HAW
HAMBURG

BACHELORTHESIS
Wolf Saure

Do tests really enable change? On the relationship between unit test coverage and maintainablity of production code.

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE
WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Wolf Saure

Do tests really enable change? On the relationship between unit test coverage and maintainablity of production code.

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Angewandte Informatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Stefan Sarstedt
Zweitgutachter: Prof. Dr. Bettina Buth

Eingereicht am: 22.10.2021

Wolf Saure

Thema der Arbeit

Do tests really enable change? On the relationship between unit test coverage and maintainablity of production code.

Stichworte

Testabdeckung, Testaufwand, Softwarequalität, Wartbarkeit, Refactoring, Metriken, Code Smells, Studie.

Kurzzusammenfassung

Softwarequalität und insbesondere der Aspekt der Wartbarkeit bestimmen zunehmend über den langfristigen Erfolg von Softwareprojekten. Refactoring dient der Verbesserung der Wartbarkeit und wird durch eine hohe Testabdeckung unterstützt. Es wird daher allgemein angenommen, dass ein Zusammenhang zwischen Testabdeckung und Wartbarkeit besteht. Dieser Zusammenhang wurde in der vorliegenden Studie für 45 Java-basierte Open-Source-Projekte auf Basis von Metriken und Code Smells statistisch untersucht. Als Ergebnis wurde eine Vielzahl positiver Zusammenhänge mit hoher statistischer Signifikanz nachgewiesen. Dies könnte ein Hinweis sein, dass Entwickler den Testaufwand für schwer wartbaren Code erhöhen. Negative Zusammenhänge als Hinweis auf positive Auswirkungen einer hohen Testabdeckung auf die Wartbarkeit wurden hingegen nur für einzelne Projekte gefunden.

Wolf Saure

Title of Thesis

Do tests really enable change? On the relationship between unit test coverage and maintainablity of production code.

Keywords

Test coverage, Test effort, Software quality, Software maintainability, Refactoring, Metrics, Code smells, Survey.

Abstract

Software quality and especially the aspect of maintainability increasingly determine the long-term success of software projects. Refactoring serves to improve maintainability and is supported by high test coverage. It is therefore generally assumed that there is a correlation between test coverage and maintainability. This correlation was statistically examined in the present survey for 45 Java-based open source projects on the basis of metrics and code smells. As a result, a large number of positive correlations with high statistical significance were found. This could be an indication that developers are increasing test effort for code that is difficult to maintain. Negative correlations indicating positive effects of high test coverage on maintainability, on the other hand, were only found for individual projects.

Contents

List of Figures

List of Tables

Acronyms

ACM Association for Computing Machinery

BC Branch Coverage

BP Code smells associated to PMD rule set Best Practices

CC Cyclomatic Complexity

CE Efferent Coupling

C&K Chidamber and Kemerer Suite

CLOC Comment Line of Code

CS Code smells associated to PMD rule set Code Style

CSV Comma Separated Values

DN Code smells associated to PMD rule set Design

DIT Depth of Inheritance Tree

HCNC High Coverage and Negative Correlations

HCNC $_{(IC/MC)}$ High Coverage and Negative Correlations – only for instruction and method coverage

HTML Hypertext Markup Language

IC Instruction Coverage

IEC International Electrotechnical Commission

IDE Integrated Development Environment

IEEE Institute of Electrical and Electronics Engineers

ISO	International Organization for Standardization
ISTQP	International Software Testing Qualification Board
JaCoCo	Java Code Coverage Library
JLOC	JavaDoc Lines of Code
LCPC	Low Coverage and Positive Correlations
LCPC $_{(BC)}$	Low Coverage and Positive Correlations – only for branch coverage
MC	Method Coverage
MI	Maintainability Index
MPC	Message Passing Coupling
MTTR	Mean (or Median) Time To Repair
n/a	not available
NOC	Number of Children
OOD	Object-Oriented Design
PMD	The meaning is not clear (cf. [PMD21c])
RAP	Rapid Application Development
SoC	Separation of Concerns
SML	Standard Meta Language
TDD	Test Driven Development
UI	User Interface
UML	Unified Modeling Language
XP	Extreme Programming

1 Introduction

Computer software has an increasing impact on more and more aspects of everyday life and on society as a whole. This development is influencing both safety and security concerns and it has an increasing impact on the competitiveness of companies and products (cf. [SL19, p. 1]).

Therefore, the quality of software is becoming a more and more crucial and important success factor (cf. [SL19, p. 1], [FB20, p. 442]). One way of establishing quality are software tests. Unit tests especially play a central role in agile software development (cf. [CS14]).

Additionally, maintainability is of particular significance in regards to quality. This attribute describes the ease with which software can be changed, in order to implement new features or to remove bugs (cf. [Int21a]). Low maintainability leads to increasing maintenance costs and can, in the long run, endangers the success of software projects (cf. [PMVV12]).

One way of increasing maintainability is the activity of "refactoring"; hereby the source code of software programs is rewritten with the aim of making it easier to read, understand and change (cf. [FBB+05, p. 53]).

It is widely assumed that there is a connection between test coverage and maintainability. With a high level of test coverage, developers can confidently make changes to the productive code (cf. [Bec01, p. 46], [CS14, p. 2], [Mar09, p. 124]). On the other hand, code changes are risky if the test coverage is inadequate and can prevent developers from performing refactoring edits (cf. [KZN12, p. 4], [Lil20, p. 272]).

Therefore, test coverage is used in some projects as an indicator to quantify the maintainability of a software product (cf. [BFWZ18, p. 128]). But are test coverage and maintainability really correlated to each other?

So far, only one older study seems to exist that actually investigated a similar question for SML and C++ projects (cf. [HS96]). However, there seem to be no studies that correlate the aspects of test coverage and maintainability for Java projects. The aim of this investigation is to close this gap.

The first part of this work (starting on page 6) lays the foundations in answering the following questions:

- What is software quality?
- What role does software maintenance play in development?
- What is meant by "maintainability" and why is this important for software development?
- What is meant by "refactoring" and in what way is this related to the aspect of maintainability?
- How can maintainability – or more generally: software quality – be measured?
- What significance do unit tests have for the quality of software?
- What kind of test coverage types exist?

The second part (starting on page 35) describes the study itself. Finally, the results can be found in section 10 on page 71.

Part I

Foundations

2 Software quality

In this chapter it will briefly be summarized how software quality can be defined and conceptualized (section 2.1) and what is meant by software requirements (section 2.2).

2.1 Definition and ISO/IEC 25010 quality model

The IEEE(1990) define software quality as "(1) The degree to which a system, component, or process meets specified requirements. (2) The degree to which a system, component, or process meets customer or user needs or expectations." [IEE90, p. 60].

A more recent but quite similar definition was given by the ISO (2011), who define software quality as the "degree to which a software product satisfies stated and implied needs when used under specified conditions" [Int21a].

From these definitions it can be concluded that there is no "absolute" quality but only a quality relative to specific requirements associated with a specific software system.

To make the concept of quality more tangible (and measurable), quality models are used. They describe the different aspects of software quality and their relations to each other. In the past, several quality models have been devised (cf. [FB20, p. 442 ff.]). In 2011, the international standard ISO/IEC 25010:2011 has been passed, which defines two separate models (cf. [Int21a]):

- the quality in use model, which names five characteristics (effectiveness, efficiency, satisfaction, freedom from risk and context coverage) that describe the outcome of the interaction between an user and a system when used within a specific context.
- the product quality model, which names eight characteristics and 31 subcharacateristics, that describe both the static (e.g. some properties of the source code) and the dynamic (e.g. how quickly a system responds to an user request) traits of a software product.

The product quality model has replaced the previous model defined within the ISO/IEC 9126 standard and is more frequently cited in literature (e.g., cf. [FB20]). Therefore, reference is only made to the second model, which is illustrated in fig. 2.1:

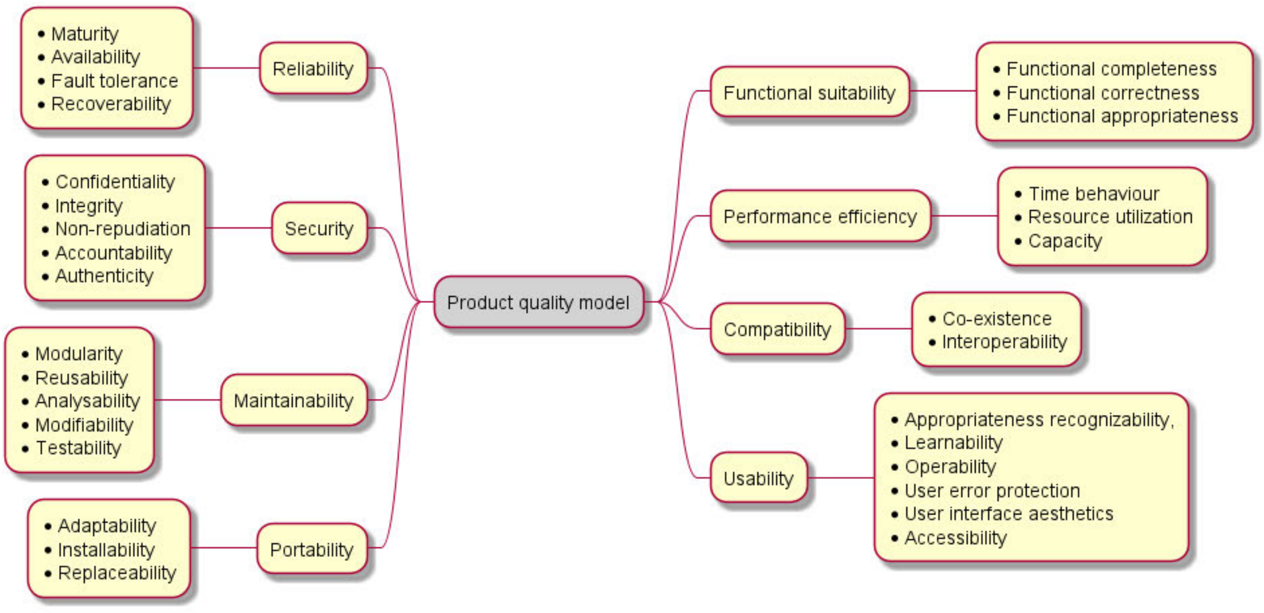


Figure 2.1: Quality characteriscs defined by the ISO/IEC 25010 product quality model (own representation based on [Int21a]).

As can be seen, the model consists of eight characteristics and 31 sub-characteristics. It is beyond the scope of this work to give a detailed description of all attributes shown above. Of interest is the characteristic of maintainability which will be described in more detail later on.

2.2 Different forms of requirements

From the definitions stated above the term requirements (= needs, expectations) is important. Pohl & Rupp (2015) differentiate three different types of requirements (for better illustration, the example of a text editor will be used):

- Functional requirements: they define which functions a system has to offer to its users (e.g. the users of the editor should be able to underline passages of text and they should be able to export a text as a pdf-file).

- Quality requirements (or non-functional requirements): they define which kind of quality goals a system shall meet (e.g. the text editor should be "performant" - the start of the application should take less than two seconds; and the editor should be "reliable" - when an error occurs, the text can still be recovered and not be lost).
- Ancillary conditions: they define conditions for the product or the development process, that should be met apart from functional or quality requirements (e.g. the editor shall be realized as a web service and be available to customers until the end of this year) (cf. [PR15, p. 8 f.]).

From the examples given above, it should be noted that the quality model does not define how the various types of quality can or should be measured. Instead, it is necessary that all stakeholder groups need to be identified and asked to give or agree on a precise definition of those aspects of quality that are relevant to them (cf. [PR15, p. 22]).

Stakeholder of software products are all people and organisations that influence these requirements. That are, of course, first and foremost customers using the system. Further stakeholder groups are people that are involved in the development and organisations or authorities that set standards or rules, which have to be abided by the software. Even people who want to damage the system (e.g. hacker groups) need to be taken into account in order to ward of attacks (cf. [PR15, p. 3, 142]).

2.3 Summary

In summary, it should be noted that software quality is based on requirements which are voiced by different groups that are somehow related to a software product. In the next chapter, the attribute of maintainability – as part of the product quality model – shall be described in more detail.

3 Maintenance and maintainability

Important (for the scope of this work) is the fact, that requirements (cf. section 2.2) evolve during the life-cycle of a software system: the system is run under unforeseen conditions, customers voice additional expectations, special use cases need to be considered or after some time of running problems occur that were not observed previously (cf. [SL19, p. 94]; additionally: [PR15, p. 140 f.]), [LL13, p. 576]).

Consequently, the software needs to be adapted to these new or changing requirements – it needs to be "maintained". But what precisely is "maintenance" and when does it happen (section 3.1)? And what is "maintainability" and why is it of special importance, at least in the long run (section 3.2 on page 11)?

3.1 Software maintenance

3.1.1 Definition and position within software life cycle

Maintenance is defined by the IEEE – with regard to software – as the "process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment" ([IEE90, p. 46]).

Here, the words "after delivery" are already hinting at the position of maintenance within the life span of a software product. The so called software live cycle consists of the following phases (cf. [IEE90, p. 68]): (a) Concept exploration, (b) Requirements, (c) Design, (d) Implementation, (e) Test, (f) Installation and checkout, (g) **Operation and maintenance** and (h) Retirement. In contrast, the development cycle encompasses only the phases until installation / delivery (cf. [IEE90, p. 73]).

Important for further understanding are the differences between existing development models. These models are used to give a logical structure to the process of software development. According to Spillner & Linz (2019), two main models can be differentiated: sequential and iterative-incremental development models (cf. [SL19, p. 53]):

- In sequential models (e.g. Waterfall- or V-Model), development phases are meant to happen one after the other, in linear order and without repetition. After completing the last phase of development, the product is delivered to its customers ([SL19, p. 53]). Behind this approach stands the conviction that it is possible to collect all requirements from all stakeholder groups in one go – and that these requirements will not change during the following development phases (which may well take several years after the onset of a project: cf. [SL19, p. 54]).
- On the other hand, in iterative-incremental development models (e.g. RAP, XP, Kanban, Scrum) a software product is delivered repeatedly to its end users. In each iteration, all phases are run through; the functionality is extended or adapted and the new version is tested, delivered and – important for the aspect of maintenance – customer feedback is obtained (cf. [SL19, p. 58]).

As a consequence, the significance of maintenance is different for these two types of development approaches. Whereas in sequential models maintenance happens not until the main development has finished, with the iterative approach the software is continuously maintained during development as well.

Of importance is the fact that changes to a software system are more expensive, the later they are made (cf. [PR15, 2]).

3.1.2 Forms of maintenance

According to the IEEE, the following forms of maintenance can be distinguished (cf. [IEE90, p. 8, 22, 55, 57]; additionally: [FB20, p. 461]): (a) preventive maintenance: for the prevention of problems which are likely to arise, e.g. potential bugs, (b) adaptive maintenance: if changes in the environment have occurred and the program needs to be adapted in order to work properly), (c) corrective maintenance: in order to resolve any bugs or faults, which were found while testing or which have been reported by customers, and (d) perfective maintenance: in order to improve some characteristics of a software product, e.g. its performance or maintainability.

Of special interest for this work is the last type of maintenance. Refactoring – i.e. the process of changing code without modifying its behaviour, e.g. to enhance its readability (cf. section 4.1) – can be viewed as a form of perfective maintenance.

3.2 What is maintainability and why is it important?

In section 2.1 the product quality model was introduced. In this model, maintainability is one of eight characteristics (cf. fig. 3.1).

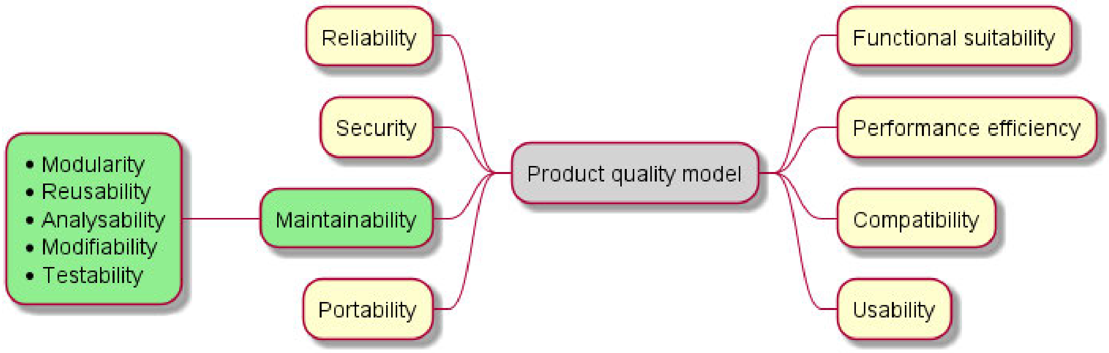


Figure 3.1: Maintainability within the ISO/IEC 25010:2011 product quality model (own representation based on [Int21a]).

3.2.1 Definition and subcharacteristics of maintainability

Maintainability is defined by the ISO/IEC 25010 standard as the "degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers" ([Int21a]).

According to this standard, maintainability can be further divided into the following subcharacteristics:

- Modularity: the degree, to which a system is divided up into separate and – preferably – independent modules or components, where a change to one component will not affect other components (cf. [Int21a]). A module can again be composed of other modules (cf. [Lil20, p. 20]).

- Analysability: the ease with which the maintainers of a system are able:
  - to find out the cause of a failure,
  - to identify which part of a system needs to be changed or
  - to estimate the impact of changes to parts of a system (cf. [Int21a]).
- Modifiability: The ISO/IEC defines this aspect as the ease with which a system can be "modified without introducing defects or degrading existing product quality" ([Int21a]). In the notes to this definition, the standard points out that modifiability is influenced both by modularity and analysability (cf. [Int21a]).
- Reusability: the degree, to which any part of a system (e.g. a method, a class or a component) can be utilized in other systems as well (based on [Int21a]).
- Testability: the ease with which it is possible to define test criteria and to implement and execute software tests in order to find out if a system or part of a system fulfils these criteria ([Int21a]).

Which of these sub-aspects of maintainability is the one that is most important? From the definition above, modifiability seems to play a central role. According to Lilienthal (2020), for long-lasting software products modularity is the key factor (cf. [Lil20, p. 66]). Other authors point out that the characteristics modifiability and reusability are important in order to diminish maintenance costs (cf. [SL19, p. 89]);

3.2.2 Maintainability and technical debt

Closely related to maintainability is the concept of technical debt. Cunningham (1992) introduced this analogy to express that existing code must be continuously rewritten and the design adapted to the implementation of new program features, in order to stay maintainable in the log run: "Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite. [...] Every minute spent on not-quite-right code counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation, object-oriented or otherwise" ([Cun92]).

Technical debt symbolizes the additional effort and costs needed for maintenance (cf. [Lil20, p. 4], [Vet13, p. 78]).

Some authors further distinguish between different forms of technical debt (defect debt, implementation debt, design debt, documentation debt and testing debt), thus indicating more precisely, what kind of maintenance effort is needed (e.g., fix defects, remove code smells, improve the architecture, update documentation or write missing tests: cf. [Vet13, p. 109], [Lil20, p. 14]). In the context of this work, only implementation and design debt are of interest.

3.2.3 Symptoms of low maintainability

Why is the aspect of maintainability relevant to software development? One way of answering this question is to point out what happens when a software product has a *low* maintainability. According to Martin (2000), the following symptoms will begin to appear [Mar00, p. 2 f.]):

- Rigidity: A system is rigid, when even simple changes take much longer than expected because of growing dependencies between modules. Thus, one modification leads to further modifications in related modules and so forth.
- Fragility: This symptom describes a software product, where modifications lead to unexpected failures in other components, that sometimes do not even seem to be related to the area where the change was introduced.
- Immobility: Here, dependencies between modules make it improbable if not impossible to reuse part of a software somewhere else. Therefore, instead of reusing a method, class or module, it is simply rewritten.
- Viscosity: A system has a high viscosity when developers are tempted to implement changes that compromise the design of a system. Either because of "difficult" design guidelines (design viscosity) or because of a slow or impractical development environment (environment viscosity).

In summary, software with low maintainability has more dependencies which leads to prolonged maintenance periods, unexpected failures and double code.

There are three main reasons for these symptoms: (a) there are changing requirements, that were not foreseen when the original design of a system was conceived; (b) the changes need to be implemented fast and (c) not all developers are familiar to the original design guidelines ([Mar00, p. 3]).

3.2.4 Consequences

As a consequence to the symptoms described above, it takes much longer to adapt software with a low maintainability. According to Visser et al. (2016), in systems with below average maintainability it takes twice the time to resolve issues or to implement enhancements than in systems with above average maintainability (cf. [VRvdL+16, p. 3]).

In regards to the different forms of maintenance (cf. p. 10): it takes longer to adapted it to changing requirements, to fix bugs (so called issues), to enhance some quality aspect or to prevent future faults.

Therefore, it comes not as a surprise that Poort et al. (2012) found out that – in the eye of software architects – modifiability is crucial for the success of software projects (cf.[PMVV12]; additionally: [VRvdL+16, p. 3]).

Visser et al. see maintainability as an "Enabler" for other quality characteristics (as depicted in fig. 3.1 on page 11), since for all kinds of changes it is necessary to find, analyse, understand and test the source code - and this is easier in systems with higher maintainability: "[...] optimizing a software system requires modifications to its source code, whether for performance, functional suitability security, or any other of the seven nonmaintainability characteristics defined by the ISO 25010" (cf. [VRvdL+16, p. 4]).

3.3 Summary

Maintenance for software products is necessary in order to fix bugs, adapt a system to changing requirements or to enhance quality attributes. Whereas in traditional development models maintenance only happens after the end of the development process, in iterative models software is maintained continuously.

Maintainability describes how easily a software product can be changed and adapted. A system with low maintainability shows several symptoms, the common cause being increasing dependencies between components. In the long run, this leads to higher development costs and can endanger the success of software projects.

In the next chapter, the concept of refactoring will be presented – this can be seen as a form of perfective maintenance with the aim of reducing technical debt.

4 Refactoring – cure for the symptoms?

In section 3.2.3, four symptoms of low maintainability have been described. Refactoring can be seen as a "cure" in this regard. In section 4.1, the idea behind this concept will be described briefly; in section 4.2 on the next page, the relevance of refactoring for maintainability will be depicted and in the remaining chapter (section 4.3) the starting points for refactoring in practice will be described in more depth.

4.1 Definition and scope of refactoring

Refactoring was defined by FOWLER ET AL. (2005) as "a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour" [FBB+05, p. 53].

To give an example: in section 5.5 on page 25 a function named CALCULATE() will be presented that has three input parameters (two numbers and one operand) and returns a numeric result. Refactoring – in a narrow sense – means that the function stays the same in regards to both input parameters and return value. Thus, the "external behaviour" will be the same – two by two should still yield four.

But, as KIM ET AL.(2012) summarize, in practice and in science different definitions are in use. In practice, not all refactoring changes are behaviour-preserving or they are combined with other modifications that *do* change the behaviour (cf. [KZN12, p. 2]). This perception is backed up by LEPPÄNEN ET AL. (2015), who found that developers view refactoring as an activity that aims to redesign a system and that would take several days rather than simple tasks like method renaming (cf. [LML+15, p. 64]).

It should be added that refactoring edits are not limited to production code. Since tests can be "smelly" as well (cf. section 5.6.2), there is an equal need for optimization (cf. [Mar09, p. 123]).

4.2 Significance of refactoring for maintainability

According to the above definition (cf. section 4.1), refactoring is supposed to affect the following quality characteristics: understandability and changeability. Both are subcharacteristics of maintainability (naming according to ISO/IEC 25010 standard: analysability and modifiability; cf. section 3.2.1). But what is the rationale behind this assumption?

In chapter 3 it was stated that as requirements evolve or bugs are found the system needs to be maintained, i.e. some source code needs to be added or changed. According to Fowler et al.(2005), without refactoring the initial design of a software system will deteriorate due to these changes: "[...] the code looses its structure. It becomes harder to see the design by reading the code. Refactoring is rather like tidying up the code" ([FBB$^+$05, p. 55]).

The need for a continuous code refurbishment is likewise expressed by the technical debt-analogy (cf. section 3.2.2): refactoring is necessary in order to repay debts or otherwise the maintenance effort will become unforeseeable (cf. [Lil20, p. 4 ff.]).

4.3 Starting points

How can developers go about the task of refactoring code? In this section, some starting points shall be illustrated.

Several books have been written with the aim to guide developers in writing code and building software that is highly maintainable (e.g. [Mar09], [BF11], [VRvdL$^+$16]). It would be far beyond the scope of this work to give a sufficient introduction on the many different aspects that need to be taken into account in order to achieve this complex task.

Instead, for the context of this study some examples shall be given on how to achieve a higher level both in analysability and modifiability – the two sub-characteristics of maintainability mentioned in the definition for refactoring.

For a short repetition, analysability means the ease with which the cause of failures can be found whereas modifiability indicates how easy it is to modify a system without introducing new defects (cf. section 3.2.1).

4.3.1 Refactoring for higher modifiability

One central cause for low changeability is coupling (cf. section 3.2.3) – i.e. dependencies between software components – which often is a consequence of suboptimal design decisions. Coupling can have different degrees of severity (cf. [LL13, p. 413]).

To improve object-oriented design (OOD), many different design patterns have been deceived. These patterns imply established solutions and best practices for design problems that developers frequently are confronted with ([GHJV15, p. 25 f.]). Many of these patterns help to reduce dependencies between components. The observer pattern, for example, decouples (a) a component that wants to inform about a change of state (b) from those components that in some ways rely on or need to know about these changes (cf. [GHJV15, p. 360]).

Additionally, there are more classical OOD principles like information hiding, which means that software components should reveal as little information about their internal structure as possible (cf. [IEE90, p. 40]). This again helps to reduce dependencies since now the internal structure of a component can be changed or improved without affecting other components.

Last but not least, there are guidelines such as the Law of Demeter, which recommends that a function or method should restrict calls to other methods to those that are directly attached to a component – which promotes decoupling as well (cf. [LHR88, p. 326]).

4.3.2 Refactoring for higher analysability

There are several ways for organizing and writing source code so that other developers have less difficulty to read it and understand how it works. This can be done on different levels.

On a higher level, there are again some common principles:

- Separation of Concerns (SoC) or Single Responsibility Principle advises that a software component should be responsible for only one specific field of duty – or otherwise it might become to complex and difficult to understand ([LL13, p. 418], [LML+15, p. 66]).

- Cohesion describes in how far the elements of a software component are related to each other, belong together and serve a common purpose (cf. [FB20, p. 375, 418]). A high level of cohesion is recommended for a comprehensible code structure (cf. [Mar09, p. 141]).

On a more fundamental level, there are many recommendations regarding (a) naming (e.g., should be meaningful and reveal the intention behind a code element: cf. [Mar09, p. 18]), (b) size of functions (should be as small as possible: cf. [Mar09, p. 34]) or (c) formatting (should be supportive for reading; e.g., additional blank lines to separate different concepts: cf. [Mar09, p. 78]).

All of these principles, patterns, guidelines and recommendations can be used as part of refactoring edits and thus help to decrease coupling or improve understandability of code.

4.4 Summary

The original idea of refactoring is to make rather small changes to the internal structure of a code element without changing its external behaviour. In practice, a broader concept seems to be in use that includes bigger changes of design and is not restricted to behaviour preservation.

Nevertheless, the overall aim is to achieve higher code quality in regards to understandability and changeability. Several principles, patterns and guidelines were deceived that help to achieve this.

In order to estimate whether a refactoring edit was successful, it is necessary to somehow measure respective quality attributes. How this can be done will be illustrated in the next chapter.

5 How can software quality be measured?

In this work, the central question is whether unit test coverage is correlated to maintainability (cf. section 1). It will therefore be necessary in due course to measure both aspects and see if a statistical relation can be proven.

In general, several aims are being pursued by measurements in regards to both the development process as well as the software product itself (cf. [FB20, p. 14 ff.], [LL13, p. 297]):

- to visualize and understand relationships between development activities and affected attributes (e.g., if there is a correlation between low maintainability and duration / costs of maintenance activities, as described in section 3.2.4).
- to monitor and control those aspects of development that are relevant to the stakeholders of a software project (e.g., to monitor the complexity of source code, which may be important to future maintainers).
- to support decisions (which attributes of a product need special attention?).
- to improve both the development process and the quality attributes of a software product to reach predefined targets (e.g., change the source code of those modules that exceed a certain complexity value).
- to make forecasts (e.g., related to costs, target dates or the level of quality that will probably be achieved).

Still, as Fenton & Bieman point out, in many software projects it is neglected to measure central development attributes (cf. [FB20, p. 12 f.]).

For measuring those attributes, quality metrics are used. The IEEE defines quality metric as follows: "[...] A function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which the software possesses a given quality attribute" ([IEE90, p. 60]).

To give an example: to calculate the size of a module, the lines of code can be counted. From the above definition, the input is the source code of the module in question and the output is the calculated size, say 125 lines of code. On first look, this seems to be a simple and straightforward matter. Still, even here several different approaches exist, depending on whether empty lines, comments or data declarations are being taken into account or not (cf. [FB20, p. 339 f.]; additionally: [Hof13, p. 250]).

Important, for this work, is the fact that there are two main approaches to measure software attributes: the dynamic and the static approach (sections 5.3 and 5.4). But first of all, some introductory explanations regarding test levels and test types are necessary (sections 5.1 and 5.2). The chapter continues with an illustration of how test coverage and maintainability can be measured (sections 5.5 and 5.6) followed by a brief description of tools necessary for this task (section 5.7). Since the topic of measurement is of major importance for the subsequent investigation, this chapter is somewhat more extensive.

5.1 Test levels

Traditionally, the following test levels are distinguished (cf. [SL19, p. 56, 62 ff.], [Hof13, p. 159]):

- Unit testing: Testing of individual and basic software components to find out whether each component meets its specified requirements.
- Integration testing: Here, it is tested whether groups of components work together as expected.
- System testing: Testing if the system as a whole meets the requirements.
- Acceptance testing: This level resembles the system testing level, but here the view of customers or later users is put forward and contractual requirements are tested more explicitly. Moreover, in contrary to system testing, the customer is typically directly involved as a tester.

For the context of this investigation, only the first level is of relevance.

5.2 Test types

In addition to test levels, several different test types are named in literature which can be grouped in functional, non-functional and structure based tests (cf. [SL19, p. 86 f.]):

- Functional tests check whether a test object fulfills functional requirements (cf. section 2.2) – e.g., if a method that multiplies two numbers returns the correct value. This kind of tests are used on all test levels (cf. [SL19, p. 88]). Test cases are designed by employing so-called black box test design techniques. This means that only the input parameters and the return value are taken into account whereas the internal structure of a component or method is irrelevant for the design of test cases (black box testing; cf. [SL19, p. 141 ff.], [LL13, p. 173 f.]).
- Non-functional tests, on the other hand, check to which degree non-functional requirements (cf. section 2.1) are fulfilled. Here, each test type is focused on a specific quality attribute, e.g. reliability tests, performance tests or usability tests (cf. [SL19, p. 89 ff.]).
- In opposition to functional tests, structure-based tests check the internal structure of a software system, i.e. the control or data flow within a component is tested and used for the design of respective test cases (white box testing; cf. [SL19, p. 192 ff.], [LL13, p. 174]).

Important, for the scope of this work, are two additional test types that are especially related to maintenance:

- Confirmation testing: this kind of testing is executed after an issue has been resolved in order to make sure that the failure does not occur anymore (cf. [Int21b]).
- Regression testing: Here, after a software modification of any kind, existing tests (often automated and only a sub-selection of all tests) are re-run to guarantee that no new failures were introduced by the changes made (cf. [SL19, p. 98 ff.]).

5.3 Dynamic testing

According to the IEEE (1990), a (dynamic) test is an "activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component. [...]" ([IEE90, p. 74]; also called dynamic analysis: cf. [Int21c]).

From this definition it should be noted that dynamic testing is only applicable for executable code. The aim of this kind of testing is to find software failures (cf. [LL13, p. 480]), i.e. behavior of a system or component, where given requirements are not fulfilled, either because the execution is too slow, the results are incorrect or because the system terminates unexpectedly (cf. [IEE90, p. 32]).

When a failure is found, it becomes necessary to locate the specific part within the source code, which causes the malfunction – this is typically called fault or defect (cf. [IEE90, p. 32], [Int21c]).

Significance of unit testing

This work focuses on unit test coverage. Why are unit or component tests especially relevant? This is best described by the test automation pyramid (cf. [Coh09]). The test pyramid is used as an analogy to show the recommended ratio of above mentioned test levels: Most tests should be automated unit tests (the base of the pyramid) and only a small percentage of tests should be dedicated to (manually) testing the user interface (UI) – the tip of the pyramid.

The rationale, why unit tests should be preferred over UI tests is the following: Unit tests (a) are much quicker than UI tests, (b) are easier to maintain, (c) are more reliable, since UI tests can easily break; and they (d) usually give precise information as to where the fault is located, thus relieving the task of finding and fixing software bugs (cf. [Coh09], [Fow12]).

The principles of Test Driven Design (TDD) (also called test-first-approach) reinforces the role of units tests. TDD advises developers to write unit tests first before doing anything else. The second step is to write only as much code as to make the test pass, typically followed by the last step of refactoring the production code to make it easier to maintain (cf. [SL19, p. 70 f.], [Mar09, p. 122 f.], [Bec01, p. 115 ff.])

5.4 Static testing

In contrast to dynamic testing, static tests are not restricted to source code but can be applied to all products that are generated as part of the software development or test process (e.g. documentation of requirements or architecture as well as test plans, user manuals or contracts; cf [SL19, p. 103 f.]).

Two different approaches

The following two approaches can be distinguished: (a) Manual reviews, where any of the just mentioned artifacts are being scrutinized by a group of experts. There are several different review types (such as informal review, technical review, inspection or walkthrough) which differ, for example, in the way the review process is organized and documented, the time and effort needed or the goals that are being pursued (cf. [SL19, p. 105 ff.], [Hof13, p. 321 ff.]). (b) Tool-supported analysis or automated static code analysis: here, software tools are used to analyze source code or any other documents that have a formal structure (e.g. UML specifications). In contrast to dynamic testing, the source code is not executed here (cf. [SL19, p. 283], [Hof13, p. 247 ff.]).

For the scope of this work, only the second approach is of interest.

Different forms of automated code analysis

Static automated code analysis can be used for several different purposes. Relevant for this work are the following approaches:

- Analysis of control flow: The control flow describes which parts of source code are being executed and which are not. It can be visualized by a control flow graph; different variants, as to how the control flow is modelled, exist (cf. [Hof13, p. 202 ff.]). For illustration, the following graph is used (cf. fig. 5.1)

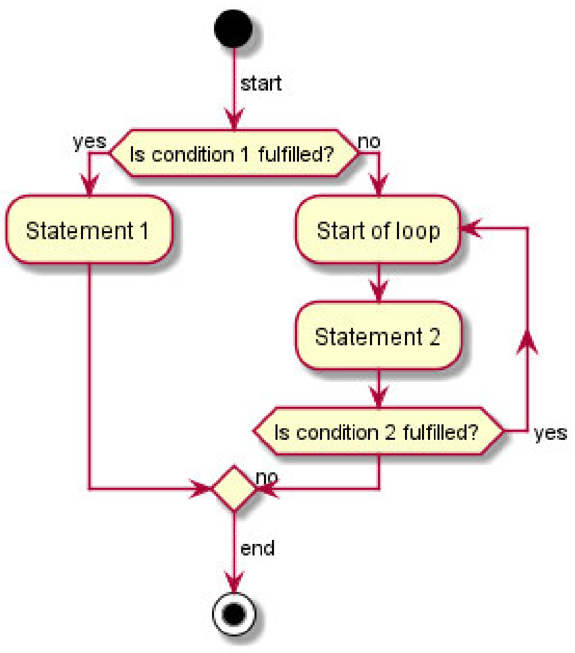


Figure 5.1: Example for a control flow graph (own representation).

This graph shows a fairly simple function, where the control flow is represented by the edges (arrows) and code statements are represented by nodes (yellow boxes plus two separate nodes representing start and end of the function). The example graph is visualizing that (a) only when condition 1 is fulfilled, statement 1 will be executed and the functions terminates; (b) otherwise statement 2 within the loop will be executed as many times as condition 2 is fulfilled (this can one time, many times or forever) and (c) finally, when condition 2 is not fulfilled anymore, the function ends as well.

Control flow analysis is necessary for calculating test coverage (cf. section 5.5).

- Calculation of software metrics: All internal attributes – i.e., attributes of the software product itself, without taking its behavior or environment into account – can be calculated by respective software tools, e.g. metrics for size, complexity, coupling or reuse (cf. [FB20, p. 88 ff.], [Hof13, p. 247 ff.]). This allows conclusions to be drawn about external quality characteristics – like maintainability (cf. [FB20, p. 90, 461 f.]). The relevant metrics for measuring maintainability will be described in more detail in section 5.6.

5.5 Measurement of test coverage on unit test level

Test coverage can be defined as the "degree to which a given test or set of tests addresses all specified requirements for a given system or component" ([IEE90, p. 75]).

Depending on test level and test type (cf. sections 5.1 and 5.2), different kinds of test coverage targets are usually defined. The overall aim is to reach a high test coverage with as few test cases as possible (e.g., cf. [Hof13, p. 175]).

On unit test level, both functional and structure-based tests are employed (cf. [SL19, p. 88, 93]). In the ensuing descriptions, the following example is used for better illustration: a function CALCULATE(INT NUM1; INT NUM2; OPERAND OP) that accepts two integer numbers and an operator (MULT for multiplication, DIV for division), executes a calculation depending on the operator and returns the result.

5.5.1 Coverage types for functional testing (black box testing)

A common way to design test cases for functional tests is equivalence partitioning. Here, the input parameters are divided up into classes, where all values of that class behave in the same way. For the example above, three numeric classes can be constructed: negative numbers, positive numbers and zero. For each class of each parameter, a representative is chosen (e.g., -5, 5, 0, MULT, DIV). Assuming a test case exists for all possible combinations, a equivalence partition coverage of 100 % is achieved (cf. [Hof13, p. 175]). Other forms of coverage are – for example – boundary value coverage or state (transition) coverage (cf. [SL19, p. 162, 170]).

5.5.2 Coverage types for structure-based testing (white box testing)

As stated above (see section 5.2), in case of structure-based testing the internal structure of a code element is used for testing (in this context, by looking at the control flow). Regarding the function CALCULATE() mentioned above, the control flow graph could look like this (cf. fig. 5.2):

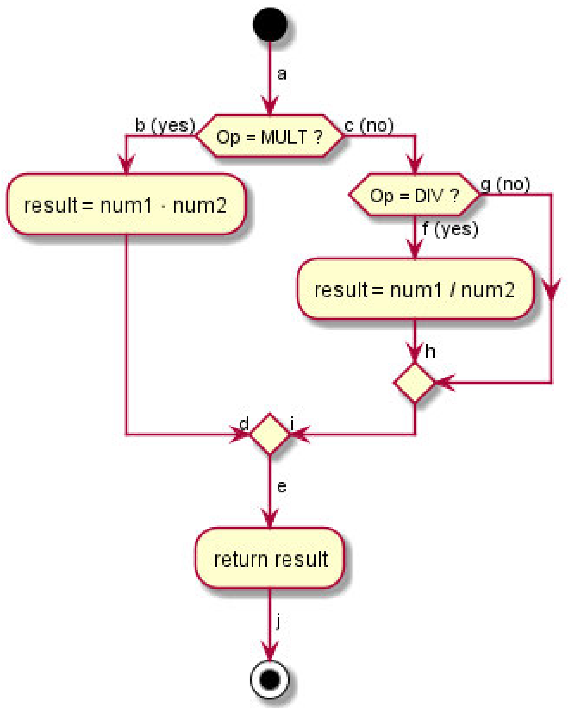


Figure 5.2: Control flow graph for function CALCULATE() (own representation).

In this graph are displayed ten edges (a to j) and nine nodes, which will be referred to in the following explanations.

The most frequently cited coverage types are the following (cf. [SL19, p. 193 ff.], [Hof13, p. 200 ff.]):

- Statement coverage: For a coverage of 100 % it is necessary that all statements (nodes) are executed at least once. In the example above, to achieve this at least two test cases are necessary (one multiplication and one division).
- Branch coverage / Decision coverage: Here, not the statements (nodes) are relevant but the branches (arrows), depending on the decision in loops, if- or case-statements or error handling situations. In case of the example, to obtain a coverage of 100 %, again both operands (MULT and DIV) must be used; additionally, some other operand (e.g., ADD for the addition of two numbers) must be used as a function argument as well in order to cover branch g.

- Condition coverage: In contrast to decision coverage, where a decisions are evaluated as a whole (true or false), condition coverage has an even higher granularity. Supposing, at the beginning of the example function it would checked in a single if-statement that numbers are both greater than -100 and smaller than 100. Here, both (atomic) conditions of the if-statement must be covered in order to achieve a full coverage.

- Path coverage: Path coverage is taking into account individual paths (i.e., a sequence of nodes and edges) through a function. For the above example, the following paths exist (for simplicity, nodes where omitted): {a, b, d, e, j}, {a, c, f, h, i, e, j} and {a, c, g, i, e, j}. In case of loops (e.g., as in the graph in fig. 5.1), it is not always possible to design test cases so that a full path coverage is achieved (or it takes too long for the tests to run).

Evidently, designing test cases to achieve a high coverage in structure-based testing is not an easy task and requires a high commitment – in regards to time effort as well as in accuracy. Still, hidden defects are more likely to be found with white-box-tests rather than black-box-tests. In the above function, for example, exists the fault that on using an operand other then MULT or DIV would lead to an unexpected situation, where a result is returned that was not assigned a value previously (cf. fig. 5.2).

5.5.3 Significance of test coverage for maintainability

Why is test coverage relevant for keeping software maintainable? Since this question touches the central idea of this work, existing sources will be quoted more fully.

Maintainability, as we have established, means that a software product can be changed in an effective way in order to fix bugs, introduce new functionality or improve the design (cf. sections 3.1.2 and 3.2.1).

One aspect frequently named in literature is that a high test coverage increases the level of confidence developers have in their software – and that this confidence enables them to change the code, as they can feel sure not to "break" any existing functionality.

As Martin (2009) writes: "It is *unit tests* that keep our code flexible, maintainable and reusable. The reason is simple. If you have tests, you do not fear making changes to the code! [...] Indeed, you can *improve* that architecture and design without fear [...] tests enable *change*" (accentuation by Martin; [Mar09, p. 124]).

Lilienthal (2020) also stresses that in order to reduce implementation or design debt (cf. section 3.2.2), the fundamental requirement is to have a high test coverage (i.e., low test debt; cf. [Lil20, p. 14]).

Beck (2001) points out that tests not only have the short-term effect of increasing confidence, but help to keep software alive longer (cf. [Bec01, p. 46]). Additionally, tests help to quickens up maintenance: "Once you have gotten used to testing, though, you will quickly notice the gain in productivity [...] you no longer spent an hour for a bug, you find it in minutes" ([Bec01, p. 46]).

Fowler (2005) supports the statements above by saying: "If you want to refactor, the essential precondition is having solid tests [...] I don't see this as a disadvantage. I 've found that writing good tests greatly speeds up my programming, even if I am not refactoring" ([FBB+05, p. 89]).

Last but not least, Choudhari & Suman (2014) add several more advantages of a high test coverage: "Extensive test coverage provides several advantages in the maintenance process such as instant feedback while working on legacy code, confidence and courage while making error-prone modifications, improved code readability, and faster impact analysis before any modifications" ([CS14, p. 2]).

Summarized, a high test coverage is deemed in several ways as essential for maintainability and thus for an effective maintenance process: (a) it increases confidence, (b) relieves the task of making changes or improving the design of existing code, (c) it speeds up the finding of defects, (d) helps to change legacy code and (e) increases readability.

5.6 Measurement of maintainability

In literature, two different approaches are frequently named (e.g. [Vet13]) in order to measure maintainability: metrics and code smells (cf. sections 5.6.1 and 5.6.2).

5.6.1 Maintainability metrics

As Fenton & Bieman (2021) point out, two different views on maintainability exist which can both be measured (cf. [FB20, p. 461 ff.]).

External metrics

The external view on maintainability deals with the development or maintenance process, i.e. with relations between a software product and its maintainers - those responsible for resolving an issue, implement a new function or improve some quality characteristics (cf. section 3.1.2).

Consequently, the so called external metrics seek to measure how efficient software can be maintained; a typical metric is the mean or median time to repair (MTTR), which is measured by keeping track of the time to recognize and analyze a problem, make the appropriate changes and do some administrative work related to solving the problem.

Additional metrics are, for example,

- the number of unresolved issues,
- the percentage of changes that lead to new faults or
- the number of modules that need to be adapted in order to implement a neccessary change (cf. [FB20, p. 462]).

Evidently, these external metrics are related to the above mentioned symptoms of low maintainability (cf. section 3.2.3).

Internal metrics

The internal view, on the other hand, deals with the software product itself without taking into account its environment. Typical internal metrics in regards to maintainability are focused on the structure of source code, its complexity, how easy it can be read or how well documented it is (cf. [FB20, p. 463 ff.]).

A plethora of metrics has been proposed; to name just the most popular: (a) McCabe's Cyclomatic Complexity (CC), (b) Efferent Coupling (CE), (c) Chidamber and Kemerer Suite (C&K), (d) The Halstead Suite, (e) Comment Line of Code (CLOC), (f) Lines of Code (LOC) or (g) Maintainability Index (MI) (cf. [ACBV20, p. 6 ff.]).

It is assumed that internal and external metrics are correlated. Since internal metrics can be measured earlier and easier, they are used to predict the external ones (cf. [FB20, p. 88 f., 462]).

For this work, too, it would have been impossible to collect external maintainability metrics. Therefore, internal metrics were used (they are described in more detail in table A.1 on page 81).

5.6.2 Code smells

Apart from metrics, another way to measure the maintainability of software are code smells (e.g., cf. [Vet13, p. 84]).

Martin Fowler (together with Kent Beck) has coined the expression code smells or bad smells to express when object-oriented code violates common design rules and should be refactored in order to achieve higher maintainability (cf. [FBB+05, p. 75 ff.], [Vet13, p. 85], [SYA+13, p. 1144]).

These smells indicate, for example, duplicated code, classes that are too large, methods that have too many parameters or classes that are not necessary (cf. [FBB+05, p. 75 ff.]).

Over the years, additional code smells have been added (even fairly exotic ones, like "energy code smells" that indicate code that might increase power consumption; cf. [Vet13, p. 202]). Furthermore, recent research activities (e.g. [VMS+19], [PMA+19]) show an increasing interest in test smells, i.e. code smells that are typical for test code (cf. [vDMvK02])

For the scope of this work, only smells related to maintainability issues in production code are of interest (cf. sections 5.7 and 7.4.3).

5.6.3 Problems and criticism

Both maintainability metrics and code smells are not without criticism.

Ostberg & Wagner (2014), for example, express the opinion that the following metrics are inappropriate (cf. [OW14, p. 33 f.]):

- Cyclomatic Complexity (CC): here, the authors point out that the complexity of a control flow graph and the complexity to understand the source code are not necessarily the same.

- Halstead Suite: according to Ostberg & Wagner, this measure is not scientifically proven.
- Lines of Code (LOC): on the one hand, the different ways to count would make this metric difficult to compare (cf. page 20); additionally, a large but structured code segment could well be easier to understand than an unstructured small segment.
- Maintaiability Index (MI): this metric is built on the already criticized metrics; additionally, according to the authors, it is not easy to understand.

More generally speaking, just because a metric and a quality attribute (e.g., complexity and maintainability) seem to be correlated, this does not mean that this metric is really a measure for that attribute (cf. [FB20, p. 463 f.]).

In regards to code smells, Zhang et al. (2011) found out that only few of the smells originally proposed by Fowler & Beck have been scientifically investigated for their effect on maintenance effort. Therefore, in their opinion there is not be enough evidence to justify the use of code smells (cf. [ZHB11]).

Additionally, Sjøberg et al. (2013) have undertaken a study regarding the maintenance effort of 12 code smells. They conclude that "the 12 types of code smells turned out to be harmless with respect to effort" ([SYA$^+$13, p. 1154]).

5.7 Tool Support

In this section software tools necessary to collect data for test coverage, maintainability metrics and code smells shall briefly be described.

Tools for test coverage

Coverage tools work as follows (cf. [SL19, p. 291]): (a) before execution, either the source code or the byte code is instrumented, i.e. "probes" are inserted into the code that register when an element (statement, branch, ...) has been covered; (b) tests are executed and coverage data is collected; (c) from a protocol file the coverage data is analysed and statistics are generated.

Most tools support several programming languages, coverage types and export formats (like HTML, CSV or PDF; cf. [YLW09, SI11]).

According to Alemerien & Magel (2014), the results reported by these tools vary significantly; the authors assume that this might be – among other reasons – due to instrumentation type (byte- or source code), size and complexity of programs or different definitions of coverage (cf. [AM14, p. 17 ff.]).

Tools for metrics and code smells

In contrast to code coverage tools, for the collection of metrics and code smells the source code does not need to executed but can be analysed statically.

Tools for finding code smells (like PMD or FindBugs) typically use configurable rules or bug patterns to recognize violations (cf. [TEM13, p. 249]). PMD, for example, uses eight different rulesets in order to indicate more precisely what type of smell or flaw was found (e.g., Code style, Design, Documentation or Security; cf. [PMD21b]). One shortcoming often reported for this type of static analysis tools is that they report a vast amount of issues that are no true faults ("false positives": cf. [VZS$^+$12, p. 21]).

For the calculation of maintainability metrics, many tools are available that support different programming languages and sets of metrics (cf. [ACBV20]).

5.8 Summary

In lieu of a summary the following mindmap gives a basic overview of the many different levels and approaches for measuring software quality (cf. fig. 5.3 on the next page).

In this work, test coverage and maintainability metrics will be measured on unit test level. Several different coverage types exist.

For the calculation of test coverage, the code needs to be executed (dynamic testing) and, additionally, the control flow for all tests needs to be analysed (static analysis) – regardless if the tests are functional or structure based. For measuring maintainability metrics and code smells only static analysis is necessary. Luckily, tools exist that support the measurement both of metrics/smells and coverage.

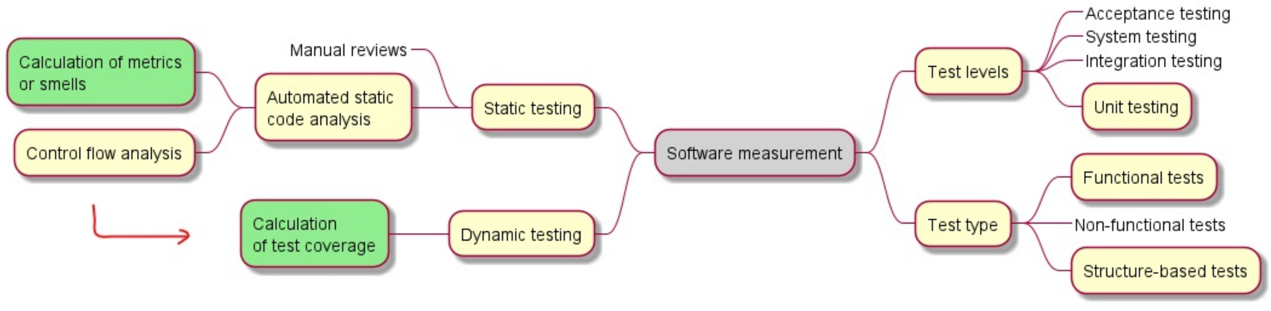


Figure 5.3: Software measurement - overview and focus of this work (own representation based on [SL19]; green: measurements central for this work; yellow: levels and approaches that are indirectly relevant).

The first part of this work has laid the foundations that were necessary for understanding background and scope of the investigation. The next part will describe the investigation itself.

Part II

Study

6 Literature research

The preliminary research question of this work was the following: Does unit test coverage influence maintainability of production code?

With this question the following chain of assumptions was pursued (cf. section 5.5.3):

1. With a high level of unit test coverage developers feel encouraged to refactor existing production code as part of perfective maintenance activities. Implicitly, it is expressed that developers will feel discouraged to refactor if the test coverage is low.

2. These code changes are likely to affect maintainability in a positive way.

The preliminary research question and its assumptions are reflected in the following draft of a model (cf. fig. 6.1):

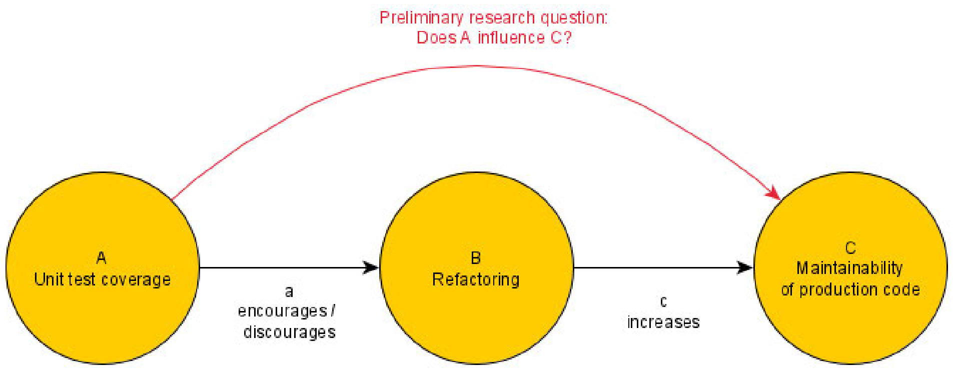


Figure 6.1: Draft of a model and preliminary research question (own representation).

Aim of the literature research is to find works related to both the preliminary research question and the assumptions on which it is based. With this, the model will be rounded up and a final research question / hypothesis will be derived.

The following search terms were used: (("test coverage" OR "code coverage" OR "api coverage" OR "coverage testing" OR "test gaps") AND ("software maintainability" OR "software evolvability" OR "software sustainability" OR "software maintenance" OR "refactoring" OR "software design")). No restrictions were made in regards to date of publishing.

The search was conducted using the following scientific libraries:

- ACM Digital Library (cf. [ACM21]): 978 results
- IEEE Xplore (cf. [IEE21]): 112 results
- ScienceDirect (cf. [Els21]): 532 results

Additionally, from those articles that were found useful the list of references were searched in order to find further related work that might be interesting.

6.1 Works related to preliminary research question

Bogner et al. (2018) have conducted an online survey with 60 participants to find out which maintenance assurance techniques are most frequently used in industry (cf. [BFWZ18, p. 127]). Interesting for the scope of this work is the fact that test coverage is the most mentioned metric for estimating and controlling maintainability (reported by one third of the participants; cf. [BFWZ18, p. 128]).

An study most similar to the preliminary research question was conducted by Harrison & Samaraweera (1996), who focus on the programming languages C++ and SML (cf. [HS96, p. 78]). The authors investigate relations between the number of test cases and design metrics (e.g., non-comment source lines (NCSL) or number of functions called by a program (N*)) and development metrics (e.g., time to fix known errors (TKE) or time to implement modifications (TMR; cf. [HS96, p. 79]) – i.e., relations to both internal and external maintainability metrics (cf. section 5.6.1).

They found positive correlations between the number of test cases and most design and development metrics at 0.05 level of significance for both programming languages (cf. [HS96, p. 80 ff.]).

That the correlations are positive instead of negative is insofar surprising as higher values for these metrics indicate worse maintainability (e.g., more time to implement changes means that a system is "rigid": cf. section 3.2.3). From the assumptions made in section 6 the opposite would be expected: i.e., higher test effort "should" be correlated with better maintainability – and that means lower values for both internal and external metrics would be expected.

The positive correlation could be explained with the influence of production code metrics on test effort. According to Toure et al. (2018), it would be too expensive to exert equal test effort for all software components. Instead, "developers have to focus on critical classes, requiring a relatively important testing effort, to ensure software quality" ([TBL18, p. 16]). To predict the expected test effort, metrics of production code (e.g., for size, complexity or coupling) can be used (cf. [TBL18, p. 17]).

6.2 Works related to first assumption

The first assumption is that developers feel encouraged to refactor when the test coverage is high (cf. page 35).

Kim et al. (2012) conducted interviews with more than 300 developers at Microsoft; additionally Leppänen et al.(2015) questioned 10 senior architects / developers at Finish software companies; both surveys had the goal to find out what kind of obstacles, risk factors and benefits interviewees associate with refactoring activity and in which situations refactoring edits are initiated (cf. [KZN12, p. 4 ff.], [LML+15]).

In regards to obstacles, indeed many developers said that low (regression) test coverage prevents refactoring activity (cf. [KZN12, p. 4]). Therefore, in addition to automated tests, a solid version control system was deemed a necessary precondition to refactoring (cf. [LML+15, p. 67]). Further obstacles named by the interviewees are the need to maintain backward compatibility (cf. [KZN12, p. 4]) and difficulties to convince customers or management (cf. [LML+15, p. 66]).

Most of the interviewed developers (75,41 %) expressed their concern about introducing new software faults and thus breaking builds as risks related to refactoring; additional reasons were increased testing cost, less time for other tasks, merge conflicts and that it is difficult to measure the value of refactoring (cf. [KZN12, p. 5]). The latter reason as well as the fear of introducing new bugs was also reported by Leppänen et al. (cf . [LML+15, p. 66 f.]).

As to the question what kind of situations made developers initiate refactoring activity, the following were named: short term changes (57 %), the need to fix a bug (46 %), poor readability or maintainability, code duplication and increasing dependencies (cf. [KZN12, p. 5 f.]) as well as performance issues or the wish for higher reliability and robustness (cf. [LML$^+$15, p. 66]).

From these investigations it can be concluded that there are many influencing factors in regards to refactoring activities. Low test coverage seems indeed to be regarded by developers as an important, but by far not the only obstacle.

That testing costs are seen as an risk factor could emphasize the precondition of solid coverage: if there are not enough tests, than prior to initiating refactoring edits more tests need to be implemented in order to gain enough courage that the introduction of new bugs will be noticed.

On the other hand, testing costs associated with refactoring can also arise because due to code changes the alignment between production and test code can break – even if the external behaviour remains unchanged (cf. [PBB16]). For example, white box tests are designed with knowledge of the internal structure (cf. section 5.2) – and refactoring edits aim to change and improve exactly these internals (cf. section 4.1). There are other reasons why the consistency between production code and tests can break – as a consequence, tests need to be adapted or added (cf. [PBB16]). Therefore, it is possible that the additional test effort related to refactoring could discourage developers from code improvements even if the initial test coverage is high.

Additionally, as Dinh-Trong et al. (2008) point out, unit tests alone are not always sufficient for detecting behaviour-changing faults as some refactoring edits may well affect more than one software component (cf. [DTGLR08, p. 255]).

6.3 Works related to second assumption

The second assumption is that after refactoring edits the maintainability of production code is higher than before (cf. page 35).

When Kim et al. (2012) and Leppänen et al. (2015) asked developers in regards to benefits of refactoring, the interview partners gave – among other reasons – the following answers: (a) improved readability (43 %), (b) improved maintainability (30 %), (c) higher ease to add new features (37 %), (d) fewer bugs (27 %) (cf. [KZN12, p. 5]) as well as (e) higher flexibility and reusability (cf. [LML+15, p. 66]).

It can be summarized that many developers share the assumption that refactoring activities can have a positive influence on maintainability – but are there any quantitative assessments to support this?

Kim et al., in addition to conducting interviews with developers, examined the impact of refactoring edits on Windows 7 in regards to number of dependencies and defects (cf. [KZN12, p. 7 f.]). They found that in binaries that were frequently refactored the increase in dependencies between two versions is significantly lower than in those that were not refactored (cf. [KZN12, p. 9]).

Furthermore, Kolb at al. (2006) reported improved maintainability metrics after several cycles of refactoring edits with the aim to reuse a legacy component. The average file size had decreased by -56.4 %, maximum function size by -79.1 % and Cyclomatic Complexity by -45.9 % (cf. [KMPY06, p. 128]).

On the other hand, a more recent longitudinal study on 25 open source projects conducted by Cedrim et al.(2016) found that only in 2.24 % of all cases refactoring edits succeeded to eliminate code smells whereas in 2,66 % new smells were introduced; in all other cases the smell density remained unchanged (cf. [CSGG16]).

These findings are backed up by research done by Bavota et al. (2015) who on three Java open-source projects investigated relations between refactoring and software quality. The authors report that only in 7 % of all cases developers succeeded in removing code smells (cf. [BDD+15]).

6.4 Other related works

Some other works were found that are not directly relevant to the research question or the two assumptions but still seem related and shall briefly be mentioned.

For example, quite a few works use static code analysis to predict maintainability (cf. [Vet13], [PGH+08], [THG20], [SOPF19])

Additionally, there are works that investigate relations between test and production code. For example, TUFANO ET AL. (2016) found relations between tests smells and code smells (cf. [TPB+16]).

6.5 Summary

Only one older work was found with an almost similar research question. HARRISON & SAMARAWEERA (1996) found positive correlations between the number of test cases and both internal and external maintainability metrics (cf. [HS96]). In addition, BOGNER ET AL. (2018) established that test coverage is the most favourite metric to estimate software maintainability (cf. [BFWZ18]).

With the results gained by literature research, the preliminary model (cf. fig. 6.1 on page 35) can be rounded up with the following relations (cf. fig. 6.2 on the next page):

- In addition to test coverage (relation a), refactoring seems to be influenced by several factors (perceived obstacles and benefits as well as the specific situation and pursued refactoring goals: relations d and e)
- Refactoring edits appear to have an backward impact on existing tests (relation b)
- According to TOURE ET AL. (2018), maintainability metrics are used to focus test effort on those software components that seem to be critical in some way (relation f). It can be assumed that an increased test effort goes hand in hand with a higher test coverage (relation g). This would explain the positive correlations between number of test cases and maintainability found by HARRISON & SAMARAWEERA.

Especially the three backward relations (b, f and g) have an impact on the research question. Previously, it was assumed that the relation between test coverage and maintainability is probably one-sided (cf. fig. 6.1 on page 35). Because of the named backward relations and the findings by HARRISON & SAMARAWEERA, this assumption no longer sounds plausible.

Instead, a bidirectional relationship between unit test coverage and maintainability is assumed and the following research question will be investigated in this work:

Is unit test coverage correlated with maintainability of production code?

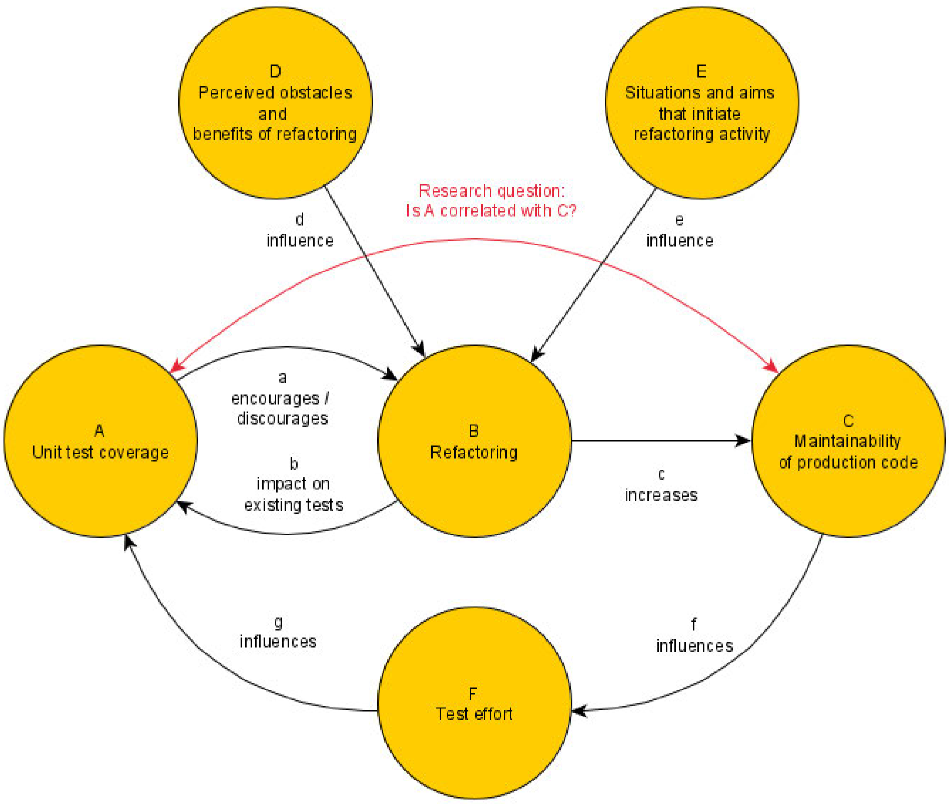


Figure 6.2: Adapted model and research question (own representation).

7 Research design

7.1 Hypothesis

According to Fenton & Bieman (2020), the first step in a scientifical investigation is to define the goal that is pursued – and that this goal can be expressed in form of a hypotheses (cf. [FB20, p. 139, 146]).

The aim of this investigation is to find an answer to the question, whether unit test coverage and maintainability are correlated (cf. section 6.5 on page 40).

In order to make maintainability quantifiable (cf. [FB20, p. 139]), internal maintainability metrics (cf. section 5.6.1 on page 29) were used. The following hypotheses were proposed:

- H0: There is no relationship between unit test coverage and internal maintainability metrics (null hypothesis).
- H1: There is a relationship between unit test coverage and internal maintainability metrics (alternate hypothesis).

Coverage types, metrics and code smells used in this investigation are described in more detail in sections 7.4.2 and 7.4.3 on page 47.

7.2 Study type

As Fenton & Bieman (2020) point out, the study type depends on

- the level of control researchers have on key variables,
- whether the events of interest have already taken place or not and

- the scale of an investigation (cf. [FB20, p. 135 ff.]).

In this investigation it was planned to analyse source code written by other development teams. As a consequence, it was not possible to control either of the variables (test coverage or maintainability metrics). Furthermore, the event of interest (writing of source code) had already taken place.

Lastly, the scale of this investigation depended on the effort needed to collect, analyse and depict the data of interest. Since many convenient open-source tools for the collection of coverage data and software metrics are available (cf. section 5.7 on page 31) and since a Java tool could be written with which it was possible to integrate the collected data (cf. fig. 7.3 on page 52), it was deemed practicable to analyse n > 30 development projects.

Therefore, a survey was chosen as the appropriate study type. According to Fenton & Bieman (2020), a survey is "a retrospective study of a situation to try to document relationships and outcomes. Thus, a survey is done after an event has occurred.[...] [S]urveys try to poll what is happening broadly over large groups of projects [...]" ([FB20, p. 136 f.]).

7.3 Selection of software projects

The target population of this investigation are all software projects that are: (a) open-source, (b) written in Java as source code language, (c) listed on GitHub for easy installation, (d) using Gradle or Maven as build tool and (e) using JUnit for unit tests.

GitHub is a development platform, which allows individual developers as well as organisations to build and maintain software. The source code of software projects is administrated within so called repositories (cf. [Git21c]). Within this work, the terms "repositories" and (software) "projects" are used synonymously.

The reason to restrict the selection of projects to those using Gradle and Maven as build tools is that it simplifies the collection of coverage data (cf. section 7.5 on page 50).

A first cursory search on Github to find out how many repositories are based on Java yielded almost six million results (cf. [Git21b]; accessed on 06.06.21). As a consequence, a sample had to be drawn.

According to Sahner (1971), three conditions must be met so that a sample allows conclusions to be drawn about the target population:

- Random sampling: each element of a population must be selectable with the same probability (cf. [Sah71, p. 14 f.]).
- Independent sampling: the selection of one element may not influence the probability with which other elements are selected (cf. [Sah71, p. 16 f.]).
- Normal distribution: The attributes of interest (in this case: unit test coverage and internal maintainability metrics) must be distributed normally within the target population (cf. [Sah71, p. 39]). If the sample is large enough ($n \geq 30$), this condition can be safely ignored (cf. [Sah71, p. 53]).

It was not possible to make any assertions as to whether coverage or maintainability metrics are distributed normally. As a consequence, a sample size of at least 30 repositories had to be achieved.

To select appropriate projects, the first approach was to send queries to the GitHub API (cf. [Git21a]) using search terms like "language:java" and randomly select repositories from the response. In the end, it proved to be not feasible to find enough projects which satisfied all selection criteria (see above).

The second approach was to look on MVNRepository for repositories which are tagged with "GitHub". Like GitHub, MVNRepository serves as a development platform, from which source code of software projects can be downloaded or directly integrated via build tools. Several different software categories are available; additionally, tags (like "spring", "security" or "database") help to identify interesting repositories (cf. [Mvn21]).

Using this approach yielded 15 pages with each page containing 10 repositories resulting in 150 projects altogether. However, it seemed unlikely that only such a small number of software projects is available. Additionally, some repositories did not contain any Java code.

Therefore, a different approach was chosen to achieve better results. A query to MVN-Repository with the search term "Java" was made, resulting in 40275 repositories. From these, 4175 repositories were tagged with "GitHub" (accessed on 07.06.21). Still, a problem remained, since only the first 50 pages (= 500 repositories) were listed.

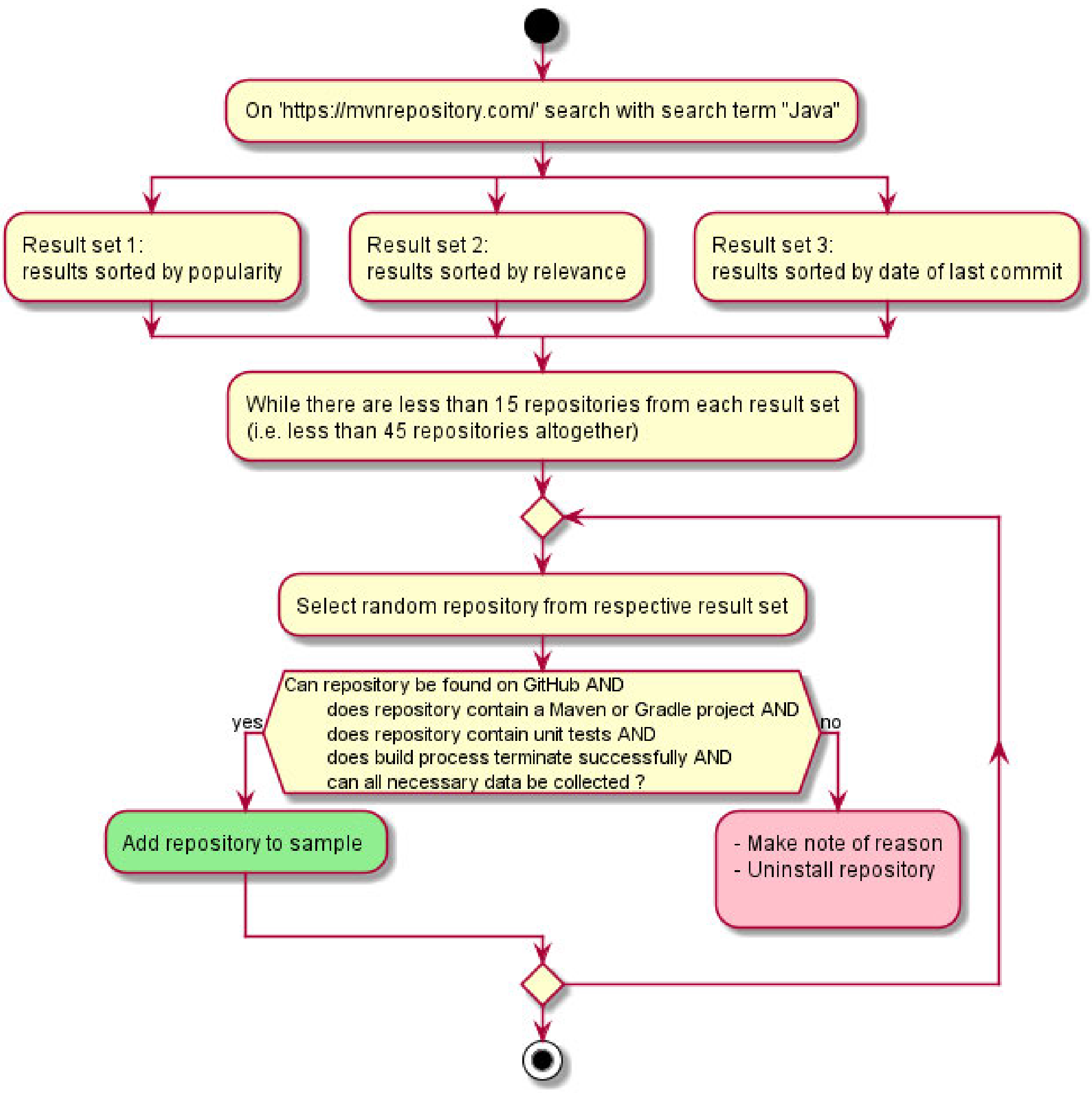

On 'https://mvnrepository.com/' search with search term "Java"
Result set 1: results sorted by popularity
Result set 2: results sorted by relevance
Result set 3: results sorted by date of last commit
While there are less than 15 repositories from each result set (i.e. less than 45 repositories altogether)
Select random repository from respective result set
Can repository be found on GitHub AND does repository contain a Maven or Gradle project AND does repository contain unit tests AND does build process terminate successfully AND can all necessary data be collected ?
yes
no
Add repository to sample
- Make note of reason
- Uninstall repository

Figure 7.1: Selection process for repositories.

As a consequence, a fully random selection was not possible since not all of the 4175 repositories had the same chance to be chosen. To mitigate this, the list of results was successively sorted by relevance, by popularity and by date of last commit. Each time, 15 repositories were randomly chosen resulting in a sample size of 45 (cf. fig. 7.1).

To gain a random sample, a self-written Java tool was used (RandomSampleGenerator). The repositories were then manually installed.

The condition of independent sampling, on the other hand, is fulfilled since the sample size is far less than 20 % of the target population (cf. [Sah71, p. 16 f.]).

7.4 Tools and examination conditions

A literature and internet research was conducted in order to identify works and webpages suitable for the selection of appropriate tools, coverage types, metrics and code smells. The following sources were used:

- Tools: Alemerien & Magel (2014), Ardito et al. (2020), Parfianowicz (2017), Shahid & Ibrahim (2011), Tomas et al. (2013) and Yang & Weiss (2009) (cf. [AM14, ACBV20, Par17, SI11, TEM13, YLW09]).
- Coverage types: Alemerien & Magel (2014), Hoffmann (2013), Shahid & Ibrahim (2011), Spillner & Linz(2019) and Yang & Weiss (2009) (cf. [AM14, Hof13, SI11, SL19, YLW09]).
- Maintainability metrics: Ardito et al. (2020) and Fenton & Bieman (2020) (cf. [ACBV20, FB20]).
- Code smells: Fowler et al. (2005) and PMD (2021) (cf. [FBB^{+}05, Mar09, PMD21b])

7.4.1 Selection of tools

The first step was to select suitable tools for the purpose of collecting unit test coverage, maintainability metrics and code smells.

Before selection, the following criteria were defined for all tools: (a) tool is still maintained in 2021, (b) open-source, (c) suitable for Java based projects, (d) availability of appropriate output formats (e.g. CSV), (e) good documentation and (f) availability of either command line interface, integration into an IDE (e.g. Eclipse or IntelliJ) or build tool integration.

Additional criteria were defined for the different tool types:

- Coverage and metric tools: they should support coverage types and metrics frequently discussed in literature.

- Code smell tools: selection of smells related to maintainability is possible.

Based on the before mentioned criteria the following tools were chosen, installed and tested for the intended purpose: (a) Coverage: **JaCoCo** (Version 0.8.8; cf. [Mou21b]); (b) Metrics: **MetricsReloaded** (Version 1.11; cf. [Lei21a]); (c) Code smells: **PMD** (Version 6.34.0; cf. [PMD21b]).

7.4.2 Selection of coverage types

The following coverage types were selected: branch-, instruction- and method-coverage, which in the following text will be referred to as BC, IC and MC.

All are supported by JaCoCo (cf. [Mou21a]) and are frequently discussed in literature (e.g. [Hof13, p. 206 ff.], [SL19, p. 193 ff.], [AM14], [YLW09]) – which were the criteria for selection.

7.4.3 Selection of maintainability metrics

As mentioned before, there are two different ways to measure maintainability: metrics and code smells (cf. section 5.6.1 on page 28). Selection criteria for both were that they had to be supported by the selected tools (MetricsReloaded and PMD) and – if possible – that they are in widespread use in scientific literature.

Metrics

Ardito et al. (2020) have conducted a literature review on the most frequently used maintainability metrics and respective tools for measurement. The authors identified 14 metrics (or metric suites) which had a citation count above the median (cf. [ACBV20, p. 13]).

Fortunately, MetricsReloaded can calculate most of these (cf. [Lei21b]). Some metrics were omitted because no calculation on class level was available (e.g. CC and CE).

Also omitted were metrics related to documentation (e.g. JLOC and CLOC), since documentation is not subjected to unit tests – and therefore not of interest for the scope of this work.

Finally, some metrics (DIT, NOC and MPC) were omitted because it showed later that in many cases these were not computable (resulting in "n/a"-values). In the final results, this was the case for 377 out of 5131 classes (7,35 %).

What remained were the following: most of the metrics employed by the Chidamber and Kemerer Suite (WMC, CBO, RFC, LCOM), two metrics from Halstead Suite (E and D) as well as LOC and STAT (a description of all metrics used in this investigation can be found in appendix A.1 on page 81). For all metrics used, higher values indicate lower maintainability.

Code smells

PMD provides the following code smell rule sets: Best Practices, Code Style, Design, Documentation, Error Prone, Multithreading, Performance and Security. A literature research gave no results to the question which rules / rule sets are especially related to maintainability.

Therefore, the following rules were selected based on experience and advice given by the authors of "Clean Code" (cf. [Mar09]).

- All rules from rule set Design expect *LoosePackageCoupling*, since this rule indicates a smell on package and not on class level (the selected rules are in following referred to as DN)
- 11 rules from rule set Best Practices (BP) related to maintainability (*ConstantsInInterface, DefaultLabelNotLastInSwitchStmt, ForLoopCanBeForeach, ForLoopVariableCount, LooseCoupling, OneDeclarationPerLine, UnusedAssignment, UnusedFormalParameter, UnusedLocalVariable, UnusedPrivateField, UseCollectionIsEmpty*)
- 15 rules from rule set Code Style (CS) related to maintainability (*BooleanGetMethodName, ConfusingTernary, FieldDeclarationsShouldBeAtStartOfClass, ForLoopShouldBeWhileLoop, IdenticalCatchBranches, LinguisticNaming, ShortClassName, ShortMethodName, ShortVariable, UnnecessaryLocalBeforeReturn, UnnecessaryModifier, UnnecessaryReturn, UselessQualifiedThis, UseShortArrayInitializer, UseUnderscoresInNumericLiterals*; a description of all Java rules can be found in [PMD21a]).

Code smells (i.e., violations of these rules) were not counted for individual rules but summarized for the rule sets named above (BP, CS, DN). A description of the process for the collection and integration of data is given in section 7.5 on the next page.

Thus, a total of 11 maintainability metrics are used in this investigation: BP, CBO, CS, D, DN, E, LCOM, LOC, RFC, STAT and WMC.

7.4.4 Statistical analysis technique

Strength of correlation (r)

To find an answer to the research question, correlations between the selected coverage types and maintainability metrics need to be measured.

The appropriate statistical analysis technique depends on several factors (cf. [FB20, p. 241 ff.]): (a) the aim of an investigation, (b) the scale of the measures (e.g., ordinal or ratio scale) and (c) the assumed distribution of data (normal or non-normal distribution).

In case of this work,

- the aim of the investigation is to explore a relationship
- for all attributes, the measurements are on a ratio scale (i.e., they can be expressed as numbers and a zero point exists: cf. [FHK+16, p. 16]) and
- it is not known how the data is distributed. Mostly, for software measures, it can be assumed that the data is not normally distributed (cf. [FB20, p. 252]).

Under these circumstances, appropriate analysis techniques are either Spearmen's rho or Kendall's tau as rank correlation coefficients (cf. [FB20, p. 244]). Since Spearmen's rho is more commonly used (cf. [FB20, p. 244, 253]), it was used for this investigation as well (in the following called r).

It takes values between -1 (negative correlation) and 1 (positive correlation; cf. [FHK+16, p. 135]). The strength can be grouped as follows (cf. [FHK+16, p. 130]):

- Weak correlation: $|r| < 0.5$ (in following: "weak r")
- Medium correlation: $0.5 \leq |r| < 0.8$ ("medium r")
- Strong correlation: $|r| \geq 0.8$ ("strong r")

Significance level (p)

As Fenton & Bieman point out, measures for association need to be accompanied by a statistical test in order to determinate the level of significance (cf. [FB20, p. 252]), in the following abbreviated as p.

The lower the significance level, the more likely findings are merely based on coincidence. Typically (and in this text), 0.01 indicates a high and 0.05 a low level of significance (in following: "high p" and "low p"). This means that values of $p \leq 0.05$ [0.01] indicate that with a probability of 95 % [99 %] a finding is not based on coincidence (cf. [Sah71, p. 101 ff.]) – in both cases, the alternate hypothesis will be accepted, but in the second case the confidence is higher.

7.5 Data collection and integration

The data was collected by using the tools mentioned above: JaCoCo (coverage data), MetricsReloaded (maintainability metrics) and PMD (code smells). All tools support CSV as export format.

Each tool was used in way to generate the respective CSV file with as little effort as possible: (a) For JaCoCo, a Gradle or Maven task was defined and executed as part of the built process; (b) For MetricsReloaded, the chosen metrics were calculated within IntelliJ IDE and (c) PMD was executed via command line (cf. fig. 7.2 on the following page).

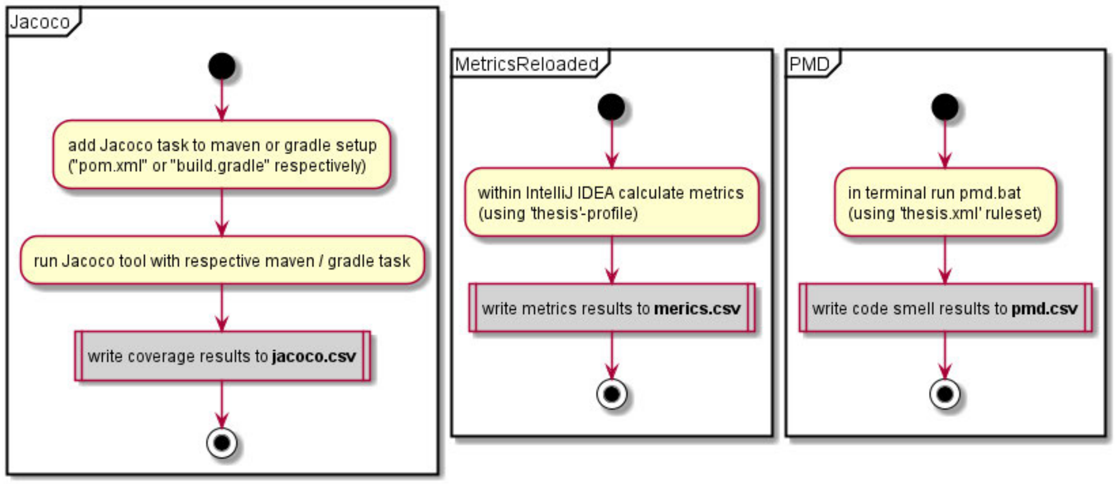

Jacoco
add Jacoco task to maven or gradle setup ("pom.xml" or "build.gradle" respectively)
run Jacoco tool with respective maven / gradle task
write coverage results to **jacoco.csv**
MetricsReloaded
within IntelliJ IDEA calculate metrics (using 'thesis'-profile)
write metrics results to **merics.csv**
PMD
in terminal run pmd.bat (using 'thesis.xml' ruleset)
write code smell results to **pmd.csv**

Figure 7.2: Processes for collecting data from repositories (executed manually).

The next step was to integrate the data from the previously generated CSV files into a common results file. For this, another self-written Java tool was used (CsvIntegrator); The integration process consists of several steps (cf. fig. 7.3 on the next page):

1. The coverage data is read; since MetricsReloaded apparently does not calculate metrics for anonymous classes, these classes are skipped. For all other classes the package name together with the name of the class is used as an unique ID. The coverage data for each class is linked to its ID.

2. The metrics data is read and the ID for each class extracted; if coverage data exists for this ID then the metrics data of this class is incorporated.

3. In the same way code smell data is handled, the difference being that code smells are summed up for each rule set (BP, CS, DN) and not considered individually.

4. After all data is extracted (coverage, metrics and code smells) and linked to unique IDs, a validation is carried out, because it showed that for some classes not all data was available. This was due to the following reasons: (a) For some classes only bytecode existed instead of source code. While JaCoCo and PMD could handle this and still generate data, MetricsReloaded could not. (b) Other cases where data could not be retrieved by all tools were interfaces with default methods.

5. Consequently, only those classes are taken into account for which all data could be retrieved; these datasets are sorted by ID and written to a CSV results file.

6. Finally, a log file with more details is generated (cf. appendix A.3).

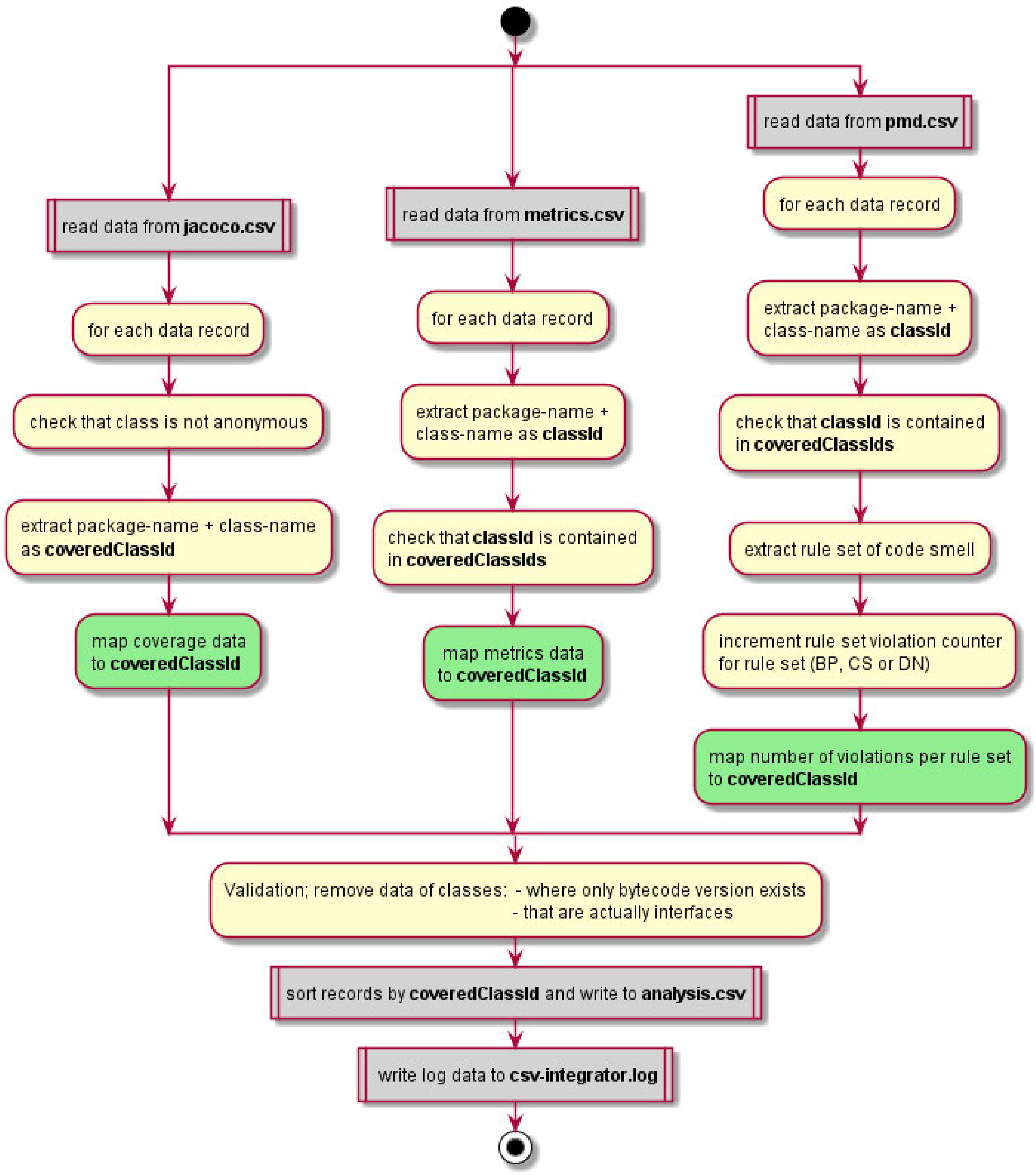

read data from **jacoco.csv**
for each data record
check that class is not anonymous
extract package-name + class-name as **coveredClassId**
map coverage data to **coveredClassId**
read data from **metrics.csv**
for each data record
extract package-name + class-name as **classId**
check that **classId** is contained in **coveredClassIds**
map metrics data to **coveredClassId**
read data from **pmd.csv**
for each data record
extract package-name + class-name as **classId**
check that **classId** is contained in **coveredClassIds**
extract rule set of code smell
increment rule set violation counter for rule set (BP, CS or DN)
map number of violations per rule set to **coveredClassId**
Validation; remove data of classes: - where only bytecode version exists - that are actually interfaces
sort records by **coveredClassId** and write to **analysis.csv**
write log data to **csv-integrator.log**

Figure 7.3: Process for integration of data, using a self-written Java tool (CsvIntegrator).

The CsvIntegrator tool was thoroughly tested on several different repositories until it could handle all cases described above.

7.6 Data evaluation

The data was analysed and visualized by using R (version 4.1.1; cf. [R F21]). Several R scripts were written and used, for the combined analysis data of all classes as well as for the individual repositories with more than 30 classes:

1. Firstly, an R script was used to calculate the significance and strength of correlations between all three coverage types and all 11 metrics (33 correlations altogether) and to write the results to CSV tables (**cors.r**)
2. The generated CSV tables were then used in other scripts,
    a) to count the number of significant correlations and again to write the results to other CSV tables (**countcors.r**) and
    b) to generate plots for visualizing significant correlations (**corplots.r**)
3. Finally, other scripts were used for generating plots to visualize the distribution for the different coverage, smell and metric types (**boxplots.r**) and for generating plots to visualize more general aspects of this study (**repoinfos.r**)

8 Results

In 43 cases of the randomly chosen repositories it was not possible to extract the data necessary for analysis, the main reason being build errors (cf. fig. 8.1).

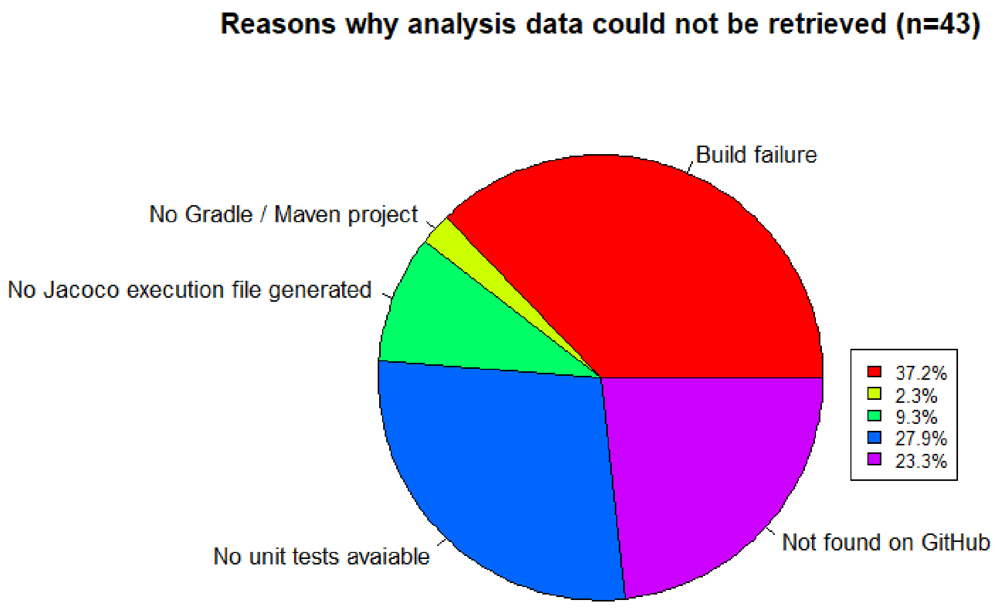


Figure 8.1: Reasons why analysis data could not be retrieved from repositories.

As a consequence, additional repositories had to be chosen until 45 projects with all necessary data (coverage, metrics and code smells) could be retrieved.

The selected repositories included in the sample contained 5131 classes for which all data (coverage, metrics and code smells) could be retrieved. On average, 114 analysable classes were included in a single repository. Four projects contained more than 600 classes (P9, R1, R9, R12) and 14 projects had less than 10 classes (cf. fig. 8.2). Tables with repository details can be found in appendix A.2.

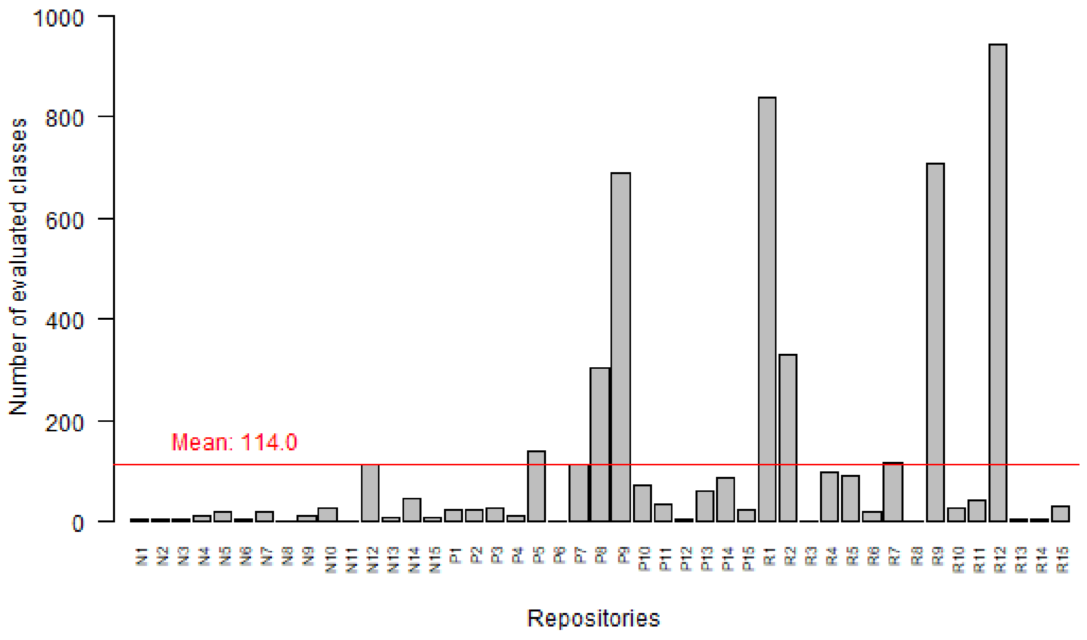


Figure 8.2: Number of evaluated classes per repository (n = 5131). N, R and P indicate the order of the result set, from which the repositories were selected: sorted by last commit (N), by popularity (P) or by relevance (R) .

8.1 Description of distributions

For a visual description of distributions, box-plots were used (cf. [FB20, 243]). In the first graphic, the distribution of coverage is visualized (cf. fig. 8.3 on the following page).

Coverage

It can be seen that the most common value (median) for BC was zero and that 75 % of all repositories had a coverage roughly below 50 %.

For both MC and IC, the coverage was much higher; in most cases, the projects had a coverage of slightly below 40 %; one quarter of all repositories had a coverage between 40 and 90 %.

In all cases, some projects managed to gain a coverage of 100 %.

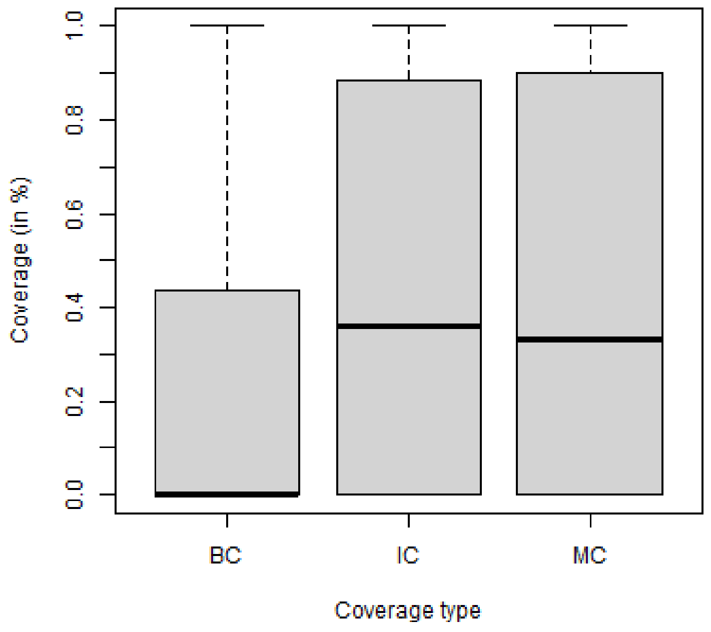


Figure 8.3: Distribution of coverage (all repositories).

Code smells

In order to visualize the full distribution of code smells and metrics, a logarithmic scale was used.

It can be seen that in most cases only few smells were found (for all smells the median was 0). But, in opposite to coverage, a high number of outliers existed, with values up to 1000 smells in some classes (cf. fig. 8.4).

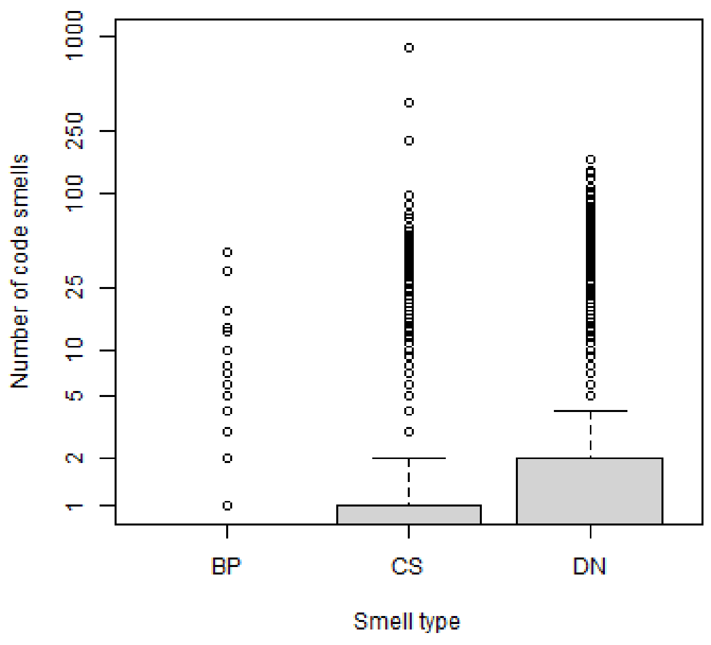


Figure 8.4: Distribution of code smells (all repositories).

Metrics

In regards to metrics, most median values were below 50 with the exception of E, for which values around 5000 were achieved. Similar to smells, many outliers existed (cf. fig. 8.5 on the following page).

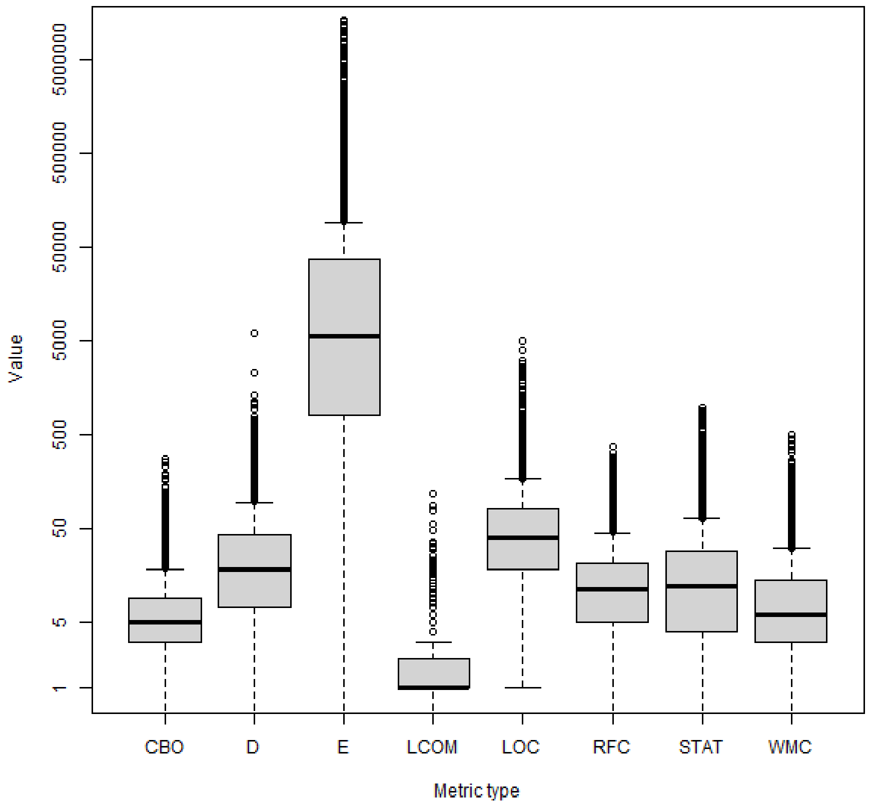


Figure 8.5: Distribution of metrics (all repositories).

8.2 Description of correlations

8.2.1 Overall results for all repositories

On the level for all projects (n=5131 classes) 20 out of 33 correlation were significant; 18 were highly significant (0.01 level) and two correlations had lower significance (0.05 level). The strength of all correlations was weak ($|r| < 0.5$; cf. section 7.4.4 on page 49). All significant correlations were positive expect two (cf. tab. 8.1 on the following page).

What stands out is that for BC all 11 correlations were highly significant whereas for the other coverage types only few relations with equal significance could be identified (IC: 3; MC:4). IC is the only coverage type with significant but very weak negative correlations.

CBO, DN and LCOM are the only metrics for which significant correlations with all coverage types were found.

Coverage	BC	BC	IC	IC	MC	MC
Metric	p	r	p	r	p	r
BP	**0,0000**	0,11	0,0403	-0,03	0,0986	-0,02
CS	**0,0000**	0,23	0,5325	-0,01	0,3230	0,01
DN	**0,0000**	0,30	0,0225	0,03	**0,0003**	0,05
CBO	**0,0000**	0,36	**0,0000**	0,26	**0,0000**	0,28
D	**0,0000**	0,42	0,1271	-0,02	0,4140	0,01
E	**0,0000**	0,40	0,1067	-0,02	0,3790	0,01
LCOM	**0,0000**	0,15	**0,0000**	0,11	**0,0000**	0,10
LOC	**0,0000**	0,38	0,4853	-0,01	0,1585	0,02
RFC	**0,0000**	0,40	0,6208	0,01	**0,0084**	0,04
STAT	**0,0000**	0,40	**0,0045**	-0,04	0,8723	0,00
WMC	**0,0000**	0,40	0,1656	-0,02	0,3616	0,01

Table 8.1: Correlations for all repositories; bold = high (0.01 level), underlined = low significance (0.05 level), red = significant negative correlations.

For a visual description, in the following graphic all significant relations are displayed (cf. fig. 8.6):

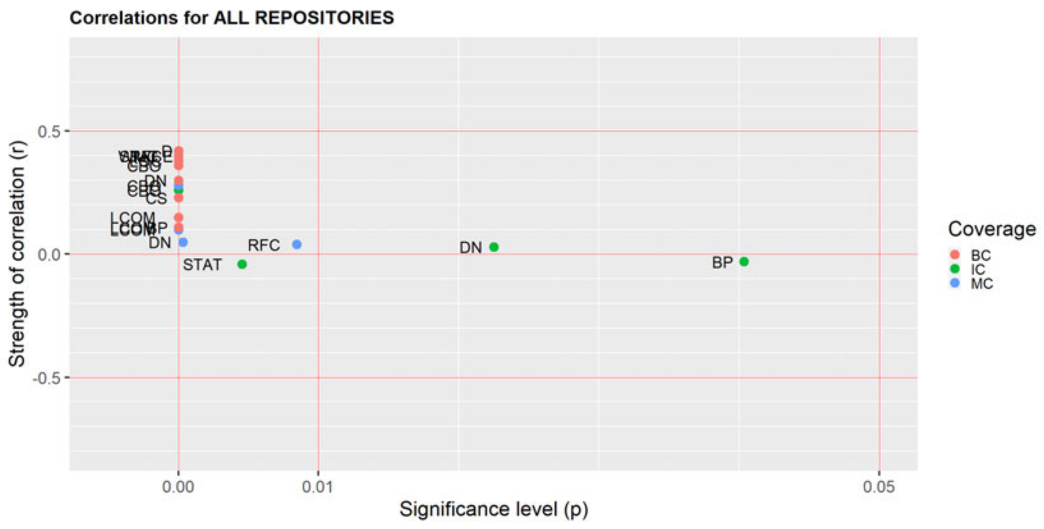


Figure 8.6: Significant correlations for all repositories.

8.2.2 Results for repositories with more than 30 classes

Correlation patterns

Since the significant correlations on an overall level for all repositories were weak and mostly positive it was then decided to have a closer look at individual projects (multi-case-study). The rationale behind this decision was that maybe in some repositories more and stronger negative correlations do indeed exist but which are – on the level of all 45 projects – levelled out by other repositories with positive correlations.

To this end, it was decided to have a look on all projects with more than 30 classes. 18 of 45 repositories were found to match this criterion.

Repository	Sum r	Pos r	Neg r	Weak r	Medium r	Strong r	Low p	High p
R9	31	31	0	31	0	0	3	28
R12	30	30	0	30	0	0	3	27
P8	29	10	19	29	0	0	3	26
R1	29	12	17	25	**4**	0	2	27
R2	26	9	17	21	**5**	0	0	26
R4	25	11	14	21	**4**	0	2	23
P13	20	9	11	8	**12**	0	3	17
N12	18	18	0	18	0	0	12	6
P14	18	11	7	10	**8**	0	5	13
P9	16	13	3	16	0	0	4	12
R5	12	12	0	11	**1**	0	2	10
R11	12	1	11	7	**5**	0	4	8
P5	11	8	3	8	**3**	0	2	9
P11	10	10	0	1	**9**	0	0	10
N14	9	9	0	5	**4**	0	2	7
P10	6	6	0	6	0	0	6	0
R7	6	6	0	6	0	0	1	5
P7	3	3	0	3	0	0	3	0
Sum	311	209	102	256	55	0	57	254

Table 8.2: Number of significant correlations in repositories with n > 30 classes; sorted by sum of relations.

As can be gained from table 8.2, the number of significant correlations (sum r) varies greatly: between three (P7) and 31 (R9). Remarkably, half of the repositories had negative correlations; roughly one third (102 of 311: 32,8 %) of all relations were negative – in case of R11, almost all relations were negative. Additionally, in 10 projects correlations with medium strength were found (55 of 311: 17,7 %).

In terms of different coverage types, it can be noticed that most correlations were found for BC (145, of which 42 had medium strength = 29,0 %) – and all were positive (cf. table 8.3). In contrast, 91 correlations for IC were found (medium strength: 13,2 %; negative: 64,8 %) and 75 for MC (1,3 %; 57,3 %).

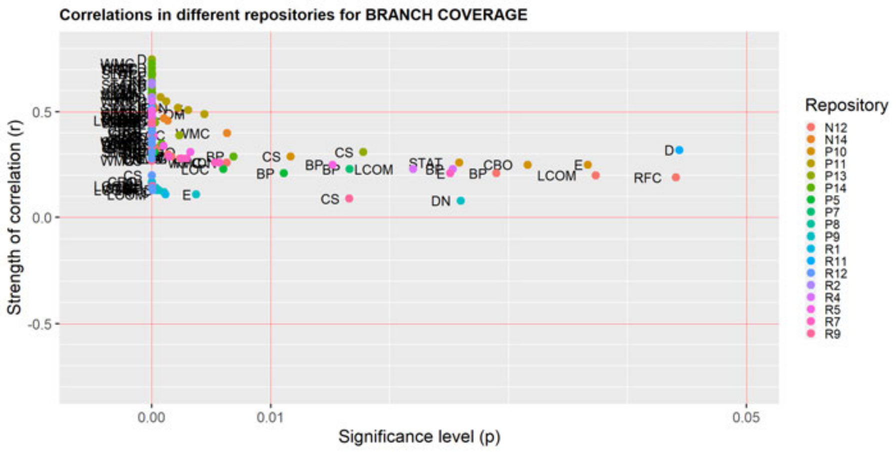
Coverage	Sum r	Pos r	Neg r	Weak r	Medium r	Strong r	Low p	High p
BC	145	145	0	103	**42**	0	17	128
IC	91	32	59	79	**12**	0	22	69
MC	75	32	43	74	**1**	0	18	57

Table 8.3: Number of significant correlations for different coverage types (for repositories with n > 30 classes; sorted by sum of relations).

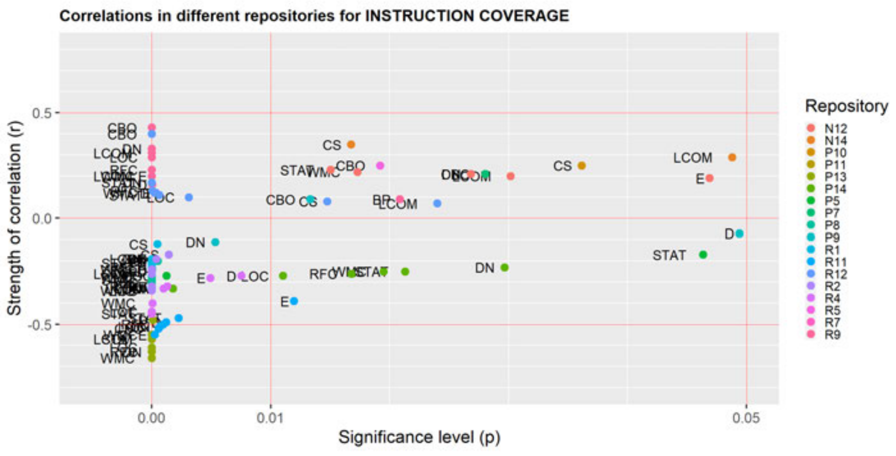
These findings are visualized in fig. 8.8 on the following page. In regards to metrics, the results are displayed in the following table (cf. 8.4).

Coverage	Sum r	Pos r	Neg r	Weak r	Medium r	Strong r	Low p	High p
STAT	34	22	12	27	**7**	0	4	30
WMC	34	21	13	27	**7**	0	5	29
LOC	33	18	15	29	**4**	0	3	30
E	32	22	10	24	**8**	0	5	27
D	31	20	11	23	**8**	0	3	28
DN	31	20	11	23	**8**	0	5	26
RFC	30	19	11	23	**7**	0	2	28
CBO	26	24	2	23	**3**	0	7	19
LCOM	25	15	10	23	**2**	0	7	18
CS	23	18	5	22	**1**	0	8	15
BP	12	10	2	12	0	0	8	4

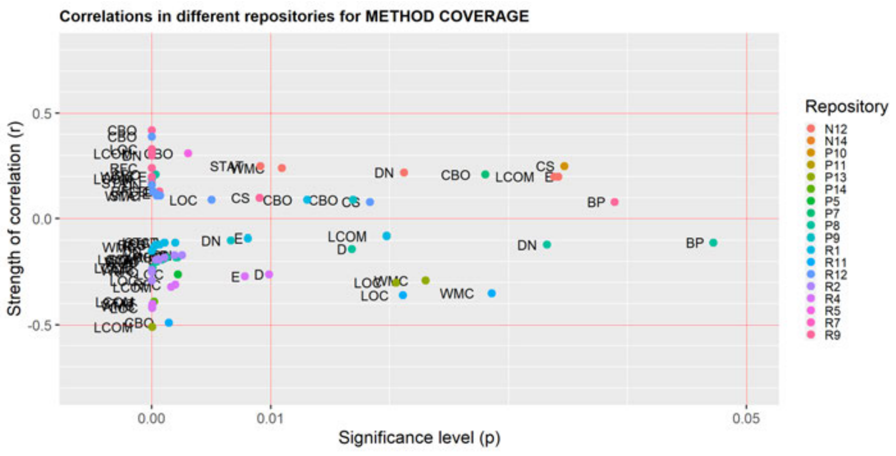
Table 8.4: Number of significant correlations for different metrics (for repositories with n > 30 classes; sorted by sum of relations).



(a) Branch coverage



(b) Instruction coverage



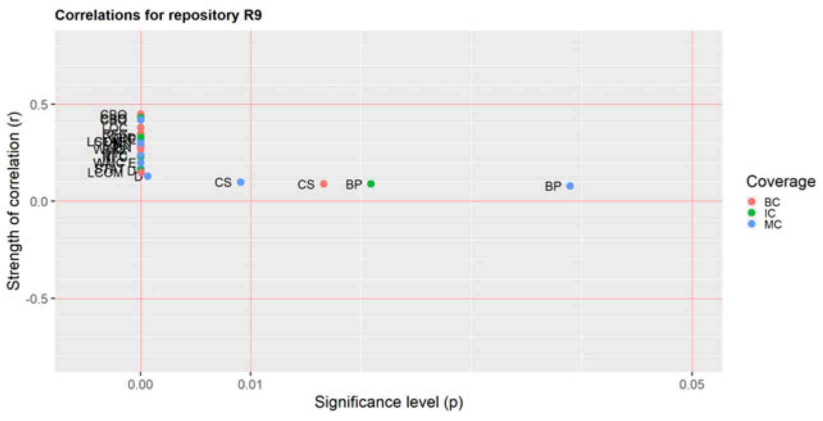
(c) Method coverage

Figure 8.8: Significant correlations for different coverage types (for repositories with n > 30 classes).

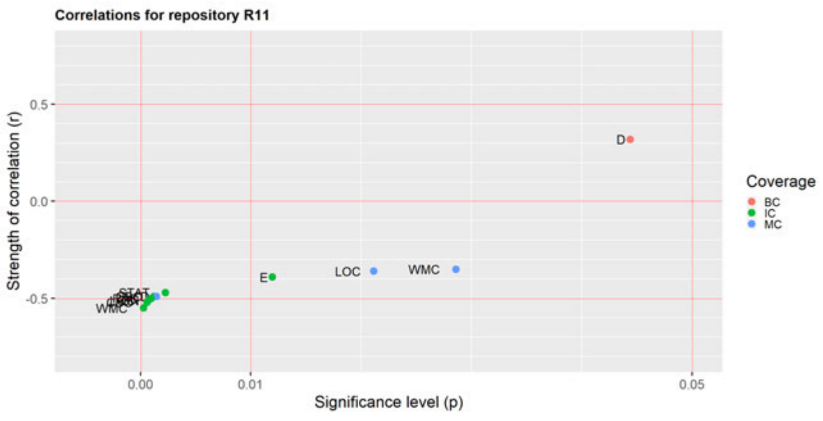
In regards to metrics, the first thing that attracts attention is that – in contrast to repositories and coverage types – there seems to be a more even distribution:

- The first seven metrics have roughly the same amount of correlations: between 30 (RFC) and 34 (STAT).
- Additionally, most metrics have more or less the same percentage of negative relations: 30 - 40 %. Higher values were only found for LOC (45.5 %), lower values for CS (21,7 %), BP (16,7 %) and CBO (7,7 %).

That the distribution of metrics is more even suggests that coverage type and repository have a stronger influence on correlation than metric types. Therefore, two repositories with "extreme" results shall be viewed more closely: one where all correlations were positive (R9) and one where mostly negative correlations occurred: R11 (cf. fig. 8.9):



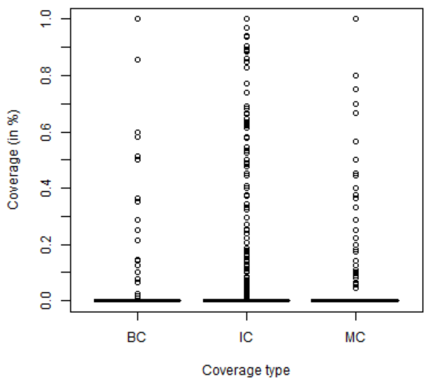
(a) Repository with only positive correlations



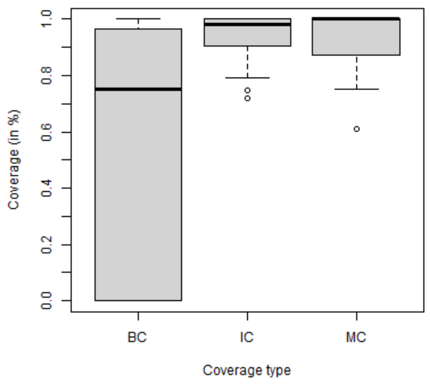
(b) Repository with mostly negative correlations

Figure 8.9: Repositories with "extreme" results.

When looking at coverage distribution for both repositories, the differences are obvious (cf. fig. 8.10):



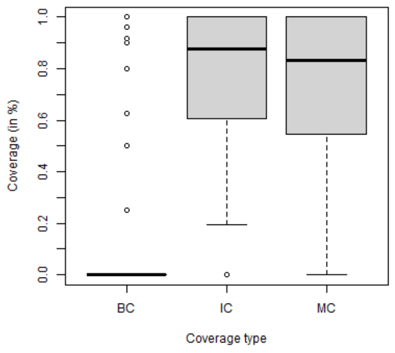
(a) Coverage for repository with only positive correlations (R9)



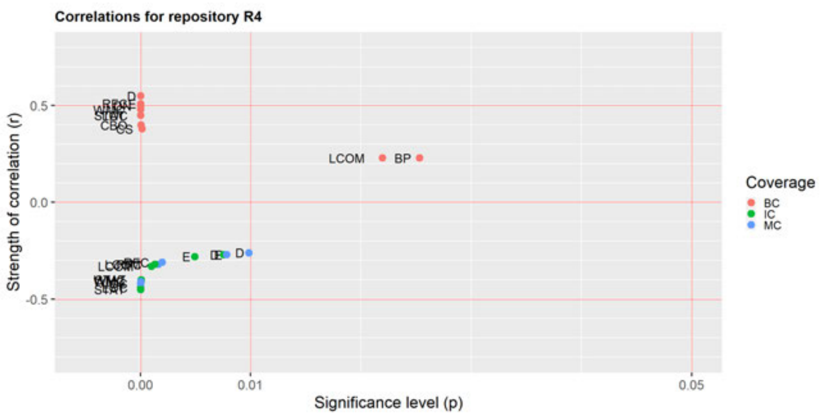
(b) Coverage for repository with mostly negative correlations (R11)

Figure 8.10: Coverage for repositories with "extreme" results.

The first pattern of low coverage and positive correlations (in following: LCPC-pattern) can be found in six other repositories as well, e.g. R12, N12 or R7 (cf. appendix A.4.1). The second pattern of high coverage and negative correlations (HCNC-pattern) could not be found with the same clarity as in R11. Instead, a mixture of both patterns seems to be more typical: LCPC for branch coverage (LCPC $_{(BC)}$) and HCNC for instruction and method coverage (HCNC $_{(IC/MC)}$) (cf. fig. 8.11).



(a) Coverage for repository R4



(b) Correlations for repository R4

Figure 8.11: Mixed LCPC $_{(BC)}$/HCNC $_{(IC/MC)}$ pattern in repository R4.

This pattern was observed for repositories P5, P14 and R13 as well (cf. appendix A.4.3 on page 90). For six repositories, no common pattern was recognized (cf. appendix A.4.4 and A.4.5). Summarized, the following patterns were found in repositories with more than 30 classes (cf. tab. 8.5):

Pattern	Number of repositories	Repositories
LCPC	7	N12, P7, P9, P10, R7, R9, R12
HCNC	1	R11
Mixed LCPC $_{(BC)}$ / HCNC $_{(IC/MC)}$	4	P5, P14, R4, R13
No apparent pattern	6	N14, P8, P11, R1, R2, R5

Table 8.5: Patterns in repositories with more than 30 classes.

For the description of these patterns only coverage types were taken into account. This is due to the fact that for coverage both a lower and a upper limit exist – whereas for maintainability metrics there is no real upper limit (cf. outliers in fig. 8.5 on page 58). Additionally, no strict definition was used; instead, it was roughly estimated whether a coverage distribution was higher or lower than the average (cf. fig. 8.3 on page 56).

Metric classes with strongest positive and negative correlations

On a higher level, which metric classes dominate positive and negative correlations? In order to find an answer to this question, the top ten significant correlations with highest positive / negative strength were identified (cf. tables 8.6 and 8.7 on the following page):

Interestingly, in the majority of all cases positive correlations could be assigned to code size – and only one coupling metric was found. In comparison, the majority of negative correlations was due to design-related metrics (design size, coupling, cohesion and design issues).

In summary, 45 projects with 5131 classes were analysed and the following results were obtained in regards to the research question.

On the level for all projects (n=45; survey):

(1) 20 out of 33 significant correlations were found, of which 18 were highly significant (0.01 level). (2) The strength of all correlations was weak. (3) All correlations were

Repository	Coverage	Metric	Classification	p	r
P13	BC	D	Code size	0,0000	0,75
P14	BC	WMC	Design size	0,0000	0,73
P11	BC	D	Code size	0,0000	0,71
P11	BC	WMC	Design size	0,0000	0,71
P14	BC	LOC	Code size	0,0000	0,71
P11	BC	STAT	Code size	0,0000	0,70
P11	BC	RFC	Coupling	0,0000	0,70
N14	BC	D	Code size	0,0000	0,69
P11	BC	LOC	Code size	0,0000	0,68
P14	BC	STAT	Code size	0,0000	0,68

Table 8.6: Top 10 significant positive correlations (for repositories with n > 30 classes); classification of metrics according to [FB20].

Repository	Coverage	Metric	Classification	p	r
P13	IC	WMC	Design size	0,0000	-0,66
P13	IC	RFC	Coupling	0,0000	-0,63
P13	IC	DN	Design issues	0,0000	-0,63
P13	IC	LOC	Code size	0,0000	-0,61
P13	IC	STAT	Code size	0,0000	-0,57
P13	IC	LCOM	Cohesion	0,0000	-0,57
P13	IC	E	Code size	0,0000	-0,55
R11	IC	WMC	Design size	0,0000	-0,55
R11	IC	CBO	Coupling	0,0000	-0,52
P13	MC	LCOM	Cohesion	0,0000	-0,51

Table 8.7: Top 10 significant negative correlations (for repositories with n > 30 classes); classification of metrics according to [FB20] and [PMD21b].

positive except two. (4) For BC all correlations were highly significant whereas for the other coverage types only few relations had equal significance. (5) IC was the only coverage type with negative correlations. (6) DN and LCOM were the only metrics that had significant correlations with all coverage types.

On the level for repositories with more than 30 classes (n=18; multi-case study):

(7) The number of significant correlations varies greatly between projects; half of the repositories had negative correlations and in ten repositories correlations were of medium strength. (8) Most correlations were found for branch coverage (and with a higher percentage of medium strength relations). (9) All correlations for BC were positive whereas IC and MC had around 60 % negative relations. (10) For metric types, the number of correlations was more evenly distributed than for repository or coverage type. (11) Several correlation patterns were discovered, the most common being LCPC, followed by a mixed LCPC/HCNC-pattern; only one case of a relatively "pure" HCNC-pattern was found. (12) The strongest positive correlations were dominated by size-related metrics whereas the strongest negative correlations were related to metrics of design.

9 Discussion

9.1 Discussion of results

More than half of all possible correlations were of high significance even if the strength of relations was weak (results 1, 2).

That the number of correlations were more evenly distributed for metric types (10) may be seen as evidence that coverage types and development practices in individual software projects have a higher influence on the relation between coverage and maintainability than the type of metrics.

Interesting is the finding that the coverage types differ in respect to number, strength and direction of correlations (3, 8, 9). That only few relations for IC and MC were found on level of all repositories can be explained with the fact that on level of individual projects roughly half of all relations for IC and MC were negative (9) – thus, on a higher level, the negative and positive correlations cancel each other out.

There seem to be more and stronger positive relations, both on the level of all 45 repositories as well as on the level for projects with more than 30 classes (3, 8). This is a similar result to what HARRISON & SAMARAWEERA (1996) reported ([HS96]). This can be interpreted as an indication that developers increase their test effort for those components that were rated as being in some way "critical" (cf. [TBL18]; cf. fig. 6.2 on page 41). That for BC all correlations were positive (9) – in contrast to IC and MC – and that LCPC was the most frequent pattern (11) can be seen as a hint that especially BC is used with the aim of thoroughly testing these complex modules.

That IC and MC had a much higher percentage of negative correlations (9) raises the question whether these coverage types indeed increase the confidence to refactor with the aim of better maintainability, as claimed by several authors (cf. section 5.5.3 on page 27).

The mixed LCPC $_{(BC)}$ / HCNC $_{(IC/MC)}$ pattern, found in four cases, may be seen as a hint that in software projects both relations can be observed simultaneously: low BC as an indicator that only critical components were tested in more depth – and a high IC / MC with the effect of increasing confidence for refactoring edits.

Interesting is the result that 70 % of the strongest *positive* correlations were dominated by code-size metrics (12). This could be seen as a hint that developers focus their in-depth testing activities on very large components in particular. In comparison, an equal percentage of the strongest *negative* relations could be assigned to metrics of design like coupling or cohesion. This might support the assumption that high test coverage encourages refactoring activities with positive effects on design attributes. This finding is related to works by other researchers who reported less dependencies and decreased complexity after refactoring edits (cf. [KZN12, p. 9], [KMPY06, p. 128]).

All in all, the evidence for a negative correlation between test coverage and maintainability was much weaker than for a positive correlation. This may be due to the many factors that influence refactoring activities (cf. [KZN12], [LML+15]).

9.2 Threats to validity

According to Wohlin et al. (2000), several problems can occur which might threaten the validity of research results (cf. [WRH+00]; cited by [FB20, p. 143 f.]):

- Conclusion validity: For this type of validity, the appropriate statistical test needs to be employed, the results need to be statistically significant and the sample size large enough. Additionally, the number of variables for which relations are investigated should not be too large.
  In this respect, no obvious threats were discovered, since only relations between pairs of two variables were investigated. For sample size and statistical test, advises in literature have been followed closely. Finally, the results were on a significance level of 0.05 or above. Still, for some observations (e.g. for the proposed patterns) only few supportive cases were found. Therefore, these propositions should be tested with a larger sample size.

- Construct validity: Most important, for this validity, is to use meaningful and robust measures.
  In this investigation, only those maintainability metrics were used, that have been

most frequently discussed in scientific literature. Still – as described in section 5.6.3 on page 30 – some of these metrics have been criticised: CC (on which WMC relies), the Halstead Suite (of which D and E were used) or LOC. In regards to code smells, suitable rules have been selected in accordance with only one source and, in addition, based on own experience. But smells were also subject of criticism (cf. section 5.6.3). Therefore, the construct validity might – at least partly – be threatened.

- Internal validity: For this validity type, some theory must exist as to how the variables under investigation are related to each other. Additionally, before the onset of an investigation, the independent variable should be measured.
  In this context, an explanatory model based on a literature findings has been proposed (cf. fig. 6.2 on page 41). It was stated that both variables are supposedly correlated; i.e. a *two-sided relationship* was assumed and reasons given for this. Thus, both variables are dependent and independent at the same time. In addition, the aim of this investigation (a survey followed by a multi-case study) was to document relations in retrospective. Therefore, it was not possible to measure the independent variable(s) beforehand. What might be criticised, though, is that it was nevertheless tried to make conclusions in regards to *one-sided relations* (for example, a positive correlation between BC and maintainability metrics means that developers probably have tested complex components more thoroughly).

- External validity: This kind of validity refers to the generalizability of an investigation. The results are generalizable, if the study environment and the study objects are close to real-world conditions.
  To achieve a higher generalizability – regarding the population of Java projects listed on GitHub – in this investigation the study objects (repositories) were selected randomly and the sample size was above 30 as proposed in literature (cf. section 7.3 on page 43). As described in that section, it was not possible to gain a fully random sample. Additionally, only a sub-population of all repositories listed on GitHub was selected due to technical reasons (requirement: Maven or Gradle as build tools). For the same reason, the selection was carried out on MVNRepository and not directly on GitHub. Therefore, the external validity might be impaired.

10 Conclusions and future work

With 18 out of 33 correlations with high (0.01 level) and two correlations with low (0.05 level) significance the alternative hypothesis (H1) is accepted: there is indeed a relationship between unit test coverage and maintainability metrics – although the strength of this correlation is weak on an overall level.

Most correlations were positive and towards branch coverage. This can be seen as an indication that developers increase testing effort for "critical" components in particular – and that branch coverage is important for this purpose.

Negative correlations – indicating positive effects of high test coverage on maintainability – were only found for few of the projects. This may be due to the many factors that influence refactoring activities. Still, it could be established that both instruction- and method-coverage are of higher relevance here.

On these grounds, it can be concluded that in regards to the proposed model there seems to be a much stronger relation in "backward" (test coverage $\leftarrow$ maintainability) than in "forward" direction (test coverage $\rightarrow$ maintainability).

Based on this work, for future research it seems of interest to explore the reasons behind the finding that individual projects differ greatly in regards to number, strength and direction of correlations between coverage and maintainability metrics. This could be done by means of qualitative research, e.g. by interviewing the involved maintainers and developers of repositories with "extreme" correlations (only positive, only / mostly negative, no correlations at all) in regards to development model, used development tools, significance of refactoring activities, views on software quality / maintainability etc.

Additionally, it might be interesting to expand the research on other metrics, coverage types or other types of software projects (proprietary software instead of open-source; different sectors of industry, etc.).

Bibliography

[ACBV20] Luca Ardito, Riccardo Coppola, Luca Barbato, and Diego Verga. A Tool-Based Perspective on Software Code Maintainability Metrics: A Systematic Literature Review. *Scientific Programming*, 2020(-):1–26, aug 2020.

[ACM21] ACM. ACM Digital library. https://dl.acm.org/, 2021. Accessed: 2021-09-09.

[AM14] Khalid Alemerien and Kenneth Magel. Examining the effectiveness of testing coverage tools: An empirical study. *International Journal of Software Engineering and its Applications*, 8(5):139–162, 2014.

[BDD$^+$15] Gabriele Bavota, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Fabio Palomba. An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software*, 107:1–14, sep 2015.

[Bec01] Kent Beck. *Extreme Programming explained. Embrace change. 7th printing.* Addison-Wesley, 2001.

[BF11] Dustin Boswell and Trevor Foucher. *The Art of Readable Code. Simple and Practical Techniques for Writing Better Code.* O'Reilly Media, Inc, 2011.

[BFWZ18] Justus Bogner, Jonas Fritzsch, Stefan Wagner, and Alfred Zimmermann. Limiting Technical Debt with Maintainability Assurance – An Industry Survey on Used Techniques and Differences with Service- and Microservice-Based Systems. In *Proceedings of the 2018 International Conference on Technical Debt*, volume 18, pages 125–133, New York, NY, USA, 2018. ACM.

[Coh09] Mike Cohen. The Forgotten Layer of the Test Automation Pyramid. https://www.mountaingoatsoftware.com/blog/the-forgotten-layer-of-the-test-automation-pyramid, 2009. Accessed: 2021-08-16.

[CS14] Jitender Choudhari and Ugrasen Suman. Extended iterative maintenance life cycle using eXtreme programming. *ACM SIGSOFT Software Engineering Notes*, 39(1):1–12, feb 2014.

[CSGG16] Diego Cedrim, Leonardo Sousa, Rohit Gheyi, and Alessandro Garcia. Does refactoring improve software structural quality? A longitudinal study of 25 projects. In *SBES '16: Proceedings of the 30th Brazilian Symposium on Software Engineering*, pages 73–82, 2016.

[Cun92] CunninghamWard. The WyCash portfolio management system. *ACM SIGPLAN OOPS Messenger*, 4(2):29–30, dec 1992.

[DTGLR08] Trung Dinh-Trong, Birgit Geppert, J. Jenny Li, and Frank Roessler. Looking for more confidence in refactoring? - How to assess adequacy of your refactoring tests -. In *Proceedings - International Conference on Quality Software*, pages 255–263, 2008.

[Els21] Elsevier. ScienceDirect. https://www.sciencedirect.com/, 2021. Accessed: 2021-09-09.

[FB20] Norman Fenton and James Biemann. *Software Metrics. A Rigorous and Practical Approach. Third Edition.* CRC Press; Taylor & Francis Group, LLC, 2020.

[FBB+05] Martin Fowler, Kent Beck, John Brant, John Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code. Sixteenth printing.* Addison Wesley Longman, Inc., 2005.

[FHK+16] Ludwig Fahrmeier, Christian Heumann, Rita Künstler, Iris Pigeot, and Gerhard Tutz. *Statistik. Der Weg zur Datenanalyse.* Springer Spektrum, Springer-Verlag GmbH: Berlin, Heidelberg, 8., überarb. u. erg. aufl. 2016 edition edition, 2016.

[Fow12] Martin Fowler. TestPyramid. https://martinfowler.com/bliki/TestPyramid.html, 2012. Accessed: 2021-08-16.

[GHJV15] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Entwurfsmuster als Elemente wiederverwendbarer objektorientierter Software.* mitp Verlags GmbH und Co. KG, 01 2015.

[Git21a] GitHub. Github API. https://api.github.com/search/repositories?q=language:java, 2021. Accessed: 2021-06-06.

[Git21b] GitHub. Github: Search language:java. https://github.com/search?q=language%3Ajava, 2021. Accessed: 2021-06-06.

[Git21c] GitHub. Github: Where the world builds software. https://github.com, 2021. Accessed: 2021-06-06.

[Hof13] Dirk W. Hoffmann. *Software-Qualität. 2. Auflage*. Springer Vieweg, Berlin, Heidelberg, 01 2013.

[HS96] R. Harrison and L. G. Samaraweera. Using test case metrics to predict code quality and effort. *ACM SIGSOFT Software Engineering Notes*, 21(5):78–88, sep 1996.

[IEE90] IEEE. IEEE Standard Glossary of Software Engineering Terminology. *Office*, 121990(1):1, 1990.

[IEE21] IEEE. IEEE Xplore. https://ieeexplore.ieee.org/Xplore/guesthome.jsp, 2021. Accessed: 2021-09-09.

[Int21a] International Organization for Standardization. ISO/IEC 25010:2011(en), Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models. https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en, 2021. Accessed: 2021-04-29.

[Int21b] International Software Testing Qualifications Board. ISTQP Glossary. https://glossary.istqb.org/en/search/confirmation%20testing, 2021. Accessed: 2021-08-16.

[Int21c] International Software Testing Qualifications Board. ISTQP Glossary. https://glossary.istqb.org/en/search, 2021. Accessed: 2021-08-16.

[KMPY06] Ronny Kolb, Dirk Muthig, Thomas Patzke, and Kazuyuki Yamauchi. Refactoring a legacy component for reuse in a software product line: a case study. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(2):109–132, mar 2006.

[KZN12] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. A Field Study of Refactoring Challenges and Benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering - FSE '12*, pages 1–11, New York, New York, USA, 2012. ACM Press.

[Lei21a] Leijdekkers Bas. MetricsReloaded. https://github.com/BasLeijdekkers/MetricsReloaded, 2021. Accessed: 2021-09-21.

[Lei21b] Leijdekkers Bas. MetricsReloaded - metricDescriptions. https://github.com/BasLeijdekkers/MetricsReloaded/tree/master/stockmetrics/metricsDescriptions/metricsDescriptions, 2021. Accessed: 2021-07-12.

[LHR88] K. Lieberherr, I. Holland, and A. Riel. Object-oriented programming: An objective sense of style. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 1988*, pages 323–334. Association for Computing Machinery, Inc, jan 1988.

[Lil20] Carola Lilienthal. *Langlebige Software-Architekturen: Technische Schulden analysieren, begrenzen und abbauen. 3., überarb. u. erw. Edition.* dpunkt.verlag GmbH, Heidelberg, 2020.

[LL13] Jochen Ludewig and Horst Lichter. *Software Engineering, 3., korrigierte Auflage.* dpunkt.verlag GmbH, Heidelberg, 05 2013.

[LML+15] Marko Leppanen, Simo Makinen, Samuel Lahtinen, Outi Sievi-Korte, Antti Pekka Tuovinen, and Tomi Mannisto. Refactoring-a Shot in the Dark? *IEEE Software*, 32(6):62–70, nov 2015.

[Mar00] Rc Martin. Design principles and design patterns. *Object Mentor*, (c):1–34, 2000.

[Mar09] RC Martin. *Clean Code: A Handbook of Agile Software Craftsmanship.* Pearson Education, Inc., 2009.

[Mou21a] Mountainminds GmbH & Co. KG and Contributors. Coverage Counters. https://www.jacoco.org/jacoco/trunk/doc/counters.html, 2021. Accessed: 2021-09-21.

[Mou21b] Mountainminds GmbH & Co. KG and Contributors. JaCoCo - Java Code Coverage Library. https://www.jacoco.org/jacoco/trunk/index.html, 2021. Accessed: 2021-09-21.

[Mvn21] MvnRepository. MvnRepository: Search, Browse. Explore. https://mvnrepository.com, 2021. Accessed: 2021-09-21.

[OW14] Jan Peter Ostberg and Stefan Wagner. On automatically collectable metrics for software maintainability evaluation. In *Proceedings - 2014 Joint Conference of the International Workshop on Software Measurement, IWSM 2014 and the International Conference on Software Process and Product Measurement, Mensura 2014*, pages 32–37. Institute of Electrical and Electronics Engineers Inc., dec 2014.

[Par17] Parfianowicz Marek. Clover 4.1 : Comparison of code coverage tools. https://atlassian-docs.bitbucket.io/CLOVER/Comparison-of-code-coverage-tools_681706101.html, 2017. Accessed: 2021-09-22.

[PBB16] Harrie Passier, Lex Bijlsma, and Christoph Bockisch. Maintaining Unit Tests During Refactoring. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, pages 1–6, New York, NY, USA, 2016. ACM.

[PGH+08] R. Plösch, H. Gruber, A. Hentschel, G. Pomberger, and S. Schiffer. On the relation between external software quality and static code analysis. *32nd Annual IEEE Software Engineering Workshop, SEW-32 2008*, 1(1):169–174, 2008.

[PMA+19] Anthony Peruma, Mohamed Wiem Mkaouer, Khalid Almalki, Ali Ouni, Christian D Newman, and Fabio Palomba. On the distribution of test smells in open source android applications: An exploratory study. In *CASCON 2019 Proceedings - Conference of the Centre for Advanced Studies on Collaborative Research - Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, volume 10, pages 193–202, 2019.

[PMD21a] PMD. Java Rules. https://pmd.github.io/latest/pmd_rules_java.html, 2021. Accessed: 2021-09-22.

[PMD21b] PMD. PMD. An extensible cross-language static code analyzer. https://pmd.github.io/, 2021. Accessed: 2021-08-23.

[PMD21c] PMD. What does 'PMD' mean? https://pmd.github.io/latest/pmd_projectdocs_trivia_meaning.html, 2021. Accessed: 2021-08-23.

[PMVV12] Eltjo R. Poort, Nick Martens, Inge Van De Weerd, and Hans Van Vliet. How architects see non-functional requirements: Beware of modifiability. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7195 LNCS:37–51, 2012.

[PR15] Klaus Pohl and Chris Rupp. *Basiswissen Requirements Engineering, 4. Auflage*. dpunkt.verlag GmbH, Heidelberg, 07 2015.

[R F21] R Foundation. The R Project for Statistical Computing. https://www.r-project.org/, 2021. Accessed: 2021-09-28.

[Sah71] Heinz Sahner. *Schließende Statistik*. Studienskripten zur Soziologier. Herausgegeben von E.K. Scheuch. Statistik für Soziologen 2. B. G. Teubner, Stuttgart, 06 1971.

[SI11] Muhammad Shahid and Suhaimi Ibrahim. An Evaluation of Test Coverage Tools in Software Testing. In *2011 International Conference on Telecommunication Technology and Applications*, volume 5, pages 216–222, 2011.

[SL19] Andreas Spillner and Tilo Linz. *Basiswissen Softwaretest*. dpunkt.verlag GmbH, Heidelberg, 06 2019.

[SOPF19] Markus Schnappinger, Mohd Hafeez Osman, Alexander Pretschner, and Arnaud Fietzke. Learning a classifier for prediction of maintainability based on static analysis tools. In *IEEE International Conference on Program Comprehension*, volume 2019-May, pages 243–248. IEEE Computer Society, may 2019.

[SYA$^+$13] Dag I.K. Sjoberg, Aiko Yamashita, Bente C.D. Anda, Audris Mockus, and Tore Dyba. Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering*, 39(8):1144–1156, 2013.

[TBL18] Fadel Toure, Mourad Badri, and Luc Lamontagne. Predicting different levels of the unit testing effort of classes using source code metrics: a multiple case study on open-source software. *Innovations in Systems and Software Engineering*, 14(1):15–46, mar 2018.

[TEM13] P. Tomas, M. J. Escalona, and M. Mejias. Open source tools for measuring the Internal Quality of Java software products. A survey. *Computer Standards and Interfaces*, 36(1):244–255, nov 2013.

[THG20] Alexander Trautsch, Steffen Herbold, and Jens Grabowski. A longitudinal study of static analysis warning evolution and the effects of PMD on software quality in Apache open source projects. *Empirical Software Engineering*, 25(6):5137–5192, nov 2020.

[TPB$^+$16] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. An Empirical Investigation into the Nature of Test Smells. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 4–15, New York, NY, USA, 2016. ACM.

[vDMvK02] A. van Deursen, L. Moonen, A van Den Bergh, and Gerard Kok. Refactoring test code. *Extreme Programming Perspectives*, pages 141–152, 2002.

[Vet13] A Vetro'. *Empirical Assessment of the Impact of Automatic Static Analysis on Code Quality.* PhD thesis, Politecnico di Torino, 2013.

[VMS$^+$19] Tássio Virgínio, Luana Almeida Martins, Larissa Rocha Soares, Ivan Machado, Railana Santana, and Heitor Costa. On the influence of Test Smells on Test Coverage. In *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*, New York, NY, USA, 2019. ACM.

[VRvdL$^+$16] Joost Visser, Sylvan Rigal, Rob van der Leek, Gijs Wijnholds, and Pascal van Eck. *Building Maintainable Software.* O'Reilly Media, Inc.; Sabastopol, 08 2016.

[VZS$^+$12] Antonio Vetro, Nico Zazworka, Forrest Shull, Carolyn Seaman, and Michele A. Shaw. Investigating automatic static analysis results to identify quality problems: An inductive study. In *Proceedings of the 2012 IEEE 35th Software Engineering Workshop, SEW 2012*, pages 21–31, 2012.

[WRH$^+$00] Claes Wohlin, Per Runeson, Martin Hoest, Magnus C. Ohlsson, Bjoern Regnell, and Anders Wesslen. *Experimentation in Software Engineering.* Springer Science and Business Media, New York, 10 2000.

[YLW09] Q. Yang, J. J. Li, and D. M. Weiss. A Survey of Coverage-Based Testing Tools. *The Computer Journal*, 52(5):589–597, aug 2009.

[ZHB11] Min Zhang, Tracy Hall, and Nathan Baddoo. Code Bad Smells: a review of current knowledge. *Journal of Software Maintenance and Evolution: Research and Practice*, 23(3):179–202, apr 2011.

A Appendix

A.1 Description of metrics

Abbr.	Full name	Description	Classification
WMC	Weighted Method per Class	Measures the sum of weights of all methods in a class. The "weight" is a measure for complexity, typically (like in MetricsReloaded) the Cyclomatic Complexity (CC). Therefore, WMC is summing up the complexity of all methods in a class.	Design Size
CBO	Coupling Between Objects	Calculates the number of classes or interfaces that are coupled to a given class. Two classes are called "coupled" to each other, if one class depends on the other (e.g. because relying on an method implemented there). In MetricsReloaded, dependencies due to inheritance are not counted.	Coupling
RFC	Response For Class	RFC is, like CBO, a coupling measure. It calculates the number of local methods as well as external methods that can potentially be called in response to a message received by an object.	Coupling
LCOM	Lack of Cohesion in Methods	LCOM captures the cohesion of a class, "charakterized by how closely the local methods are related to the local instance variables in the class" ([FB20, p. 419]). A value of more than 1 means that the class is not very cohesive and most likely needs to be split up.	Cohesion
D	Halsteads Difficulty	An estimate for the effort needed to maintain the source code; Roughly said, it is calculated by counting the elements of expressions (operands and operators).	Code Size
E	Halsteads Effort	Related to Halsteads Difficulty; To calculate E, D is multiplied by the so called volume of a class, i.e. the number of bits needed to store the source code.	Code Size
LOC	Lines of code	Calculates the number of lines of code in each class. In MetricsReloaded, the comments are counted, but whitespace is omitted.	Code Size
STAT	Number of Statements	Counts the number of statements in a class.	Code Size

Table A.1: Description of metrics (cf. [ACBV20], [FB20], [Lei21b]); Classification according to [FB20].

A.2 Repositories

ID	Name	Type	Commit	Classes	Link to repository on GitHub
R1	HTSJDK	Gradle	8466c82	839	https://samtools.github.io/htsjdk/
R2	JSQLParser Library	Maven	8eb3d9a	331	https://github.com/JSQL Parser/JSqlParser
R3	Hermod Java Ser Descriptor API	Maven	9fb87de	2	https://github.com/hermod /hermod-java-ser-descriptor-api
R4	Java Restify HTTP Client	Maven	baad629	97	https://github.com/ljtfreitas /java-restify
R5	Java UnRar	Gradle	8b6d9f9	91	https://github.com/junrar /junrar
R6	Java SQL ODBC	Maven	aaa47de	20	https://github.com/nbbrd /java-sql-util
R7	Clj DS	Maven	6a45a1d	115	http://github.com/krukow/clj-ds
R8	Java Property Utils	Maven	382b0d1	1	https://github.com /ansell/property-util
R9	Java Stellar SDK	Gradle	67b2690	707	https://github.com /stellar/java-stellar-sdk
R10	Java Diff Utilities	Maven	3d5343c	28	https://github.com/java-diff-utils/java-diff-utils
R11	Java JSON Schema Generator	Maven	3369728	40	https://github.com/janlabrie /jsonschema-generator
R12	Langx Java Reflection AspectJ	Maven	2593558	942	https://github.com/fangjinuo /langx-java
R13	Hermod Java Ser API	Maven	e170dda	3	https://github.com/hermod /hermod-java-ser-api
R14	Generex	Maven	cda5912	5	https://github.com/mifmif /Generex
R15	Java SemVer	Maven	1f4996e	29	https://github.com/zafarkhaja /jsemver

Table A.2: Random sample of GitHub repositories (found on mvnrepositories.com with search term "Java" and results sorted by relevance).

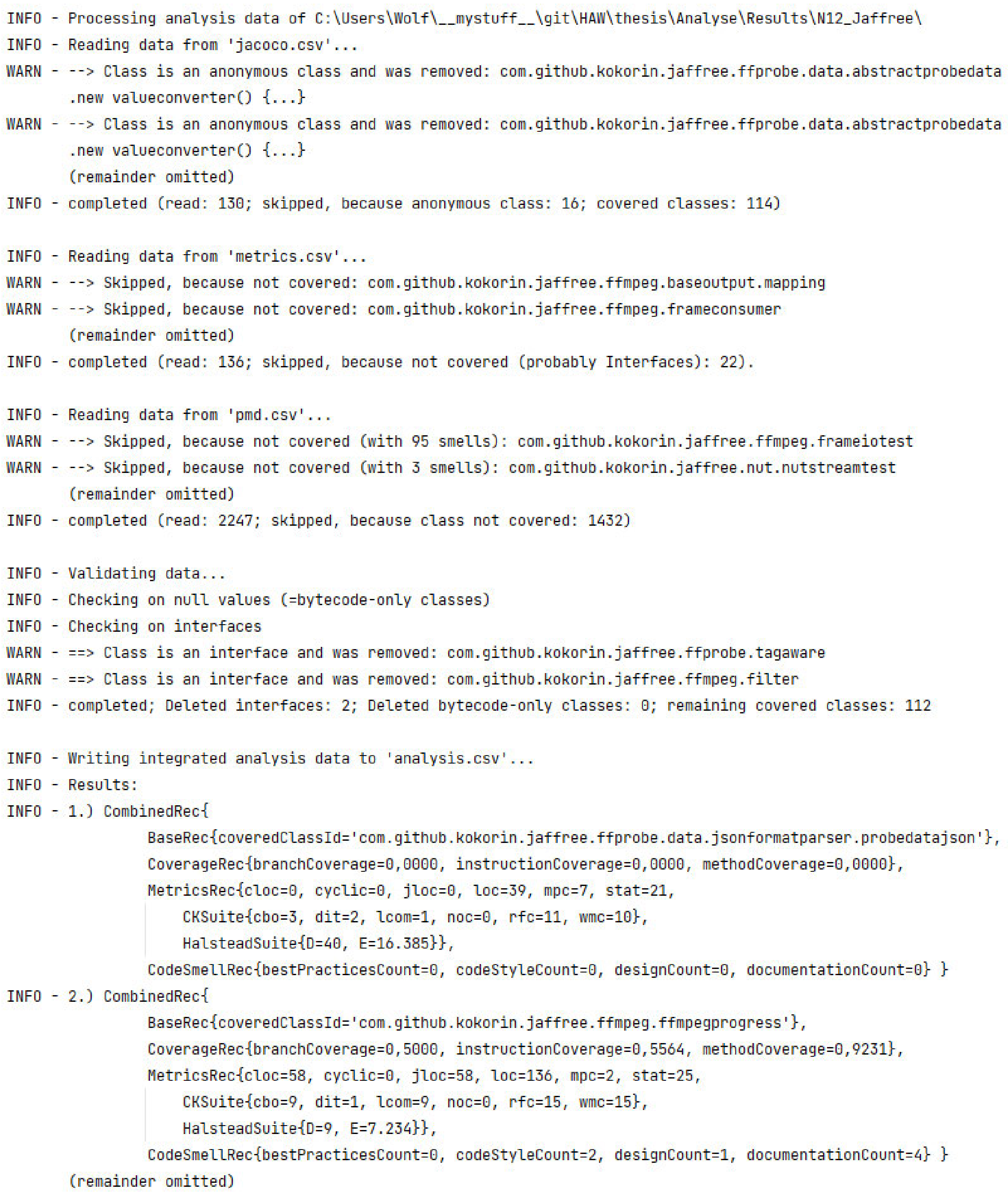
ID	Name	Type	Commit	Classes	Link to repository on GitHub
P1	Embedded Redis	Maven	b855803	22	https://github.com/kstyrc/embedded-redis
P2	RxJava 3 Interop Library For Java 8	Gradle	ea8510a	21	https://github.com/akarnokd/RxJavaJdk8Interop/
P3	Wget	Maven	17ce5f4c	28	https://gitlab.com/axet/wget
P4	Text Recorder	Gradle	b393426	11	https://github.com/naturs/TextRecorder/
P5	JSON Schema Validator	Gradle	d265c6b	140	https://github.com/java-json-tools/json-schema-validator
P6	Docker Java	Maven	4f8f7b9	1	https://github.com/docker-java/docker-java
P7	DevUtility Internal	Maven	260e02f	112	https://github.com/eagle6688/devutility.internal
P8	Sqlhelper Dialect	Maven	6f53c34	303	https://github.com/fangjinuo/sqlhelper
P9	Picard	Gradle	77b9159	689	http://broadinstitute.github.io/picard/
P10	Chicory Core	Maven	a88b05b	73	https://github.com/sviperll/chicory
P11	Simple ODS Library	Maven	1ac2e81	32	https://github.com/miachm/SODS
P12	CID	Maven	d6b6a70	3	https://github.com/ipld/java-cid
P13	JNBIS	Maven	71c302b	60	https://github.com/kareez/jnbis
P14	XsdParser	Maven	78ba41c	87	https://github.com/xmlet/XsdParser
P15	Software and Algorithms	Maven	34c07b6	21	https://www.github.com/KevinStern/software-and-algorithms

Table A.3: Random sample of GitHub repositories (found on mvnrepositories.com with search term "Java" and results sorted by popularity).

ID	Name	Type	Commit	Classes	Link to repository on GitHub
N1	Lambda Factory	Maven	7428576	5	https://github.com/Hervian /lambda-factory
N2	Ksuid Creator	Maven	fd1039d	4	http://github.com/f4b6a3 /ksuid-creator
N3	JTools	Maven	59860	5	https://github.com/levkopo /JTools
N4	Geocalc	Maven	4f14539	12	https://github.com /grum-limited/geocalc
N5	Iban4j	Maven	a1fa1c7	18	https://github.com /passion-java4/iban4j
N6	ActivitiUtils	Maven	7e14121	4	https://github.com /AppUn-defined/javaUtils
N7	JD CLI Root	Maven	2972045	20	https://github.com/kwart/jd-cli
N8	Java Xid	Maven	40ca6e2	1	https://github.com/0xShamil /java-xid
N9	Azure DevOps Remote Configuration	Gradle	063cf43	11	https://github.com /davidpolaniaac/azure-devops-remote-configuration-for-java
N10	AWS Lightweight Client Java	Maven	205373a	26	https://github.com /davidmoten/aws-lightweight-client-java
N11	Protobuf Swagger Mapper	Maven	97dbbed	1	https://github.com /robert2411/protobuf-object-mapper
N12	Jaffree	Maven	29e32e1	112	https://github.com/kokorin /Jaffree
N13	Bot Core	Gradle	a3119d7	6	https://github.com /dreamhead/object-bot
N14	Param Validator	Maven	3ac7d42	46	https://github.com/CatDou /param-validator
N15	Caesar	Maven	e3097f5	7	https://github.com/Glusk /caesar

Table A.4: Random sample of GitHub repositories (found on mvnrepositories.com with search term "Java" and results sorted by date of last commit).

A.3 Data integration process - log file

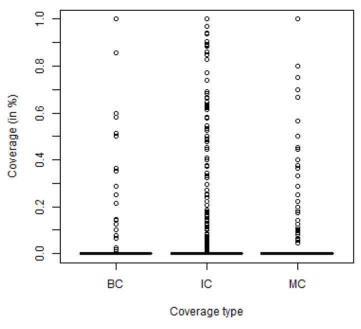


```
INFO - Processing analysis data of C:\Users\Wolf\__mystuff__\git\HAW\thesis\Analyse\Results\N12_Jaffree\
INFO - Reading data from 'jacoco.csv'...
WARN - --> Class is an anonymous class and was removed: com.github.kokorin.jaffree.ffprobe.data.abstractprobedata
       .new valueconverter() {...}
WARN - --> Class is an anonymous class and was removed: com.github.kokorin.jaffree.ffprobe.data.abstractprobedata
       .new valueconverter() {...}
       (remainder omitted)
INFO - completed (read: 130; skipped, because anonymous class: 16; covered classes: 114)

INFO - Reading data from 'metrics.csv'...
WARN - --> Skipped, because not covered: com.github.kokorin.jaffree.ffmpeg.baseoutput.mapping
WARN - --> Skipped, because not covered: com.github.kokorin.jaffree.ffmpeg.frameconsumer
       (remainder omitted)
INFO - completed (read: 136; skipped, because not covered (probably Interfaces): 22).

INFO - Reading data from 'pmd.csv'...
WARN - --> Skipped, because not covered (with 95 smells): com.github.kokorin.jaffree.ffmpeg.frameiotest
WARN - --> Skipped, because not covered (with 3 smells): com.github.kokorin.jaffree.nut.nutstreamtest
       (remainder omitted)
INFO - completed (read: 2247; skipped, because class not covered: 1432)

INFO - Validating data...
INFO - Checking on null values (=bytecode-only classes)
INFO - Checking on interfaces
WARN - ==> Class is an interface and was removed: com.github.kokorin.jaffree.ffprobe.tagaware
WARN - ==> Class is an interface and was removed: com.github.kokorin.jaffree.ffmpeg.filter
INFO - completed; Deleted interfaces: 2; Deleted bytecode-only classes: 0; remaining covered classes: 112

INFO - Writing integrated analysis data to 'analysis.csv'...
INFO - Results:
INFO - 1.) CombinedRec{
                BaseRec{coveredClassId='com.github.kokorin.jaffree.ffprobe.data.jsonformatparser.probedatajson'},
                CoverageRec{branchCoverage=0,0000, instructionCoverage=0,0000, methodCoverage=0,0000},
                MetricsRec{cloc=0, cyclic=0, jloc=0, loc=39, mpc=7, stat=21,
                    CKSuite{cbo=3, dit=2, lcom=1, noc=0, rfc=11, wmc=10},
                    HalsteadSuite{D=40, E=16.385}},
                CodeSmellRec{bestPracticesCount=0, codeStyleCount=0, designCount=0, documentationCount=0} }
INFO - 2.) CombinedRec{
                BaseRec{coveredClassId='com.github.kokorin.jaffree.ffmpeg.ffmpegprogress'},
                CoverageRec{branchCoverage=0,5000, instructionCoverage=0,5564, methodCoverage=0,9231},
                MetricsRec{cloc=58, cyclic=0, jloc=58, loc=136, mpc=2, stat=25,
                    CKSuite{cbo=9, dit=1, lcom=9, noc=0, rfc=15, wmc=15},
                    HalsteadSuite{D=9, E=7.234}},
                CodeSmellRec{bestPracticesCount=0, codeStyleCount=2, designCount=1, documentationCount=4} }
       (remainder omitted)
```

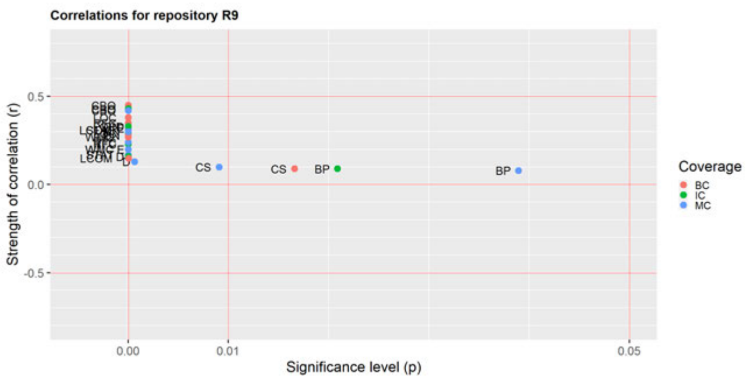
Figure A.1: Example log file for data integration process (repository N12).

A.4 Patterns

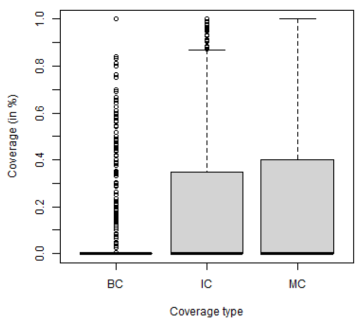
A.4.1 LCPC-Pattern (low coverage + positive correlations)



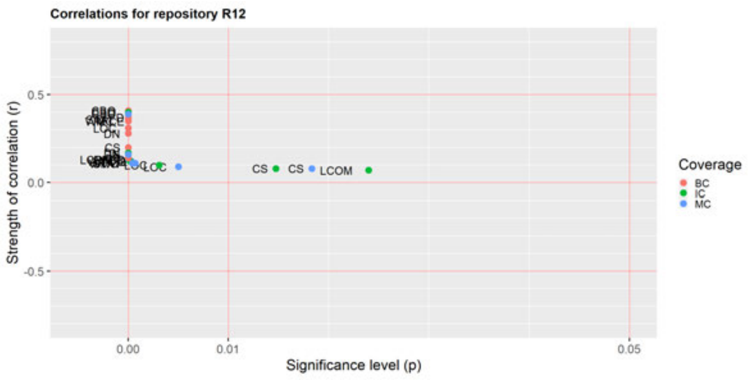
(a) Coverage for repository R9



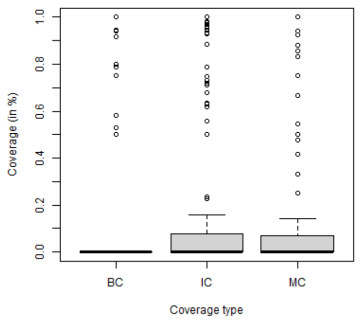
(b) Correlations for repository R9



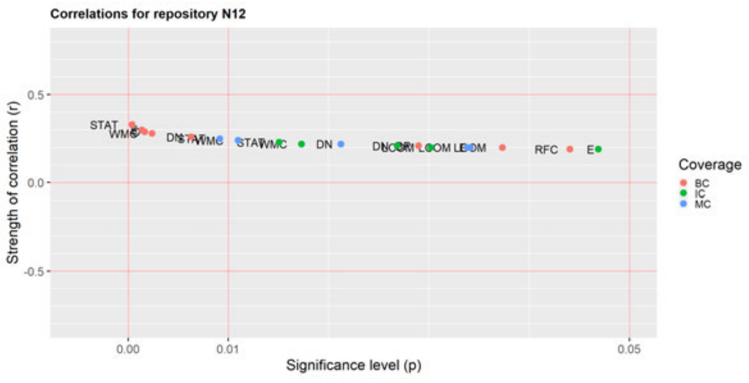
(c) Coverage for repository R12



(d) Correlations for repository R12

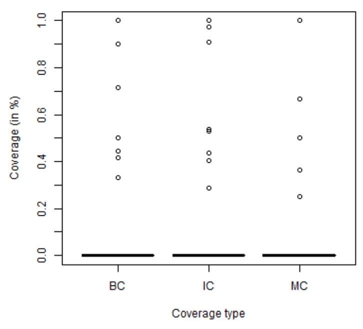


(e) Coverage for repository N12

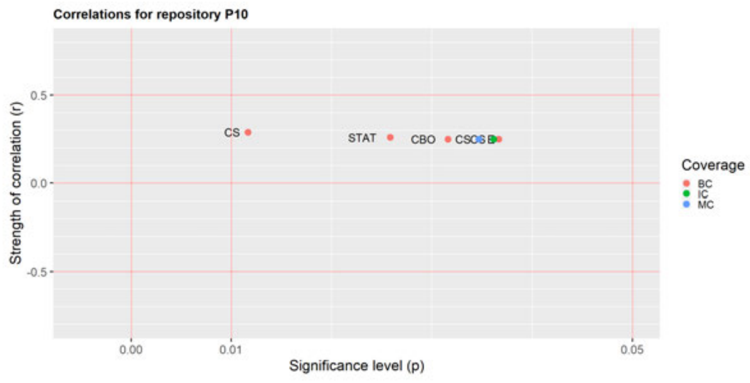


(f) Correlations for repository N12

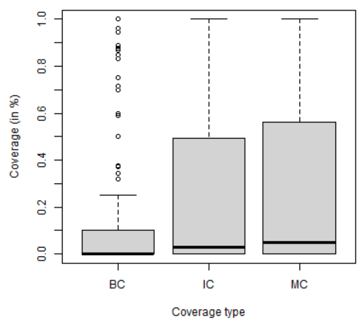
Figure A.2: LCPC-Pattern in repositories R9, R12 and N12.



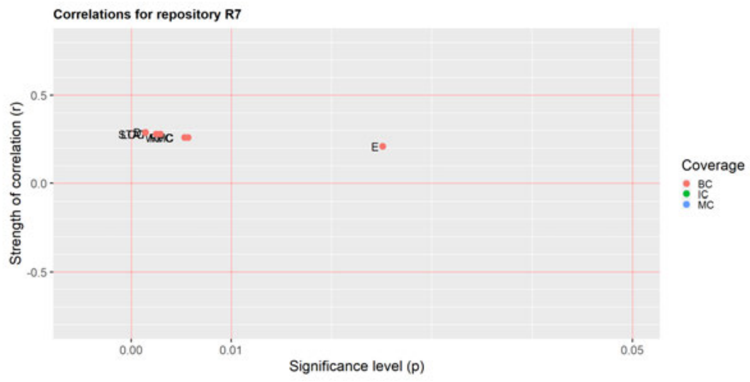
(a) Coverage for repository P10



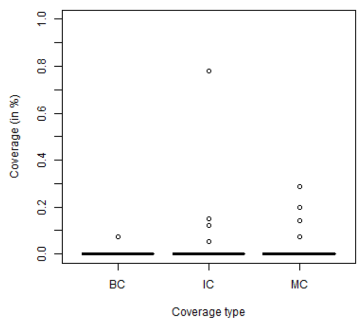
(b) Correlations for repository P10



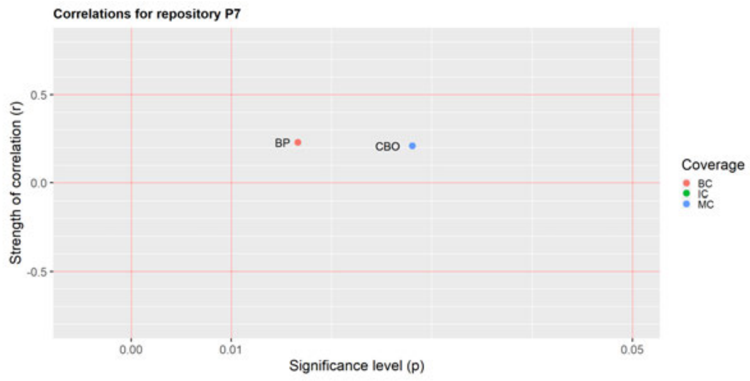
(c) Coverage for repository R7



(d) Correlations for repository R7

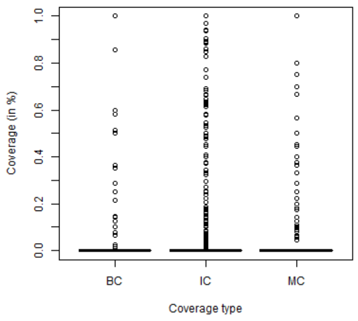


(e) Coverage for repository P7

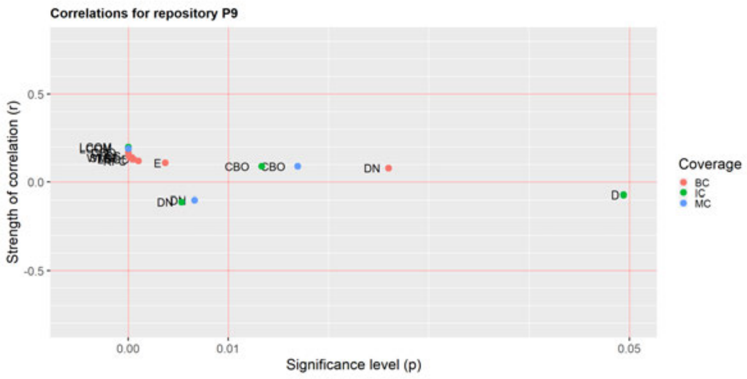


(f) Correlations for repository P7

Figure A.3: LCPC-Pattern in repositories P10, R7 and P7.



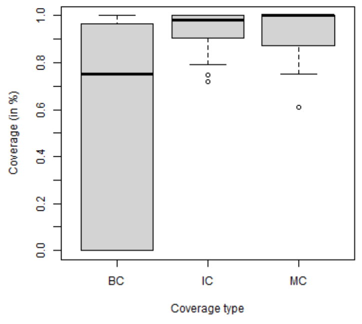
(a) Coverage for repository P9



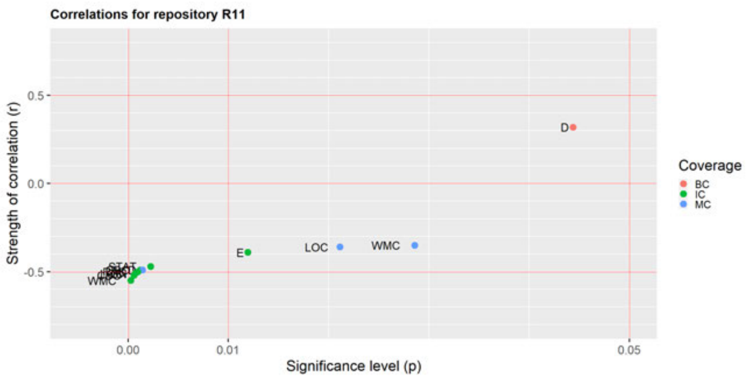
(b) Correlations for repository P9

Figure A.4: LCPC-Pattern in repository P9.

A.4.2 HCNC-Pattern (high coverage + negative correlations)



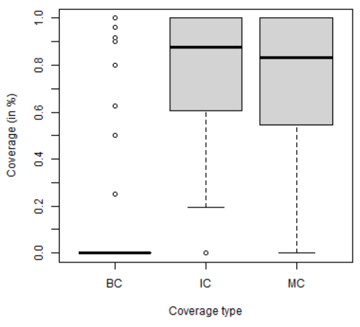
(a) Coverage for repository R11



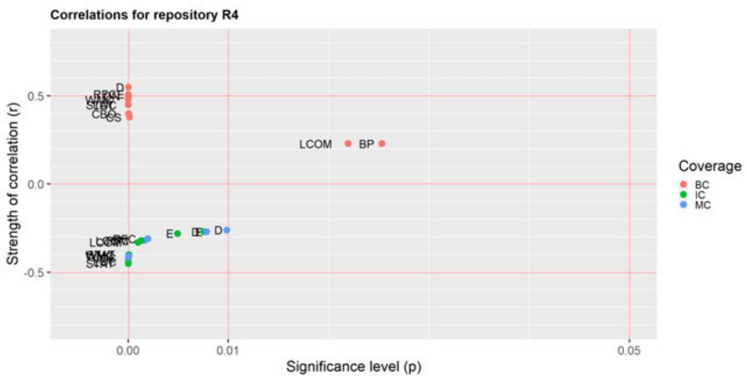
(b) Correlations for repository R11

Figure A.5: HCNC-Pattern in repository R11.

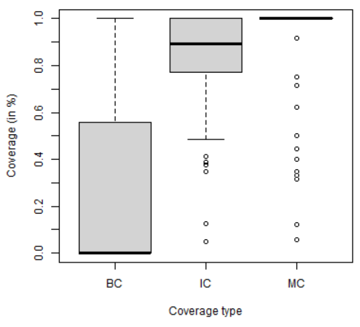
A.4.3 Mixed pattern: LCPC $_{(BC)}$ + HCNC $_{(IC,MC)}$



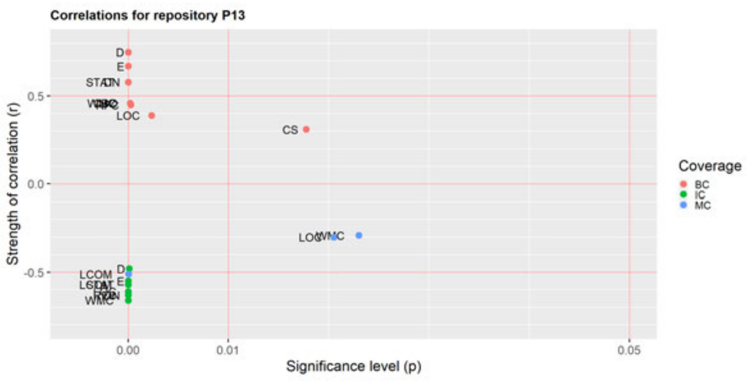
(a) Coverage for repository R4



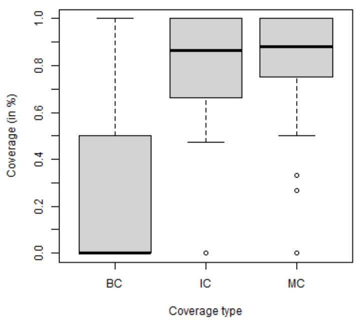
(b) Correlations for repository R4



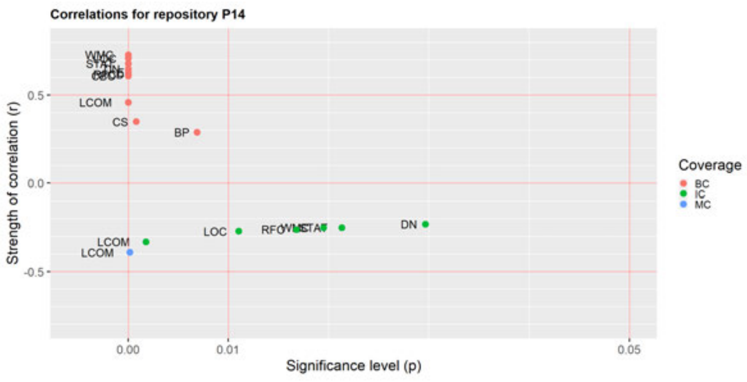
(c) Coverage for repository R13



(d) Correlations for repository R13

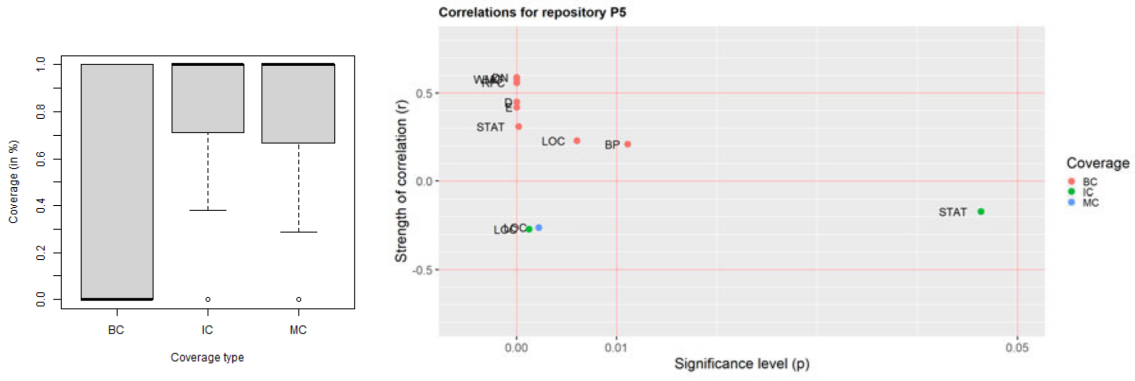


(e) Coverage for repository P14



(f) Correlations for repository P14

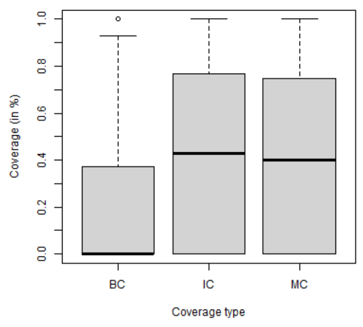
Figure A.6: Mixed LCPC $_{(BC)}$ / HCNC $_{(IC,MC)}$ in repositories R4, P13 and P14.



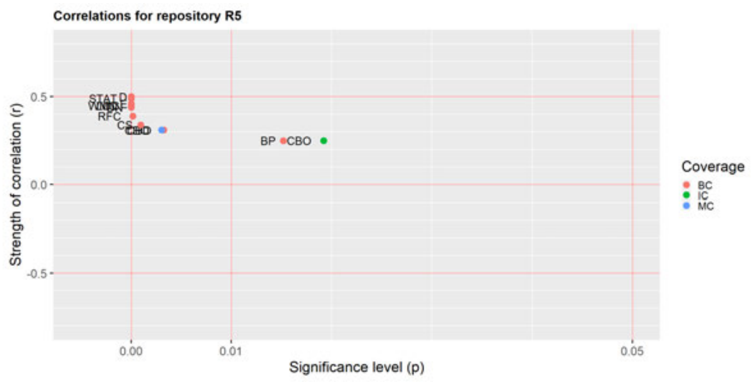
(a) Coverage for repository P5 (b) Correlations for repository P5

Figure A.7: Mixed LCPC $_{(BC)}$ / HCNC $_{(IC,MC)}$ pattern in repository P5.

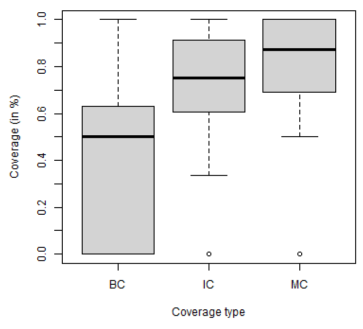
A.4.4 Repositories with positive correlations and unclear pattern



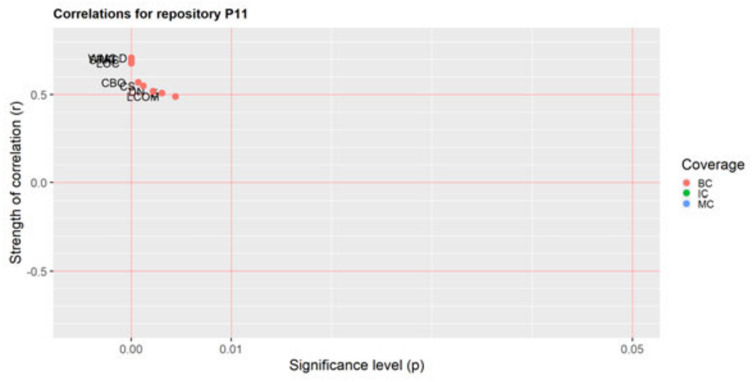
(a) Coverage for repository R5



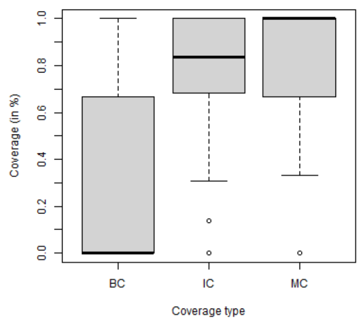
(b) Correlations for repository R5



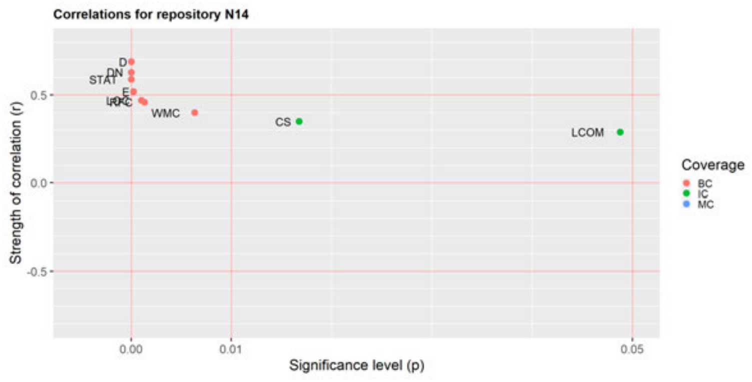
(c) Coverage for repository P11



(d) Correlations for repository P11



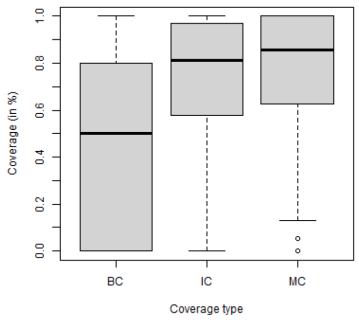
(e) Coverage for repository N14



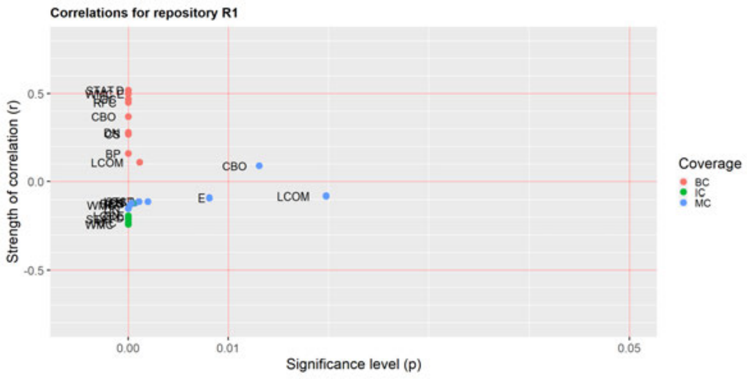
(f) Correlations for repository N14

Figure A.8: Repositories with only positive correlations and unclear pattern.

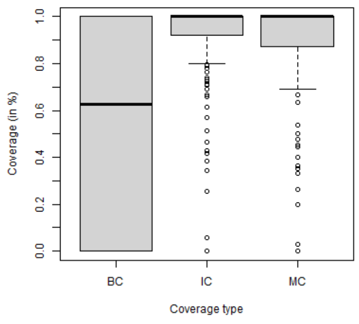
A.4.5 Repositories with mixed correlations and unclear pattern



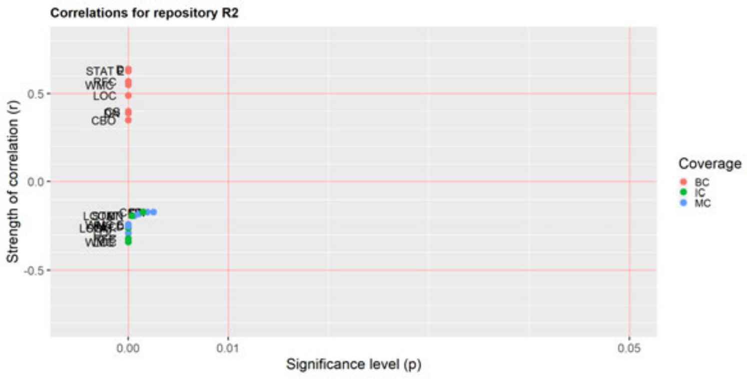
(a) Coverage for repository R1



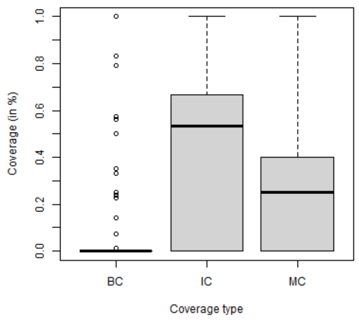
(b) Correlations for repository R1



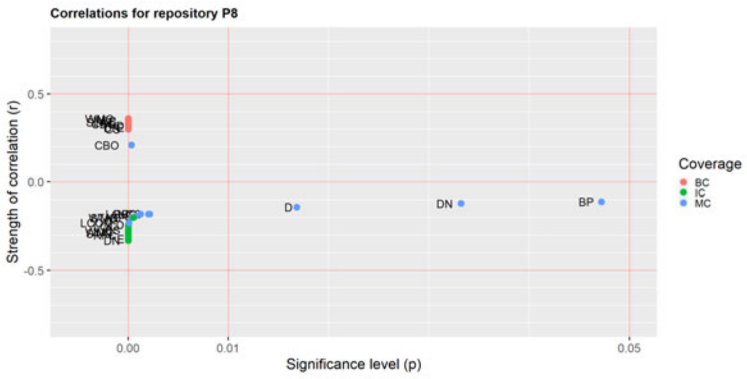
(c) Coverage for repository R2



(d) Correlations for repository R2



(e) Coverage for repository P8



(f) Correlations for repository P8

Figure A.9: Repositories with mixed correlations and unclear pattern.

Glossary

Code smells This term is used to express when object-oriented code violates common design rules and should be refactored in order to achieve higher maintainability.

Cohesion Cohesion describes in how far the elements of a software component are related to each other, belong together and serve a common purpose. A high level of cohesion is recommended for a comprehensible code structure.

Coupling Coupling describes the situation when software components depend on each other. A system with low coupling is – in general – easier to maintain.

Dynamic test By use of dynamic tests the source code is executed in order to find failures, i.e. cases where requirements are not fulfilled.

Fragility A symptom of low maintainability that describes a software product, where modifications lead to unexpected failures in other components

Immobility A symptom of low maintainability, where dependencies between modules make it improbable if not impossible to reuse part of a software somewhere else.

Maintainability The ease with which a system can be maintained. Maintainability is one of eight software attributes defined within the product quality model. Sub-characteristics are Modularity, Analysability, Reusability and Testability.

Maintenance Development activities with the aim to change a software system in order to implement new or adapt existing features, fix/prevent problems or improve quality attributes.

Quality metric Metrics are used to monitor and compare quality characteristics of software systems. The source code is analysed and reduced to a single number that describes, for example, the complexity of a component.

Product Quality Model A model proposed by the ISO/IEC 25010 standard that describes the quality of software products in terms of of eight quality characteristics with further 31 subcharacteristics.

Refactoring Changes made to the internal structure of software with the aim to make it easier to understand and cheaper to modify; in a narrower sense, these changes should be rather small and not alter the observable behaviour.

Rigidity A symptom of low maintainability. A system is rigid, when even simple changes take much longer than expected because of growing dependencies between modules.

Software requirements Needs or expectations that users or other relevant groups have in regards to a software product. These can be either functional requirements (the functions that a system should have) or nun-functuctional requirements (quality goals in terms of reliability, maintainability, performance etc.).

Software Quality The degree to which a software product fulfils requirements voiced by relevant stakeholder groups.

Stakeholder Groups of persons or organisations that have an influence on software requirements.

Static code analysis Form of analysis where the source code is inspected without executing it; this is usually done in an automated way with the support of software tools.

Test coverage Describes the degree to which specified requirements for a component or system are tested. Different coverage types exist, e.g. statement coverage or branch coverage.

Technical debt Represents the additional effort needed for maintenance and is a different way of characterizing the maintainability of a software product.

Unit tests Tests that are designed to check how far requirements for individual software components are fulfilled without considering how these components work together.

Viscosity A symptom of low maintainability. A system has a high viscosity when developers are tempted to implement changes that compromise the design of a system.

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Hamburg	**22.10.2021**	[redacted]
Ort	Datum	Unterschrift im Original