. In: *Proceedings of the XIII Asia Pacific Software Engineering Conference*. APSEC ’06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 61–70. ISBN: 0-7695-2685-3. doi:

- 10.1109/APSEC.2006.57. URL: <http://dx.doi.org/10.1109/APSEC.2006.57>.
- [Git14] GitHub. *GitHub Collaboration Platform*. Jan. 2014. URL: <http://github-media-downloads.s3.amazonaws.com/GitHub.Quick.Facts.pdf> (visited on 01/18/2014).
- [GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. “DART: directed automated random testing”. In: *ACM Sigplan Notices*. Vol. 40. 6. ACM. 2005, pp. 213–223.
- [Goo11] Google. *A fall sweep*. Oct. 2011. URL: <http://googleblog.blogspot.de/2011/10/fall-sweep.html> (visited on 10/14/2011).
- [Hen93] Scott Henninger. “Locating relevant examples for example-based software design”. UMI Order No. GAX93-20432. PhD thesis. Boulder, CO, USA, 1993.
- [Hen97] Scott Henninger. “An evolutionary approach to constructing effective software reuse repositories”. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 6.2 (Apr. 1997), pp. 111–140. ISSN: 1049-331X. doi: 10.1145/248233.248242. URL: <http://doi.acm.org/10.1145/248233.248242>.
- [Hol04] Reid Holmes. “Using Structural Context to Recommend Source Code Examples”. Masters Thesis. University of British Columbia, 2004.
- [HM05] Reid Holmes and Gail C. Murphy. “Using structural context to recommend source code examples”. In: *Proceedings of the 27th international conference on Software engineering. ICSE ’05*. St. Louis, MO, USA: ACM, 2005, pp. 117–125. ISBN: 1-58113-963-2. doi: 10.1145/1062455.1062491. URL: <http://doi.acm.org/10.1145/1062455.1062491>.
- [HW07] Reid Holmes and Robert J. Walker. “Supporting the Investigation and Planning of Pragmatic Reuse Tasks”. In: *Proceedings of the 29th international conference on Software Engineering. ICSE ’07*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 447–457. ISBN:

- 0-7695-2828-7. doi: 10.1109/ICSE.2007.83. URL: <http://dx.doi.org/10.1109/ICSE.2007.83>.
- [Hum08] Oliver Hummel. “Semantic Component Retrieval in Software Engineering”. PhD thesis. University of Mannheim, 2008.
- [HA04] Oliver Hummel and Colin Atkinson. “Extreme Harvesting: test driven discovery and reuse of software components”. In: *Proceedings of the 2004 IEEE International Conference on Information Reuse and Integration, 2004. IRI 2004*. Nov. 2004, pp. 66–72. doi: 10.1109/IRI.2004.1431438.
- [HA06] Oliver Hummel and Colin Atkinson. “Using the Web as a Reuse Repository”. In: *Reuse of Off-the-Shelf Components*. Ed. by Maurizio Morisio. Vol. 4039. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, pp. 298–311. ISBN: 978-3-540-34606-7. doi: 10.1007/11763864_22. URL: http://dx.doi.org/10.1007/11763864_22.
- [HA10] Oliver Hummel and Colin Atkinson. “Automated creation and assessment of component adapters with test cases”. In: *Component-Based Software Engineering*. Springer, 2010, pp. 166–181.
- [HJ12] Oliver Hummel and Werner Janjic. “Towards Better Comparability of Software Retrieval Approaches Through a Standard Collection of Reusable Artifacts”. In: *Proceedings of the Seventh International Conference on Software Engineering Advances (ICSEA 2012)* (Nov. 2012), 450 to 458.
- [HJ13] Oliver Hummel and Werner Janjic. “Test-driven reuse: Key to improving precision of search engines for software reuse”. In: *Finding Source Code on the Web for Remix and Reuse*. Springer, 2013, pp. 227–250.
- [HJA07] Oliver Hummel, Werner Janjic, and Colin Atkinson. “Evaluating the efficiency of retrieval methods for component repositories”. In: *Proceedings of the International Conference on Software Engineering and Knowledge Engineering (SEKE), Boston*. 2007.

- [HJA08] Oliver Hummel, Werner Janjic, and Colin Atkinson. “Code Conjurer: Pulling Reusable Software out of Thin Air”. In: *IEEE Software* 25.5 (Sept. 2008), pp. 45–52. ISSN: 0740-7459. DOI: 10.1109/MS.2008.110. URL: <http://dx.doi.org/10.1109/MS.2008.110>.
- [HJA10] Oliver Hummel, Werner Janjic, and Colin Atkinson. “Proposing software design recommendations based on component interface intersecting”. In: *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*. RSSE ’10. Cape Town, South Africa: ACM, 2010, pp. 64–68. ISBN: 978-1-60558-974-9. DOI: 10.1145/1808920.1808936. URL: <http://doi.acm.org/10.1145/1808920.1808936>.
- [Hum+06] Oliver Hummel et al. “Improving Testing Efficiency through Component Harvesting”. In: *Proceedings of the Brazilian Workshop on Component Based Development: WDBC 2006*. CESAR. Recife, Brazil, Dec. 2006.
- [Ino+05] Katsuro Inoue et al. “Ranking Significance of Software Components Based on Use Relations”. In: *IEEE Trans. Softw. Eng.* 31.3 (Mar. 2005), pp. 213–225. ISSN: 0098-5589. DOI: 10.1109/TSE.2005.38. URL: <http://dx.doi.org/10.1109/TSE.2005.38>.
- [Jac+99] Ivar Jacobson et al. *The unified software development process*. Vol. 1. Addison-Wesley Reading, 1999.
- [Jan07] Werner Janjic. “Realising High-Precision Component Recommendations for Software-Development Environments”. Diploma Thesis. Chair of Software Technology, University of Mannheim, Dec. 2007.
- [JA12] Werner Janjic and Colin Atkinson. “Leveraging software search and reuse with automated software adaptation”. In: *ICSE Workshop on Search-Driven Development - Users, Infrastructure, Tools and Evaluation (SUITE)*, ICSE’12. June 2012, pp. 23–26. DOI: 10.1109/SUITE.2012.6225475.

- [JA13] Werner Janjic and Colin Atkinson. “Utilizing software reuse experience for automated test recommendation”. In: *8th International Workshop on Automation of Software Test (AST), 2013*. IEEE. 2013, pp. 100–106.
- [JHA10] Werner Janjic, Oliver Hummel, and Colin Atkinson. “More archetypal usage scenarios for software search engines”. In: *Proceedings of 2010 ICSE Workshop on Search-driven Development: Users, Infrastructure, Tools and Evaluation*. SUITE ’10. Cape Town, South Africa: ACM, 2010, pp. 21–24. ISBN: 978-1-60558-962-6. doi: 10.1145/1809175.1809181. URL: <http://doi.acm.org/10.1145/1809175.1809181>.
- [JHA14] Werner Janjic, Oliver Hummel, and Colin Atkinson. “Recommendation Systems in Software Engineering”. In: ed. by Martin P. Robillard et al. Springer, 2014. Chap. Reuse-Oriented Code Recommendation Systems.
- [Jan+11] Werner Janjic et al. “Discrepancy Discovery in Search-Enhanced Testing”. In: *Proceedings of the 3rd International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation*. ACM. 2011, pp. 21–24.
- [Jan+13] Werner Janjic et al. “An Unabridged Source Code Dataset for Research in Software Reuse”. In: *Proceedings of the Tenth Working Conference on Mining Software Repositories (MSR’13)*. IEEE Press. San Francisco, CA, USA, 2013, pp. 339–342.
- [JSO14] Yahoo Group on JSON. *Introducing JSON*. Jan. 2014. URL: <http://www.json.org/> (visited on 03/24/2014).
- [Ker95] Norman L. Kerth. “Pattern Languages of Program Design”. In: ed. by James O. Coplien and Douglas C. Schmidt. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1995. Chap. Caterpillar’s Fate: A Pattern Language for the Transformation from Analysis to Design, pp. 293–320. ISBN: 0-201-60734-4. URL: <http://dl.acm.org/citation.cfm?id=218662.218684>.

- [KL86] John C. Knight and Nancy G. Leveson. “An experimental evaluation of the assumption of independence in multiversion programming”. In: *IEEE Transactions on Software Engineering* 1 (1986), pp. 96–109.
- [KL90] John C. Knight and Nancy G. Leveson. “A reply to the criticisms of the Knight & Leveson experiment”. In: *ACM SIGSOFT Software Engineering Notes* 15.1 (1990), pp. 24–35.
- [KS98] Adam K Kolawa and Roman Salvador. *Method and system for generating a computer program test suite using dynamic symbolic execution of Java programs*. Tech. rep. US Patent 5,784,553. Parasoft Corporation, July 1998.
- [Kru92] Charles W. Krueger. “Software reuse”. In: *ACM Comput. Surv.* 24.2 (June 1992), pp. 131–183. ISSN: 0360-0300. doi: 10.1145/130844.130856. URL: <http://doi.acm.org/10.1145/130844.130856>.
- [LFL98] Thomas K. Landauer, Peter W. Foltz, and Darrell Laham. “An introduction to latent semantic analysis”. In: *Discourse Processes* 25.2–3 (1998), pp. 259–284.
- [LT12] Mathias Landhäußer and Walter F. Tichy. “Automated Test-Case Generation by Cloning”. In: *Proc. of the 7th International Workshop on Automation of Software Test (AST 2012)*. June 2012.
- [LM89] Beth M. Lange and Thomas G. Moher. “Some strategies of reuse in an object-oriented programming environment”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’89. New York, NY, USA: ACM, 1989, pp. 69–73. ISBN: 0-89791-301-9. doi: 10.1145/67449.67465. URL: <http://doi.acm.org/10.1145/67449.67465>.
- [Laz+09] Otávio Augusto Lazzarini Lemos et al. “Applying test-driven code search to the reuse of auxiliary functionality”. In: *Proceedings of the 2009 ACM symposium on Applied Computing*. SAC ’09. Honolulu, Hawaii: ACM, 2009, pp. 476–482. ISBN: 978-1-60558-166-8. doi: 10.1145/1529282.1529384. URL: <http://doi.acm.org/10.1145/1529282.1529384>.

- [Lem+07] Otávio Augusto Lazzarini Lemos et al. “CodeGenie: using test-cases to search and reuse source code”. In: *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ASE ’07. Atlanta, Georgia, USA: ACM, 2007, pp. 525–526. ISBN: 978-1-59593-882-4. DOI: 10.1145/1321631.1321726. URL: <http://doi.acm.org/10.1145/1321631.1321726>.
- [Man+05] David Mandelin et al. “Jungloid mining: helping to navigate the API jungle”. In: *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. PLDI ’05. Chicago, IL, USA: ACM, 2005, pp. 48–61. ISBN: 1-59593-056-6. DOI: 10.1145/1065010.1065018. URL: <http://doi.acm.org/10.1145/1065010.1065018>.
- [McI69] M. Douglas McIlroy. *Mass Produced Software Components*. Tech. rep. NATO, 1969, pp. 138–155.
- [MPG10] Collin McMillan, Denys Poshyvanyk, and Mark Grechanik. “Recommending source code examples via API call usages and documentation”. In: *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*. RSSE ’10. Cape Town, South Africa: ACM, 2010, pp. 21–25. ISBN: 978-1-60558-974-9. DOI: 10.1145/1808920.1808925. URL: <http://doi.acm.org/10.1145/1808920.1808925>.
- [Med14] Slashdot Media. *SourceForge*. Jan. 2014. URL: <http://www.sourceforge.net> (visited on 01/16/2014).
- [Mey92] Bertrand Meyer. “Applying ‘design by contract’”. In: *IEEE Computer* 25.10 (1992), pp. 40–51.
- [MMM98] A. Mili, R. Mili, and R. T. Mittermeir. “A survey of software reuse libraries”. In: *Ann. Softw. Eng.* 5 (Jan. 1998), pp. 349–414. ISSN: 1022-7091. URL: <http://dl.acm.org/citation.cfm?id=590631.590637>.
- [Muş+12a] Kivanç Muşlu et al. “Improving IDE recommendations by considering global implications of existing recommendations”. In: *Proceedings of the 2012 International Conference on Software Engineering*. IEEE Press. 2012, pp. 1349–1352.

- [Muş+12b] Kivanç Muşlu et al. “Speculative analysis of integrated development environment recommendations”. In: *ACM SIGPLAN Notices* 47.10 (2012), pp. 669–682.
- [Mye79] Glenford J. Myers. *The Art of Software Testing*. John Wiley & Sons, Inc., 1979.
- [MS04] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, Inc., 2004. ISBN: 0471469122.
- [Nea96] Lisa Neal. “Structure-based editors and environments”. In: ed. by Gerd Szwillus and Lisa Neal. Orlando, FL, USA: Academic Press, Inc., 1996. Chap. Support for software design, development and reuse through an example-based environment, pp. 185–192. ISBN: 0-12-681890-8. URL: <http://dl.acm.org/citation.cfm?id=242222.242511>.
- [OB88] Thomas J. Ostrand and Marc J. Balcer. “The category-partition method for specifying and generating functional tests”. In: *Communications of the ACM* 31.6 (1988), pp. 676–686.
- [Red13] RedMonk. *The RedMonk Programming Language Rankings: June 2013*. July 2013. URL: <http://redmonk.com/sogrady/2013/07/25/language-rankings-6-13/>.
- [Rei09] Steven P. Reiss. “Semantics-based code search”. In: *Proceedings of the 31st International Conference on Software Engineering*. ICSE ’09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 243–253. ISBN: 978-1-4244-3453-4. doi: 10.1109/ICSE.2009.5070525. URL: <http://dx.doi.org/10.1109/ICSE.2009.5070525>.
- [RAO92] Debra J. Richardson, Stephanie Leif Aha, and T. Owen O’malley. “Specification-based test oracles for reactive systems”. In: *Proceedings of the 14th international conference on Software engineering*. ACM. 1992, pp. 105–118.
- [RWZ10] Martin Robillard, Robert J. Walker, and Thomas Zimmermann. “Recommendation Systems for Software Engineering”. In: *IEEE Software* 27.4 (July 2010), pp. 80–86.

- [Roy70] Winston W Royce. “Managing the development of large software systems”. In: *proceedings of IEEE WESCON*. Vol. 26. 8. Los Angeles. 1970.
- [SO03] Thomas L. Saaty and Mujgan Ozdemir. “Negative priorities in the analytic hierarchy process”. In: *Mathematical and Computer Modelling* 37.9 (2003), pp. 1063–1075.
- [SE05] David Saff and Michael D. Ernst. “Continuous Testing in Eclipse”. In: *Proceedings of the 27th International Conference on Software Engineering*. ICSE ’05. St. Louis, MO, USA: ACM, 2005, pp. 668–669. ISBN: 1-58113-963-2. doi: 10.1145/1062455.1062600. URL: <http://doi.acm.org/10.1145/1062455.1062600>.
- [San96] N. Sanders. “Automated testing using executable formal specifications”. In: *Proceedings of the International Conference on Software Engineering: Education and Practice*. Jan. 1996, pp. 176–181.
- [Sea99] Robert C. Seacord. “Software engineering component repositories”. In: *Proceedings of the International Workshop on Component-Based Software Engineering*. 1999.
- [SHW98] Robert C. Seacord, Scott A. Hissam, and Kurt C. Wallnau. “Agora: A Search Engine for Software Components”. In: *IEEE Internet Computing* 2.6 (Nov. 1998), pp. 62–70. ISSN: 1089-7801. doi: 10.1109/4236.735988. URL: <http://dx.doi.org/10.1109/4236.735988>.
- [SA06] Koushik Sen and Gul Agha. “CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools”. In: *Computer Aided Verification*. Springer. 2006, pp. 419–423.
- [Soc14] Audio Engineering Society. *The Digital Revolution*. May 2014. URL: <http://www.aes.org/aeshc/docs/recording.technology.history/digital.html> (visited on 05/22/2014).
- [Som10] Ian Sommerville. *Software Engineering*. 9th ed. Addison-Wesley Publishing Company, 2010.

- [SWH11] Matt Staats, Michael W. Whalen, and Mats P.E. Heimdalh. “Programs, tests, and oracles: the foundations of testing revisited”. In: *Software Engineering (ICSE), 2011 33rd International Conference on*. IEEE. 2011, pp. 391–400.
- [Str83] Howard Straubing. “A combinatorial proof of the Cayley-Hamilton theorem”. In: *Discrete Mathematics* 43.2 (1983), pp. 273–279.
- [Szy02] Clemens Szyperski. *Component software: beyond object-oriented programming*. Pearson Education, 2002.
- [TT10] Gerald Teschl and Susanne Teschl. *Mathematik für Informatiker: Band 1: Diskrete Mathematik und Lineare Algebra*. Vol. 1. Springer DE, 2010.
- [TX07] Suresh Thummalapenta and Tao Xie. “Parseweb: a programmer assistant for reusing open source code on the web”. In: *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ASE ’07. Atlanta, Georgia, USA: ACM, 2007, pp. 204–213. ISBN: 978-1-59593-882-4. doi: 10.1145/1321631.1321663. URL: <http://doi.acm.org/10.1145/1321631.1321663>.
- [TIO14] TIOBE. *TIOBE Programming Community Index for January 2014*. Jan. 2014. URL: <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>.
- [USL08] M. Umarji, Susan E. Sim, and Cristina Videira Lopes. “Archetypal internet-scale source code searching”. In: *Open source development, communities and quality* (2008), pp. 257–263.
- [Vou90] Mladen A. Vouk. “Back-to-back testing”. In: *Information and software technology* 32.1 (1990), pp. 34–45.
- [Wal+98] Stephen Walker et al. “Okapi at trec-6: Automatic ad hoc, vlc, routing, filtering and qsdr.” In: *The 6th Text REtrieval Conference (TREC-6)* (1998), pp. 125–136.
- [Wey82] Elaine J. Weyuker. “On testing non-testable programs”. In: *The Computer Journal* 25.4 (1982), pp. 465–470.

- [Ye01] Yunwen Ye. “Supporting Component-Based Software Development with Active Component Repository Systems”. PhD thesis. Department of Computer Science, University of Colorado, Boulder, CO, 2001.
- [YF02] Yunwen Ye and Gerhard Fischer. “Supporting reuse by delivering task-relevant and personalized information”. In: *Proceedings of the 24th International Conference on Software Engineering*. ICSE ’02. Orlando, Florida: ACM, 2002, pp. 513–523. ISBN: 1-58113-472-X. DOI: 10.1145/581339.581402. URL: <http://doi.acm.org/10.1145/581339.581402>.
- [Yok+03] Reishi Yokomori et al. “Java program analysis projects in Osaka University: aspect-based slicing system ADAS and ranked-component search system SPARS-J”. In: *Proceedings of the 25th International Conference on Software Engineering*. ICSE ’03. Portland, Oregon: IEEE Computer Society, 2003, pp. 828–829. ISBN: 0-7695-1877-X. URL: <http://dl.acm.org/citation.cfm?id=776816.776972>.
- [ZW95] Amy Moermann Zaremski and Jeannette M Wing. “Signature matching: a tool for using software libraries”. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 4.2 (1995), pp. 146–170.

D

Index

- abstract syntax tree, 119
- adaptee, 57
- adapter, 57
- Agora, 41, 103
- assertEquals, 124
- assertion, 20
- assertTrue, 124
- AST, 119
- autoboxing, 147
- B-tree, 116
- back-to-back testing, 6, 178, 183
- background agent, 52
- Brahmagupta, 198
- business objects, 127
- Caterpillar's Fate, 51
- characterizing test case, 187
- class under test, 17, 122
- Code Conjurer, 5, 45, 52, 87, 165
- Code Finder, 73
- Code Genie, 46
- code inspiration, 38
- Code Recommenders, 84
- code scavenging, 92
- code-based search, 155, 156
- CodeBroker, 75
- CodeFinder, 41
- CodeGenie, 81
- CodeRank, 46
- collection, 116
- component under test, 17
- concolic testing, 4
- continuous speculative testing, 174
- continuous testing, 174
- correct output, *see* expected result
- CUT, 17, 111
- DCOM, 17
- denotational semantics methods, 139
- deployment, 137
- design by contract, 139
- design prompter, 38
- design scavenging, 92
- digital revolution, 3

directed automated random testing, 4
Discourse Model, 76
document, 116
domain expert, 136
drag and drop reuse, 87
dynamic symbolic execution, 4

Eclipse, 78
Eclipse Code Recommenders, 5
Enterprise Java Beans, 17
equivalent mutant, 7
evaluation
 ex ante, 7, 91, 174
 ex post, 6, 174
everything is an object, 147
example recommendation, 78
exception test, 144
execution, 17
execution driver, 194
expected result, 18, 20
Extreme Programming, 49

false positive, 182, 216
false positives, 199
Fetcher, 75
fixture, 123
function oriented query, 44

Gang of Four, 56
genetic algorithms, 4
GitHub, 117
Glass-Box Reuse, 38
Google Codesearch, 41, 43
goto fail bug, 4, 23

heartbleed bug, 4
human oracle, 6

implementation, 135
in-between class, 57
information hiding, 59
interface mismatch, 55
Invocation, 114
invocation, 17
invocation table, 143
IPO model, 16

Java, 17
JUnit, 17, 20

k-permutation, 59
kiddie testing, 19
Knight and Leveson Experiment, 184
KobrA, 17
Koders.com, 43
Krugle, 43

Listener, 75

maintenance, 137
Merobase, 41, 43, 50, 56, 87, 106, 118
Merobase Query Language, 44, 109
method, 17
MongoDB, 115
multi-version testing, 190
Multi-Version Testing Environment, 192
mutation testing, 7

n-version programming, 178, 182
non-functional requirements, 137
not-invented-here syndrome, 136

- nutch, 117
- NVP, 182
- object-oriented query, 44
- Ohloh Code, 43
- open-source revolution, 33
- operation, 17
- operational semantics methods, 149
- oracle, 6, 19, 20, 40
 - golden, 19
- PageRank, 46
- PARSEWeb, 84
- PECOS, 17
- penetration testing, 137
- plain old Java objects, 27
- pragmatic reuse, 92
- Precision, 48
- Presenter, 76
- proactive approach, 75
- provided interface, 109, 139
- pull approach, 75
- push approach, 75
- Quick Fix, 89
- RCI, 76
- reactive approach, 75
- Recall, 48
- regular expression, 153
- repository problem, 103
- representation problem, 68
- required interface, 56, 109
- requirements, 135
- retrieval, 138
- reuse, 5
 - library, 40
 - test, 40
 - test-driven, 39
- reuse-by-memory, 77, 95
- reuse-oriented recommendation system, 67
- reuse-oriented software testing, 6
- ROCR, 67
- ROCR System, *see* ROCR
 - S^6 , 41, 106
- search
 - definitive, 10, 134, 137
 - interface-based, 39
 - speculative, 10, 36, 134, 141
 - test-driven, 44, 46, 83, 89
 - value-based, 149, 150
- search scenarios, 134
- Search-Enhanced Testing, 9, 13, 56, 182
- SENTRE, 8, 104, 115, 118, 133
- SET, 182
- smoke testing, 135
- SOFA, 17
- software testing, 4, 15
- software under test, 17
- Software-Reuse Environment, 87
- Sourcerer, 41, 106
- SPARS-J, 43
- specification, 135
- specification-based operational semantics method, 150
- speculative analysis, 7, 170
- Strathcona, 5, 78

stress testing, 137
system under test, 6

test, 19, 20
test automation, 4
test case, 20, 21, 26
test case values, 17, 20, 111, 114
test oracle, 5
test result, 20
test reuse scenarios, 134
test suite, 21
test-driven development, 81, 135
test-driven reuse, 49, 135, 179, 187
test-driven search, 156
test-sheets, 143
testing, 137
 discrepancy driven, 40
tests, 21

uses- and calls-relation, 139

wildcard, 141
wrapper class, 147