Realizing Automated Test Recommendations in Software Development Environments

Diploma Thesis

by

Oliver Erlenkämper

presented at

Chair of Software Engineering
Prof. Dr. Colin Atkinson
Department of Business Informatics and Business Mathematics
University of Mannheim

Supervisor:
Dipl.-Wirtsch.-Inf. Werner Janjic

15.06.2013

Abstract

Software testing is a mainly manually performed and thus very labour intensive process. Beside time, it demands a high amount of domain knowledge, concentration and problem awareness from the developer. Although software reuse is a well examined area –in both academia and industry – it is mainly focussed on the reuse of different kinds of documentation and program code. In this thesis we create a client-side recommendation system for the novel idea for an automated test recommendation approach that is based on lessons learned from traditional software reuse and recommendation. While most existing testing-assistance systems help a developer by providing information about various coverage criteria only ex post, we want to support the developer pro-actively while writing the test and create as little overhead as possible during his work. Thereby we benefit from the lessons learned in the area of ”traditional” software reuse and apply them in a kind of test reuse for test recommendation approach. To validate our theoretical considerations, we present a tool that will help writing tests with less effort.

Contents

List of Abbreviations

API	Application Programming Interface
AST	Abstract Syntax Tree
CC	Code Conjurer
CUT	Component Under Test
DTO	Data Transfer Object
FQDN	Fully Qualified Domain Name
IDE	Integrated Development Environment
JDK	Java Development Kit
JRE	Java Runtime Environment
OO	Object-Oriented
SDD	Search-Driven Development
SR	Software Reuse
ST	Software Testing
TDD	Test-Driven Development
TT	Test Tenderer
UI	User Interface

List of Figures

List of Tables

CHAPTER 1

Introduction

Over the last decades, both Testing and Reuse have become valuable companions in Software Engineering. Nowadays, there is no doubt, that goal-oriented, comprehensive and organized testing increases the quality of software [1] and it is thus very important for successful software development [2]. Testing is widely supported by mature testing frameworks [3] as integral part of today's software development. Reuse is also considered significantly beneficial as it reduces defect density and thus increases productivity [4], which has even been confirmed for industrial practice [5, 6]. While the combination of these promising technologies is considered beneficial as well [7], a unified implementation of *Test Reuse* is also still missing [8]. Interestingly, there is support available for both disciplines (e.g. [9,10] respectively [3]), but not for both *together*. Reasons for this can be found in inadequate databases, unattractive, pure web search engines or simply too high effort for the adaptation of reuse candidates to own needs [11].

This consequently means that developers are expected to discover even the most subtle failure on the one, but also, that they can not fall back on support for this issue on the other hand. It is striking that an adequate, unified, and integrated solution could tackle this deplorable state of affairs and thus provide rich benefit for test developers.

In this work the prototypical tool *Test Tenderer* is introduced, which aims to shift Test Reuse to application level. Test Tenderer is designed as an Eclipse plugin and utilizes the popular JUnit framework for unit testing in Java. It utilizes information about tested components, methods, and assertion statements extracted from test cases of a cooperating database. With information collected from the local project Test Tenderer is capable of searching for appropriate artifacts, adapting them to the project-specific needs, investigating their impact on the test under development, and finally of providing convenient insertion functionality for the specific knowledge.

In order to provide the user with rich value, Test Tenderer is also designed to speculatively analyze the assertion proposals and to provide ex-ante coverage information for each proposal. This enables this plugin to illustrate the impact of insertion before it actually takes place. Besides that, the user will be provided with the opportunity to inspect the source code of suitable reuse candidates. Finally, Test Tenderer automates continuous testing by recurring background JUnit runs and is hereby able to provide developers with non-intrusive live feedback about their testing effort.

The following chapters lay the foundations for this work, explain design and implementation details, and finally summarize the impact of this work on the field of Test Reuse: Chapter 2 presents the theoretical background, especially for Reuse, Testing, JUnit and Speculative Analysis. In chapter 3, the design of the developed plugin is illustrated. In particular the steps of Preparation, Search, Speculative Analysis and Proposal Generation are highlighted. Chapter 4 provides an insight into the implementation, starting with the integration of the plugin into Eclipse. Furthermore details about the communication with the database are given followed by particulars about the architecture. Finally, the work flow is illustrated. Chapter 5 evaluates the results against the theoretical background. In addition, the system performance

as well as Assisted Reuse is subject to this assessment. Chapter 6 concludes the work and provides starting points for future work and improvements.

CHAPTER 2

Foundations

In this chapter the foundations of this work are presented. These primarily include the areas of Testing, JUnit, Reuse and Speculative Anlyse. In the respective sections a broad insight into the subject is initially provided. At the end of each section the information presented above are combined and reduced to the specific application scenario of this work. Finally, a brief summary of the main findings follows at the end of the chapter.

2.1 Software Testing

In short, Software Testing (ST) is "the process of executing a software system to determine whether it matches its specification and executes in its intended environment" [2]. In more detail, it is also discovering and eliminating errors, ensuring consistent functionality and avoiding rework requirements. Goal-oriented, comprehensive and organized testing avoids unnecessary costs [12, 13], increases the quality of software [1] and is thus very important for successful software development [2]. ST should thus be an essential companion of any software development process.

2.1.1 Motivation for Testing

Some studies indicate that the cost to fix a software problem after delivery is up to 100 times higher than if the problem had been detected and eliminated during the design phase [12]. In addition, sometimes between 40 and 50 percent of the total project budget are spent on avoidable rework. If funds are no longer available at the end, faulty software may even be delivered to the customer. According to [12] up to 50% of software contains non-trivial defects. The problem here is that these bugs often become apparent only when they are reported by the user.

Reasons for failure

Assumed, a reported bug is in fact an error, the reason for a missed early discovery may be one of the following:

- the code executed by the user was not tested
- the execution order of statements in the actual usage differs from the tested one
- the particular user input was not tested
- the applicability to the user's operating environment was not tested

Certainly, this list could be expanded by numerous situations. The underlying problem is nevertheless the same: inadequate testing. However, 60% of the defects may be eliminated through peer review and the

defect introduction rate is reduced by up to 75% through the use of suitable personnel with appropriate testing discipline [12].

Consequences of failure

Besides the already mentioned financial consequences of reworking effort, even bigger problems could result from inadequate tested software. These so-called "war stories" clearly show how important it is to test. The following examples were taken from [14] and [1].

- In 1999, the Mars lander, respectively the Mars Climate Orbiter, crashed when the landing procedure was initiated. The reason for the loss of this 338 kilogram NASA space probe was a software design error. Two independently developed modules were designed with different units of measure, which led to a “misunderstanding” between these software units. While one of them used the English units to compute thruster data, the other module expected metric units. This failure lead to the loss of much money and prestige.
- In 1996 the Ariane 5 rocket exploded 40 seconds after take off. Reason for this safety self-destruction was a system crash caused by a software failure. The internal reference system tried to convert a 64 bit floating point number to a 16 bit unsigned integer. Unfortunately the result of 32.767 was beyond the limits of this 16 bit machine and the conversion failed. The loss amounted to $ 7 billion.
- The London Ambulance system was equipped with a new dispatch control software in 1992. Unfortunately the system was not able to cope with the huge volume of 1.500 emergency calls per day and the software broke down. Even worse: the system didn’t crash, but behaved incorrectly. Positions of vehicles were incorrectly recorded, with the consequence that a possibly large number of ambulances were sent to the same location, while other emergencies were completely ignored for several hours. The assumed repair cost was about £9 million, what almost sounds affordable compared to the fact that 20 lives could have presumably be saved.

These examples show the need for full and adequate software testing at each development stage. Not only against the background of impending (although supposedly insured) costs of repair or compensation payments, but also because human lives may depend on it.

2.1.2 Classification by type

When classified according to type, software testing can be divided into four categories which are independent from but complementary to each other [14]. Consequently, Security Testing, Reliability Testing, Performance Testing, and Correctness Testing can be distinguished.

Security Testing

The goal of security testing is to discover weaknesses of the system, which may be used to harm the system, from an internal as well as an external perspective. Consequently, the following disciplines have to be taken into account:

- *Security Auditing & Scanning*
  The subjects of investigation are the operating systems on which the software is to run, as well as the one it was developed on. The goal is to identify weaknesses in these systems and, if applicable, in the network.
- *Vulnerability Scanning*
  This task is usually accomplished with the help of standardized software. The goal is to identify possible leaks which may harm the system.

- *Risk Assessment*
  Risk Assessment aims at estimating the risk of using the software. It is accomplished by analyzing the necessities and requirements of the potential users. The goal is to identify the probability of loss related to these risks.

- *Posture Assessment*
  This analysis type aims at the identification of the position among competitors in the context of security.

- *Penetration Testing*
  Penetration Testing is conducted in order to recognize whether a system provides potential loopholes. The goal is to secure these vulnerabilities and thus to prevent the access of unauthorized users to the system.

- *Ethical Hacking*
  The attempt to access the system without permission is in the focus of Ethical Hacking. With a large number of penetration tests ethical hackers try to discover leaks of the system and therewith to intrude into the system. For this purpose ethical hackers use the same techniques that would also be used by unauthorized hackers.

Performance Testing

Performance testing deals with the feasibility of a system to serve the users in a timely and reliable manner [14]. It thus aims to evaluate the performance of the system with respect to real world scenarios. In order to test the performance of a system, two particular methods are mainly applied.

- *Load testing*
  The key object of Load Testing is the ability of the system to handle the requests, respectively the number of users accessing the system [14]. A load test provides the system with maximum load and investigates whether the system is able to handle it. Load tests are often conducted for web applications, as the server needs to be able to bear the load even during peak hours [14].

- *Stress testing*
  Stress Testing applies load tests beyond the limits of the system and hereby provides the opportunity to inspect the behavior of the system under unanticipated conditions. For this purpose, the system is "stressed" with random operation sequences, huge load, partly beyond the limits, for long periods [14]. This technique therefore also serves reliability measurement purposes.

Reliability Testing

The main purpose of reliability testing is the discovery of failures of a system before deployment [1, 14]. A key subject of investigation is the robustness of the system, which can be determined with Robustness Testing. Together with stress testing, the overall reliability of the system can be assessed and the risk of using the software can be estimated. The deployment of the system can then be finally initiated on the basis of this information.

- *Robustness testing*
  In order to test the robustness of a system several non-expected values may be applied to check the behavior of the software. These can be purely invalid values, as well as values below the specified minimum and beyond the required maximum [14]. The goal of robustness testing is to determine whether the system is able to handle invalid parameters in order to avoid failure or unexpected output values [14].

Correctness Testing

Besides Reliability Testing, Correctness Testing is the second major area of interest [15]. The goal here is to ensure a correct behavior of the system. Common testing approaches of this technique are White-Box Testing, Black-Box Testing, and Gray-Box Testing, although their applicability is not limited to this testing type.

- *White-Box Testing*
  In a White-Box Testing scenario, the structure and the source code of the component under test are known [1]. For this reason, this testing technique is also known as Structural Testing [14]. The goal is to derive oracles from the inspected implementation in order to build appropriate tests [1, 14, 15]. These oracles are subsequently tested against the de-facto behavior of the units under test to check their functionality for correctness. Appropriate test cases are usually created by the developer often already during the development of the tested system.
- *Black-Box Testing*
  In contrast to White-Box Testing, in a Black-Box Testing scenario only the functionality and the interfaces of the components under test are known. The internal structure is completely hidden. Tests can be derived from external software descriptions, such as specifications, requirements and design [1]. For this reason, black-box testing is also known as *functional testing* [14]. Tests are usually conducted by specific testers, which are usually not the developers of the system under test.
- *Gray-Box Testing*
  This testing technique represents a mixture of these both approaches. It combines the openness of the white-box approach with the structured and functionality-oriented methodology of Black-Box Testing. For this purpose, the test cases are derived by the developer, but designed only based on the functionality of the system. This procedure ensures that that no potentially error-prone statements are neglected.

2.1.3 Classification by level

Testing takes place at several stages in software development. A popular model for the different implementation and testing stages is the v-model (Figure 2.1, in the version of [14]), which supposedly goes back to Barry Boehm in the late 1960's. This model exists in numerous versions, but two key points are present in all current versions. (1) there are four[1] different levels of testing, namely Unit Testing, Integration Testing, System Testing, and Acceptance Testing and (2) testing is to be considered from the early development stages[2].In other words, adequate tests should be designed accordingly even in the early design phases of the software product. Of course, the testing *process* can only take place when the software is implemented. Nevertheless should the outcomes of these test lead to refinements of the regarding design.

Unit Testing

Unit Testing is viewed as essential for software quality, because with this technique even subtle or deeply hidden failures may be discovered [13]. For this purpose, several "smallest testable piece[s] of software, which may consist of hundreds or even just a few lines of source code" [16], are tested. The goal of these unit tests is to check the behavior of the unit under test and thus to ensure, that the specified functionality and design specifications are met [16]. Typically, units are represented by methods or (at most) classes. Unit tests are often written by the developer of the corresponding unit [1], which is useful, because an estimation of functionality and especially of the expected results is much easier for

[1] in Figure 2.1 the first and second level are summarized

[2] The dashed arrows indicate the necessity of considering Testing from the early development states

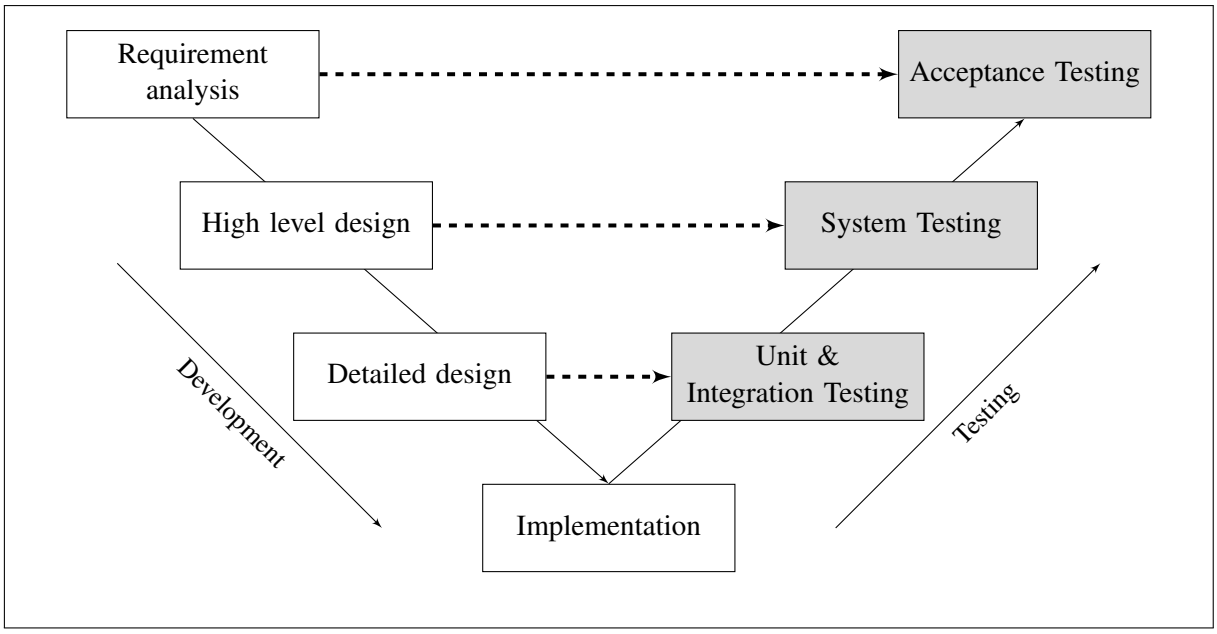


Figure 2.1: V-Model

the developer than for an uninvolved tester. Unit testing can be done with functional and/or structural testing techniques [14]. However, unit testing may be problematic, if the tested units are not completely independent. This requires additional testing effort, because the units possibly need to be modified to be isolated testable [14].

Integration Testing

The next higher level of consideration is the Integration Testing. In contrast to unit tests, the focus is on reviewing the overall behavior and interaction of the units. Integration Testing aims at "verifying that each component interacts according to its specifications as defined during preliminary design" [16]. Subject of according tests is thus usually the interface of the particular unit. The level of dependence between units is referred to as coupling. The higher the coupling rate, the more tests should be performed for the *overall behavior* of the interacting units. This effort is hampered by the fact, that additional code is likely to be added in order to test the units sufficiently. This is due to possibility, that some states may not arise in a sequential interaction when the elements show high coupling rates. In contrast, at a low coupling rate, the interfaces of the units become important. Consequently, these interfaces should then become the primary test subject. However, there are recommendations for reduction [14], since finally a good design should have low coupling:

- only data should be passed, not control information
- no undesired data should be passed
- the number of parameters to be passed between two units should be minimized
- no complete data structures should be passed
- no global variables should be declared

All in all, this may be the most complex test step, depending on how accurately the different units were developed. However, unit and integration testing are the primary subjects of *Regression Testing*, which aims at repeating test runs after every alteration and which has been widely scaled up to industrial systems [17]. Regression tests are continuous automated tests that check the functionality of a unit, a program or a system with corrective, perfective, adaptive, or preventive changes.

System Testing

System testing aims to test the entire software in its expected environment [14] and is performed after unit and integration testing. The means for an end for this testing level are (not exclusively) functional testing techniques. Because a system, in contrast to software, is a combination of multiple components, such as the program, the environment, the hardware, the operation system etc., it is essentially important to ensure the fulfillment of all functional requirements. Furthermore, non-functional requirements are taken into account, such as security, performance, reliability, or recovery of the system [16]. This task is performed by explicit testers. In this phase, design changes are likely not feasible and may cause limited functionality. However, as this is the first time the whole system is operated in the target environment, this step is inevitable to ensure final quality [14].

Acceptance Testing

Acceptance Testing may be seen as extension of System Testing [14] and can be started as soon as the system tests are finished. Usually, system tests are conducted together with the respective customer [1, 14] and aim to ensure, that the system meets the customer's needs [1]. For this purpose, the final product is presented to the customer, who in turn begins to evaluate the usability of the program and how precisely the requirements have been realized. If the product is not designed for a specific customer but for anonymous users, acceptance testing is not feasible. In such cases, the software is tested by *potential* customers with the help of two alternative procedures:

- *Alpha Testing*
  Alpha Testing is performed by a distinct number of selected potential customers under the supervision of developers. It generally takes place at the developer's site.
- *Beta Testing*
  Beta testing is usually accomplished by many different users without any involvement of the developers. The versions for this type of testing are called beta-versions. The goal is to gather up information about the performance as well as possible failures of the system.

In summary, acceptance tests are tests conducted about the whole system with the goal of meeting the customers' requirements. The main focus of this test step is the usability of the system and the intent is to "verify that the effort required from end-users to learn to use and fully exploit the system functionalities is acceptable" [16].

2.1.4 Testing Techniques

Static Testing

Especially in the early stages of software development, static testing methods are widely applicable. All methods, that do not require the execution of the program, respectively any code, fall in this category. Established in the early phases of software development, this method gives good results at reasonable cost [14]. This technique is often used for the inspection of documents and documentations created in each development phase. This testing technique therefore serves the purpose of verifying the software.

Dynamic Testing

Dynamic Testing includes all techniques that require the execution of the software. Subject of this technique is therefore the *functionality* of the system. For that reason, it serves the purpose of validation, by which the functionality is tested against real world environments. Consequently, domain-specific knowledge is necessary [1]. It is only with dynamic testing methods that failures can be experienced and the reasons for these may be identified [14]. Dynamic testing is thus inevitable to ensure the reliability of the system.

Functional Testing

In an functional testing scenario, the structure of the component under test is unknown. This is why the functionality is the central point of investigation. This testing technique is therefore also called Black-Box Testing. Functional testing aims primarily at finding test cases that make the software fail [14]. Furthermore, every possible functionality is attempted to test. As the execution of the program is inevitable for this technique, it also serves validation purposes. It is applicable to every testing level and thus helps testers to efficiently and effectively find software faults [14]. Subcategories of functional testing are [14]:

- *Boundary Value Analysis*
  Boundary Value Analysis is a popular testing technique which focuses on the creation of a test on or close to boundary values of the system, which have a higher probability of detecting a fault in the software [14].
- *Equivalence Class Testing*
  The idea of this testing technique is the clustering of test cases into categories of same behavior, since usually numerous tests exist for the basically same code.
- *Decision Table Based Testing*
  Decision Table Based Testing is popular for the testing of complex logical relationships, especially under circumstances where the output depends on many conditions and decisions [14]. The derived decision tables serve as a blueprint for the generation of the test cases, as every possible conditional state represents one test case.
- *Cause-Effect Graphing Technique*
  This technique is a systematic method for the creation of test cases. Decision graphs are created on the basis of the inputs a program expects and the decision-based output it produces (=effect). It is practically only usable for limited unit testing of small programs, because the complexity of the cause-effect graph quickly increases.

Structural Testing

Structural testing can be equated with White-Box Testing, as the *structure* of the component under test forms the basis for any investigations. Test cases are usually designed from the source code, which hereby becomes the central document for investigation. The goal is to understand the implementation details as well as the internal structure. This also means that test cases can be created much easier and more targeted, what makes this a very popular technique [14]. Common testing methods are:

- *Control Flow Testing*
  This technique is very popular, as it is simple and effective. It is based on the identification of paths, which are sequences of statements in the program under test. A quality measure for control flow testing is the coverage, i.e. the “percentage of source code that has been tested with respect to the total source code available for testing ” [14].
- *Data Flow Testing*
  In order to test every statement of a program, Control Flow Testing may be not sufficient. Variables may be initialized outside the tested unit and the data flow may not be testable with the previous technique. Data Flow Testing aims to test the variable specific flow during execution, respectively the path between *definition* and *usage* of a specific artifact and thus takes global variables into account. A popular approach is the test of all definition-usage-paths (*du-paths*) [1, 14].
- *Slice based testing*
  This technique simplifies the testing process through the partitioning of the program into slices

with respect to the variables and their location inside the program. These slices may be executed independently.

- *Mutation Testing*
  This testing technique is basically a measure of test quality. For the generation of the so called "mutants'" particular statements in the source code are altered. The modified components are subsequently tested. If the test fails, this mutant is called a "killed mutant", whereas those passing the test are denoted with "live mutant". The resulting mutation score is the quotient of the number of killed mutants divided by the amount of total mutants and represents the quality of the testing suite. Ranging from 0 to 1 this value indicates how sensible the test suite actually is. A value near 1 indicates, that the test responds to nearly any change, whereas a score near 0 means, that the test absolutely neglects changes in the source code. Such a test would then be considered a bad test.

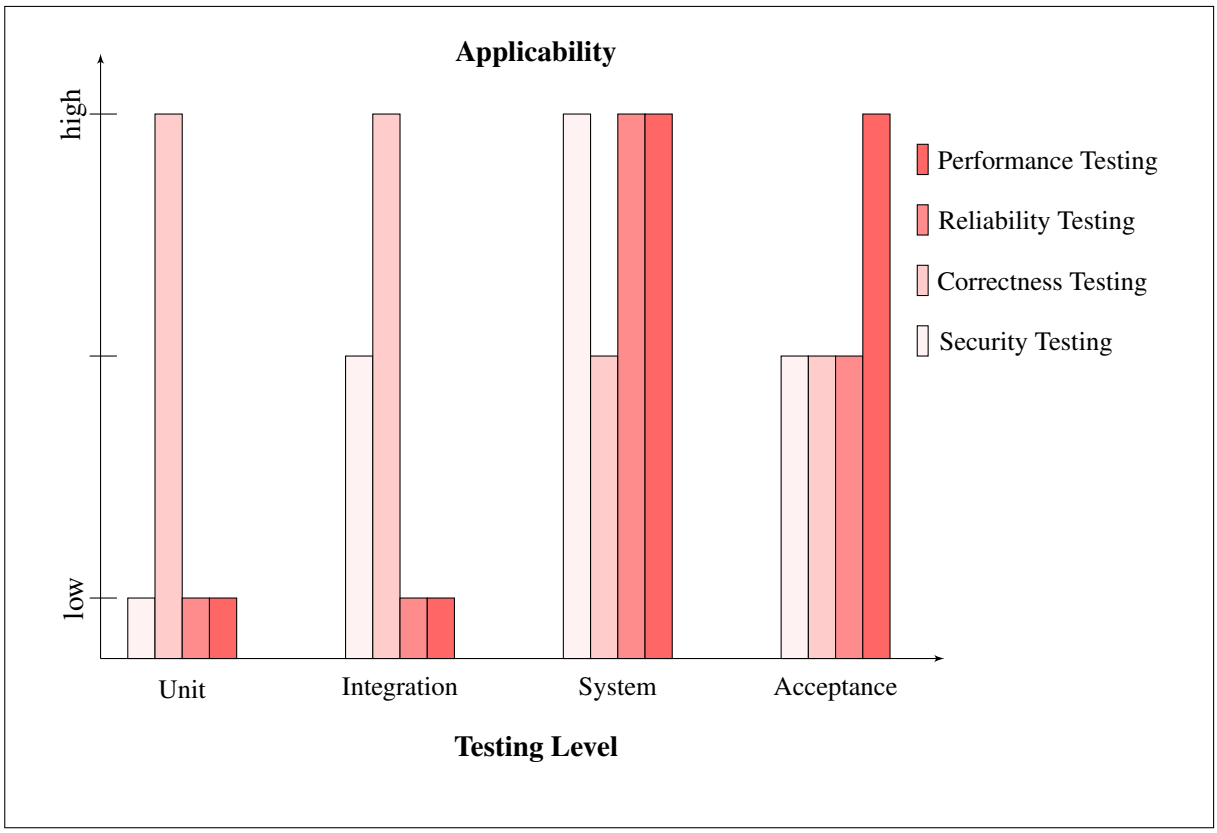


Figure 2.2: Applicability of testing types for each level

Figure 2.2 attempts to illustrate the previous findings, especially to map the applicability of the testing types to the different phases. However, due to the high level of generality, there may be valid scenarios that do not conform to this scheme.

Due to the scope of this work the following relates to the **dynamic** testing of the **structure** and the corresponding **control flow** of **units** based on source code, regarding their **correctness** in a **white-box** scenario.

2.1.5 Coverage Criteria

Since "complete testing" is an impossible task due to an effectively infinite number of inputs, formal coverage criteria need to come into play [1]. However, with the help of these quality measures it is feasible to find faults and to assure, that the software is of high quality and reliability [1]. Graph coverage criteria are the major quality measures in use [1]. They aim to inspect a graph representation of source code on how many of its branches and nodes are "visited" by a test. For that purpose, the source code has to be transformed into an abstract graph, which is usually the control flow graph. Based on these abstract representations, several criteria can be applied to measure the test quality. Figure 2.3 shows an exemplary graph representation with nodes n_0 to n_2 and edges e_1 to e_3. The nodes may represent statements as well as methods, the edges represent conditional and/or sequential dependencies. At node n_0, for example, two subsequent steps are possible depending on the decision taken in this node. Consequently, either edge e_1 will lead to the execution of node n_1 or edge e_2 to the execution of node n_2. In the following, selected structural graph coverage criteria will be introduced. If applicable, the appropriate measure is exemplary explained with the help of the simple example given in figure 2.3. To better understand the following sections various graph theoretical definitions are given here [1]:

- *Node*
  A node often represents a statement or basic block in a testing context. In graph theory a node is sometimes also called a *vertex*.
- *Edge*
  In testing scenarios edges are usually identified with the structure they represent, often as branches. In graph theory, edges are sometimes denoted with *arc*.
- *Path*
  A path is a sequence of nodes, where each node pair is connected by an edge of the graph. Paths have a start node and an end node. The length of the path is the number of contained edges.
- *Test Path*
  A test path usually represents the execution of a test case. However, a single *test path* may correspond to many test cases as well as no test case, if the path is unfeasible.
- *Simple Path*
  Paths are called simple, if every included node only occurs once.
- *Independent Path*
  An independent path has, compared to all other paths, at least one new node or an edge in its sequence from the starting node to its final node [14].
- *Prime Path*
  A path is considered prime path if it is a simple path and is not contained in any other simple path. With other word, a prime path is a simple path of maximum length.

Node coverage

A familiar and simple criterion is given with node coverage. The basic idea is to visit every statement in the source code and thus to ensure fully executed functionality. In order to achieve full node coverage, the control flow paths of a set of tests need to cover every possible node in the graph. Generally, this criterion is not only applicable to statements, but also to methods in a class. However, not every test will execute every statement what derives a percentage measure for the entirety of tests. Using the example of figure 2.3 a full node coverage may be achieved with a path $\{n_0, n_1, n_2\}$. Not exclusively two distinct measures may be defined based on this coverage criterion:

- **Instruction Coverage**
  The instruction coverage provides information about the de facto amount of executed instructions

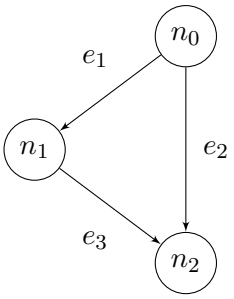


Figure 2.3: Graph example

by a test in a source code. For this purpose the amount of executed statements in contrast to the total number of statements in the test serves as measure. This value can be calculated for any root of the tree, i.e. for methods or whole classes. Since this is a structure-based criterion, the value may be calculated as sum of the value of its leafs. In other words: the instruction coverage for a class is the sum of the instruction coverages of its methods.

- **Method Coverage**
  In contrast to instruction coverage this indicator points at the visiting of the methods. The calculation here is quite simple. The comparison of the overall amount of methods in a class to the number of methods that are actually executed by a test serves as measure. However, multiple method invocations do not result in a different value, what limits the significance of this criterion. Furthermore the meaningfulness for the quality of the test is limited. Nevertheless this criterion may help to identify "dead code" [1].

Edge Coverage

This quality measure represents the number of visited edges when using a specific test path. Compared to the total amount of edges in the tested unit, a percentage value can be derived indicating the coverage value. Using the example of figure 2.3 and assuming the edges e_1 and e_2 are complementary, a full edge coverage can only be achieved by two tests. The respective test paths would be $\{n_0, n_1, n_2\}$ and $\{n_0, n_2\}$. At this point the difference to node coverage is obvious: all nodes could be visited without the usage of all edges, as only the first path would satisfy node coverage. For this reason, edge coverage is a stronger criterion than node coverage. The common implementation of Edge Coverage is Branch Coverage, which reflects this criterion in a programmatic environment. Branches are usually initiated by if-statements or other conditional statements in the source code. However, this can be a deceptive criterion as the else statement is not always used. This particular characteristic can lead to an incomplete coverage value and should thus be considered for coverage enhancements attempts.

Line Coverage

Line Coverage quantifies the number of code lines visited in relation to all lines of code. A line is recognized as visited as soon as only one statement in it is executed. As to achieve maximum coverage both all nodes and all edges must be visited, Line Coverage is thus an intermediate criterion between node coverage and edge coverage. It basically has the same requirements as those two criteria, but provides more realistic results, because non-existent branches will not cause a deterioration of the result.

Prime Path Coverage

The usage of this criterion keeps the number of test cases down, but its application can be problematic. Although a prime path may contain feasible simple paths, the path itself may be unfeasible [1]. In this

case the prime path needs to be resolved to its simple paths. The coverage criterion indicates how many prime paths are covered with a particular test. The value represents the ratio to the total number of prime paths.

Complete Path Coverage

The basis for the calculation of this coverage value are all paths of a program, respectively the relation of paths covered by a test and the overall amount of paths. Although this criterion seems to be extensive, it is often not feasible since graphs with cycles cause an infinite number of paths and hence an infinite number of test requirements. For this reason, this criterion is usually not used in practice.

Cyclomatic Complexity Coverage

The Cyclomatic Complexity represents the number of independent paths through a program [14]. This measure was introduced in 1976 by Thomas J. McCabe and uses the cyclomatic number as basis for calculation [18]. The complexity of a graph is based on the number of edges, nodes and connected components and is a measure for interdependence of units. The cyclomatic complexity can be calculated with $V(G) = e - n + 2p$ where $V(G)$ is the cyclomatic complexity, e the number of edges, n the number of nodes, and p the number of connected components [14]. If a graph is a SESE[3] graph, and every node and every edge is reachable from the entry node, the number of connected components is one. The Complexity Coverage represents a measure for the missed complexity in relation to the overall complexity of the tested unit [19]. In other words: this criterion measures the amount of independent paths covered by the test and provides a percentage value related to the total number of independent paths. For example, figure 2.3 has a cyclomatic complexity of two. An independent path $\{n_0, n_1, n_2\}$ would lead to an incomplete complexity coverage, although e.g. node coverage would be fully achieved. In summary, this measure takes the overall structure, the different paths, and the interconnectedness of different components of the program into account and is thus more reliable than solely node or edge coverage.

2.1.6 Implications

Based on the considerations in this section various requirements for the implementation of a proposed solution can be derived (see Table 2.1). Especially the integration into an Integrated Development Environment (IDE), an inspection opportunity for the source code as well as the usage of different coverage criteria are recommended.

2.2 JUnit

JUnit[4] was initially created by Kent Beck and Erich Gamma in the late 1990's. Over the years, it has evolved to the de facto standard testing framework [20]. As the name indicates, JUnit applies Unit Testing to Java. JUnit is applicable to basically any type, such as individual methods, classes or even complete complex components. For this purpose JUnit also provides functionality for the set up of the test, the initialization of variables and dependent classes, and finally compares the returned values of the tested components with particular predefined values. It is thus a valuable companion for Test-Driven Development (TDD). The complexity of the data structure determines the difficulty of writing appropriate tests. However, even for complex structures JUnit is fully applicable. There are basically two different types of JUnit, which still exist in coexistence today. With the evolution of JUnit 3.x to JUnit 4 the architecture and the semantics have changed. In order to still be compatible to existing tests designed for the earlier version, the current fourth version contains all the functionality that was

[3] Single Entry - Single Exit (SESE) graphs have exactly one entry node and one final node

[4] http://www.junit.org

Aspect	Implications
Unit Testing	Since unit tests are usually created by the developer of the component under test, an integration into an IDE should be pursued to foster the ease of use.
Dynamic Testing	The previous recommendation is supported at this point. As this technique demands the execution of tests, an IDE integration is appropriate.
White-Box Testing	For the derivation of tests the developer must be able to inspect the test code for the derivation of test cases. This opportunity should thus be given in an implementation.
Structural Testing	For the assessment of the control flow the coverage criteria introduced in 2.1.5 should be integrated to provide a measure of quality.

Table 2.1: Implications derived from Testing theory

already given in version 3. However, the JUnit 4 framework provides more flexibility and is therefore more suitable for large and complex projects. In the following the structure and functionality of JUnit 4 will be in the focus, since it is downward compatible and thus JUnit 3 tests can be executed as well. Nevertheless, hints regarding the JUnit 3 equivalent will be given in relevant places.

2.2.1 Difference between JUnit 3 and JUnit 4

The major differences between JUnit 3 and JUnit 4 are of a structural nature. JUnit 3 depends on various requirements for the naming of methods, or the need for the extension of the *TestCase* class, whereas JUnit 4 provides much more flexibility. The main differences are:

- **No TestCase extension needed**
  While in JUnit 3 tests still need to extend the *org.junit.TestCase* class, JUnit 4 has abandoned this requirement. Due to this need for inheritance in JUnit 3, freedom was severely restricted. The test and the Component Under Test (CUT) always need to be implemented in separate documents. In contrast, JUnit 4 offers the possibility to integrate test methods directly into the CUT and thus enables “built-in testing”. A distinction into test classes and CUT classes is thus not necessary - although still possible and useful.

- **Naming conventions omitted**
  The requirements for the naming are quite strict in JUnit 3. Thus, the test methods must necessarily begin with "test" otherwise they will not be executed. Even the preparatory and subsequent functions - each for the test and for its methods - must follow a precise naming. Through the introduction of annotations in JUnit 4 this necessity has been abolished.

- **Annotations included**
  JUnit 4 makes use of the *Annotations* introduced in Java 5.0. The use of these supplements allows the inclusion of metadata into the Java source code. Consequently JUnit 4 is able to omit the naming requirements and the resulting rigid test structure, and provides flexibility and built-in testing. The fixed methods for preparation and clean-up are replaced by the corresponding remarks. The name of the method is therefore irrelevant.

- **Exception testing**
  Exception testing is simplified, as test methods may be annotated with an expected exception to be thrown. For that purpose, the *@Test* annotation accepts the inclusion of a parameter *expected*,

which indicates the throw of a subsequently defined exception. The need for try/catch blocks within the methods is thus omitted. However, this is primarily useful for simple exception tests. A detailed exception testing may nevertheless demand a more complex method structure.

- **Parameterized testing**
  Since JUnit 4 parameterized tests can be implemented. Therefore a factory method may be chosen, which produces the input data for the test. When a test run is started, instances for the cross-product of these data items and the test methods are created. This enables to reuse data and thus decreases the effort of test writing. In addition, this allows the generation of random values for the test in a central place and a simple way.

- **Theories**
  A special feature introduced in JUnit 4 is *Theories*. They allow more flexible and expressive assertions, since a Theory is able to capture the behavior of a test in multiple scenarios. Assumptions may be defined, which have to be met by given DataPoints before the test will continue to run. However, if a DataPoint does not satisfy these prerequisites the system silently continues execution with the next item and the test may still succeed. The DataPoints can be either directly specified or with the help of corresponding annotations generated by a factory class. The use of Theories is therefore very flexible, reduces cost and provides broader results.

2.2.2 Test Suites

Test suites represent an accumulation of test cases. Usually the test cases are grouped by their scope and target. That means, that all tests concerning a specific component under test are grouped into one test suite. The goal of this consolidation is to automate testing and thus to foster regression testing [20]. The sufficient annotation is given with *@Suite* in conjunction with the *@RunWith(Suite.class)* annotation.

2.2.3 Test Cases

The structure of a JUnit test is completely equivalent to other Java classes. In the test cases the package is declared first, followed by imports, then the class definition with its fields, variables, and methods. When creating a JUnit 4 test the only things to pay attention to is the correctness and completeness of JUnit-specific imports, and that the path of the JUnit library is included in the class path. JUnit 4 provides several annotations for test fixture, each two for the marking of set-up and tear down methods, and each on class level and for test methods. Hereby is is possible to prepare the data for the whole test, or for each method, and to perform different cleaning activities after each method, respectively at the end of the whole test [3]. Figure 2.4 shows an example JUnit 4 test containing methods with distinct annotations. For the fixture of a test the following annotations are available:

@BeforeClass	Methods with this annotation are executed before any testing activity is started. It is helpful for the preparation of data, the instantiation of variables etc.
@AfterClass	This annotation is the counterpart of @BeforeClass. Accordingly, the corresponding method is invoked after all tests have been executed. It is useful for clean-up activities, for the safeguarding and, if required, for the re-establishment of a consistent system state.
@Before	Methods annotated with this type are executed prior to *each* test method. It may be used to reset variables or objects.

```java
package test;

import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

public class FixturesTest {


    @BeforeClass
    public static void setUpBeforeClass() throws Exception {
        System.out.println("@BeforeClass");
    }

    @AfterClass
    public static void tearDownAfterClass() throws Exception {
        System.out.println("@AfterClass");
    }

    @Before
    public void setUp() throws Exception {
        System.out.println("@Before");
    }

    @After
    public void tearDown() throws Exception {
        System.out.println("@After");
    }

    @Test
    public void firstTestMethod() {
        System.out.println("@Test firstTestMethod");
    }

    @Test
    public void secondTestMethod() {
        System.out.println("@Test secondTestMethod");
    }
}
```

Figure 2.4: Example JUnit 4 Test

@After	This annotation is the counterpart of @Before. Methods annotated with that type are executed after each test method. It is helpful for temporary clean-ups and the recovery of initial values.
@Test	Test methods are marked with this annotation. During a test run gradually all methods annotated with @Test are executed. If methods with @Before/@After annotations exist, these are executed prior/after each test method.

The usage of these annotations is not exclusive, which means, that methods can be provided with several annotations. It may possibly make sense, for example, to perform the same operations before and after each test method. In this case, the appropriate method would simply be annotated with *@Before and @ After*, which would lead to a preceding and subsequent execution. In a JUnit 3 environment these options are not available because the corresponding methods for preparation and follow-up activities are dependent on a predetermined, explicit naming. The method names used in figure 2.4 correspond to the requirements of JUnit 3 except the test methods. The test methods must by convention start with "test"

to be executed. In addition, the test class must extend the *junit.framework.TestCase* class to act as JUnit 3 test at all. The use of multiple methods for a particular purpose or the reuse of individual methods for multiple purposes is therefore excluded. However, the order of execution of the methods does not differ. A test run with the exemplary test of 2.4 would thus deliver the output of figure 2.5.

@BeforeClass
@Before
@Test firstTestMethod
@After
@Before
@Test secondTestMethod
@After
@AfterClass

Figure 2.5: Test Run Result

This illustrates the order of processing. For a JUnit 3 compliance this example had to be appropriately adapted, but would then be fully functional and the processing would take place in the same order.
If all relevant preparations have been made and all clean-up functionality has been integrated, the test is ready to be filled with test functionality. The task of testing itself is handled by the method annotated with *@Test*. A test may contain any number of test methods, which are processed sequentially.

2.2.4 Test Methods

The testing itself takes place in the test methods. Within these methods every desired functionality can be included. This comprises the instantiation of variables, the modification of fields, usage of libraries and so on. As long as a method can be fully executed and no exception occurs, it satisfies the test. This may be sufficient for certain scenarios, but usually the *state* or the *value* of a variable or an object, or the result of some function is much more interesting and worth testing than the ability of the compiler to create objects and map modifications. For the inspection of these states and values JUnit provides different predefined statements to check whether a value or condition meets the expectations. If the particular assertions are fulfilled, the test run is regularly continued. Otherwise the test fails and stops with a distinct message. The syntax of these statements is quite simple. The majority of the statements are passed an *expected* and an *actual* value, of which the latter is usually the result of some calculation and is thus often represented in form of a method invocation statement. Both values are compared on execution. The type of statement here determines the comparison method. The assertion statements may be equipped with an additional argument as first parameter holding a particular message to be displayed on test failure.

In the following the different assertion types are introduced with their specific details:

- **assertTrue(condition)**
  This assertion statement simply checks, whether a condition is true or not. Consequently this operation expects one boolean argument and causes the test to fail, if this argument is false.

- **assertFalse(condition)**
  This statement represents the opposite of the previous statement, although its behavior is comparable. However, it fails if the condition is true.

- **assertEquals(expected, actual)**
  This statement is one of the most used commands. Both *expected* and *actual* can be represented by many different data types. However, the same type must be used. It comprises inter alia the

classes *Object* and *String*, as well as the primitives *int*, *double*, *float*, *long*, *boolean*, *byte*, *char*, and *short*. Due to the capability of comparing *Objects*, this command is universally applicable. For the comparison of floating point data types (such as *double* or *float*) the statement can be equipped with a *delta* value representing the threshold in which the comparison shall be. In the latest version, this information is required for these types.

- **assertNull(object)**
  As the name indicates, this statement evaluates, whether the object is null. If this condition is met, the test continues execution without notice. The argument may be of any type.

- **assertNotNull(object)**
  This assertion type is the opposite of the previous statement for convenience. Consequently, this assertion fails, if the object is null.

- **assertSame(expected, actual)**
  This particular assertion type compares two objects at their identity. That means, that two objects of same type, with same content, but of different instance will *not* be considered equal. Both *expected* and *actual* may be represented by any *Object*, what makes this statement universally applicable.

- **assertNotSame**
  The opposite of the previous statement is given with this type for convenience. This assertion is fulfilled, if the *Object*s are not equal based on their *identity*.

- **assertArrayEquals(expected, actual)**
  This statement compares two arrays for similarity. In contrast to the preceding assertion types this statement refers to the *values* and thus two different arrays with the same content will be perceived equal. Both *expected* and *actual* may be an arrays of any type of *Object* or of any primitive type.

Basically the different assertion statements are interchangeable. Thus, for example, the statements "assertEquals(expected, actual)" and "assertTrue(expected.equals(actual))" are fully equivalent in terms of the test result. However, JUnit provides additional useful information when using *assertEquals* which can improve the quality of the test. Only in this case, a notice is issued, which value was taken by the *actual* argument. That information is not provided when using *assertTrue*. In addition, at the example above two comparisons had to be made: first, whether *actual* equals *expected*, and secondly, whether the result is true. This is likely to adversely affect the running time. Since version 4.5 a new assertion type found its way into JUnit:

- **assertThat(value, matcher)**
  This assertion statement was built on top of a project called JMock1 and utilizes the instrumentation of *org.hamcrest.CoreMatchers*. Through subject-oriented syntax of the *Matcher*, this statement primarily improves readability of statements and failure description. In addition, this assertion type is given a special meaning in conjunction with *Theories*, likewise introduced in this version. A combined usage of these two techniques allows an automated generation of test values, as the *Theories* principle acts as a filter that uses assumptions about values to dismiss inappropriate entries.

2.2.5 Other interesting Annotations

In addition, the JUnit framework provides several annotations for the creation of tests, for the work flow, and for behavioral changes to test methods. The most flexible of these annotation is the **@Rule** annotation. Rules provide very flexible modification of the behavior of any test method. Besides the opportunity to extend existing or implement custom Rules the Base Rules provided with the current JUnit

distribution contain a variety of functionality. The **@Rule** annotation enhances the following classes of the *org.junit.rules* package [3]:

- *TemporaryFolder*
  Annotated objects of type *TemporaryFolder* allow a simple creation of temporary files and folder, which are guaranteed to be deleted after the completion of the test.

- *ExternalResource*
  Components may be annotated indicating, that the marked class provides individual set-up and tear-down methods. These classes have to extend the *ExternalResource* class and thus inherit an *after* and a *before* method. These methods are invoked before respectively after each test method. This allows to reset classes that interact with the test, e.g. as tested component or as supporting provider of information and/or functionality.

- *ErrorCollector*
  The *ErrorCollector* Rule enables a continued execution of tests, even if a previous test reported a failure. For this, a special *ErrorCollector* has to be declared, which collects the reported errors.

- *Verifier*
  The *Verifier* class basically works like the *ErrorCollector*, but provides verification checking for a whole test. With the help of the *verify* method the result of the test can be examined in detail.

- *TestWatcher*
  With the help of a *TestWatcher* a detailed investigation of the testing action can be applied. It provides methods for any event, that may occur during a test run. Hereby customized messages and actions are applicable, e.g. if a test fails, succeeds, starts, or finishes.

- *TestName*
  Utilizing the class *TestName* makes the current test name available to test methods. Hereby a fine grained supervision of the test method behavior is possible. In addition, this helps to customize error messages for convenience.

- *TimeOut*
  With the help of the *TimeOut* rule it is possible to set a time-out counter for each test methods. If the test run exceeds the specified limit, the test automatically fails. This helps to prevent deadlocks and thus to avoid failing termination.

- *ExpectedException*
  The utilization of this class allows to specify expected exception types in-test. It basically works like the *ErrorCollector* class and ensures the continuation of the test, since the collected exceptions can be queried finally.

- *RuleChain*
  The class *RuleChain* allows the chain up and ordering of different *TestRules* and hereby serves structuring and readability purposes.

Another introduced annotation is the **@ClassRule**. It is similar to the *ExternalRessource* Rule, but on class level. This statement leads to an invocation of the required *before()* and *after()* methods just before any other action is performed. These methods are even executed before the method annotated with *@BeforeClass*, respectively after the *@AfterClass* method in the test. It thus serves the purpose of preparation of interacting components. The utilization of this annotation is especially lucrative if the required processing is expensive and thus otherwise an interference with the running test is threatening.

2.2.6 IDE integration

JUnit is fully integrated in many IDEs, as well as in Eclipse. It offers a graphical User Interface (UI) and hereby simplifies testing. The UI provides detailed information about the tests that have been run, occurred errors, and reported failures. An indicator bar directly informs about the success or the failing of a test by color. If a test passed, the bar is shown in bright green, whereas a dark red bar indicates a failed test run. Another text field provides information about the reasons for failure, if this is the case. Figure 2.6 shows the visual representation of a successful, respectively a failed JUnit run. However, Janjic and Atkinson complain, that existing tools evaluate test quality only ex post [21] and thus a valuable support is limited. In addition they propose an enhancement of the idea of continuous testing [1] to provide the user with valuable information about the benefits of intended testing efforts beforehand [21].

2.3 Software Reuse

Software Reuse (SR) generally refers to the use of existing software artifacts instead of newly implementing the desired functionality. Already in the 1960s the demand for reusable components has been proposed to solve the problem of realizing large and reliable software systems quickly and cost-effectively [22]. Although the principle of Reuse has been widely acknowledged so far [4, 23], a widespread, systematic Reuse is still lacking [5]. Even though Reuse is considered significantly beneficial as it reduces defect density and thus increases productivity [23], which has even been confirmed for industrial practice [6, 24], there are still difficulties in finding and utilizing components meeting the desired requirements [11]. A reason may be, that the utilization of Reuse is more than the provision of an appropriate search engine for tests [21].

2.3.1 Types of Reuse

According to [5] there are numerous conceptual facets of which to view SR. They are categorized into six perspectives:

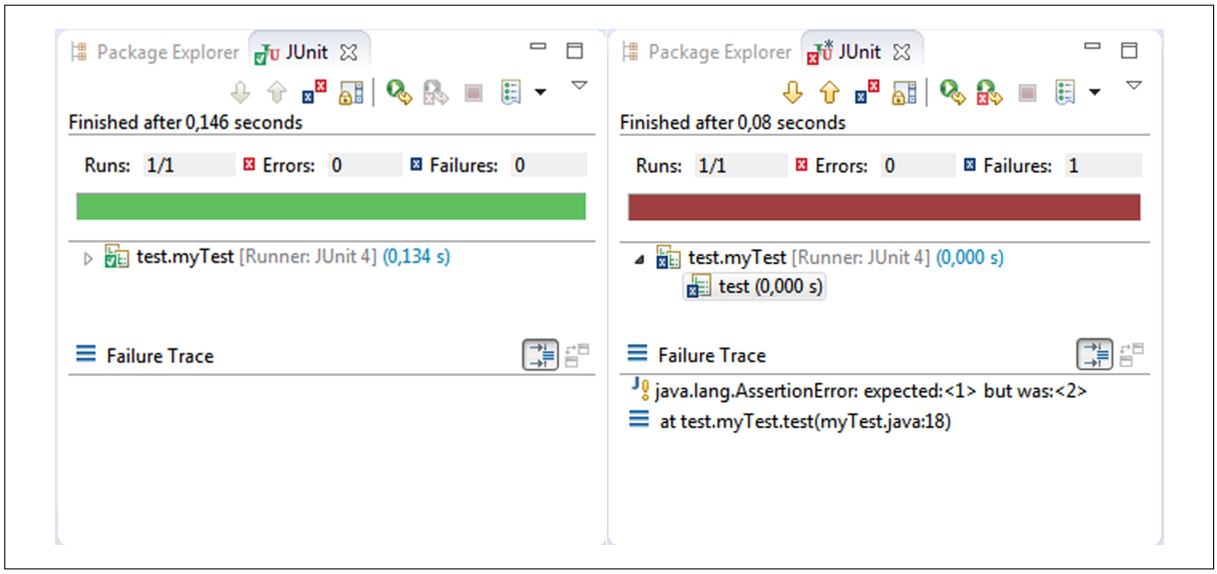


Figure 2.6: Example JUnit results

Substance Perspective

Idea Reuse — This type of Reuse refers formal ideas or concepts, such as generic algorithms. Since the re-used knowledge is not tied to components or systems, the areas of application for this Reuse type is virtually universal. However, the reused knowledge has yet to be made programmatically available, which is associated with considerable effort.

Artifacts Reuse — The Reuse of parts is one of the most promising approaches and particularly in demand in Object-Oriented (OO) technology. Characteristics for the suitability of artifacts are quality and reliability. Besides these requirements the ability of adaptation, and domain-specific collections are crucial for successful Reuse.

Procedures Reuse — This type of Reuse deals with the formalization and encapsulation of procedures, which are to be accommodated in appropriate collections. However, the preparation of the data is challenging. Nevertheless, this enables the combination of different reusable processes for the purpose of creating new and more complex ones.

Scope Perspective

Vertical Reuse — Vertical Reuse denominates the Reuse within the same domain or application area. The goal is to develop generic models for the creation and assembly of the systems. The better defined the domain is, the bigger advantages of Reuse are assumed. However, the analysis of the domain as well as the identification and development of appropriate models are crucial for application.

Horizontal Reuse — Horizontal Reuse focuses, in contrast to vertical Reuse, on the transfer of generic parts of components to different application scenarios. The major tasks in this category are the establishment of libraries, the creation of classification principles and the improvement of interoperability. In the end horizontal Reuse may be applied in component markets [11].

Mode Perspective

Planned Reuse — This paradigm focuses the level of formalism, with which Reuse is applied. For Planned Reuse, the number of regulations, the details of guidelines and systematic, as well as the existence of performance measurement tools for Reuse are relevant. Planned Reuse is usually instrumented on project or company level and often linked to economical performance indicators.

Ad-hoc Reuse — This form of Reuse is basically the state of practice. It represents the informal mechanism of Reuse of components from general libraries and is thus also called opportunistic Reuse. It is practiced by individuals rather than on project or on company level. As the fragments in public databases are usually not designed for Reuse, an adaptation is often inevitable. Thus, room for improvement exists in the enhancement of the corresponding databases, in the simplification of search opportunities, and in the empowerment of retrieval mechanisms.

Technique Perspective

Compositional Reuse — This Reuse type focuses on the composition of existing artifacts as functional modules for the assembly of new systems. It relies on well known collections, reliable repositories, and standard interfaces. Besides its applicability to almost any software component, compositional Reuse is mainly applied to source code. Main tasks for a beneficial application are sufficient component selection and retrieval capabilities as well as component adaptation and integration feasibilities.

Generative Reuse — This type refers to the application of Reuse on specification level. Specifications can be utilized e.g. for the generation of code and thus offer high potential. However, the investigation of the domain, the processes, and the architecture as well as the derivation of vocabularies, grammars and meta generators is challenging and thus a scale up to industrial level is difficult.

Intention Perspective

Black-Box Reuse — Black-Box Reuse denotes the Reuse of components without any modification. Usually, these components are packaged but provide well-defined interfaces for interaction. This approach is very promising in reliability and quality, but the development of reusable components may be expensive. Especially the verification and the attestation of performance of the component is complex under differing conditions. Nevertheless, Black-Box Reuse components are promising for a usage in component markets [11].

White-Box Reuse — The most common Reuse approach is White-Box Reuse. It allows the modification and adaptation of software components and thus to tailor the artifacts to individual needs. White-box Reuse is often used in ad-hoc scenarios. Nevertheless, knowledge of the domain is needed to be able to presort components and enable a straight-forward adaptation.

Product perspective

The type of artifact to be reused ranges from source code, structures, specifications, up to documentations (text) [4]. Each of these "products" provides pros and cons for Reuse.

Source Code Reuse — The Reuse of code is state of practice. Most of today's tools and methods focus on code Reuse. Although the utilization of source code for Reuse is surely possible and impartially, this can be a labor-intensive task. With the evolution of higher-ordered Reuse artifacts, this methodology will become obsolete. However, for the creation of these artifacts, code Reuse may be useful.

Design Reuse — The Reuse of designs is basically promising - but also challenging. Especially the applicability in the target environment is to be considered decisively. At least Reuse may be partially or indirectly possible with the help of OO methodology. However, finding suitable components is a central aspect to performance. Therefore, a fixed specification of the functionality is essential.

Specification Reuse — On the basis of generative Reuse the Reuse of specifications is most promising. However, their design and code-specific implementations have to be available and capable for Reuse.

Object Reuse	The Reuse of objects becomes increasingly important, as it fits seamlessly into OO programming. This approach is already used today for the development of libraries. However, Reuse at object level is struggling with the same challenges as any Reuse of composite artifacts. Thus, the implementation of suitable interfaces and the creation of adequate repositories with appropriate retrieval functionality is essential.
Text Reuse	The Reuse of text seems promising, since nearly all documents are created for humans and thus Reuse potentially promises huge benefits. Information retrieval and text mining technologies provide assistance for the indexing and manipulation of text.
Architecture Reuse	As whole architectures represent the most coarse-grained units for Reuse, an application is very challenging. However, investigations on the specific domain and the identification of generic designs and subordinate units may be sufficient.

This work deals with a **white-box** Reuse of **artifacts** on the basis of **source code** in an **ad-hoc** scenario and use the acquired information for a **compositional** refactoring of the functionality in a **vertical** application scenario.

The reason for this categorization is the objective of this work: the automated Reuse of tests. For this purpose a database is used, which currently includes approximately 200,000 Java tests using the JUnit framework [8]. Considering the fact, that JUnit tests are reused to enrich JUnit tests, a **vertical** character can be identified. Since the existing classes are only given with their **source code**, a Reuse can only take place on the basis of these code **artifacts**. Due to this fact, it is also essential, that these artifacts are visible to the user to provide the opportunity of inspection and adaptation. Although it is our goal to prepare the code to be inserted as well as possible for the user, this corresponds to a **white-box** character of Reuse. Nevertheless, these data have to be analyzed in structure and content to have a starting point for systematic Reuse. This step is inevitable, since the goal is to integrate the Reuse functionality into an Eclipse plugin for the purpose of automated evaluation and easy application of these artifacts. As the Reuse itself happens spontaneously through the use of the Eclipse proposal computer (and by basically any user) the Reuse mode can be considered **ad-hoc**. Using the data provided, the user is therefore able to enrich an existing or to **compose** a new test. These specific requirements are shown in figure 2.2 together with the corresponding recommendations for successful Reuse.

2.3.2 Benefits & Constraints

Applying the fixation of the type of the previous section, several benefits and constraints exist for the application of Reuse. To start first with the requirements as derived from the constraints of table 2.2, the following conditions must be met to facilitate a valuable Reuse:

- The reused resources have to be highly adaptable
- A set-up of domain specific collections, which means the creation of a test database, is inevitable
- Targeted preparation of the supporting database in order to provide all needed information
- An open-source character reveals from the nature of white-boxed Reuse, this may but foster the participation of yet uninvolved contributors
- The data has to be processed to ensure high integration feasibility
- The domain needs to be fixed, or intensively examined

Aspect	Benefits	Constraints
White-box Reuse	+ high adaptability + opportunity of inspection + often open-source licensed	- possibly the *need* for adaptation - protection of intellectual property difficult (cf. [11])
Artefacts Reuse	+ widely used + supporting technology directly available (e.g. database)	- adaptation ability required - domain-specific collections
Source Code Reuse	+ state of practice + no preprocessing needed + tool support	- labour intense - becomes obsolete with the provision of higher-order artifacts
Ad-hoc	+ state of practice + informal + spontaneously available	- fragments in public libraries usually not designed for Reuse - adaptation usually necessary - possibly insufficient databases
Compositional	+ modular application + no specifications needed	- sufficient component retrieval capabilities needed - adaptation and integration feasibilities needed
Vertical	+ applicability increased by same domain affiliation + system models do possibly already exist	- laborious domain analysis

Table 2.2: Benefits & Constraints of the selected Reuse approach

These requirements serve as a blueprint for the composition and implementation of the required components. In particular, the preparation of data and the adaptation to the specific needs must not be neglected. Furthermore, it is important to well define the interfaces of the database to obtain a simple, fast and useful source of information. Despite these supposedly high requirements Reuse offers great potential in its application to tests. It is additionally needful to consider the maintaining of *existing* benefits:

- The opportunity to manually adapt and inspect the recommended source code needs to be offered
- The user shall not be demanded to pre-process the received information
- The lookup and query process needs to be simple in order to support informal, spontaneous usage
- Any specifications needed have to be extracted from the existing project to avoid the need of manual definition

These findings represent a template for the implementation of a suitable solution. Consequently they influence the structure of the database, the design of search result objects as well as the functionality of the Eclipse plugin *Test Tenderer*.

2.3.3 Test Reuse

Basically the findings of 2.3.2 are also applicable to Test Reuse due to their general applicability. However, as Test Reuse is specific within the field of Reuse, several additional requirements exist. One possible source for enhancements can be found in the field of automated adaptation. Especially for tests, the name of a class or a method is basically irrelevant for the determination of congruence in functionality. Thus, the *behavior* of a test, respectively a testing artifact, is much more interesting and promising than the name [8]. Although this statement is generically valid for the field of Reuse, this issue is especially supported by the existence of behavioral knowledge about the tested component in the test itself. This results in the opportunity to analyze tests and therewith to enhance the recommendation quality [8].

2.3.4 Tool support

Information from tests, or tests themselves are nowadays already used to search for suitable Reuse candidates. For this purpose search engines are available (e.g. Merobase [10] or Sourcerer [25], which also offer appropriate IDE integration (e.g. Code Conjurer [9] or CodeGenie [26]). However, these tools offer a search for suitable *components*, not for *tests*. But the application to tests is promising, not only because the Reuse of test cases saves tedious work [7]. With the use of functional test information the process of test-driven search will continue to improve and lead to more and better results [8]. However, also for this purpose an IDE integration is inevitable, since “pure web-based search interfaces do not attract developers” [21].

2.4 Speculative Analysis

The idea of this technique is both simple and promising. Basically it provides the opportunity to investigate *future* actions the developer may perform [27]. For this purpose, today standardized hardware, multi-core architecture, or cloud-computing can be used to calculate consequences of possible actions in the background. The user can be given ex-ante feedback about the effects of the changes they may be considering to the software [27]. As the user is therewith able to make better founded decision, this technique increases software quality as well as developer productivity [27]. However, today’s testing software focuses on the investigation and assessment of past or present states and neglect future development states.

Application areas

This methodology can be applied to various scenarios. Brun et al. provide examples for the "quick fix" support of common IDEs and version control systems [27], but the potential scope is unrestricted.

- **Quick Fix**
  Current IDE's provide some kind of "quick fix" support, that helps the user to perform distinct actions depending on the current situation. This may be an auto completion proposal for incomplete statements, integration or creation of new artifacts, or other modifications. The great difficulty for the user is to identify the impact of each change on other components or the system. It could for example happen, that given tests will not succeed with the changes, or even that the project no longer compiles or the structure becomes inconsistent. In that case the user would have to undo the changes in order to hopefully restore the previous, functioning system state. However, with the help of Speculative Analysis these risks can be avoided. The IDE can to estimate the effect of each quick fix and to deliver this information to the user beforehand. The basis for decisions will therewith be significantly wider for the user, which results in a lower probability of incorrect decisions and thus certainly also avoids unnecessary reversal processes.

- **Version Control**
  For the support of collaborative working nowadays several version control systems exist. They provide users with the opportunity to simultaneously work on distributed projects. However, this can also be associated with problems. If, for example, two collaborators work on the same artifact, the merging of these files may cause conflicts, compilation errors, or tests that fail, although they could previously be successfully executed. Speculative Analysis can hence help to avoid such constraints by simulating the merging process. If the analysis showed, that problems would likely exist, the developer is able to postpone the merger or to directly eliminate the reason for failure.

Speculative analysis has been applied to the collaboration system Crystal [28, 29] and provides precise results [29].

Applicability for tests

Speculative Analysis can also be applied to tests. In this conjunction, beneficial information would be whether the test succeeded or failed or the consequences of the changes to the test coverage. However, this scenario only makes sense if Reuse is applied. Otherwise, the question arises, why a user should speculatively analyze the future impact of a test, if it is simply possible to run this test and to inspect the result. Furthermore the input would have to be derived from somewhere or specified by the user. In short, this scenario is senseless without Reuse. However, speculative analysis provides unprecedented opportunities. If a user was provided with proposals for tests and could therewith estimate the impact on the coverage of the test suite, this information could be used especially for the selection of appropriate candidates. This would not only help to save work, but also to increase the test quality.

Challenges

A key challenge for Speculative Analysis is the *breadth* of speculation [27]. Since basically any state may be examined, the set of possibilities quickly increases. Another driver for the size of the set is the analysis *depth*, since basically the speculative processing may be conducted with several iterations. However, with the increase of the possible future actions the cost for the analysis increase excessively. It is therefore inevitable to actively manage (and if necessary reduce) the set of possibilities. Furthermore, technical issues need to be addressed in order to ensure feasibility and consistency [27].

- **Breadth & Depth**
  As the program perhaps needs to be executed for every single calculation, the speculative analysis

process requires significant computation to explore a large number of sets [27]. However, both the *breadth* and the *depth* influence the overall effort. With *breadth*, the amount of simultaneously available (or applied) different scenarios is denoted, whereas *depth* refers to the number of iterations, the speculative analysis is performed for. Using b as the number of concurrent possibilities and d as iteration counter, the number of overall computations $C_{d,b}$ is given with $C_{d,b} = b^d$. Assuming, that for every iteration 10 possible options exist, a iteration depth of only 2 would cause 100 potentially complete runs of the system. Depending on the complexity of the system this task is likely not feasible in an appropriate time span. Solutions for this problem can be found in an active reduction of the set of possibilities, and a limitation of the depth, e.g. to 1. The latter also makes sense as the possible modification candidates may change from step to step and thus a recalculation can be anyway necessary.

- **Shared resources**
  Another issue to tackle is the usage of shared resources. The main objective is to ensure consistency and to avoid failure through unavailable or blocked components. This may for example be project files, that are needed for the calculation and which may be locked by the file system when one process requests access [27]. In this case, the file system will usually block the usage of the particular file to avoid inconsistencies. If the speculative analysis process is not feasible to handle such situations, a parallel computation will not be possible. This again leads to longer overall execution time, since the different possibilities can only processed sequentially. In the worst case, the whole task may even fail. A means to an end may be in memory compilation or advantages in rapid cloning of the project's development states [27].

2.5 Summary

In this chapter the foundations for the remainder of this work were illustrated. The treated areas include Testing, JUnit, Reuse and Speculative Analysis. Derived from the findings the following assumptions have been made:

- **Testing**
  The Testing focus of in this work lies on *dynamic* testing of the *structure* and the corresponding *control flow* of *units* based on source code on their *correctness* in a *white-box* scenario. The various implications of Table 2.1 should be reflected in an IDE implementation in order to provide maximum benefit.
- **Reuse**
  Reuse will be conducted in form of a *white-box* Reuse of *artifacts* on the basis of *source code* in an *ad-hoc* scenario and use the acquired information for a *compositional* refactoring of the functionality in a *vertical* application scenario. For this purpose, both a suitable database as well as an effective IDE integration are profitable.
- **JUnit**
  JUnit 4 is an extensive and reliable tool for the testing of Java units. Because of backward compatibility the handling of a JUnit 3 tests can as well be guaranteed.
- **Speculative Analysis**
  Speculative Analysis has been identified as a suitable technology to shift continuous testing from an ex-post usage to ex-ante application. However, some obstacles have to be considered in order to create a workable solution.

In addition, one issue is to be mentioned in this section as it is relevant for the remainder of this work, but neither fits into another section nor provides enough scope to be represented in a separate section. Since the goal of this work is to develop an Eclipse IDE plugin for the Reuse of Tests and to integrate these

recommendations into the Eclipse Proposal Computer, the *Ranking* of the proposals also influences the quality and usability of the recommendations.

- **Ranking**
  A survey conducted about intelligent code completion system showed that recommendations, which are context-sensitively ranked and thus more relevant for the user, dramatically outperform unranked proposals and thus have the potential to enhance a developer's productivity [30]. Consequently, this issue must also be remembered for the design of the plugin.

CHAPTER 3

Design

In this chapter the design of our program Test Tenderer (TT) will be described. Beside the clarification of the desired target functionality the project structure will be explained not only to enable the reader to understand and reproduce our approach but to offer the opportunity to extend the developed tool. In addition, details about the functionality will be given, especially how it processes the information from the test-driven search to provide the user with various features.

The main objectives of this tool are (1) the utilization of Search-Driven Development (SDD) and Reuse with the help of a code database, (2) to combine both techniques and apply this to tests, and (3) an integration into the Eclipse framework to simplify the Test Reuse process. To achieve these goals a plugin for the Eclipse Integrated Development Environment (IDE) was developed since this constitutes a popular framework and provides needful starting points for the integration of a new functionality. This approach allows us to shift the search process to the background and thus to fully automate. As a result the user is constantly supported with suitable tests, classes, and methods found in the database. Based on these data the user is offered the opportunity to inspect, verify or "crib" code fragments – or simply to get inspired by the way other users solved a specific task. Additionally, our tool generates, ranks, integrates, and – as an output – recommends valuable completion proposals in form of JUnit assertion statements to simplify the testing process. Alongside these primary tasks, to enable Test Tenderer to offer test recommendations and generate appropriate proposals, several complementary tasks had to be fulfilled. Despite these, or maybe *because* of these, additional value was created by supplementary features. As a result e.g. a feature called Continuous Coverage Calculation, a function that automatically evaluates the test results and calculates the current coverage of the currently edited document in the background and thereby provides the user with instant feedback about his testing efforts, could be added.

The first section describes the domain and depicts the problems related to today's Reuse efforts. Section 3.2 to section 3.5 depict an overview of the features. Starting with the Preparations (3.2) the significance and the characteristics of the search process (3.3) will be discussed in detail. Based on the results from this section the application of Speculative Analysis (3.4) will pave the way for the Proposal Generation (3.5). In these passages different aspects of provided functionality and the obstacles, which have been overcome, will be highlighted. Finally this chapter closes with a summary in section 3.6.

3.1 Domain

The domain of our work is actually composed of several smaller areas of interest. Specifically, it combines the areas of Software Reuse (SR) (in particular Test Reuse), Testing and Search-Driven Development (SDD). The domain of this topic therefore includes all the actors and actions of those individual areas.

Developer	Creator of a specific program or functionality and/or associated tests
Test	Code to test a specific component
Component	the component which is created/tested
Repository	Knowledge storage with code artifacts for reuse, e.g. internet, database
IDE	Integrated Development framework the developer works with, e.g. Eclipse
Quality	Measure for excellence of the developed code

The entities *Test* and *Component* are directly connected with each other since there is no point in a test without a tested object. Even with a Test-Driven Development (TDD) approach this condition can exist only temporarily, because sooner or later a test needs to be targeted to one ore more specific components. On the other hand the opposite situation is basically possible but neglected in the context of our work: as the reuse of tests is primary scope of this work a developmental behavior without testing is not assumed. The *Repository* represents any structured storage such as an online database. The item *Quality* represents a generalized measure of excellence. In a testing context the code coverage of a test serves as benchmark. With *Developer* the actual subject of observation respectively the addressee of the support offered by our plugin is marked. Figure 3.1 depicts the required activities for the reuse of tests a developer usually has to comply with.

3.1.1 Problem domain

Today commendable development of software includes the likewise creation of code and tests at par. These two tasks also exist independently of whether the code is written first or a TDD approach is applied. Either way, the developer must start with one of both and can not neglect the other part. To not "reinvent the wheel" the developer will likely perform SDD and reuse existing artifacts which may be found in different repositories within the internet. Without the use of specialized code search engines, the developer is likely to use a default search engine and herewith try to achieve adequate results. However, this is probably a Sisyphean task. Additionally the user may not necessarily know how to use specialized search engines since they may have distinct requirements for the structure of a query. In general, these machines are already at a loss at differing names because they often only look for textual matches and a lookup by functionality is not part of the search process. Furthermore, if results are provided, these must not necessarily be reliable. Related to the reuse of *tests* this fact is even more aggravated. But let us suppose the developer was lucky and found appropriate artifacts to his requirements, even then he still is responsible for the the inspection and evaluation of the results, by which the applicability is still dramatically reduced. Again supposed, that even this task was manageable and successful, the user is still responsible for the validation of the test, particularly whether the inserted code fulfills the needs and improves the quality. Despite this, however, the fact remains that the user still has to interrupt his work to search for and evaluate the relevant information. These exactly are the points where TT joins by supporting the user with automated lookup and evaluation processes.

3.1.2 Objectives and Positioning

To bridge these problems TT is designed to target and support all relevant tasks in the reuse scenario of Figure 3.1. More precisely, TT aims to abolish the manual search process by the provision of autonomous background searching. Subsequently the display of the discovered code leads to decreased effort for the user to inspect the results, especially since an automated evaluation can be performed beforehand.

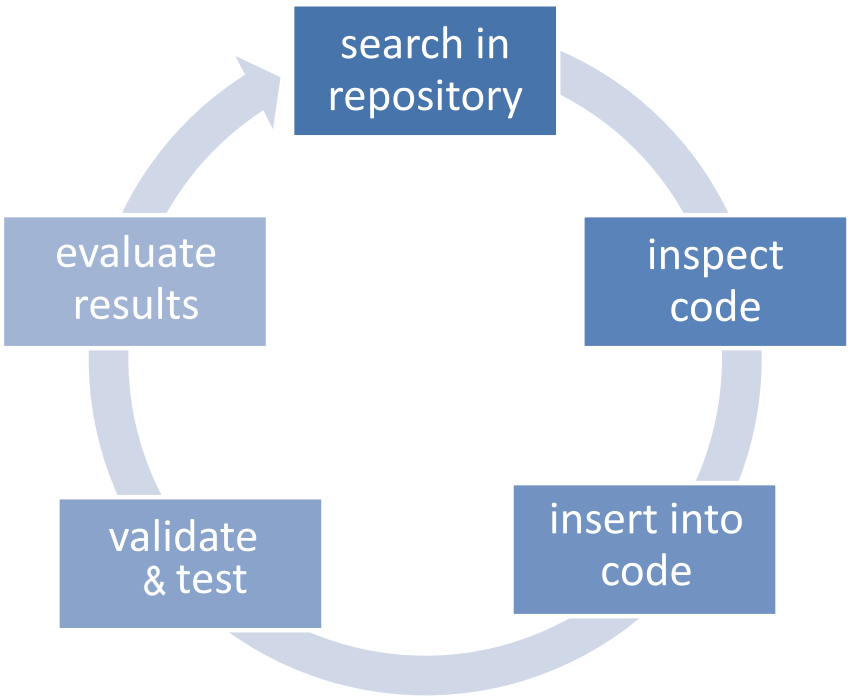


Figure 3.1: A typical Reuse scenario

Additionally the integration of discovered fragments will be simplified by the integration into the IDE. This also leads to drastically reduced distraction as the user does not need to "leave" the IDE to perform a search. Finally an automated validation of the specific test completes the profile.

To simplify SDD for the user the tool Code Conjurer (CC) is currently already available, which significantly simplifies and automates the search process. For this purpose CC comes as an Eclipse plugin and provides the user with code recommendations based on a background database search for specific possibly reusable artifacts and an integrated display in the IDE. Thus CC bridges the gap of distraction related to conventional search behavior, i.e. searching in a web browser. CC works in the latest version for components under development as well as for tests as point of origin for a search. However, there are two issues with CC, and its debatable whether these can be considered flaws, which TT aims to target. The first one is that CC is only able to recommend *components* and not *tests*. And secondly, CC provides its suggestions only in a View and neglects the opportunity to even more increase the ease of use by integrating into Eclipse's proposal system. These issues provide starting points for TT to automatically and systematically apply Reuse to tests. Consequently and complementary to CC, TT's task is to find just *tests* for given components and tests. To not retackle an already solved issue, TT needs to work in an integrated environment, too. Similar to CC the Eclipse framework is found a feasible environment for TT. To even more reduce distraction of the user TT integrates into the recommendation process of Eclipse to enable quick access to the recommendations.

3.2 Preparations

In order to provide the user with valuable results, several conditions must be met. For this purpose the local artifacts need to be prepared. Thus, the recognition of the corresponding Component Under Test (CUT), whose functionality the test aims to check, is inevitable. This information is the centerpiece of any further investigation. In addition the JUnit test type needs to be clarified. The difference in the type of test may be considered as negligible in many cases, and to only refer to a different type of structure

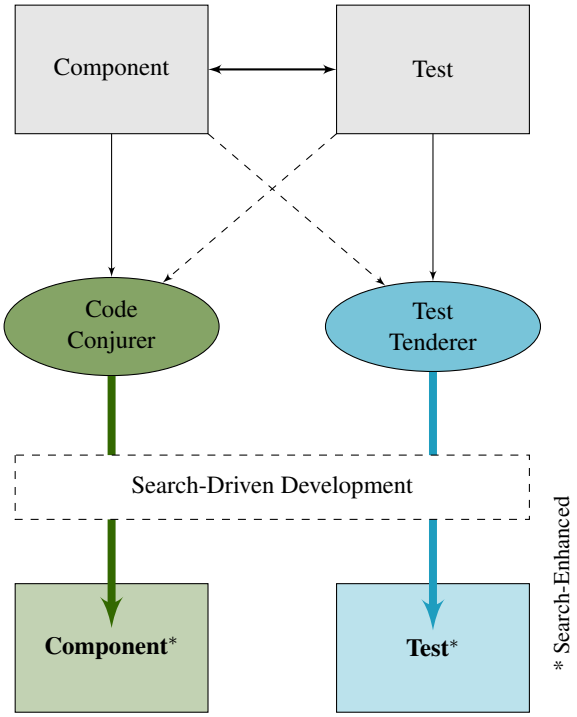


Figure 3.2: Test Tenderer Target Scope

and syntax. But since JUnit 4 tests may be included directly in the CUTs this means, however, that *Test* and *CUT* would originate in the same artifact. For the information collection and object creation process code visitor patterns (4.5) were widely used

3.2.1 Test Preparation

First of all, the determination whether the user is working on a test is inevitable. As only tests are processed herewith all stands and falls. The second required information is the JUnit test type since different kinds of tests require different processing. Furthermore, knowledge about the tested components is essential. And lastly the interface signature (cf. fig. 3.4) of the test needs to be calculated. For all of these issues again different code visitor elements (4.5) are used which can be queried for the desired information. If the fragment is recognized as test additional investigation is instituted. Although the type signature of the test is rather uninteresting for the further process it is feasible to calculate it. It serves, due to its uniqueness, as perfect identifier for a specific test. The way more critical and interesting analysis is the investigation for CUTs and their interface signature, since ultimately not identically structured tests, but the ones testing the same components shall be found. Finally, a *Test* object is thus obtained that, apart its own information (such as type signature and test characteristics), also contains information about the objects of the tested components and thus serves as starting point for further examination and structural assessment, especially for the search process.

3.2.2 Component Under Test Discovery

One of the most essential tasks for the whole capabilities of TT is the correct discovery of the CUTs that belong to a test. They form the key objects of any investigation since they are the starting point for any further information creation activities. As simple as it may sound, however, the identification of suitable candidates is not trivial. That is, because in addition to the (1) identification of these prospects, (2) the associated paths have to be resolved, (3) the code of the detected targets has to be examined, the

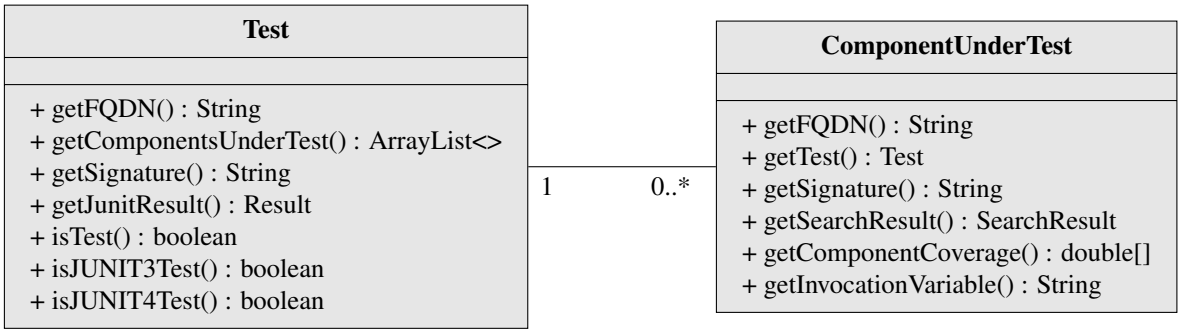


Figure 3.3: Interrelationship of Tests and ComponentsUnderTest

information has to be (4) structured and finally (5) connected to the Test to ensure consistent objects for further processing.The role of the discovery mechanism is therefore manifold and crucial. To make matters worse, Tests and CUTs can be more closely interlinked than expected. This happens primarily through the use of JUnit 4 tests that are located within the tested component. In this case, the Test and CUT objects refer to the same object, resulting in that the test (as seen from the model view) is "self testing". For the purpose of discovery the test is observed for instantiations of classes and analyze their role in the test. Helpful information for this are also the results from the test preparation task. If a component is actually tested or at least a usage is suspected, and it is thus recognized as a CUT, a specific object for further application and instrumentation is created.The inspection of the imports is additionally serviceable as well as the identification of the assigned variable.

3.2.3 Application and Interaction

If the current component under development is recognized as a test a specific *Test* object is created which enriches itself with additional information. This information includes, among other things, the test properties, the Fully Qualified Domain Name (FQDN) as well as the type signature. The *Test* object additionally investigates its own source code to determine the CUT candidates and subsequently creates *ComponentUnderTest* objects for every recognized aspirant. Besides the *Test*-sided setting of association variables (i.e. the test and the invocation variable) this object again fills its information containers on its own and therewith prepares for further processing. On initiation of the search an independent retrieval process is conducted for each of the components a test holds, based on their individual signature. The objects are afterwards enriched with the information received from the server and consequently after the lookup process every *CUT* holds its own individual search result. However, some information placeholders remain vacant at this point and are filled up during further activities. An example for this is the component coverage, that is not set until the first basic coverage generation is accomplished. This also applies to the JUnit result in the *Test* objects. To obtain the interrelationship structure of figure 3.3 this principal is even applied if a *Test* is considered a JUnit 4 test and thus also represents its corresponding *CUT*. In this case, two objects are created though, although both objects to be filled with information from the same source. In summary, these two types in the end contain all the relevant information needed. Figure 3.3 illustrates the interdependence of the *Test* and *ComponentUnderTest* objects and provides an exemplary insight into the internal structure of these objects.

3.3 Search

For the lookup process the SENTRE[1] database was enhanced and configured to closely interact with TT and to cope with the specific requests. To ensure further instrumentation and provide the opportunity to

[1] SENTRE - Search-ENhanced Testing with REuse, http://sentre.se-testing.info/

$$C_i \left(m_1(p_{(1,1)}, \ldots, p_{(1,n)}) : r_1; \ldots \underbrace{m_k(p_{(k,1)}, \ldots, p_{(k,n)}) : r_k;}_{\text{signature of method } k} \right)$$

C_i	:	class name	m_k	:	method name
$p_{(k,n)}$	:	parameter types	r_k	:	return type

Figure 3.4: Interface signature/Query format

extend the application area to other tools and services the required Application Programming Interface (API) of SENTRE was accordingly designed. It thus accepts simple signature requests and returns complex data structures holding the desired information.

3.3.1 Query format

The reduction of the server's demands for the request to the plain usage of the interface signature helps to preserve easy application and usage. The relevant signature here is but the one of the corresponding CUT, not of the test itself, or of all of the CUTs if more than one component is tested at once. The reason for this is, that the interface signature of a class contains the class name, the methods, and their parameter and return types and thus marks a similar structure and herewith again announces similar functionality. And this is just the similarity necessary as the search is conducted just to find the variety of tests that all test functionally equivalent objects. The signature of the test itself would hence not lead to reasonable result. To this effect the signature of the CUTs is the only interesting information. For a successful interaction with the server the signature must correspond the scheme of fig. 3.4. The C_i in this case represent the names of the classes, which in turn usually contain several methods that are given with their method signature. The methods pattern is encapsulated into parenthesis to provide a comprehensive and clear structure. The several single method signatures start with the method name m_k. Moreover, all its parameters (p_n) and the output type (r_k) are considered. The different parameter types are subordinated to the method, again in parentheses, and separated by commas. The output type is symbolically separated by a colon from the method to emphasize the result type character. Therefore, the method signature is reminiscent of a method call or the methods interface description. The different methods are separated by semicolons and the last method entry is terminated by a semicolon, too. A closing parenthesis marks the end of the signature. Notwithstanding this formal definition the composition of a signature in practice is much more intuitive. To give an example, the signature *Calculator(add(int,int):int;sub(int,int):int;)* refers to a *Calculator* class with the methods *add* and *sub*, both expecting two *primitive integers* as parameters and returning again a *primitive integer*.

3.3.2 Result structure

After the submission of the query the server processes the request and searches for corresponding matches in its repository. To ensure the most comprehensive supply the server supports two search modes: a *standard search* and a *relaxed search*. While in standard mode the query is not relieved and thus only results exactly matching the original request are returned, the response from the relaxed search is much richer. By unclenching the class name matching process a lot of more candidates are found and returned in the results set. This leads to significant larger result sets, but at the expense of accuracy. That means, returned to the example above, that not only *Calculator*s but, for example, also *CalculatorTest*s, *CalculatorFactory*s, *MortageCalculator*s, or *TemperatureCalculator*s etc. qualify for recommendation.

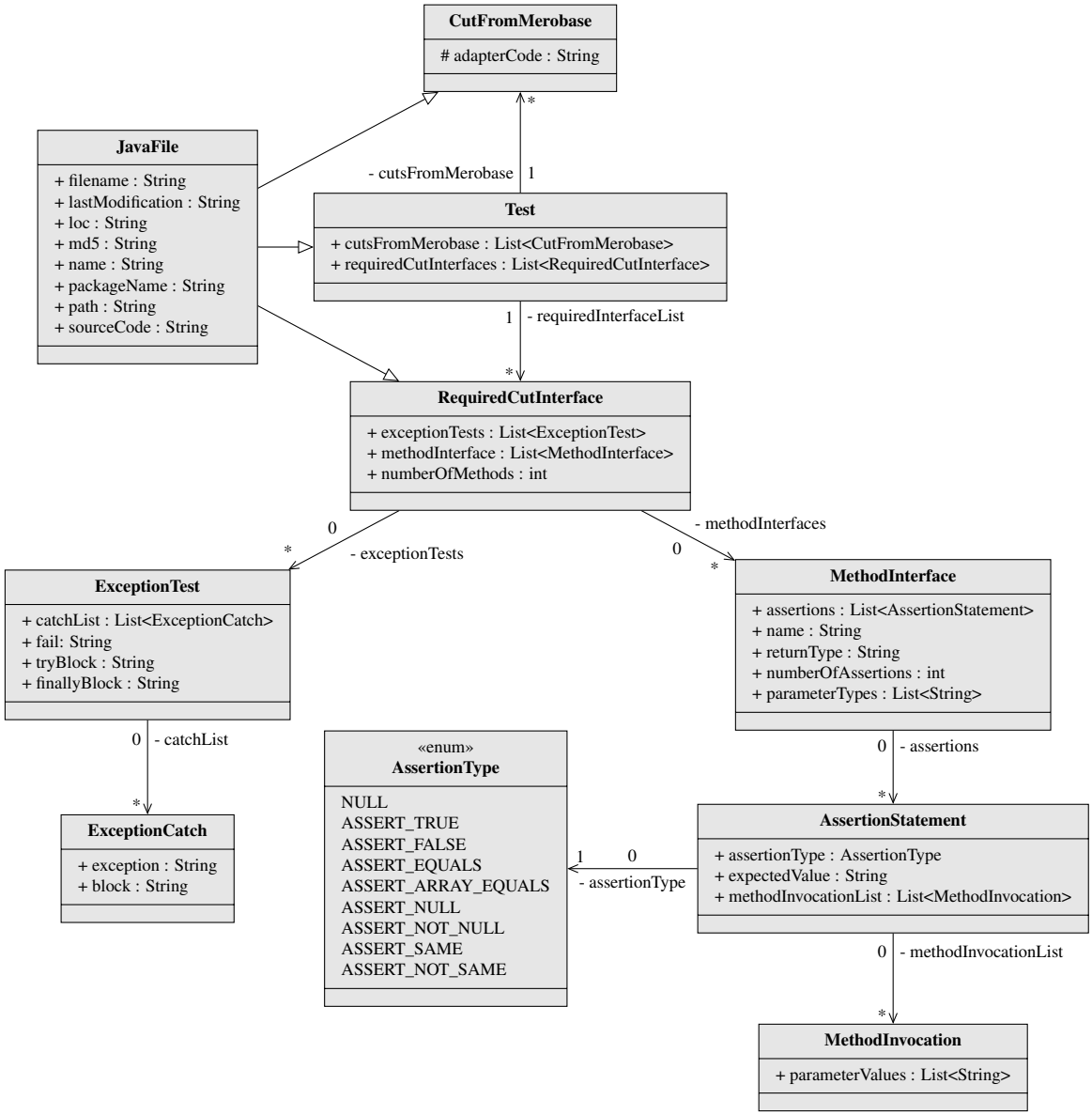


Figure 3.5: Data Transfer Object used by SENTRE / TT

This may be stimulating for a developer, but it is not necessarily helpful since the desired functionality may not be covered by the received contents – even if the similar method signature suggests similar results. In the latest version the relaxed search is offering an additional parameter as threshold which allows the only usage of this search type. Since initially a strict search is automatically performed and the search is only relaxed if the number of results go below the specified limit, the currently implemented relaxed search satisfies all needs and simplifies the search process and the implementation. The threshold is basically adjustable but currently set to 1. In practice, this means that in TT the relaxed search is only performed if a strict search does not deliver any result.

If the search process has been successful and results exist, the information exchange between server and plugin is processed by the use of a Data Transfer Object (DTO) the server creates. This object is then transferred to the requesting client and there transmuted into new complex objects with additional properties and capacities. Regrettably the DTOs are not able to map complex data structures and are thus basically only information containers. Nevertheless, with additional processing the information

from the DTOs is encapsulated and enriched by functionality. This enables us to quickly derive and display structured and finished information for the exposure of tests and the integration into the proposal computing process. Figure 3.5 shows the simplified[2] class structure of the results as received from the server.

3.3.3 Test Recommendations

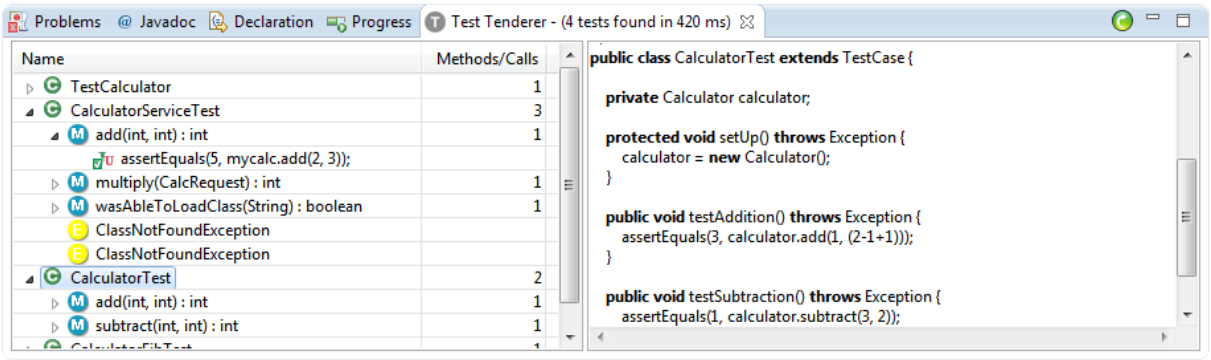


Figure 3.6: Search View

The enriched search objects serve as basis for the provision of the search result information. The objective here is to provide the user with structured and detailed information and thus to enable a simple inspection of the provided code. The advantages are obvious: users may get inspired by the work of others, resolve deadlocks, discover new paths to the goal, or simply get hints how to solve a specific issue. For targeted and easy use the tests are assigned the names of the methods of the CUT, and not the identifier of the test methods itself. The idea here is that the user probably will not care about the name of the test methods, since he is usually looking for test*ed* methods, or methods that he wants to test, respectively. With this result the different classes, their methods with the attached assertions, as well as the discovered exception tests are displayed to the user in the *Search View* registered in Eclipse. This view is bisected: on the left side the view presents the result set in a tree structure with additional information in attached tables. On the right side additional information is displayed, such as the source code of the test or method which is selected in the tree. Here, the tests represent the highest levels and form the roots of each *test sub-tree*. Attached as nodes are the methods that have been found in the parsing process of the corresponding class and which are called by assertion statements in the original test. Those instructions again form nodes of the respective method and therewith the leafs of the test sub-tree. In addition the *Exception Test*s found in the database are subordinated to the tests itself. On selection of these nodes the user is also displayed information in the right preview pane, here in form of the try/catch-block of the exception. Figure 3.6 shows this structure implemented in TT. This hierarchy easily enables users to browse the different results, to possibly identify additional methods, observe the testing behavior of other developers, or to obtain ideas how to test a specific method - or even to “crib” code. Summarized, to comply with the principle of Reuse.

3.4 Speculative Analysis

Undoubtedly most scientists will agree that it is not possible, in general, to predict the future. This is because it is impossible to know all the (mostly unknown) variables and to estimate or simulate their

[2]In the original data structure, all variables are private or protected, and there are corresponding getter and setter methods. For the purpose of illustration these methods were completely excluded and the variables were set to public.

interaction. Similarly, it seems at the first glance impossible to determine the influence of a proposal on the coverage of a test before it is inserted in the code. But since precisely that is a particular goal of our work, i.e. to provide assistance with the selection of an appropriate proposal to the user, the coverage information is inevitably needed beforehand. Some kind of "clairvoyant method" had thus somehow to be found to simulate the usage of the proposals and to estimate the potential coverage change. Fortunately unlike the real future, the "future" of a test, however, can be estimated at predictable effort since the variables are rather well known and manageable. By these requirements the means to our end was found with *Speculative Analysis*. Using this methodology we are able to assess the impact of the application of a proposal for the test in advance. For the calculation of the coverage the speculative analysis algorithm of figure 3.7 evolved. Our first approach was to start with the creation of a temporary copy of the project in order to avoid any interference or damage to the original code. For that we simply copied each file and each directory from the original project to the system's temporary folder. This resulted in redundancy as the libraries are copied as well. On the other hand this ensured that every component was always up-to-date and simplified the recomposing of the project's class path and its dependencies for the temporary project. In the latest implementation the speculative analysis is performed in-memory. This eliminates any hard disk transfer delay and cost and speeds up the analysis process dramatically. For every assertion proposal we have received from the performed search we create a temporary java objects. It is an exact copy of the original source the user wants to insert the proposal in, but with a new randomly generated name to avoid conflicts. In this code we insert the proposal on the appropriate position. We now have a source to speculatively analyze. As we are working with a *copy* of the original source the next step is not a breeze. Basically with the information about class path settings and used packages from the original project the Java compiler is easily able to compile the generated file, but this step was quite tricky during the implementation. Especially to find a way to systematically unravel the original class path, the imported packages, the used libraries, and to ensure the consistency of all dependencies was quite challenging. With that information we are able to equip the compiler with all the information needed for successful compilation. Since we want to calculate the coverage of this test we have to observe the corresponding CUTs. At first glance this may sound odd – but since the test with its assertion statements basically only invokes methods in its CUTs and different assertions execute different code and thus lead to different code coverage the observation of the CUTs is the key.

The coverage of a test is the *accumulated coverage of its components* under test.

When the compilation of the temporary test is successful, the generated class file object is loaded by a custom class loader and then executed by a *JUnit Core*, which starts the test and collects information about the test result – especially if a test fails or succeeds. In background, with the help of the JaCoCo library (compare Table 3.1), we collect data and therewith prepare information about the coverage. This knowledge is finally stored in the proposal objects for further processing and displaying.

3.4.1 Coverage Criteria Selection & Adjustment

The TT preferences menu offers users the possibility to individually calibrate the sorting and displaying methodology. The user may select one of the following criteria as a dominant sorting criterion. According to the chosen entry TT ranks the proposals in an appropriate way. The order indicates increasing significance. In addition to discrete coverage criteria, TT provides a calculated coverage criterion that enables the user to customize the desired weighting of the "standard" coverage criteria and so the sorting of the results. The different weights can be accelerated via the settings menu of TT (3.8). The proposals are subsequently sorted by and displayed in the order of the calculated individual relevance. For each criterion a percentage can be assigned to adjust the intensity of impact of the respective measure size. TT offers the following six coverage criteria: *Method Coverage*, *Branch Coverage*, *Line Coverage*, *Instruction Coverage*, *Complexity Coverage*, and *Mixed Coverage*.

```
1  create temporary copy of project*;
2  for every proposal in search result do
3      create temporary copy of original source / file*;
4      insert proposal into temporary object;
5      compile temporary file;
6      if compilation was successful then
7          calculate coverage;
8          store coverage information in proposal object;
9      end
10 end
11 delete temporary project*;
```

*only necessary for off-memory compilation

Figure 3.7: Speculative Analysis

All of them, except for Mixed Coverage, are fixed coverage criteria and hence an individual adjustment is not contemplated. The static techniques used (1-4) are of limited significance since they do not take the program flow into account – but they are time-efficient and thus suitable for our live recommendation system. Nevertheless – or exactly for this reason – these criteria are supplemented by (5) the control-flow criterion *Complexity Coverage*. The last item (6) but points to the individually weighted scale mentioned above. On selection the user is granted access to the acceleration panel of the settings menu (fig. 3.8) where the weights of the desired other "standard" criteria may be adjusted. The favored weight is accordingly computed. In the settings menu the users are also able to apply two different filters: (a) to show only proposals that improve the coverage, and (b) to mask proposals that fail the JUnit test . At this point it should be noted that the exclusion of failed tests primarily serves clarity. Indeed, it is precisely these tests, which are valuable because they are questioning the existing implementation.

In the development phase the *failing* tests are in fact the most *valuable* tests.

On application of the settings, TT saves the modifications and uses them for the ranking of the proposals. As a matter of course TT applies the weighted coverage to the ranking, if Mixed Coverage is chosen. These settings only concern the proposal generation for the live recommendation process, the display of the tests in the search view is not affected.

3.4.2 Coverage Calculation

For the calculation of the coverage of source code several tools and libraries exists which, however, differ in usability, functionality, and the provided measures. Figure 3.1 provides an overview of different tools, especially about the provided measures for coverage. Most of the inspected tools offer either Line Coverage (LC) or Statement Coverage (SC). The next widely supported measure is Branch Coverage (BC) which is anyway part of $2/3$ of the tools. The criterion Method Coverage (MC) is only supported by four of the tools, but thereby still twice as often supported as Relational Coverage(RC) and Complexity Coverage (Cc) each with two occurrences. Path Coverage (PC) is only supported by the *Quilt* library. Besides these informational differences, the usability of the selected library was a dominant criterion. As a result we decided to use the *JaCoCo* library as it provides the best fit between functionality, provided measures and applicability. This package provides different methods for the calculation and provision

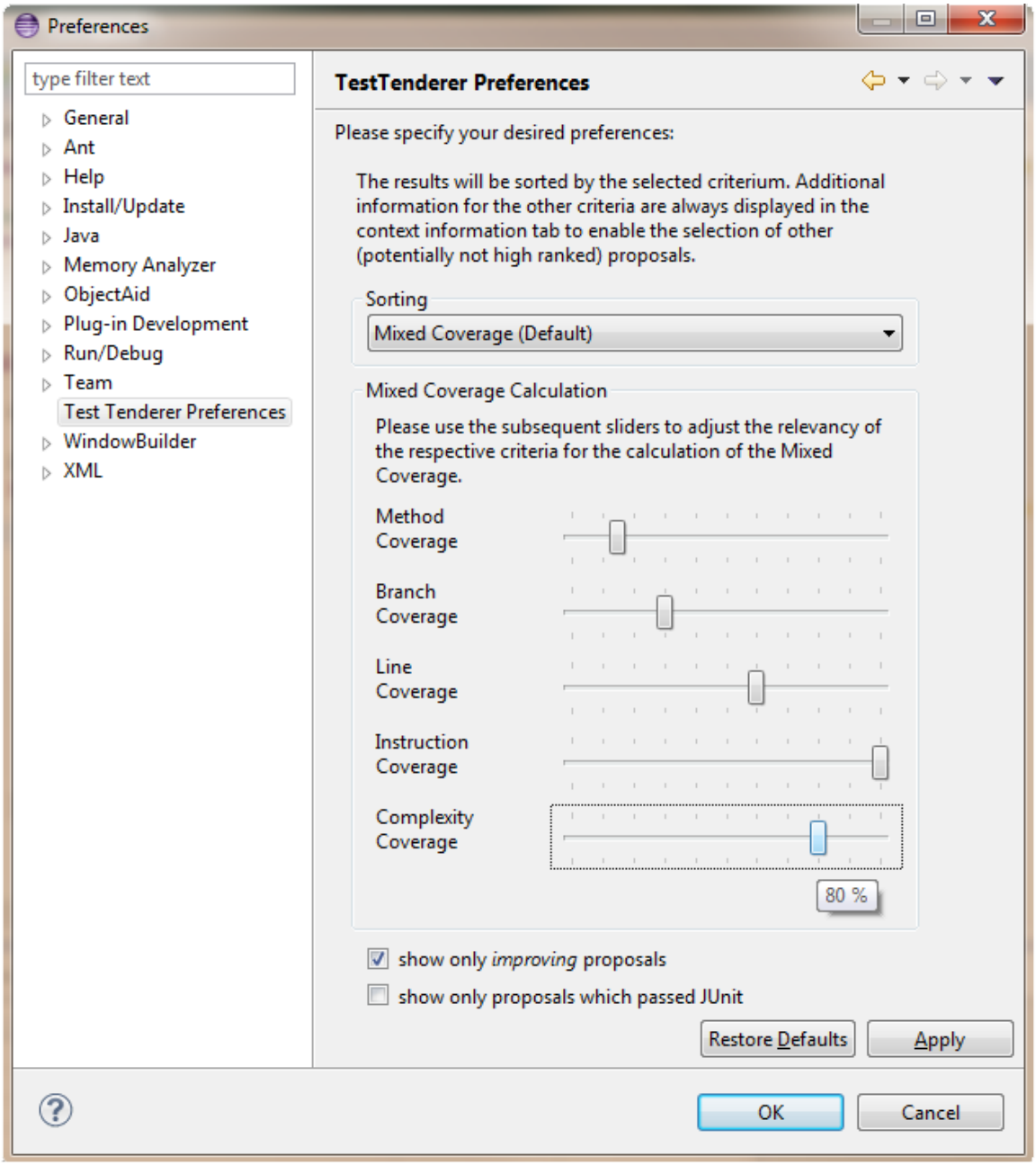


Figure 3.8: Test Tenderer coverage settings menu

of the coverage by means of different criteria as well as distinct information derived from the inspected code. In TT we included the criteria provided by JaCoCo one to one. Users may choose a dominant criterion for the sorting of the resulting proposals. The proposal computer accordingly sorts and displays the results in the appropriate order. Additionally we define the new coverage criterion *Mixed Coverage* that enables the user to set particular weighting and thus to apply individual preferences to the sorting and display of the proposals. With the computation of the *weighted mean* (3.1) we are additionally able to take the users' preferences into account. The different coverages here are indicated with c_i whereas the w_i refer to the assigned weights. Consequently, this means that in our implementation $i = 5$ as we work with the five coverage criteria offered by the JaCoCo library. Both the coverages c_i and the weights w_i

$$\frac{\sum_{i}^{i}(w_i c_i)}{\sum_{i}^{i} w_i} \text{ with } 0 \leq c \leq 1 \text{ and } 0 < w \leq 1 \tag{3.1}$$

Name	URL	Year	MC	LC	SC	BC	RC	PC	Cc
Grobo	http://groboutils.sourceforge.net/	2003		•		•			
Quilt	http://quilt.sourceforge.net/	2003		•	•	•	•	•	
NoUnit	http://nounit.sourceforge.net/	2006	•		•				
InsECTJ	http://insectj.sourceforge.net/	2005	•		•				
Cobertura	http://cobertura.sourceforge.net/	2010		•		•			•
EMMA	http://emma.sourceforge.net/	2005	•	•	•				
CodeCover	http://www.codecover.org	2011		•	•	•	•		
Coverlipse	http://coverlipse.sf.net/	2009			•	•			
JaCoCo	http://www.eclemma.org/jacoco/	2013	•	•	•	•			•

Table 3.1: Comparison of Coverage Calculation Tools

in this case are floating numbers between 0 and 1 representing a percentage value. This allows to include the weighting for each of the criteria on the overall score. At the same time this methodology ensures that if a criterion should be absolutely weighted and the other criteria are neglected the same result is achieved than if the corresponding criterion is exclusively selected as sorting and weighting measure in the settings menu (fig. 3.8).

3.4.3 Basic Coverage

With the term *Basic Coverage* TT denominates the coverage of the test *without* the insertion of a proposal. In other words the basic coverage is the current coverage of the test as is. It is both displayed in the *Coverage View* and used for the calculation of the improvement value of the proposal in the *Proposal Computer*. Here, the difference between the target coverage of the proposal and the basic coverage yields the deviation. As a matter of fact the basic coverage can only be calculated if the test is compilable. For this reason several triggers were integrated indicating whether a test is executable or not. In addition these triggers are able to request an update if the test compilation is not possible at the moment of request. These requests are subsequently processed as soon a test becomes compilable again.

3.4.4 Continuous JUnit Coverage

Besides the usage of the calculated basic coverage for the ranking of the proposals TT herewith provides the user with feedback about the currently achieved coverage for the test and thus possibly points out a way to go. This information is displayed in the *Coverage View*. Here the user receives detailed information about the current coverage subdivided into the different coverage criteria. Additionally the preferred and individually calculated *Mixed Coverage* is displayed. Although reasonable trigger mechanisms exists TT offers a manual refresh function for convenience in this view (fig. 3.9). This functionality has been integrated because the actualization is not always necessarily required for the view and an ongoing forced recalculation is not sufficient. However, on changes affecting the coverage this view is updated, though. Furthermore, the view comes up with a JUnit indicator. This colored feedback bar indicates the success of a JUnit run as it does the JUnit view as well. This indicator basically handles three different

states. If the JUnit run is successful the bar is displayed in bright green, on failure in dark red. If there are constraints that hamper a JUnit run (e.g. compilation errors) the bar pales to grey. In addition helpful information is reported in the text field beneath the bar. In case of success the duration of the JUnit run is displayed, whereas the reason for failure is reported on disappointment. With the help of this view the user thus gets an instant feedback about the recent testing efforts. As a result this represents a powerful feature and eases the testing process since recurrent JUnit runs herewith are a thing of the past.

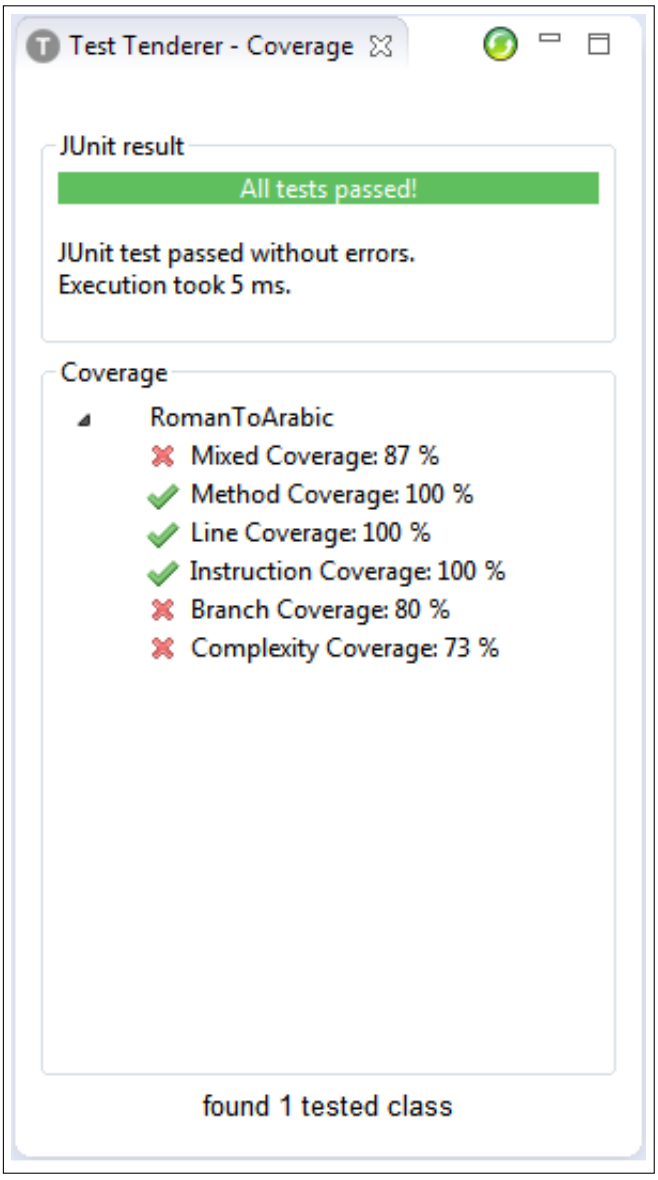


Figure 3.9: Coverage View

3.5 Proposal Generation

Since the proposal of different assertions in a test development scenario can be very advantageous and helpful, Test Tenderer is designed to provide users with live proposals containing distinct information about the achievable code coverage that would be attained with the usage of the specific code snippets. This can be achieved by Speculative Analysis (see section 3.4 for detailed information). Every assertion

is processed for potential coverage by a speculative coverage calculation. This information is then used to provide and recommend the proposals to the user. On activation of the Eclipse Proposal Computer the calculated information about the target coverage is displayed in form of Assertion Proposals that can - with one click - easily be inserted in the source code. Another sort of proposals are simultaneously provided, namely the Method Proposals. Those are suggestions for methods that have been found in tests and the corresponding CUTs, but are not yet present in the currently tested source code. If, for example, a user works on a *Calculator* class and has only implemented an *add* method so far but in the search result a test is found containing an assertion testing a *subtract* method, TT offers the user the opportunity to directly insert the *subtract* method taken from the corresponding database CUT into the local CUT. Last but not least the Exception Tests are mapped into Exception Proposals and subsequently displayed as well. Of course, they also provide insertion functionality but contrary to the *Method Proposal*s not the CUT of the test but the test itself is affected by the insertion. To be compatible with the Eclipse Proposal Computer the *Proposal* objects have to implement functionality for at least the (a) display of information in the *ProposalComputer* and (b) special code modification behavior on application if selected in it.

3.5.1 Assertion Proposals

Measured against the requirements of the Eclipse Proposal Computer interface requirements the *AssertionProposal* objects are very manageable. Notwithstanding, this class provides a variety of information. These objects offer, besides different methods for text to be displayed, insertion behavior. Furthermore, they accommodate access points for retrieval of the associated coverage, the also associated test result or even simple variables that indicate whether an assertion is actually compilable or whether it is a "improving" or "worsening" proposal. All this allows TT to provide the users with qualitative information and thus to provide a basis for decision and thus enabling them to make a sensible choice. An *AssertionProposal* textually presents itself in the *ProposalComputer* with the notion given in figure 3.11 for any k-th assertion. The JUnit indicator i_k symbolizes a successful or failed JUnit test run and exists in three versions: a green check mark, a red and a gray X (Figure 3.10). The green icon indicates that the speculative analysis with the selected proposal was successful and the test passed the JUnit test run. The successful speculative analysis can also be read by the red indicator. However, in contrast to its green counterpart, the test was not successful. If no successful speculative analysis could be performed, this is illustrated by the gray X in conjunction with an accordant message in the information area. The T_K refer to the

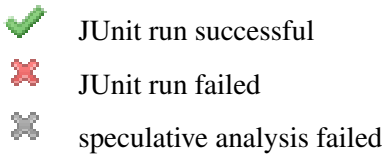


Figure 3.10: JUnit indicator

different *AssertionTypes* of figure 3.5 which were already used for the DTOs. The variable v_k comes from the associated CUT and is discovered during the parsing process by the *TestParser* (see section 4.5.2 for details). It is used in the statement to ensure that non-static methods as well may be invoked. Such a method is denoted with the m_k as found in the search result and existing in the CUT. Together with its arguments (a_k) and the assertion type T_k these three variables form a method call, which again is the first argument of the surrounding JUnit Assert statement. The second argument of this command is the expected value e_k which is compared against the return of the call under the terms prescribed by the type (see section 2.2 for details). The whole values are derived by the search result and assembled by the *ProposalGenerationJob*. Appended to the expression follows the target coverage c_k which has been calculated with Speculative Analysis. For comprehension and display purposes this value is presented in whole percent without any decimal places. Additionally another information is given with the δ_k value that illustrates the enhancement in coverage compared to the currently capped range. The δ can theoret-

ically be negative, but only if execution fails due to erroneous proposals. Thus, this negative coverages of the failed proposals are filtered out to not confuse the user with "negative coverage enhancements". More convenient examples of such proposal list entries can be found in figure 3.12 which implements the above definition.

$$i_k \underbrace{T_k(v_k.m_k(a_k), e_k);}_{\text{JUnit Assert statement}} \text{ - } c_k\,\%(\pm\delta_k\,\%)$$

i_k	:	JUnit indicator	a_k	:	arguments for method in CUT
T_k	:	assertion type	e_k	:	expected value
v_k	:	variable for CUT	c_k	:	target coverage
m_k	:	method in CUT	δ_k	:	delta to basic coverage

Figure 3.11: Assertion Proposal - textual representation

3.5.2 Method Proposals

In contrast to the Assertion Proposals the *Method Proposals* are not processed, but simply assembled. For this purpose TT examines the tests from the search result as well as the local CUT and compares the methods tested and accordingly the methods of the CUT. If in the search result a method is discovered for which assertions exist, and if such a method cannot be found in the local CUT, a *MethodProposal* object is created with the respective information taken from the database. This information includes the name of the method, the number of assertions found in the search result, and the method body taken from the database CUT. The method proposals are inserted into the proposal computer subsequently and thus offer the user to simply enrich the tested object's source code. To illustrate this from another point of view: while the assertion proposals aim to evolve the *test* the method proposals target at the enhancement of its *tested components*.

Assertion Proposals and *Exception Proposals* enrich tests, *Method Proposals* enhance CUTs

The visual representation in the *ProposalComputer* is trivial. The icon used is but to be mentioned as TT indicates Method Proposals with a blue "M" and therewith clearly contrasting with the other proposals for quick identification and comprehension. But even to display simple text and to enable the proposal for application, the objects have to store distinct information. As the *MethodProposal* is only valid for one component it thus holds a pointer to the CUT it is created for. Due to this it can access all necessary information from and about the CUT. The method body for the later insertion is already passed over during the creation of the objects to avoid unnecessary calling and processing effort. Since practically different implementations for similar methods exist, TT creates several Method Proposals for ostensibly equal methods. This may – dependent on the specific scenario – lead to an abundance of proposals, but as they may be de facto different the user has to have the opportunity to choose different implementations. When the user finally made a choice the desired implementation is inserted in the corresponding component.

3.5.3 Exception Proposals

Another type of proposal is the Exception Proposal type. During the parsing process the sources in the database are also investigated for Exception Tests. In the result set of a conducted search these exception tests are thus also included (cp. fig. 3.5). These tests form the basis for the Exception Proposals. The

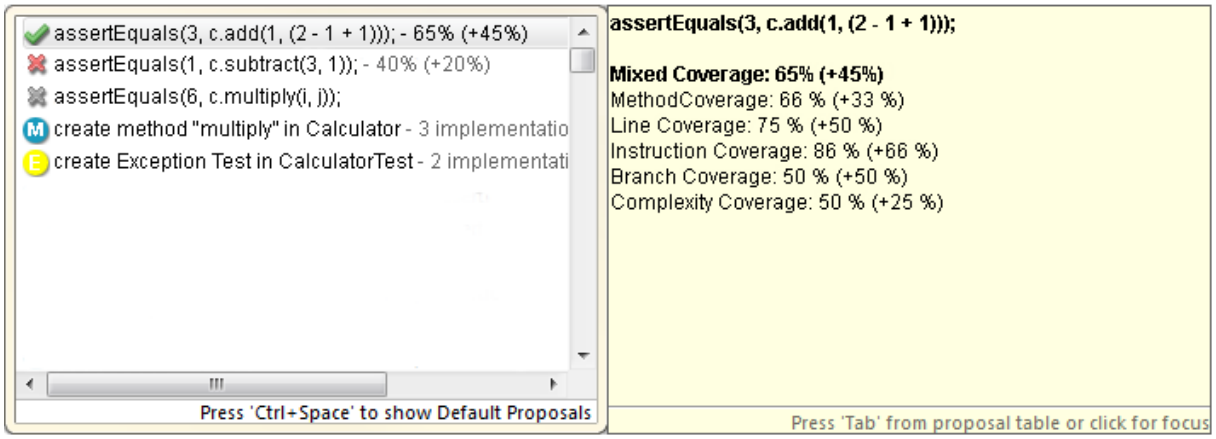


Figure 3.12: Test Tenderer proposal computer

information held by the exception tests is used to construct a body, which again is used for display and insertion purposes. Therefor the *try* block, the *catch* list, the *finally* block, and information about the *type of exception* this exception test aims to handle is extracted from the exception test. The subsequently internally assembled code body is based on the generic structure of try / catch blocks. During the speculative analysis it is inserted in the test in order to calculate the relative coverage of this proposal. For this task the exception body is wrapped into a randomly named method structure to avoid collisions with other methods or structural boundaries on insertion. The same procedure is used on application by the user, merely the generated method is inserted into the test itself. Visually the Exception Proposals contrast the others by the usage of an individual icon – TT indicates this proposal type with a yellow “E”. The textual representation indicates the behavior on application and is a simple advice that an exception test method of the distinct type is created in the test. Notwithstanding that the exception type of the original test is available, a generic Exception catch is inserted to avoid dependency resolution constraints.

3.5.4 Proposal Computer Integration

In Eclipse’s Proposal Computer the generated recommendations are displayed in the order of relevance. This relevancy is a presentation of the ranking which is applied during the generation process. The *Assertion Proposals* are assigned a relevance according to their coverage improvement, i.e. the difference of the target coverage compared to the actual coverage. The higher the potential, the higher the relevance, i.e. the ranking. Subsequently the method proposals are placed, finally followed by the Exception Proposals. The different proposals additionally provide information which may be helpful for the user. Figure 3.12 exemplary shows proposals generated by TT. On the left side in the *proposal list* the different recommendations are presented while the right (light-yellow shaded) *information area* contains information about the selected proposal. The green hook and the red cross respectively indicate whether the proposal succeeded or failed the JUnit test. A grey cross indicates that the speculative analysis could not be performed. The blue “M” indicates a method proposal that offers easy application to the local CUT. Here the method body of the database CUT is used, and on application subsequently inserted. The same is shown for the Exception Proposals, except that a yellow “E” is used as an identifier. The proposal list items are when appropriate enriched with additional information for a brief intelligence. So the Assertion Proposals are enriched with coverage information that depend on the selection made in the settings menu. In addition, the difference to the actual coverage is indicated in brackets. This choice is reflected in the information area, too, as the desired dominant criterion is highlighted in bold text. Besides this redundant information the other coverage criteria are displayed as well. This may help the user to get a feeling for the different criteria and as the case may be adjust the individual preferences. The Method

Proposals are in turn equipped with the name of the method. In the information area these proposals show up with a preview of the code to insert. The Exception Proposals indicate the kind of Exception tested and provide an insight to the source code of this Exception test method in the information area as well.

3.6 Summary

With our tool Test Tenderer we are able to tackle some problems of a conventional Reuse approach by automation. TT provides Assertion Proposals and Exception Proposals for test enhancement, and Method Proposals for CUT enrichment. In addition TT creates value by the provision of coverage information in advance of the application of a specific assertion proposal. With the help of this functionality the demand for an ex-ante evaluation of testing effort as stated in [21] could be satisfied. The user is hereby able to evaluate the results beforehand and thus to come to an informed decision. Beyond that, TT offers the opportunity to individually weight the different criteria to adjust the desired measure of excellence. Finally with the help of the provision of the Continuous Coverage Calculation in the Coverage View recurrent JUnit runs become obsolete. The concept of continuous testing [1] could hereby be realized.

CHAPTER 4

Implementation

In this chapter different aspects of the implementation will be highlighted. The first section is dedicated to TT as Eclipse Plugin itself starting with the integration into the underlying framework. This is followed by section 4.2 covering the cooperating database and the interaction with the plugin. In section 4.3 the architecture as well as selected activities of Test Tenderer (TT) are reviewed. Hereafter, the process workflow is highlighted in section 4.4. Section 4.5 is dedicated to Visitor Patterns, which were widely used in the development of TT. The last passage closes with a brief summary about the requirements of the development and starting points for potential implementation enhancements.

4.1 Eclipse Plugin

The Eclipse Integrated Development Environment (IDE) was identified as the ideal platform for the development of this tool due to its reliability and availability, but especially since its plugin-driven functionality and its extension ability allows developers to freely contribute to the framework. With this feasibility TT was assumed to be able to exploit the potential of the available data discovered in the multitude of tests in the database. This popular framework has its origins in the early 2000s, when a subsidiary of IBM decided to create an integrated framework. Since then 8 new versions were released. The latest stable version is Eclipse Juno (version 4.2), which was utilized for the development of TT.

4.1.1 Extension Points

To enable and ensure extendability, Eclipse basically offers the opportunity to "dock" to various so-called Extension Points in order to extend functionality [31]. For Test Tenderer (TT) the Extension Point *org.eclipse.jdt.ui.javaCompletionProposalComputer* was used to register the proposal computer, as well as *org.eclipse.ui.views* to contribute the views to the UI. The first "cooperation" aims at an integration into the recommendation system of Eclipse and allows to contribute customized proposals. The second linkage offers the opportunity to interact with the user by visually contributing to the framework's user interface. As we developed TT as a plugin and decided to offer the users the opportunity to adjust several settings we also "docked" our preferences to the *org.eclipse.ui.preferencePages* extension point to ensure smooth integration in Eclipse's preferences menu structure.

4.1.2 Views

Views are part of the workbench and fulfill specific UI tasks. While several built-in views, such as a Package Explorer to browse and display the project structure and its components, or the Console view that displays the application's output, exist, developers are enabled to contribute additional Views freely to the IDE and to equip them with individual displaying features. As the name alludes, Views are part of Eclipse's UI. TT makes use of this ability twice.

Search View

The first UI extension, the Search View supports the user with the structured results from the search. The results are displayed in a so called *TableTree*. The characteristic of those trees is that they are structurally built like trees, but each node has a table-like substructure. Within this table information can be displayed in different columns. However, for the tree in the Search View one column is entirely sufficient. TT here consistently provides the number of nodes subordinated to the respective item. That is in case of a test the number of different tested methods found for this test, and in case of a method item selection the number of subordinated assertions. On the right side of the view a multiline *org.eclipse.swt.custom.StyledText* pane is used to inform the user about the content of the selection. If e.g. a test class item is selected in the tree, the source code is displayed on the right. The users can hereby inspect the work of other developers, gather ideas for their own code development, or can simply copy code snippets from the displayed source code. In order to foster the comprehensibility of the code, and thus to enable the user to easily read and understand the displayed source, a code formatting mechanism has been integrated to "beautify" the result. The text is therewith given the appearance of formatted by an Eclipse code editor. Thus, for example, words like "package", "class", "public", etc. are highlighted in bold. The applied hierarchical structure and the provided information will enable developers to estimate the value and appraise the impact of the results in a convenient and comprehensible way.

Coverage View

The second UI contribution is TT's Coverage View, supporting the user with coverage information about the test under work. This View makes use of different elements of the *org.eclipse.swt.widgets* package to individually model and display the coverage information for each component under test. For the presentation of the different coverage criteria the *Tree* class of this package is used. In particular, for every discovered component under test a single tree is inserted and denoted with the name of the particular CUT. Subordinated to each root node the different coverages that are achieved by the test for this component are displayed. Depending on the particular coverage criterion the leafs indicate the achieved score with a distinct icon. That means, that TT denotes imperfect coverages with a red cross and completely covering criteria with a green hook. In addition to this central tree element different *Label*s are used for information purposes. The JUnit indicator bar, for example, visually informs the user about the result of the latest JUnit run with the use of different colors. Other applications of the *Label* are just to provide the user with textual feedback, such as result of the JUnit run or whether a test is compilable or not. The Coverage View updates on demand. That means, if an internal method requests an update, the coverage information is recalculated and refreshed. This happens on changes to the signature of the test, altered assertion-statements in the test, or if a search was conducted. If an update is not instantly possible (e.g. due to a currently not compilable test) a corresponding flag is set to request an early recalculation. When this particular flag is set, TT triggers the update on any next change event in the editor, such as a key stroke or the activation of an editor.

Activation Surveillance

One particular aspects will be additionally mentioned as its solution was simple but not trivial. The problem is, that views can only be addressed and updated if they currently exist in the workbench. If the views are not active an exception is thrown stating that the "widget is disposed" which can quickly disturb the update process and thus lead to an inconsistent UI state. A promising approach seemed to be to question the views instances for whether they are active or not, that means whether they exist in the workbench. If a view was considered not active it is displayed with the help of the *org.eclipse.swt.widgets.Display* class in conjunction with the current *org.eclipse.ui.IWorkbenchPage*. This approach seemed feasible and appeared to be goal-oriented and cost-effective. However, the remaining task to acquire information about the state of the view was not trivial to solve. Finally we were able to collect the desired information. For that purpose the *org.eclipse.ui.IPartListener* was added to the implemented interfaces, and registered

as *listener* with the *IWorkbenchPage*, as it provides connecting points for the collection of information about this view. It holds several serviceable methods, among them a *partActivated* and a *partClosed* trigger method, which are invoked on the particular event. With the use of an internal boolean indicator, which is modified on the particular events, it was thus possible to appropriately react to the particular view state and thus to avoid failure by addressing inactive views.

4.1.3 Proposal Computer

The Eclipse IDE provides several proposal computers that simplify the programming process and enable the user to comfortably find and observe desired functionality. When activated (by pressing CTRL + Space), Eclipse suggests completion proposals enriched by additional information about the specific functionality of the envisaged component, required constructor details, or possible alternatives. This proposal mechanism is usually very promising and eases the development process considerably. TT's Proposal Computer implements the *org.eclipse.jdt.ui.text.java.IJavaCompletionProposalComputer* interface and thus provides several triggering and information processing mechanisms. For instance the interface requires the implementation of the methods *computeCompletionProposals*, *computeContextInformation*, *getErrorMessages*, *sessionStarted*, and *sessionEnded*. Since we registered this class with the Eclipse system these methods are autonomously invoked by Eclipse on demand. In this connection the first three elements act as information providers for the IDE and supply Eclipse with a list of *org.eclipse.jface.text.contentassist.ICompletionProposal* objects, respectively when context information is requested a list of *org.eclipse.jface.text.contentassist.IContextInformation*. In our implementation only the first supplier is used as we are adding the Proposal-specific context information to these objects directly. Furthermore the proposals are only filtered at this point, since the construction is accomplished asynchronously in the *ProposalGenerator* and the actual proposals are hosted in the central singleton class *TestTenderer* for simple and cross-class access. That means, that Eclipse's call of this method is simply forwarded, the filters applied comply with the settings in the preferences menu. The session-related methods serve as triggers and are invoke at the beginning of a *ProposalComputer* session, respectively at the end. Their existence allows to react with preliminary and with follow-up activities. This ensures, for instance, that with the start of a session no calculation is started if TT is inactive, and that all jobs are stopped as soon as a session ends.

4.1.4 Preferences

In order to engage in the preferences structure of Eclipse all registered preferences classes have to implement the *org.eclipse.ui.IWorkbenchPreferencePage* interface and - in order to provide customized behavior - to extend the class *org.eclipse.jface.preference.FieldEditorPreferencePage*. The pages are filled with different elements from *org.eclipse.swt* package. For the composition of the preferences classes the WindowBuilder[1] Java GUI designer was used. The implementation was straightforward.

4.1.5 Runtime Environment

To benefit from the latest features of Java we decided also to use the latest version 7 (Update 17). This decision had actually forces the user to also use this latest version to enable the usage of the plugin. Since we have to perform various source code changes and thus need additional functionality from the Java Development Kit (JDK), whose availability is a further and quite major hurdle, the decision whether to rely on the topicality of the customarily installed Java Runtime Environment (JRE) or to demand installation became dispensable. That means that the user is required to have a valid JDK installed. If only a JRE is found on the target system TT displays an appropriate message and denies activation.

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

4.2 Database

The data basis for the recommendations afforded the Merobase [9, 10] database, a component search engine with currently 9.433.422 entries, of which solely 7.898.005 items are Java classes[2]. These elements were analyzed for their capacity of a test and – if a test case was found – thoroughly examined. All discovered tests were subsequently migrated to the SENTRE database. Through the exploration process the structure, the signature, and finally the information from the assertion statements was extracted to offer these data in the web interface as well as to TT for recommendations. Additionally, the tests were examined for calls on other tested methods. These invocations were examined and the signatures of the invoked methods were extracted to provide the user with suggestions as to what methods could also be interesting. Moreover, the data sets were investigated for exception tests, which were also extracted for further usage. The data thus collected form the basis of the data model that allows structured representation in the web interface and a purposeful use in TT.

4.2.1 Preparation of the data

The parsing of the tests in the database was accomplished in our group. The main goals were to prepare the data for a usage in a web search engine and to interact with TT. For the web search scenario especially a lean data structure, a reliable indexing mechanism and consequently a fast lookup process were the measure of things. For this purpose the Merobase [9, 10] was used and its data were parsed for tests. As a result a discrete database on the basis of the collected data could be established. The *SENTRE* database is designed for the utilization of tests for reuse. The parsed data were assembled to a more complex, but consequently better utilizable structures. Figure 3.5 in chapter 3 illustrates this structure. With these collected information and the corresponding structure we opened the gate to the world of Test Reuse.

Assertion

Since TT besides the display of suitable code also aims to provide the user with assistance via Eclipse's proposal computer, the extracted assertion calls available in the JUnit framework were examined in detail. As described in 2.2, these assertion calls basically all function on the same principle: an *Assert* call is passed both an *expected value*, and an *actual value*, wherein the later usually is the result of a method call. JUnit then compares both values and indicates whether the expected value matches the actual (calculated) value. We thus had to not only look for the assertion itself, but for the arguments and for the invoked methods, too. If an assertion call was found, the information about the invoking method and its arguments were extracted and together with the expected value stored in the result set. The collected data therefore comprise conclusions about the functionality of the tested method – even though it could depict a failing test.

Methods

The extracted methods exist in SENTRE only with their method signature. These signatures actually do not represent the methods *in* the test, but rather methods of the CUT associated *with* the test, and originate in previously discovered assertion statements or on the call *within* JUnit statements.

Exceptions

The information about exception tests needed for the provision of Exception Proposals have been directly taken from the source code of the test. If a test with a try/catch body was found, the entire code was stored in the corresponding object.

[2] as of June 2013

4.2.2 Utilization of the Data Transfer Objects

As the results have to be easily transferable via the internet to be available to TT, a straight structure for the transmission objects was pursued. Consequently, the data sets were packed into Data Transfer Objects (DTO), for which the structure of figure 3.5 was agreed. To extend the objects of its role as simple data containers and thus to improve local functionality structurally similar domain objects extending the search result objects are created by TT, when a search was successful. The creation and transmission of the objects finally work fine, as long as only data stored in SENTRE is affected. But as we (during the development of the tool) discovered the need for additional information and especially for data regarding the corresponding CUTs we had to find a way to integrate the parental information from the Merobase into our objects. This requirement was initiated by the wish to have the opportunity to reuse the source code of the original CUT in the Method Proposals and thus, if desired, to be able to display and to insert the code into the local CUT. To avoid huge redundancy we bypassed the option of storing the whole project structure in the new database by agreeing on only storing the source code of the respective CUT in the corresponding object. Although this enrichment was planned, this goal has not been achievable within the course of this work. The main reasons for this shortcoming have been the requirements to and thus the complexity of an appropriate parser and the effort this entails. Therefore an alternative index structure has been created, consisting of value pairs of tests and potentially associated CUTs. This index is predicated on simple investigations of imports, calls, and variable declarations in the specific test. In the current implementation, the database request therefore includes two steps. In the first step the complex object structure stored in SENTRE (and displayed in fig. 3.5) is obtained while in step two the associated cuts are requested. Internally, the encapsulated SENTRE data objects from step one are subsequently enriched with the information from step two. Finally therewith TT is able to provide *Method Proposals* offering the application of the original method body of the CUT from the database to the locally tested component. Summarized, TT hence receives all information needed for the generation of the proposals and the exposing the results. The structure here supports TT to easily find and utilize the required information. For example, if all assertions shall be found, an iteration over the various classes and methods is sufficient to retrieve the *AssertionStatement*s for the calculation of the Assertion Proposals (3.5.1). Likewise this structure finally enables the straightforward calculation of the Method Proposals (3.5.2) as well as of the Exception Proposals (3.5.3).

4.3 Architecture

In the development of our prototypical application we strictly pursued a layered architecture to facilitate further improvement and easier maintainability. We thus created distinct layers - *Application*, *Domain*, and *UI* - in our implementation what is reflected in the package structure. The package names consequently correspond to the respective responsibility: we placed every processing class in the *Application layer*, all functionality for user interaction in the *UI layer*, and the data objects in the *Domain layer*. Additionally we added the packages *Actions*, *Util*, and *Tests*. The first of these additional aggregations contains the actions for Eclipse, such as the *ActivationAction* which enables TT to run. The Tests package contains the tests for our program and is located in the main source folder as we abstained to split the tests to the specific sub packages. In addition, a Util package is available in almost every layer containing specific utility functionality as well as static methods and constants for the particular package. Since the search process represents the only transmission process in our application we omitted a further differentiation of a network/transport layer and the search was placed in the domain layer for convenience.

4.3.1 Application Layer

The application layer contains all the classes and procedures needed for any calculation purposes and is thus the layer controlling the work flow. Figure 4.1 depicts[3] the corresponding package. With the activation of TT the plugin registers an *BackgroundAgent* which observes the user behavior on the basis of performed changes in the currently opened editor. At this juncture it reports to the main class of the plugin: the central singleton class *TestTenderer*. This unit acts as the central steering and control instances. It it responsible for the whole work flow and the integrity of the system. It even contains the proposal information required by the *ProposalComputer*, which is registered in and thus questioned for proposals by Eclipse. For this purpose this servant needs to implement the *org.eclipse.jdt.ui.text.java.IJavaCompletionProposalComputer* interface to interact with the IDE. The *ProposalComputer* itself has little independent functionality and basically only acts as an "interface" for the recommendation system of Eclipse. The reason for this limitation of functionality to a simple filter handler lies in the fact that the proposals are calculated in the background. Thus this class only acts as an information dealer, not as a processor as the name may suggest. The proposal generation and coverage calculation tasks are initiated by the *TestTenderer* singleton and accomplished with the help of the *ProposalGenerator* job, respectively the *CoverageGenerator* class. On request the different proposals are generated by the *ProposalGenerator* which extends the *org.eclipse.core.runtime.jobs.Job* and therewith inherits the ability to contribute to Eclipse's progress bar. This job additionally acts as managing instance for the coverage calculating *CoverageGenerator*s. These information creators enrich the previously generated domain layer objects by speculative coverage information. As this is accomplished asynchronously, the *CoverageGenerator* implements the *java.lang.Runnable* interface.

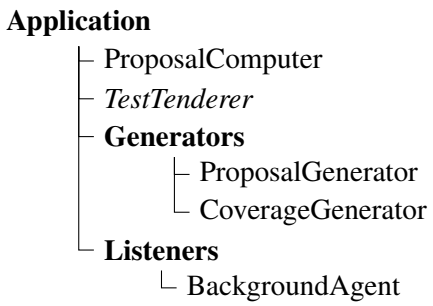


Figure 4.1: Application layer structure

4.3.2 Domain Layer

The domain layer contains all the elements of the domain, i.e. all classes that actually represent "real world objects" as given in Figure 4.2. The most important and most intensively used objects are the *Test* and the *ComponentUnderTest*. Both are linked with each other in a $1 : n$ relationship, that means that one *Test* may hold multiple *ComponentUnderTest*, but every CUT object belongs to only one corresponding test. Both classes provide access to distinct information related to the particular object. Thus, the *Test* besides simple information like the simple name or the Fully Qualified Domain Name (FQDN) hosts knowledge about its associated components, what kind of test it represents, the result of the last JUnit test run, and of course the source code that represents this object. The latter is, however, is provided in the form of a *org.eclipse.jdt.core.dom.CompilationUnit* with which the instances are also constructed. The same construction requirements are used for the *ComponentUnderTest* objects, which are additionally passed the parent *Test* instance to established the mentioned concatenation. A special circumstance

[3] for the purpose of clearer presentation some helper classes and packages have been removed

is that these objects construct themselves. If a test is instantiated with a *CompilationUnit* as parameter, it examines its own source code and enriches itself with information. This task is accomplished with the help of the *TestParser* (see section 4.5.2 for details). Furthermore, it examines the source for possible CUT candidates and creates - if the investigation was successful - appropriate *ComponentUnderTest* objects. The dependencies on objects in the project are resolved within the *TestParser*, which procures the *CompilationUnit*s of the candidates on the basis of the given *JavaProject*. The so created *ComponentUnderTest* objects show a similar behavior on creation and examine and enrich *themselves* with necessary and useful information. Therefore they come up, in addition to however trivial information (such as simple name or FQDN), with knowledge of the associated file, the corresponding search result, the name of its own methods, the achieved basic coverage from the test, or as the case may be the variable that was assigned to this component in the associated *Test*. Because not all bases for information retrieval are available upon creation, some are amended later. This concerns for example the coverage information, which can of course only be stored after it has been calculated. In addition, this also applies to the search result, which understandably can only be assigned, when a suitable search is performed. Another important participant in the work flow is thus the *Search* class in the corresponding *Search* package, which holds functionality for the search itself and operations for the construction of the desired data structure derived from the search results structure. After the execution of a search the *Search* object therefore wraps all the objects in the result set to instrumented counterparts for advanced processing capabilities. The classes starting with an "i" thus represent those "instrumented" correspondents of the objects received from the search server and are consequently also located in the *Search* package. They provide enriched functionality and are tailored to TT's specific needs. And they hereby contain all information needed for the entire processing activities. In addition to the wrapping functionality the *Search* class holds several information derived from the search process, e.g whether the search was successful and the duration of the search process, and provides different connection points for the retrieval of the result or subordinated information. A further characteristic is additionally worth mentioning. Given that the *Search* objects already interact with the *Test*s instances to perform a distinct search for each of their CUTs, they simultaneously submit the search result to the corresponding *ComponentUnderTest*. Last but not least the *Domain* layer contains the *Proposals* package, which again contains the different types of *Proposal*, which they all extend. The super class implements three interfaces: (1) the *org.eclipse.jdt.ui.text.java.IJavaCompletionProposal* interface, which is demanded by the Proposal Computer to enable integration, utilization and display of the recommendations, (2) the *org.eclipse.jface.text.contentassist.ICompletionProposalExtension2* interface, which adds the opportunity to handle trigger characters with modifiers and to visually indicate the selection of the proposal, and (3) the *org.eclipse.jface.text.contentassist.ICompletionProposalExtension6* interface, which allows styled ranges in the display string.In conjunction with the necessity of the IDE for the implementation of the first element, this also provides the general basic functionality. Access points for the string to display, for the image to use, for the relevance (which determines the rank), for additional information to display in a supportive information frame, or for contextual information are herewith provided. In addition this interface encourages to specify the insertion functionality with the demand for a specific *apply* method. For convenience additional methods are included in this *Proposal* class, which provide secondary information e.g. whether the *Test* with the specific proposal passed the JUnit test run or whether its particular coverage is improving or worsening compared to a given *BasicProposal*. The classes *AssertionProposal*, *MethodProposal*, and *MethodProposal* inherit this functionality as they are all used for the recommendation system. The class *BasicCoverage* in turn extends the *AssertionProposal* class for convenience, as this already provides some extra setter and helper methods. In the *ExceptionProposal* the only deviation is a method to obtain the underlying *iExceptionTest*. Besides these mentioned deviations the subtypes only *change* the functionality of the superclass *Proposal*, but do not *extend* it beyond that. The *Context* class is an extension of the *org.eclipse.jdt.ui.text.java.ContentAssistInvocationContext* class, which is passed by the *ProposalComputer* as container for contextual information. This information includes the caret position at which the recommendation process was triggered, the prefix the user already typed

in before activation, and the document the proposal request was started in. This information usually has to be thoroughly extracted. The *Context* class simplifies this process with the provision of automated methodology. As the *AssertionProposal*s demand the underlying information for a correct insertion behavior the *Proposal*s are contemporaneously with the triggering of the *ProposalComputer* enriched with a corresponding *Context* object.

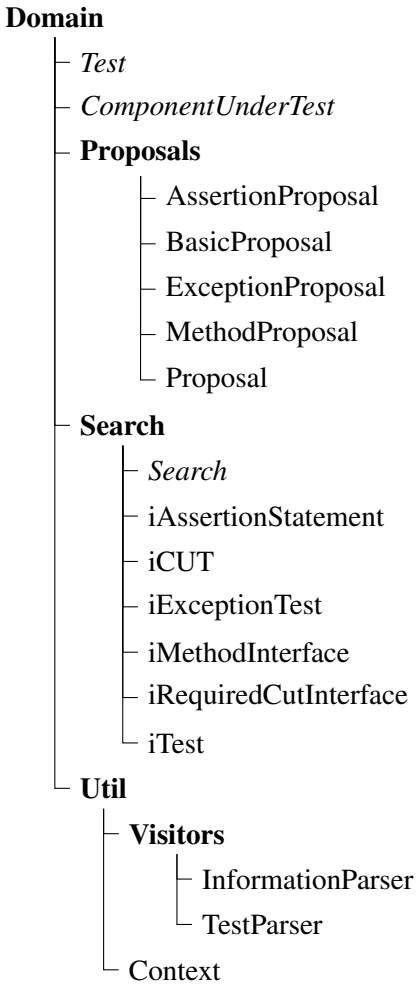


Figure 4.2: Domain Layer Structure

4.3.3 User Interface Layer

The User Interface (UI) layer contains all elements, which contribute to or can be accessed by the Eclipse workbench (see Figure 4.3). The *Activate* class implements the *org.eclipse.ui.IWorkbenchWindowActionDelegate* interface to be addressable by Eclipse. This *Action* is linked to a button in the Eclipse toolbar and consequently its *run* method is invoked by a click on the according button. Its functionality is comparatively trivial. It just calls an *activate* method in the *TestTenderer* singleton and passes whether the toggle button is selected or not. With that information TT starts or stops the *BackgroundAgent*. The UI *Listener* package is basically only supporting the different components in the *Views* package. It just comprises two interfaces and two enumerations. As the naming indicates the *CoverageEventListener* interface is implemented by the *CoverageView* and aims to deal with *CoverageEvent*s, whereas the *SearchEventListener* interface is implemented by the *SearchView* and reacts to *SearchEvent*s. Both

interfaces extend the *org.eclipse.ui.IPartListener2* interface to enable activity surveillance of the views, since these have registered with the workbench for this purpose. Additionally both listeners require a *notify* method which handle the different event types in the corresponding view. Although each view is only registered once in the *TestTenderer* instance and with Eclipse, the decision to appeal the views with listeners was taken, however, for the purpose of extensibility. The registration of the views is enabled by the extension of the *org.eclipse.ui.part.ViewPart* class, which provides functionality needed by Eclipse. These features include a method for the creation of the object or the part control, and the opportunity to focus the contribution in the workbench. With the additional implementation of the above interfaces the views beyond that become available for the managing *TestTenderer* instance and therewith even for the subordinated objects. Besides the two *View*s the package of same denomination contains two layout classes, which both serve the *SearchView* with the layout of the used *TableTree*. The *TableContentProvider* is responsible for the representation of the "table" part of the *TableTree*, whereas in turn the *TreeContentProvider* manages the display of the "tree" component. In order to enable interaction with the *org.eclipse.jface.viewers.TreeViewer.TreeViewer* in the *SearchView* both have to implement a distinct interface. In case of the *TableContentProvider* this is the *org.eclipse.jface.viewers.ITableLabelProvider* interface, for the *TreeContentProvider* the implementation of the *org.eclipse.jface.viewers.ITreeContentProvider* interface is sufficient. Both interfaces demand functionality for a structured management of the content, such as the text to display in a distinct column of the table, or the image to display for a particular node of the tree. The *Preferences* package contains all elements needed for the contribution to Eclipse's preference menu. More precisely, TT subscribes two items to the preferences. Firstly, the generically named *Preferences* page with basic adjustments for the search server and a bug reporting functionality and secondly, the *CoveragePreferences* providing adjustment capabilities for the different coverage weights are submitted. Both classes do not differ in functionality, but only in content. They implement the *org.eclipse.ui.IWorkbenchPreferencePage* interface and are therewith integrable. In addition, both extend the *org.eclipse.jface.preference.FieldEditorPreferencePage* class and therewith indicate the hosting of field editors. Simultaneously with that extension these pages acquire the ability to provide specialized behavior for application and default value restoring. These initial default values are held by the *PreferenceInitializer* class, an extension of the *org.eclipse.core.runtime.preferences.AbstractPreferenceInitializer*. It initializes the different preferred values in the *org.eclipse.jface.preference.IPreferenceStore* provided by the plugin *Activator* class with the particular defined value. For a structured assignment and retrieval process, the *PreferenceConstants* class provides several identifiers, which afford a fail safe variable assignment. In the same way the default values are restored, if requested.

4.3.4 Utilities & other classes

Furthermore, still two classes exist in the architecture, that are subordinated to any other package due to their functionality or meaning (see Figure 4.4). The most important item is the *Activator* class which represents the interlink between TT and Eclipse. As an extension of the *org.eclipse.ui.plugin.AbstractUIPlugin* class it serves a basic initiator of the plugin and starts TT with the help of the *activate* method in the *TestTenderer* class. The second "homeless" class is the *BugReporter* which was created and integrated for bug reporting purposes. This reporter manages the flow of automatically created bug reports to a dedicated *Bugzilla* database. Finally, to provide access to globally used methods and objects a central *Commons* class is located in the *Util* package. This class e.g. provides indices for the different coverage criteria, which are internally usually stored in a *double* array, simple calculation functionality for the conversion of *double* values d_j with $0 \leq d_j \leq 1$ to an *integer* percent representation i_j with $0 \leq i_j \leq 100$, or retrieval methods for globally used icons and constants.

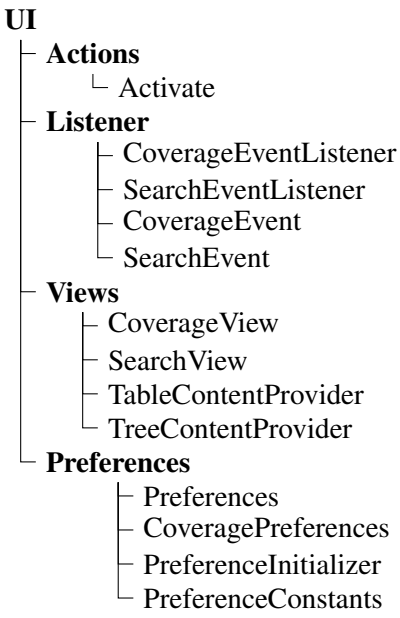


Figure 4.3: User Interface layer structure

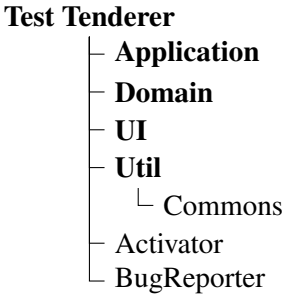


Figure 4.4: Utilities layer structure

4.4 Work Flow

Usually Eclipse hinders plugins to automatically start on launch of the framework to prevent itself e.g. from memory leaks, performance deficiency, and a too long startup time. This modus operandi is called *lazy start* mechanism. But since the support of our plugin is very promising we decided to activate it by default. Nevertheless, our plugin observes nearly any code change and is thus very "attentive", so the opportunity to (at least temporarily) disable the observation seemed reasonable on the other hand. Accordingly TT may be deactivated via a button we added to Eclipse's main command bar. However, on start-up TT registers an *BackgroundAgent* with the *org.eclipse.jdt.core.JavaCore* to enable an appropriate surveillance of the editors. This listener consequently reports to the *TestTenderer* singleton class and informs about any changes in the editor with the reporting of an *org.eclipse.jdt.core.ElementChangedEvent*. *TestTenderer* subsequently triggers several tasks depending on the actual state of the system. Primarily this is maintenance, the surveillance of the system and the project the user is working on, the execution of a search, and the generation, respectively the update, of proposals. Appendix A provides a complete overview of the work flow. In the following the major activities are presented in detail.

4.4.1 Surveillance & Test preparation

On every change event TT inspects the current state of the system and evaluates the current work of the user. Based on this evaluation TT decides which actions to perform. For this purpose first of all needs to be clarified, whether the user is actually working on a test. To access the information, TT initially creates a *Test* object based on the current editor content. On creation, this object examines the underlying source code for evidence of a test. Furthermore, the *Test* class continues self-investigation for corresponding *ComponentUnderTest*s. If the resource under inspection is not considered a test, no further activities are started. In this case the *SearchView* as well as the *CoverageView* are notified with the corresponding "NO_TEST" event. These Views subsequently update on their own and display corresponding messages. If the artifact but is a test, TT continues with the inspection of the *Test*'s signature. This information consists of the name of the *Test* and the interface signature of each *ComponentUnderTest* and thus serves as identifier. If this signature has not changed since the last evaluation the test is considered not modified and consequently nothing is done. However, if the newly calculated test identifier changed, it is subsequently compared to other signatures TT previously inspected and for which a search has already been conducted and stored. If such a previously saved search result was found, this data set is re-initiated and consequently reused. Otherwise a new search is performed. Figure 4.5 provides a detailed insight into these activities.

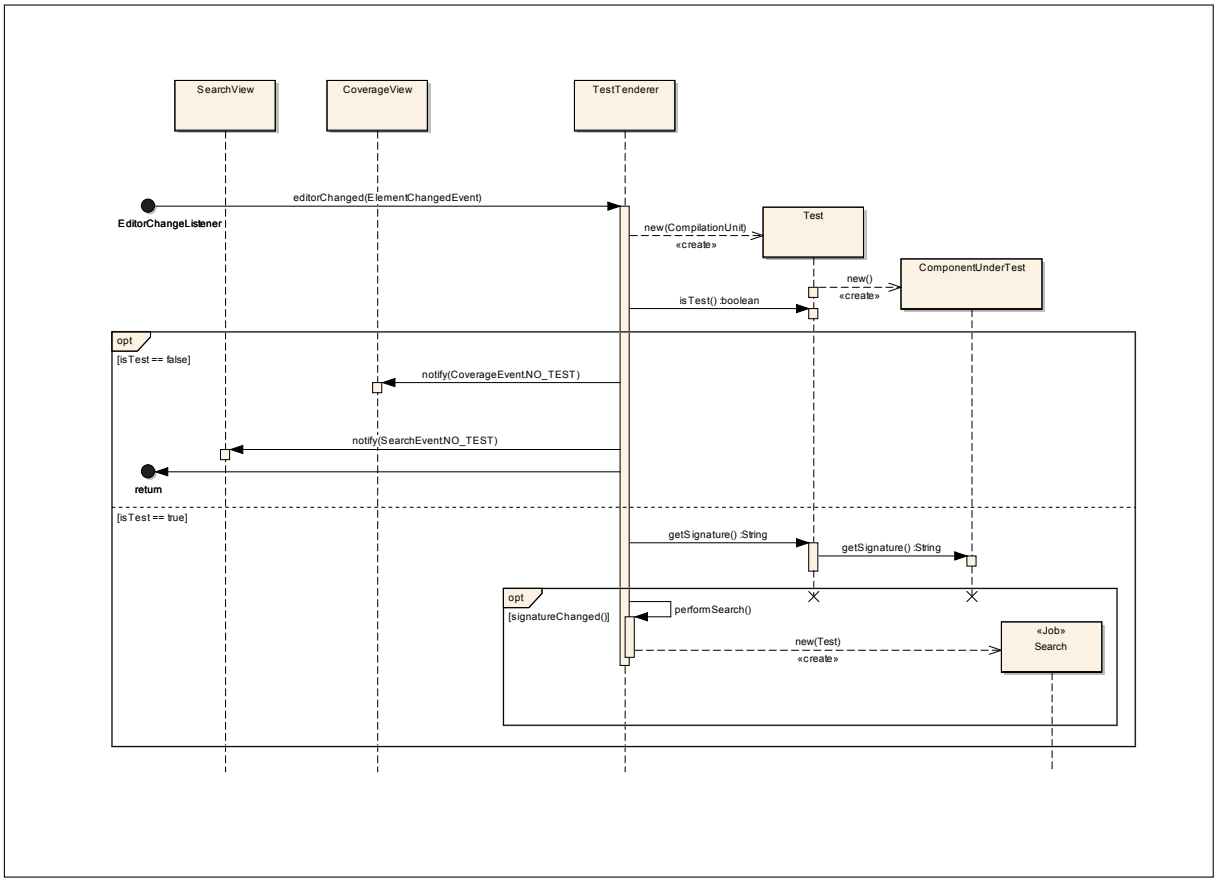


Figure 4.5: Test Preparation

4.4.2 Performing a search

In order to avoid unnecessary network traffic a search is only conducted when the signature of the *Test* changes and if the "new" signature does not correspond to any previously performed and stored *Search*. If these conditions are met, TT creates a new *Search* and passes in the new *Test* object. The *Search* job subsequently requests all *ComponentUnderTest* objects from the *Test* and starts a search for the particular component. The query here is constructed on the basis of the signature of the concerning *ComponentUnderTest*. When this lookup process is finished and the server returns the result set, the corresponding objects are enriched with the search result. Finally, the *Search* finishes with the provision of a *SearchEvent* to the *TestTenderer* instance. On that basis TT informs the *SearchView*, which in turn updates depending on the type of the reported *SearchEvent*. Figure 4.6 shows the entire decision and processing sequence.

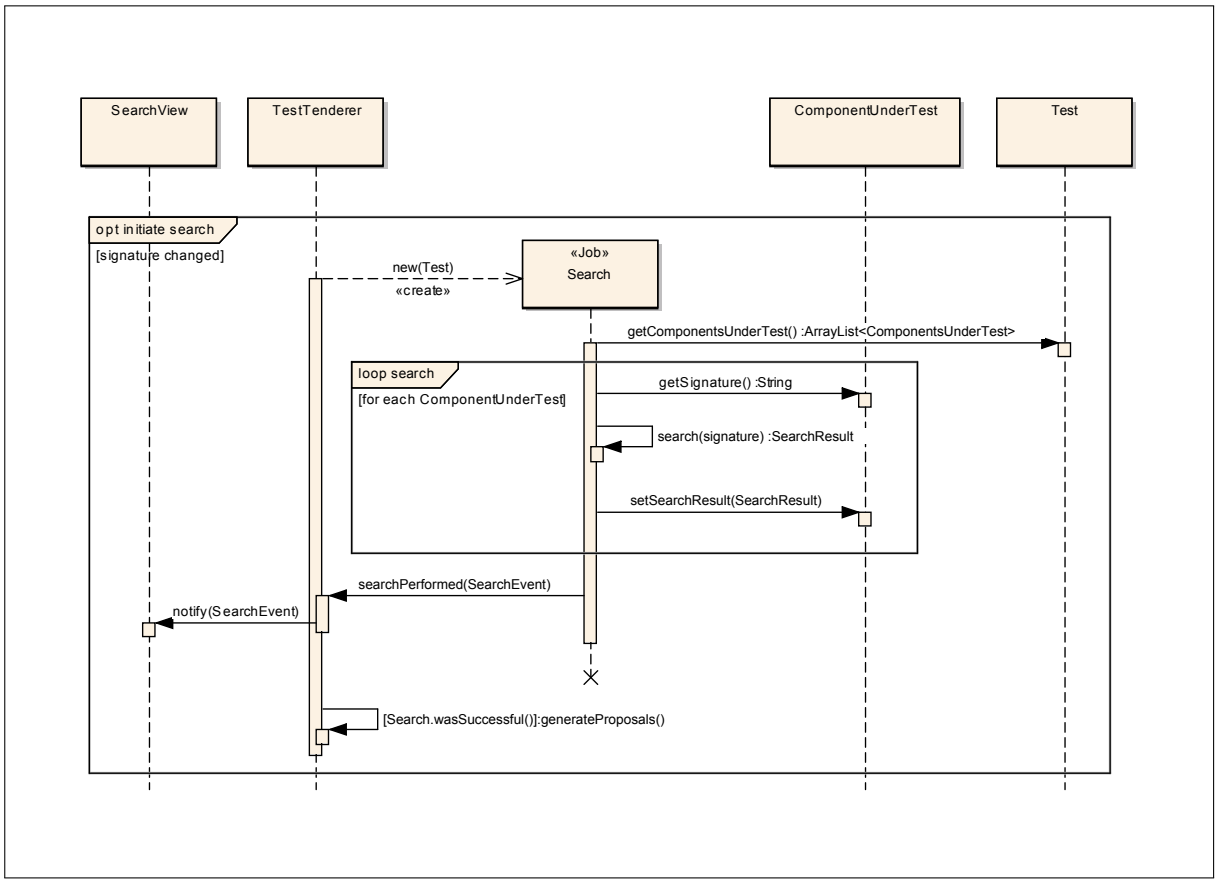


Figure 4.6: Search

4.4.3 Generating proposals & Calculating coverage

The generation of the proposals and the enrichment of the Proposal objects with the respective target coverage information is the most complex task. This step is to be chronologically classified after the search process. The proposal *creation* process is only triggered, if the search has been successful. Otherwise the proposals, if any exist yet, are updated on the target coverage information based on the current test. It should be mentioned again that the entire update process is interrupted if the current code under development is not considered a test or if the signature has not changed. However, we assume that the source is recognized a test and the search was successful. In that case TT starts with the calculation of the basic

coverage by the use of the *CoverageGenerator*. This generator subsequently calculates the coverage of the *Test* as-is and enriches the also passed in *BasicProposal*. In addition, the *CoverageGenerator* assigns each *ComponentUnderTest* discovered in the *Test* object their particular coverage. This information is subsequently used to notify the *CoverageView* and thus to display the coverage information. The basic coverage information is also used for the proposals as the coverage *improvement* the proposals achieve is also calculated and displayed in the Proposal Computer. After the calculation of this basic reference values TT generates the different *Proposal*s with the help of the *ProposalGenerator* job. This object is responsible for and manages the whole proposal creation and coverage calculation process. Based on the provided *Search* and the list of *ComponentUnderTest* it assembles the different assertion statements and therewith creates the *AssertionProposal*s. Furthermore the tested methods found in the *Search* are compared to the methods present in the *ComponentUnderTest* objects. If a method is found for that on the one hand tests were found, but on the other hand if such a method does not exist in the *ComponentUnderTest*, a new *MethodProposal* is created, which provides the source code of the corresponding database CUT method. With that information these *MethodProposal*s provide functionality for the inspection and insertion into the local CUT. The same is done with the exception tests found in the *Search*, with the only difference, that they address the test and not CUT. Arising objects on this are *ExceptionProposals*. Once the proposal generation is completed, the *ProposalGenerator* starts with the calculation of the coverage. For this, the *java.util.concurrent.ExecutorService* is used that takes the thread management based on a pre-set value. In our implementation we set this value to $max(cores, 1)$ to ensure that constantly enough computing power, i.e. at least one core, for the IDE is exclusively available. In other words, TT uses the maximum available power for the calculation, but ensures that there is always enough computing power for main operations. The *Executor* then automatically starts a *CoverageGenerator* and causes the coverage calculation for each passed proposal. As this task is performed asynchronously the *CoverageGenerator* instances report to the *ProposalGenerator* via the *done* method as soon as they are finished with the coverage calculation. Within this computation process the *Proposal* objects are already enriched with the distinct coverage information, what enables the parent job to only request for the proposals itself and prevents from additional computing effort. For each *Proposal* obtained in such a way the synchronized *addProposal* in *TestTenderer* is invoked and the finished proposal is handed over. With this last step now all required proposals are available in the central *TestTenderer* instance in order to pass on the request of the *ProposalComputer*. Figure 4.7 illustrates the whole proposal generation and coverage calculation process.

4.5 Visitor Patterns

The need for parsing methodology applies to almost all processes of the work flow. In the phase of *Preparation* the source standing for the *Test* needs to be investigated and the CUTs have to be identified to create the *ComponentUnderTest* instances. Besides that specific data, nonetheless also simpler pieces of information (such as the name or the FQDN) are needed for further processing. However, not only do the local components need to be examined in a highly structured way, but the search results need to reveal the required information only by specific investigation. In the *Search* phase the results, especially the received source codes, have to be thoroughly investigated for the identification of corresponding tested components. Underlying this issue is the fact that currently two independent web services have to be queried, one each for the tests and for the CUTs, and thus the sources of the tested components are not automatically available. Finally also the provision of method proposals makes use of these structured investigations, as the method to apply is only given by its source code and the creation of an appropriate *Method* object is rather complicated. For the purpose of parsing we make use of the Abstract Syntax Tree (AST) framework of the Eclipse IDE. It intends to map plain Java source code in a tree form. This shape is reflected in the *org.eclipse.jdt.core.dom.CompilationUnit*s, which are widely used as joint data type in TT. These again are subtypes of the *org.eclipse.jdt.core.dom.ASTNode* class. The *ASTNode* in turn is the abstract superclass of all AST node types. For the structured work off of the different nodes

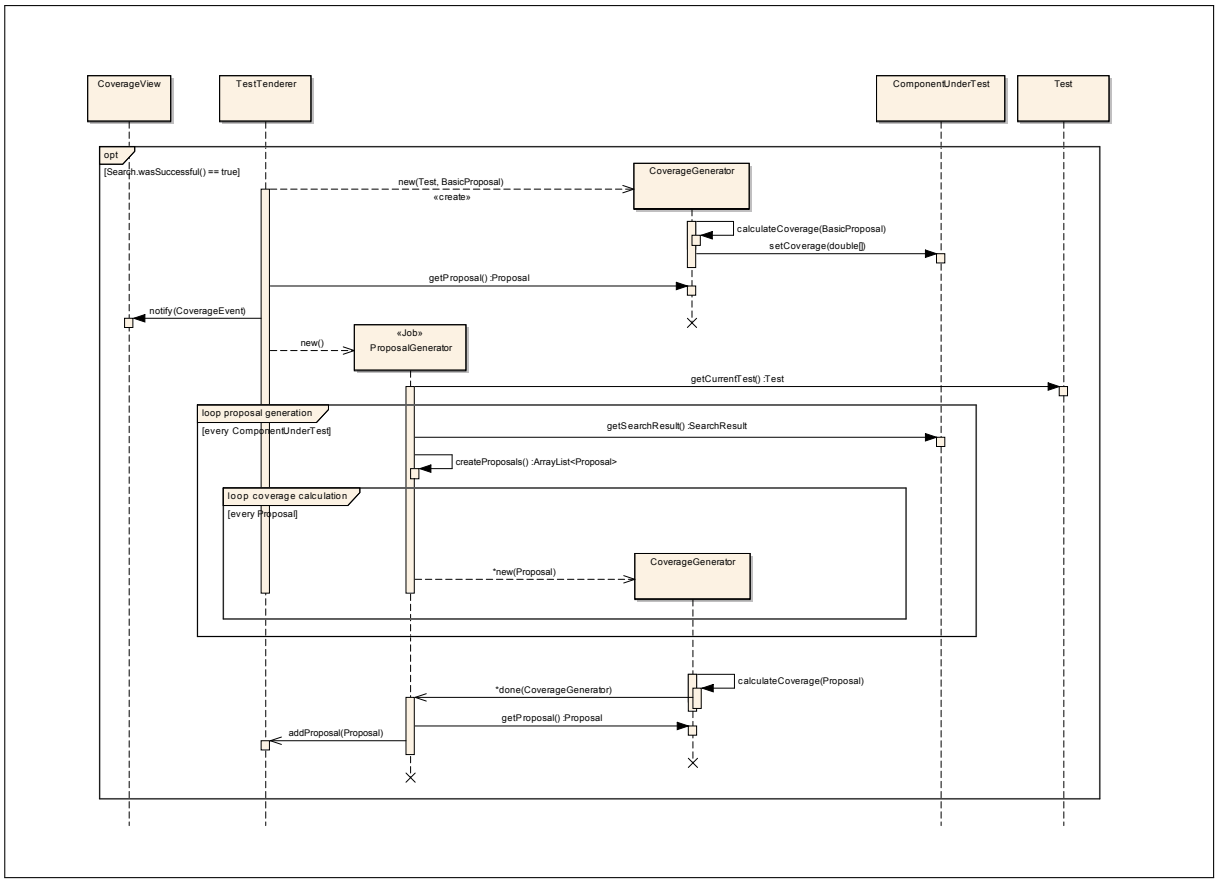


Figure 4.7: Proposals & Coverage Generation

in the AST the *org.eclipse.jdt.core.dom* package provides an *ASTVisitor* class, which exactly fulfills this demands: applied to any *ASTNode* derivative it starts at the root of the tree and sequentially "visits" all leaves of the particular node. This class utilizes the principle of *Visitor Design Pattern* [32] for each node of the AST. With the help of a specially implemented Visitor that extends the *ASTVisitor* it is thus possible to inspect, extract, or even to modify any specific information of the associated AST. In an early approach several visitors for each specific *type of information* were used. Therefore, a *NameParser*, a *CUTParser*, a *TestParser*, a *FQDNParser* plus a few more existed. In the current implementation the amount of this parsers has been clearly reduced and they have been aligned with the *field of information* they are responsible for. To make this clearer, currently an *InformationParser* is used, which investigates a given applicant and subsequently provides *every* required unstructured information, such as the name or the FQDN. The *TestParser* now holds *all* functionality for the *whole* preparation process, i.e. the recognition of a test, its test type, and its tested components.

4.5.1 InformationParser

This *ASTVisitor* extension acts as central information hoarder for different classes, in different processes and in different contexts. This is mainly because it is able to examine *any* given *CompilationUnit* for different required unstructured information. Although the information gathered by this parser affects multiple object creation and other general work flow activities, it basically does not provide functionality for this purpose itself - clearly contrasting with the *TestParser* (4.5.2).

Competences

Its range of information on offer comprises in particular:

- the simple type name
- the Fully Qualified Domain Name
- a list of the contained methods
  (in form of *org.eclipse.jdt.core.dom.MethodDeclaration*s)
- a list of the names of the methods
- the signature of the particular *ASTNode*

As this parser is basically applicable to any *ASTNode*, it is thus universally usable. However, the given AST *must contain* the desired information. For example, the FQDN may not be resolved for e.g. an *ASTNode*, that in fact is a *MethodDeclaration*, as it just does not contain the required information. That means that in this aspect a finer gradation leads to a lower information scope. Nevertheless, it is widely used to enrich existing objects, to compare components, and it is even marginally involved in the creation of objects in some cases.

Application areas

Specifically, this parser is inevitable for the UI as the displaying of any names and signatures base on its appropriate competences. In addition, the FQDN retrieval ability is widely used. This applies to the *Preparation*, as the *Test*s and *ComponentUnderTest*s are equipped with this information. Furthermore the *Proposal Generation* process is inevitably dependent on the *InformationParser*, since the *MethodProposal*s are created with the original CUT methods from the database.

Procedure

The *InformationParser* inherits *visit* methods for only three different types, but is nevertheless able to calculate and provide the above capabilities. Starting at the root of the AST the visit at an AST node of type *PackageDeclaration* is most likely the first opportunity of information retrieval. As this information is needed for the "assembly" of the FQDN it is thoroughly extracted and stored. By visiting the *TypeDeclaration*, the name is read out and stored for direct provisioning and for the compilation of the signature. Furthermore the name is needed for the construction of the FQDN. Finally, any visit at a *MethodDeclaration* AST node serves the purpose of method collecting and method name provisioning. Simply by this accumulated data all the information required can be calculated and assembled. Thus, the FQDN is a composition of the package information and the class name. The signature is an assembly of the type name and information extracted from the *MethodDeclaration*s, such as the type of the parameters and the return value, which is accessed by further detailed investigation. The lists of methods and corresponding method names are composed in the same step.

4.5.2 TestParser

This parser serves only one purpose, and the application is limited to only one class. Nevertheless, it is essential for the whole system, as it is inevitable for the recognition of the type, for the resolving of dependencies, and thus for the creation of *Test* objects.

Competences

In contrast to the *InformationParser* this class is not limited to the retrieval of data and thereby to a passive role in a creation process, but contains an own functionality for the generation of objects. Nevertheless, different information is provided, and the offering here is wide. Summarized, the *TestParser* provides the following functionality:

- evaluation, whether a *Test* object in fact *is* a test
- determination of the kind of test (JUnit 3 or JUnit 4)
- identification of accompanying CUTs
- compilation of a list of CUTs, respectively their FQDNs
  (for the ability to query the server accordingly)
- creation of the corresponding *ComponentUnderTest* objects

The use of this parser may be universal, since it depends on the pass of a *Test* object only for the creation of the *ComponentUnderTest* objects . As a consequence, this feature is not available in a generic usage scenario. All other information may be read from any *ASTNode*s. However, as already mentioned for the *InformationParser*, this can only be done if the desired information is contained in the AST.

Application areas

The field of application is limited to only two scenarios. First, the TestParser helps to create the *Test* objects themselves and to fill them with information, and consequently builds the *ComponentUnderTest* instances. Secondly, it is used to search for the FQDNs of corresponding CUT candidates within the source code of the tests contained in the search result. These identifiers are required to get the code of the candidates, since this information-gathering process is still split in two independent tasks and a second web service needs to be queried for this specific information. With a refinement of the main web services that fact is omitted in the future.

Procedure

For the determination whether the source is a test the parser observes the source for JUnit-specific substance. Furthermore it distinguishes the kind of test by independently searching for JUnit 3 plus JUnit 4 specific artifacts and derives two independent indicators. For this evaluation task the *visit*-methods for the *CompilationUnit* itself and for *MethodDeclaration*s are overridden. When visiting a *CompilationUnit* the *TestVisitor* is able to query the unit for its superclass. If this equals the *junit.framework.TestCase* we consider the test a JUnit 3 test and set the corresponding flag. Similarly, we used the visitor method for *MethodDeclaration*s to identify JUnit 4 tests. We therefore request the annotations for every visited method. If just one method contains an *org.junit.Test*-annotation the *Test* is identified as JUnit 4 equivalent. On the basis of this gathered information the test qualities as well as the test type is derived. For easy application and for quick check an extra variable is used representing an OR-conjunction of the both indicators. The distinction of the test type is necessary since JUnit 4 provides built-in testing, that means that test methods may reside in the tested component itself. In other words: since the CUT adheres its own test, the source of the corresponding class basically serves two roles, namely as base for a *ComponentUnderTest* and for its own *Test* object, and thus needs further processing. This necessity also applies to all other CUTs, which makes the identification mechanism extremely important. For the purpose of CUT discovery a total of three *visit* methods are used. Not only for the imports, respectively the *org.eclipse.jdt.core.dom.ImportDeclaration*s, and the variable declarations, in form of *org.eclipse.jdt.core.dom.VariableDeclarationFragment*s, but also for instructions, i.e. the *org.eclipse.jdt.core.dom.ExpressionStatement*s, these *visit* methods are enriched with functionality.

As the usage of these methods only serves information-gathering purposes, the creation of the objects is thus to be done in an additional task. The identification process comprises four steps:

1. **Examination of the imports**
   Here, all imported objects are cached, because at this point it cannot be decided which import may possibly belong to a CUT candidate

2. **Investigation of the declared variables**
   This is not only necessary to estimate which imports do not refer to a CUT, but also to know by which variable the CUT may be requested. This particular information is inevitable for the *AssertionProposals*, as they have to construct method calls within the assertion statements on that basis.

3. **Analysis of the instruction statements**
   As the *visit(ExpressionStatement)* method is called for *any* instruction, initially those statements have to be filtered out, that are not representing a JUnit Assert-statement according to the pattern shown in figure 3.11. This goal is accomplished with the help of regular expressions, respectively the combination of a *java.util.regex.Pattern* and a *java.util.regex.Matcher*. However, if a valid assertion statement is found, it is further investigated. The tested method as well as the variable corresponding to the class, that contains the method, are identified. If the variable was already found in the previous steps, nothing more is done. But if this identifier has not been added so far, it joins the list of pointers for candidates. At this point the collected import statements are purged. Imports, that have no corresponding variable declaration in the source code and are not represented in any assertion statement, are assumed not belonging to a CUT candidate.

4. **Creation of the *ComponentUnderTest* objects**
   Finally the *ComponentUnderTest* objects are to be created. For this purpose the information from the previous steps are processed in reverse order. They are gradually dissolved to ultimately have pairs of the instantiated variables and the FQDN of the corresponding CUT. The FQDN is either resolved from the imports, or from the package declaration, depending on where the corresponding source is located. The FQDN is used to allocate just this corresponding component. If the lookup was successful, the candidate's *CompilationUnit* is used to construct the *ComponentUnderTest* object. Finally each of those newly created objects is enriched with the information of which variable is used in the test to address this particular component, rather than to invoke their methods.

4.6 Summary

Although overall a harmonious tool has been created, the integration of the various functionality was not always easy. Especially the joining of a search functionality with a speculative analysis, JUnit and beyond that to integrate this all in one Eclipse plugin was a demanding task. For this purpose, parsing functionality became a central aspects. In addition, performance-issues had to be solved, because the system is intended to run in the background and to not affect the user. Finally, a major task was also to present the results to the user in a clear way. Even if the tool is running stably and meets the expectations, there are always ways for improvement. The object structure of the search results was discussed extensively, and there is surely room for improvement in the future. Especially the fact that currently two different web services need to be addressed in order to get the search result and the sources of tests and CUTS is not ideal. An enhancement of the parsing mechanisms, both for the local classes and the database contents would thus increase performance. However, to solve these issues it may be necessary to leave the Eclipse IDE and the AST framework and to create the entire parsing methodology from scratch.

CHAPTER 5

Evaluation & Related Work

Since no existing recommendation tools focus on the Reuse of tests [21], an evaluation of the tool developed in this work against other implementations is not feasible. However, a comparison against the basic principles of Reuse and Testing is possible and consequently conducted in the following. In addition, the implemented tool Test Tenderer (TT) is evaluated against a classical, unassisted Reuse approach as illustrated in figure 3.1 in section 3.1.1. Furthermore general requirements regarding system performance and bandwidth efficiency are discussed in this second section.

5.1 Theoretical Evaluation

In the following TT will be evaluated against the demands, requirements, and opportunities of the underlying theoretical foundations. For this purpose already in chapter 2 implications for a goal oriented implementation were derived. This section thus discusses the degree of realization of these suggestions in est Tenderer.

5.1.1 Testing

The requirements of 2.1 could be widely met. However, several implications could be derived to support usability and ease of use. For the purpose of maximal support of the testing process, these implications had to be realized.

+ The demand for the support of Unit Testing could be achieved by the integration of TT into the Eclipse framework and furthermore the integration of JUnit into TT. As basically JUnit supports a standalone integration into Eclipse, this requirement could have been addressed by the JUnit plugin only. However, TT is able to provide live information about the test result with the help of its Coverage View and therewith supports continuous testing as claimed in [21]. For this purpose, the underlying test is autonomously executed in the background and the JUnit result is extracted and subsequently displayed.

+ In addition, several coverage measures indicate the quality of the test. The criteria *Statement Coverage*, *Method Coverage*, *Line Coverage*, *Branch Coverage*, and *Complexity Coverage* serve the purpose of indicating the quality. They hereby provide the user with rich instant feedback about recent testing effort. An additional criterion, *Mixed Coverage*, could be created to support the application of different weights for the standalone coverage criteria.

+ Although the calculation of the basic coverage and the actual result of the test is processed autonomously, the user retains full control over any changes to the test. The proposals offered by TT are given with full source code and either displayed in the Proposal Computer or in the Search View,

depending on the actual usage scenario. In the View, complete test cases can be inspected for the purpose of manual assessment. In the Proposal Computer, the different proposals also provide insight into their internal structure. This is not only true for the Assertion Proposals, but also for the Method Proposals and the Exception Proposals.

Besides these valuable implementations, however, some promising approaches remained unattained.

– Unfortunately, very promising approaches like *Data Flow Testing* could not be implemented. This is mainly due to the fine granularity of the applied parser, which was just designed to support a general applicability of the information extracted from the Reuse candidates in the database. In addition, no coverage measurement tool could be identified, which supports according criteria.

– The promising innovations introduced with JUnit 4.5 remain unattained. This is primarily because the database contains almost only JUnit 3 Tests. For example, annotated test methods could give hints to the tested component with their method name, which would ease the parsing process.

– Some coverage criteria could not be applied, such as the valuable *Prime Path Coverage*. The reason for that is the limited functionality of open source coverage tools, which do only support distinct criteria. A means to an end could be the implementation of a high performance coverage measurement tool.

5.1.2 Reuse

Several tools already showed, that Reuse can be successfully conducted [9, 26], but only for the development of components. Tests to date have not been subject to tool supported Reuse [21]. Nevertheless, this work has shown that the Reuse of tests is feasible. However, for this purpose several obstacles had to be overcome in almost any aspect of the chosen approach to exploit the benefits of reuse discovered in 2.3.

+ As the implementation was planned for an *Ad-Hoc* scenario it had to be ensured, that a sufficient database provides TT with valuable data. Although the Merobase [10] contained several JUnit test cases these components could not be used one-to-one as they were not designed for Reuse. As a consequence the new database SENTRE was created containing only the *tests* from Merobase.

+ In addition to the newly created database the tests were thoroughly parsed with a very fine gradation. Only the basic information were taken from the source code to enable universal applicability. With that approach the domain-specific problems of Reuse could be overcome. This issue was especially demanded to enable *Vertical Reuse*.

+ However, due to the fine gradation of the parsing result, the information had to be briefly reassembled to be usable. Nevertheless this requirement provided the opportunity to simultaneously add adaptation feasibility by the use of a distinct result object structure. Hereby the need for adaptation could be satisfied, which was demanded by the Artifact Reuse and the Compositional Reuse aspect.

+ The Reuse of *Source Code* was considered truly labor-intense. In turn, however, it was implied that no preprocessing would be required. This may be true for manual source code inspection and reuse, but was not feasible in an automated scenario. However, as the only source of information was the source code of the tests in the database, a preprocessing step was separated. With the help of that upstream process, the information stored in the source code could be extracted time-independent. Hereby precious time could be saved for the recommendation process since only the adaptation and the display of the results had to be accomplished on demand.

Although this work showed that automated Reuse of tests is basically possible, several limitations exist:

– The code base for TT was built from public open-source repositories and thus the parsing process delivered very inconsistent results. While some of the acquired projects were very well tested and consequently various information could be extracted, other projects were not of any usable value. This may be caused by the open-source character of the artifacts, since it may be assumed, that unprotected intellectual property will not be of superior quality. A means to an end would here be the application of a quality filter or a purposeful collecting of components, which are explicitly designed for Reuse.

– Not every test in the database could actually be used. Due to partly complicated dependencies in the source projects some relations could not be resolved. In addition, dependent libraries which were used in the source project could not always be accessed. However, if they were accessible in the project, the internal structure could not be inspected due to the compiled form of libraries.

One issue is especially worth mentioning, since it constitutes a potential source of danger:

Although the interface or the name of a reused component may correspond to the demanded artifact, the functionality may largely differ from the intended or demanded behavior. As basically every code may be included in the executable body of the artifact, this represents a significant security risk. This issue is even aggravated, as the different artifacts may be included in the source code, either as Method Proposals in the tested component or as Exception Proposals in the test. For example, one method was identified which contained - for whatever reason - a system console output of a nursery rhyme. Although this statement was (even in the specific context) totally useless, it could be integrated into the local source code. However, this could also be a malicious command and thus cause serious damage to the system. Fortunately, and for that good reason the inspection of the code is possible at all times. To even mitigate this risk the verification of the code is demanded at all affected areas.

5.1.3 Speculative Analysis

The methodology of Speculative Analysis [27] was fully applied for the simulated calculation of the target coverage of the Assertion Proposals. As these proposals were to be displayed in the Eclipse Proposal Computer and a multidimensional presentation is not supported, the simulation depth was consequently set to one. However, a higher iteration depth was basically possible to further automate the recommendation process. As a consequence it would be possible to offer the opportunity to add several proposals at once and hereby maximize the target coverage. However, this would mean that the user would be restricted in his ability to inspect the code since a representation in the Proposal Computer would be reduced to a summary.

5.1.4 Ranking

As proposed in [30] the proposals were appropriately ranked to provide maximal benefit. Consequently, a ranking mechanism was implemented that calculates the rank dependent on the improvement of the respective proposal compared to the basic coverage of the test. As a result, the most improving proposals are assigned the highest rank and are thus shown on top of the proposal list. As a separate Proposal Computer was registered with the Eclipse IDE, the demand for context aware ranking could be neglected.

5.2 Practical Evaluation

This section evaluates the practical application of TT. For this purpose the automated support of TT is compared to an unassisted Reuse approach. Furthermore, it is investigated to what extent TT affects system performance and whether it hinders the general work flow.

5.2.1 Assisted Test Reuse

TT was designed from the beginning to reflect a classic Reuse approach, but beyond that to overcome the associated problems and thus to ease the Test Reuse process. To recap the scenario of Figure 3.1 the main steps for a Reuse of tests in a unassisted scenario comprise:

(1) repository lookup
(2) code inspection
(3) code insertion
(4) test validation
(5) result evaluation

Preliminary summarized, TT supports all of these steps. In order to get valuable results from code search engines users often need to "learn" a specific query language. Besides the effort that is caused by this requirement, pure web-based search interfaces are simply not attractive [21]. For this reason the repository lookup process was fully automated. TT inspects the source code of the component under test in the local workspace, constructs an appropriate query and finally questions the SENTRE database for Reuse candidates. The user does not need to interfere in this process and consequently does not have to deal with query formats, grammars, or other specific requirements demanded by the search engine. However, even the web interface of SENTRE provides advantages compared to other search engines, as the query structure reflects a simple type signature format. It is thus even manually possible to query the database without huge effort. The task of code inspection is also fully automated. TT investigates the results returned from SENTRE on their applicability for the current project. As the query was was built an the basis of the type signature and thus the functionality of the component under test is already reflected, the quality and the applicability of the search results is likely high. Nevertheless, TT speculatively simulates a usage and rejects inappropriate candidates in advance. Simultaneously TT calculates the impact of the insertion of appropriate artifacts on the coverage of the test under development. TT is consequently able to provide the user with specially tailored proposals of high value on demand. If the user decides to insert a specific proposal in his code TT automatically inserts the corresponding code at an appropriate position. Furthermore TT immediately starts a result evaluation in form of a basic coverage calculation to update the Coverage View, which again informs the users instantly about consequences of the code insertion. Besides the coverage information TT also indicates, whether the test failed or succeeded and hereby substitutes for a manual test validation. All these steps are fully automated and do thus not interfere the work flow.

5.2.2 System Performance

The goal of this work was to implement a *non-intrusive* system for the Reuse of Tests from remote repositories. Therefore the computing power consumption of TT had to be constantly paid attention to, especially sufficient overall system performance had to be guaranteed. This goal was thus consequently pursued. As a result and due to the high capacities of today's computer systems, the user usually does not even notice that background calculations are performed. But not only the computing power is basically limited, also the caused network traffic or insufficient bandwidth may result in unfavorable constraints.

+ As speculative test runs are not less time-consuming than manual test runs and for every proposal an individual run needs to be executed, the whole speculative calculation process was shifted to the background. Furthermore several variables were integrated to always ensure sufficient computing power for the main work flow.

+ In addition the whole speculative calculation process was strictly parallelised. Consequently every calculation and every test run is performed in a distinct thread. As a result the system is used with maximal but non-intrusive effort, but only for a very short period of time.

+ In addition, the number of parallel processes was limited to the number of available cores of the system. This ensures that tasks of high priority, such as the operation system or the user interface, do not have to suffer from lacking resources.

+ Although it could often be neglected in times of flat rates and high-speed Internet connections yet a buffer mechanism for the conducted searches was installed. This means that already performed searches are not repeated, but reused. For this purpose the various proposals have to be recalculated. However, since this step had to be done anyway for new searches this circumstance is not an additional time driver.

5.3 Summary

In total, the defined requirements could be widely met. Especially the automation of the Reuse process provides rich benefit, saves time and supports continuous testing. With a smart arrangement of the program components and the integration of context-sensitivity a resource efficient implementation could also be realized.

CHAPTER 6

Conclusion & Future Work

6.1 Conclusion

Although both *Testing* as well as *Reuse* are widely considered beneficial for Software development, *Test Reuse* has been largely neglected - so far. This is also reflected in a respective tool support for each particular discipline. However, none of them faces the challenge of Test Reuse itself. The reason for this may be the different peculiarities of each discipline, which result in a distinct number of non-trivial obstacles. A major constraint for Reuse has been identified in the lack of adequate databases, given that the few existing are typically difficult to use and only provide reasonably good results. Searches for tests basically provide great potential for delivering accurate results, since information about the behavior of the tested components is already given in the tests. The functional information about these components can then be considered as an ideal indicator for the estimation of similarity. However, an automated search process plays a crucial role in this process. Beyond that, an automated database lookup process also provides the opportunity to pre-process the demanded information and thus to automatically adapt the artifacts to the specific needs of the developer.

The developed plugin *Test Tenderer* is able to overcome these obstacles and to provide rich functionality for an automated Reuse of Tests. For this purpose, the Eclipse Integrated Development framework is utilized to fully automate the Test Reuse process. The type of support ranges from the inspection of the local project, an automated lookup for suitable artifacts in the supporting Test Reuse database SENTRE, and the verification of similarity and applicability to an adaptation of the received components to the project-specific characteristics. In addition, the plugin continuously provides various information about the test quality on the basis of six coverage measures. Furthermore, it attains a shift from an ex-post quality measurement to an ex-ante coverage provision through Speculative Analysis, which renders it capable of supporting the essential task of regression unit testing in a developer-friendly and non-intrusive way. As an interruption of the development work flow is to be strictly avoided, Test Tenderer is designed to work as a backer for unit test developers. Moreover, it instruments the popular unit testing companion JUnit to instantly provide feedback about recent testing effort to the developer. Test Tenderer does therefore not only provide additional features, but also simplifies existing processes. Continuous testing is herewith no longer just a concept as it is reflected lively in this tool.

Concluded, the developed plugin is truly capable of unifying the principles of *Reuse* and *Testing* in a developer-friendly, non-intrusive, and valuable way. In short: *Test Tenderer* pulls *Test Reuse* into practice.

6.2 Future work

Just because TT combines several areas of interest, there are also a variety of starting points for improvements and future work. Although the setup of the SENTRE database was not in the scope of this work, the development was cooperative. Not just because SENTRE's primary goal was to serve Test Tenderer, but also early experience during the development of the tool influenced the structure and the functionality of the repository. This is the reason why at this point, some recommendations are made in this direction. As the following pointers to future work concern various areas, they are given in bullet points to better distinct the different issues.

- Test Tenderer instruments the JaCoCo library for the calculation of the different coverages. Although this library was outstanding among the other examined tools, it basically focuses on control flow coverage criteria. However, especially data flow based coverage criteria (such as Prime Path Coverage) are known as beneficial, as they take state-based information into account. Consequently the quality of the testing measures could be enhanced by the integration of appropriate functionality. One possible solution would be the creation of an extensive quality assessment tool, providing various coverage measures and integration opportunities.

- The code base of SENTRE is built from open-source repositories and thus the result volume and quality clearly vary. In addition, several artifacts are not usable due to unresolvable dependencies of the source project. However, the whole project had to be parsed and even then it could not be guaranteed that the components are applicable. A means to this end would be a purposeful collection of components, explicitly designed for Reuse. As the open-source community is widespread and constantly enjoys greater popularity, an extensive crawling of appropriate repositories could thus reveal promising candidates.

- The existence of a valuable collection of reusable artifacts would also enable the establishment of a component-market mechanism as proposed in [11]. If fully functional components or tests could be provided, the range of usage could probably be extended. In addition, distinct marketing mechanisms could be applied as well. For example, the integration of user-based recommendations in the manner of "people who used this artifact also used..." proposals would create additional value.

- The execution of the code demanded by functional testing is possibly risky. As explained in section 5.1.2, some artifacts which contain partly "strange" functionality were identified in the database. Although in the particular example the statement triggered only an output to the system console, this might also be a harmful command. However, a static testing methodology is no alternative due to the lower quality of derivable information. It would thus be promising to find a way to protect the user from unwanted consequences. One possible solution might be an execution and a subsequent investigation in a "sandbox" before the artifact is actually used in the "real" workspace.

- As proposed by [5], the Reuse of higher-order objects is promising. However, the artifacts in this work are based on source code. The code artifacts have been thoroughly examined and very fine grained information has been extracted, which provides easier adaptability and fosters universal applicability. Nevertheless, the Reuse of whole tests, especially an impact and quality assessment, is still missing. With the integration of entire test cases it would thus be possible to implement further functionality, such as Search-Enhanced Testing [33].

- On the other hand the users may be currently confused by changing search results. This is due to the current search mechanism, which awaits the type signature of a component under test to find appropriate tests for this component. However, even on small signature changes repeated searches are initiated which are not always coherent. That means that in a subsequent search components discovered in a previous search are possibly not any longer included. Thus, a matching on method

level would increase the stability of the search results. In contrast to the previous demand, this would lead to a finer gradation. A way out of this predicament might be an automated construction mechanism, which assembles test cases on demand based on the requirements of the test under development.

- Currently a relaxed search mechanism ensures that more similar tests are found. For this purpose, the class name is relaxed in several steps and consequently also "similar" tests are returned as described in 3.3.2. However, synonymous class names are currently not considered. A means to this end could be the integration of a lexical database, such as WordNet [34].

- Currently, the tests in the database are investigated on their source code to predict their particular tested component. However, this is an error-prone task, since practically all classes instantiated in the test could be the desired artifact. A possible solution could be directly established in the tests. JUnit is considered the de facto standard testing framework for Java unit tests. Although several serviceable annotations found their way into this framework (cf. chapter 2.2) with the latest version, one particular annotation type is missing. An annotation pointing at the respective components under test would help to ease the identification of the correspondent artifacts. This would provide developers with additional structure and, moreover, would enable a quite easier identification of the desired component in the parsing process.

Besides these particular suggestions for the specific Test Reuse scenario, especially the area of Reuse demands larger improvements. Due to complex object structures, included libraries, reflection and injection practices, and many other implementation-specific details Reuse is often not applicable to generic scenarios. Many classes and implementations are just *too* specific for a different environment. Consequently, Test Reuse faces the same problems, as tests are mostly bound to just that specific components for specific contexts. Future work should thus primarily focus on enhancements in Reuse itself. A possible approach could for example be Reuse on method level, as the chances to find compatible components could be potentially higher. However, these changes and improvements had to be applied to the whole Reuse process including the databases, the search engines, and finally the tools.

Index

Bibliography

[1] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, January 2008.

[2] J.A. Whittaker. What is software testing? and why is it so hard? *IEEE Software*, 17(1):70–79, 2000.

[3] JUnit. http://junit.org/. visted 01.06.2013.

[4] Charles W. Krueger. Software reuse. *ACM Comput. Surv.*, 24(2):131–183, June 1992.

[5] R. Prieto-Diaz. Status report: software reusability. *IEEE Software*, 10(3):61–66, 1993.

[6] Waileung Ha, Hongyi Sun, and Min Xie. Reuse of embedded software in small and medium enterprises. In *2012 IEEE International Conference on Management of Innovation and Technology (ICMIT)*, pages 394–399, 2012.

[7] M. Landhausser and W.F. Tichy. Automated test-case generation by cloning. In *2012 7th International Workshop on Automation of Software Test (AST)*, pages 83–88, 2012.

[8] W. Janjic and C. Atkinson. Leveraging software search and reuse with automated software adaptation. In *2012 ICSE Workshop on Search-Driven Development - Users, Infrastructure, Tools and Evaluation (SUITE)*, pages 23–26, 2012.

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

[10] Werner Janjic, Oliver Hummel, Marcus Schumacher, and Colin Atkinson. An unabridged source code dataset for research in software reuse. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, page 339–342, Piscataway, NJ, USA, 2013. IEEE Press.

[11] T. Ravichandran and Marcus A. Rothenberger. Software reuse strategies and component markets. *Commun. ACM*, 46(8):109–114, August 2003.

[12] B. Boehm and V.R. Basili. Software defect reduction top 10 list. *Computer*, 34(1):135–137, 2001.

[13] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *2007 Future of Software Engineering*, FOSE '07, page 85–103, Washington, DC, USA, 2007. IEEE Computer Society.

[14] Yogesh Singh. *Software Testing*. Cambridge University Press, November 2011.

[15] Jiantao Pan. Software testing. http://www.ece.cmu.edu/ koopman/des_s99/sw_testing/, 1999. visted 06.06.2013.

[16] Antonia Bertolino and Eda Marchetti. A brief essay on software testing. *Software Engineering, The Development Process. Wiley-IEEE Computer Society Press,*, 2005.

[17] Juan Jin and Fei Xue. Rethinking software testing based on software architecture. In *2011 Seventh International Conference on Semantics Knowledge and Grid (SKG)*, pages 148–151, 2011.

[18] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.

[19] Arthur H. Watson, Thomas J. McCabe, and Dolores R. Wallace. Structured testing: A testing methodology using the cyclomatic complexity metric. *NIST special Publication*, 500(235):1–114, 1996.

[20] Frank Westphal. *Testgetriebene Entwicklung mit JUnit & FIT: Wie Software änderbar bleibt*. Dpunkt Verlag, 1., aufl. edition, November 2005.

[21] Werner Janjic and Colin Atkinson. Utilizing software reuse experience for automated test recommendation. In *International Workshop on Automation of Software Test (AST 2013) co-located with ICSE 2013*, San Francisco, USA, May 2013.

[22] M. Douglas McIlroy, J. M. Buxton, Peter Naur, and Brian Randell. Mass-produced software components. In *Proceedings of the 1st International Conference on Software Engineering, Garmisch Pattenkirchen, Germany*, page 88–98, 1968.

[23] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. How reuse influences productivity in object-oriented systems. *Commun. ACM*, 39(10):104–116, October 1996.

[24] M.T. Baldassarre, A. Bianchi, D. Caivano, and G. Visaggio. An industrial case study on reuse oriented development. In *Proceedings of the 21st IEEE International Conference on Software Maintenance, 2005. ICSM'05*, pages 283–292, 2005.

[25] Sushil Bajracharya, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina Lopes. 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*, page 681–682, 2006.

[26] Otávio Augusto Lazzarini Lemos, Sushil Krishna Bajracharya, and Joel Ossher. CodeGenie:: a tool for test-driven source code search. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, OOPSLA '07, page 917–918, New York, NY, USA, 2007. ACM.

[27] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. Speculative analysis: exploring future development states of software. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, FoSER '10, page 59–64, New York, NY, USA, 2010. ACM.

[28] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. Proactive detection of collaboration conflicts. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, page 168–178, 2011.

[29] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. Early detection of collaboration conflicts and risks. *IEEE Transactions on Software Engineering*, pages 1–1, 2013.

[30] 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, page 213–222, New York, NY, USA, 2009. ACM.

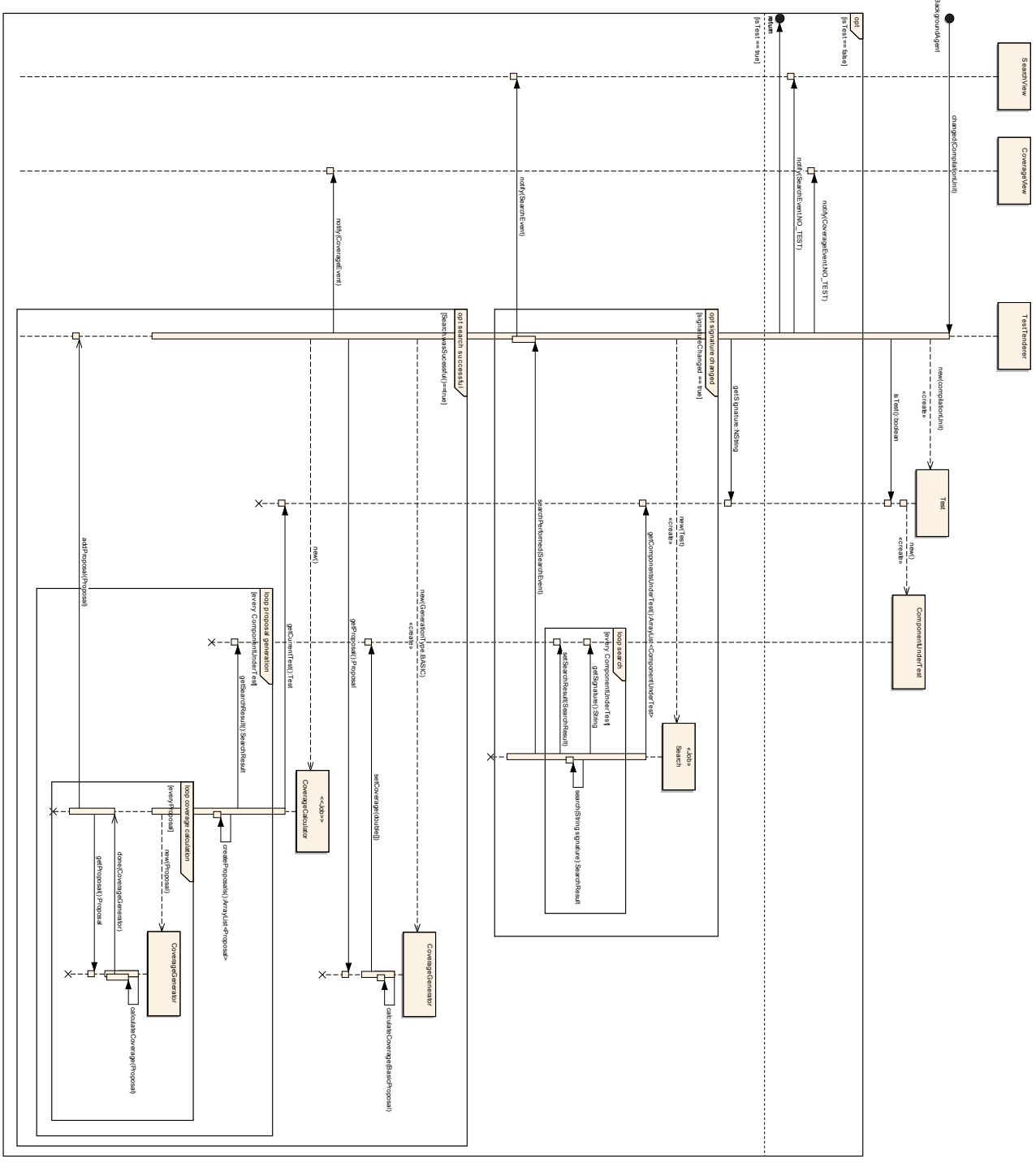
[31] Eclipse - the eclipse foundation open source community website. http://www.eclipse.org/. visted 02.04.2013.

[32] John Vlissides, R. Helm, R. Johnson, and E. Gamma. Design patterns: Elements of reusable object-oriented software. *Reading: Addison-Wesley*, 49, 1995.

[33] C. Atkinson, O. Hummel, and W. Janjic. Search-enhanced testing: NIER track. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 880–883, 2011.

[34] Christiane Fellbaum, editor. *WordNet: An Electronic Lexical Database*. A Bradford Book, May 1998.

Appendix A


SearchView
CoverageView
TestTestGenerator
BackgroundAgent
opt
[isTest == false]
return
[isTest == true]
changedCompilationUnit()
notify(SearchEvent(NO_TEST))
notify(CoverageEvent(NO_TEST))
notify(SearchEvent())
notify(CoverageEvent())
opt [search successful]
[SearchWasSuccessful() == true]
opt signature changed
[signatureChanged == true]
new(compilationUnit())
«create»
isTest():boolean
getSignature():String
Test
new()
«create»
ComponentUnderTest
new()
«create»
searchPerformed(SearchEvent())
getComponentsUnderTest():Array<ComponentUnderTest>
loop search
[every ComponentUnderTest]
getSignature():String
setSearchResult(SearchResult)
search(String signature):SearchResult
«obj»
Search
new()
getCurrentTest():Test
addProposal(Proposal)
loop proposal generation
[every CompilationUnit Test]
getSearchResult():SearchResult
«obj»
CoverageCalculator
loop coverage calculation
[every Proposal]
new(Proposal())
createProposals():ArrayList<Proposal>
getProposal():Proposal
doneCoverageGenerator()
CoverageGenerator
calculateCoverage(Proposal)
getProposal():Proposal
new(GenerationType.BASIC)
«create»
setCoverage(double[])
CoverageGenerator
calculateCoverage(BasicProposal)

Acknowledgements

“Above all else I especially thank my family for the valuable support during the last year. Your understanding and your confidence gave me the strength to finally complete my studies – and thus to finish an important chapter in life.

I also specially thank *Werner Janjic* for the challenging support, *Benjamin John* for the valuable assistance, and *Steven Griffiths* for his detailed linguistic remarks.

Without all your backing, this probably would not have been possible.”

Ehrenwörtliche Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mannheim, 15.06.2013

Oliver Erlenkämper