Fachhochschul-Masterstudiengang
SOFTWARE ENGINEERING
4232 Hagenberg, Austria

Mining Software Repositories for the Effects of Design Patterns on Software Quality

Masterarbeit

zur Erlangung des akademischen Grades
Master of Science in Engineering

Eingereicht von

Johann Aichberger, BSc

Betreuer:	Andreas Hannes Schuler, BSc MSc,
	FH OÖ Forschungs und Entwicklungs GmbH
Begutachter:	Dipl.-Ing. (FH) Dr. Josef Pichler

September 2020

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

This printed thesis is identical with the electronic version submitted.

Date Signature

Abstract

Design patterns are reusable solutions for commonly occurring problems in software design. First described in 1994 by the *Gang of Four*, they have gained widespread adoption in many areas of software development throughout the years. Furthermore, design patterns have also garnered an active research community around them, which investigates the effects that design patterns have on different software quality attributes. However, a common shortcoming of existing studies is that they only analyze the quality effects of design patterns on a relatively small scale, covering no more than a few hundred projects per case study. This calls into question how generalizable the results of these small-scale case studies are.

Pursuing more generalizable results, this thesis conducts a much larger-scale analysis of the quality effects of design patterns. To accomplish this, software metric and design pattern data for 90,000 projects from the *Maven Central* repository is collected using the metrics calculation tool *CKJM extended* and the design pattern detection tool *SSA*. Correlations between design patterns and software quality attributes are then analyzed using software metrics as proxies for software quality by following the methodology described by the QMOOD quality model. The results of the analysis suggest that design patterns are positively correlated with *functionality* and *reusability*, but negatively correlated with *understandability*, which is consistent with the results of existing smaller-scale case studies.

Kurzfassung

Entwurfsmuster sind wiederverwendbare Lösungen für häufig auftretende Probleme beim Softwareentwurf. Erstmals 1994 von der *Gang of Four* beschrieben, haben sie im Laufe der Jahre in vielen Bereichen der Softwareentwicklung weite Verbreitung gefunden. Darüber hinaus haben Entwurfsmuster auch eine aktive Forschungsgemeinschaft um sich versammelt, die die Auswirkungen von Entwurfsmustern auf verschiedene Softwarequalitätsmerkmale untersucht. Ein allgemeines Manko bestehender Studien ist jedoch, dass sie die Qualitätseffekte von Entwurfsmustern nur in einem relativ kleinen Maßstab analysieren. Konkret untersuchen selbst die größten bestehenden Fallstudien nur einige hundert Projekte. Dies wirft die Frage auf, wie verallgemeinerbar die Ergebnisse dieser klein angelegten Fallstudien sind.

Um Ergebnisse mit einem höheren Grad an Verallgemeinerbarkeit zu erreichen, werden in dieser Masterarbeit die Qualitätseffekte von Entwurfsmustern in einem deutlich größeren Maßstab analysiert. Dafür werden Daten über Softwaremetriken und die Verwendung von Entwurfsmustern für 90.000 Projekte aus dem *Maven Central* Repository mit Hilfe des Softwaremetrik-Berechnungswerkzeugs *CKJM extended* und des Entwurfsmuster-Erkennungswerkzeugs *SSA* gesammelt. Darauf aufbauend werden Korrelationen zwischen Entwurfsmustern und Softwarequalitätsattributen analysiert, wobei Softwaremetriken unter Verwendung des QMOOD-Qualitätsmodells als Surrogat für die untersuchten Softwarequalitätsattribute eingesetzt werden. Die Ergebnisse der Analyse zeigen, dass Entwurfsmuster positiv mit *Funktionalität* und *Wiederverwendbarkeit*, aber negativ mit *Verständlichkeit* korreliert sind, was mit den Ergebnissen bestehender kleiner angelegter Fallstudien übereinstimmt.

Contents

Chapter 1

Introduction

1.1 Design Patterns in Software Engineering

Design patterns represent reusable solutions for commonly occurring problems in software design. An initial set of 23 design patterns was first identified and formally described by Gamma, Helm, Johnson, and Vlissides, who are often referred to as the *Gang of Four* (GoF), in their book *Design Patterns: Elements of Reusable Object-Oriented Software* in 1994 [30]. Included in this book are design patterns such as *Singleton*, which ensures that only a single instance of a class can exist throughout the runtime of a program, *Decorator*, which allows to dynamically extend the functionality of an object at runtime, and *Observer*, which describes how changes to the state of an object can be communicated to other objects.

Throughout the years, design patterns have gained widespread adoption in many areas of software development. Not limited by boundaries like industry, programming language, project size, etc., they have, in many cases, become the gold standard for how to solve specific design problems, making up as much as 10–30% of the total number of classes in many projects [6]. This makes design patterns an essential tool in the repertoire of software developers and software architects alike, because knowledge of design patterns makes it easier to understand and maintain existing code that uses them [48] and also provides a shared vocabulary for discussions on the design level [33].

Since the initial set of 23 design patterns was published by the *Gang of Four* in their 1996 book, many other types of design patterns have been identified and formally described by various authors. Among these patterns are, for example, patterns of enterprise application architecture [29], security patterns [76], and microservice patterns [71]. While the existence of such a wide variety of design pattern types further cements the importance that design patterns have achieved in the software engineering community at large, this thesis only focusses on the 23 GoF design patterns.

1.2 Existing Research on Design Patterns

Perhaps prompted by the popularity of design patterns in practical software engineering, the topic of design patterns has also garnered an active research community around it. Research in the field of design patterns is still ongoing, seeing around 30–40 publications per year in recent years [52]. As shown in Figure 1.1, investigated sub-topics of design pattern research include areas such as design pattern development (25% of publications), design pattern usage (19%), design pattern mining (19%), and quality evaluation of design patterns (10%), as well as a few other less commonly investigated sub-topics.

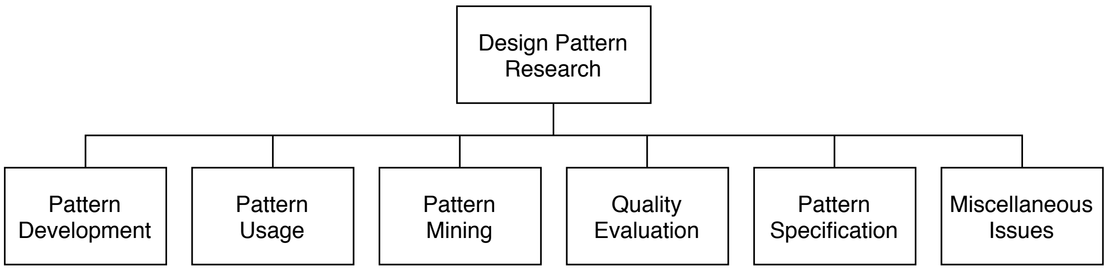

Figure 1.1: Topics of design pattern research [52].

For this thesis, quality evaluation of design patterns is the most relevant sub-topic. Researchers in this field are working towards a better understanding of the quality effects that design patterns have on the software projects in which they are applied [95]. By providing information on the quality trade-offs that the use of design patterns entails, progress in this field can help software engineers make more informed decisions about the use of design patterns [78]. Due to the widespread adoption of design patterns, even minor discoveries in this area have the potential for far-reaching impacts.

Many such discoveries about the quality effects of design patterns have been made in the past, showing both positive as well as negative effects for a diverse set of software quality attributes [4]. For example, both Khomh and Gueheneuce [46] as well as Hussain et al. [35] have found that the *Observer* design pattern appears to have a positive effect on reusability, whereas Vokác et al. [91] have found the *Visitor* design pattern to be detrimental to software maintainability. However, existing literature does not always agree on whether specific design patterns have a positive or a negative effect on certain software quality attributes [95].

One key shortcoming that might explain some of the contradictory results in design pattern research is that a large number of studies only investigate the quality effects of design patterns on a rather small scale. For example, a majority of case studies only analyze a single project or a relatively small number of projects [95], which has been cited as a threat to validity at multiple occasions in both primary [14][36] and secondary research [4][95]. After all, software projects can be very different from each other, so one might assume that the effects of design patterns also vary across projects, which brings into question how generalizable the results from smaller-scale studies really are.

1.3 Research Goals and Questions

To achieve more generalizable results than those found in existing small-scale case studies, the primary goal of this master's thesis is to conduct a large-scale multiple-case study about the quality effects of design patterns.

Large-scale, for the purpose of this thesis, means an investigation that analyzes at least one or two orders of magnitude more projects than the largest existing studies do. Given that these studies cover on the order of tens to hundreds of projects, as seen in studies such as the ones that were published by Hussain et al. [35] in 2017 and Ampatzoglou et al. [6] in 2015, this master's thesis should cover at least thousands if not tens of thousands of projects or more.

Regarding investigated software quality attributes, the focus should be on the quality attributes defined by the QMOOD metrics suite [12]. The six quality attributes that are part of this metrics suite are *effectiveness*, *extendibility*, *flexibility*, *functionality*, *reusability*, and *understandability*. All six of these quality attributes are based on the software quality definitions in the ISO9126 standard [39]. Furthermore, each quality attribute is calculated as a weighted sum of different software metrics. Thus, the software quality definition used in this thesis also follows the definitions in the ISO9126 standard, and software metrics are used as proxy measures for the different software quality attributes.

Given these clarifications, the main research question that has to be answered to accomplish the stated research goal is:

RQ1: Are there correlations between design pattern use and QMOOD quality attributes?

Since QMOOD defines six quality attributes, this main research question can be further broken down into the following sub-questions:

RQ1.1: Are there correlations between design pattern use and *effectiveness*?
RQ1.2: Are there correlations between design pattern use and *extendibility*?
RQ1.3: Are there correlations between design pattern use and *flexibility*?
RQ1.4: Are there correlations between design pattern use and *functionality*?
RQ1.5: Are there correlations between design pattern use and *reusability*?
RQ1.6: Are there correlations between design pattern use and *understandability*?

To aid reproducibility of the achieved results, all code that is written as part of this thesis is made publicly available through two projects hosted on GitHub: *Qualisign*[1], which performs the data collection, and *Qualisign Analysis*[2], which performs the data analysis.

Additionally, SQL dumps of the collected data are published on Zenodo[3]. This ensures that the full dataset is easily accessible for review and further use.

[1]`https://github.com/jaichberg/qualisign`
[2]`https://github.com/jaichberg/qualisign-analysis`
[3]`https://zenodo.org/record/3731872`

1.4 Structure of the Thesis

The remainder of this thesis is composed of Chapter 2: Background, Chapter 3: Methodology, Chapter 4: Results, Chapter 5: Discussion, and Chapter 6: Conclusion. Short summaries of the main contents of these chapters are presented in following paragraphs. **Chapter 4: Results** presents the results of the conducted statistical data analysis. These results consist of two main parts. In the first part, descriptive statistics are used to provide an overview of the software metric and design pattern data that is contained in the created dataset. The second part then shows the results of the quality analysis, thereby answerin

Chapter 2: Background provides supporting background information on topics including software quality measurement, software metrics, and mining software repositories. Furthermore, this chapter outlines the current state of research about the quality effects of design patterns, covering not only the results of existing research, but also the most commonly used research methods that were employed to achieve these results.

Chapter 3: Methodology describes the data collection and processing pipeline through which the necessary software quality and design pattern data is retrieved and analyzed. Subtopics covered in this chapter are, for example, (i) the process through which the used pattern detection tool was selected, (ii) the criteria based on which a suitable repository of projects to analyze was identified, and (iii) the preprocessing and analysis steps that were performed on the collected data.

Chapter 4: Results presents the results of the conducted statistical data analysis. These results consist of two main parts. In the first part, descriptive statistics are used to provide an overview of the software metric and design pattern data that is contained in the created dataset. The second part then shows the results of the quality analysis, thereby answering the research questions regarding the quality effects of design patterns.

Chapter 5: Discussion interprets the achieved results and compares them to the results of existing studies that have investigated the quality effects of design patterns. Furthermore, this chapter also discusses the reproducibility of this thesis as well as threats to validity. Included types of threats to validity are threats to construct validity, threats to internal validity, threats to external validity, and threats to reliability.

Chapter 6: Conclusion provides a summary of this thesis and discusses how the achieved results contribute to the current state of design pattern research and software engineering practice. Finally, this chapter lists potential starting points for future studies that could improve upon the achieved results.

Chapter 2

Background

2.1 Defining Software Quality

According to the *Software Engineering Body of Knowledge* (SWEBOK) [17], a commitment to software quality is an integral part of software engineering culture. Consequently, every software engineer should be familiar with basic quality concepts, characteristics, and values, and should be able to apply this knowledge to the software systems they are developing [17]. To accomplish this, a commonly accepted definition of software quality is needed that defines what exactly software quality is and how it can be measured. After all, a high level of quality can only be achieved if all stakeholders work towards the same quality goals, which, in turn, requires a shared understanding of the different aspects that software quality entails.

One of the more authoritative definitions of software quality comes from the *ISO/IEC/-IEEE International Standard for Systems and Software Engineering Vocabulary* [40]. It defines software quality as “[the] capability of [a] software product to satisfy stated and implied needs when used under specified conditions”. Through this definition, the standard not only signifies that software quality can be affected by requirements that are not explicitly stated, but also makes it clear that the context in which a software product will be used has to be taken into consideration when determining the product’s quality. However, even though these are important clarifications to make, more detailed definitions of individual software quality attributes are necessary to fully capture the diverse set of factors that influence the quality of software products.

Such detailed definitions of software quality attributes can be found in software quality models [23], many of which have been proposed throughout the years [56]. Early models include the ones described by McCall et al. [54] in 1977, Boehm et al. [16] in 1978, and Dromey [26] in 1996. These models focus on basic quality attributes like *maintainability* and *portability* that are relevant across all domains and types of software [56]. Newer models often extend these basic models with additional attributes that are tailored to specific domains [56]. For example, the *SQO-OSS* model by Spinellis et al. [83] introduces attributes such as *mailing list quality* and *developer base quality* that are used to evaluate the quality of an open-source project’s community.

The software quality model that is most widely used today [92] is the *Software Product Quality Requirements and Evaluation* (SQuaRE) model defined by the ISO25010 standard [38]. First published in 2011, SQuaRE supersedes the quality model defined by the ISO9126 standard [39]. It is a hierarchical model consisting of 8 top-level attributes and 31 sub-attributes (see Figure 2.1). SQuarRE also provides short descriptions for both the top-level attributes as well as the sub-attributes. For example, the top-level attribute *reliability* is described as “[the] degree to which a system, product or component performs specified functions under specified conditions for a specified period of time”, whereas its sub-characteristic *fault tolerance* is described as “[the] degree to which a system, product or component operates as intended despite the presence of hardware or software faults”.

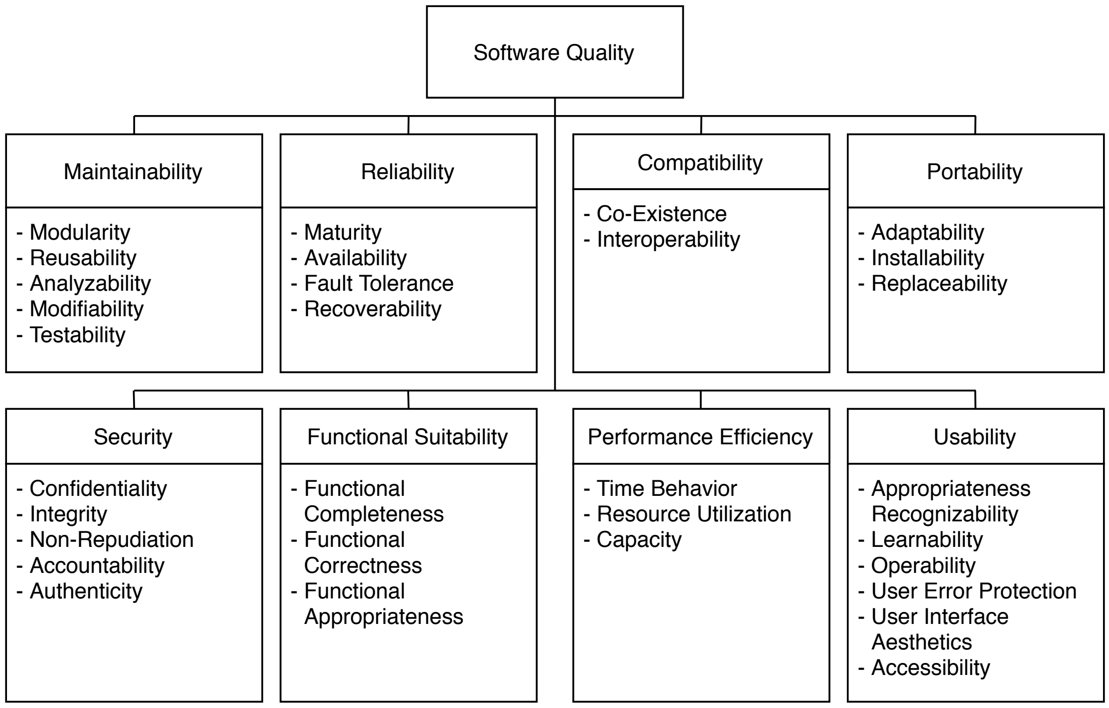

Figure 2.1: Software quality model defined by the ISO/IEC 25010 standard [38].

Prior to the ISO25010 standard’s publication in 2011, the software quality model that was most widely used [63] was the one described by ISO25010’s predecessor, the ISO9126 standard [39]. While the old ISO9126 standard was officially withdrawn when the new ISO25010 standard was published, the old standard is still commonly referred to even today. This is due to ISO9126’s widespread popularity throughout its 20-year long lifetime from 1991 to 2011. The main difference between the two standards is that ISO9126 is less comprehensive than ISO25010, containing only 6 top-level attributes and 27 sub-attributes. However, apart from the missing attributes, the two standards are largely equivalent. Thus, conclusions drawn based on one standard can oftentimes be directly transferred to the other.

2.2 Software Metrics as a Proxy for Software Quality

While the definitions in the ISO25010 standard are sufficiently detailed to be used as a basis for discussions with customers and other stakeholders, they lack quantitative metrics that can be used to measure the described quality attributes [23]. For example, the quality attribute *reusability* is defined as “[the] degree to which an asset can be used in more than one system, or in building other assets” [38]. This aptly communicates the general idea of reusability on a level that is not only understandable for software developers, but also for stakeholders that are less familiar with the software development process. However, due to the abstract nature of the definition, it remains unclear which changes could be made on a code level to improve reusability. Furthermore, the definition does not allow for a direct comparison of reusability across projects, and the same restrictions apply for the remaining quality attributes defined by ISO25010.

A commonly employed strategy that enables comparable, automatable, and reproducible assessments of software quality is to use software metrics as a proxy measure for software quality attributes [10]. Software quality attributes such as maintainability, reliability, and portability are not measured directly then. Instead, they are calculated from a weighted sum of software metric values [97]. Since the results obtained by these calculations are numeric values, quantitative assessments of software quality become possible through this. Additionally, since the calculation of software metrics and derived quality attributes does not require any subjective value judgments, the assessment process can be easily automated. Therefore, the use of software metrics as a proxy for software quality makes it much easier to regularly assess quality trends in a project and to measure the quality of much larger projects than a manual quality assessment process would allow.

The primary quality assessment model used in this thesis is the *Quality Model for Object-Oriented Design Assement* (QMOOD), which was described by Bansiya and Davis [12] in 2002. QMOOD is a hierarchical model that is loosely based on the ISO9126 quality model. For the calculation of quality attributes, QMOOD defines a three-step process. First, eleven design metrics are calculated. Then, each metric is mapped to one of eleven design properties such as coupling, cohesion, and polymorphism. Finally, the six quality attributes that QMOOD covers are calculated as a weighted sum of the design properties. A graphical representation of this process is shown in Figure 2.2, and the relationships between metrics, design properties, and quality attributes are shown in Table 2.1.

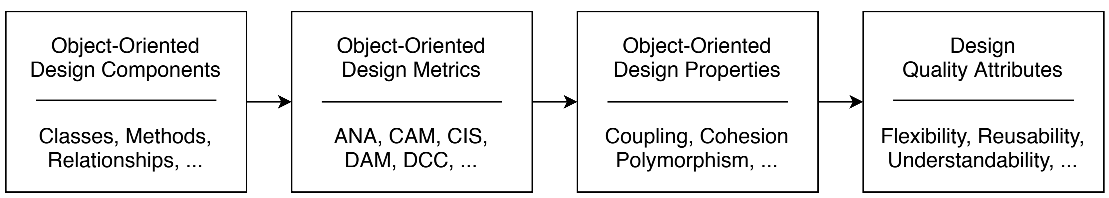
Level 4
Object-Oriented Design Components
Classes, Methods, Relationships, ...
Level 3
Object-Oriented Design Metrics
ANA, CAM, CIS, DAM, DCC, ...
Level 2
Object-Oriented Design Properties
Coupling, Cohesion Polymorphism, ...
Level 1
Design Quality Attributes
Flexibility, Reusability, Understandability, ...

Figure 2.2: Calculation of software quality attributes in QMOOD [12]. Values of the lower-numbered levels are calculated from the values of the higher-numbered levels.

Level 3 Design Metrics	Level 2 Design Properties	Level 1 - Quality Attributes Reu.	 Flex.	 Und.	 Func.	 Ext.	 Eff.
DSC	Design Size	$+0.50$	-	-0.33	$+0.22$	-	-
NOH	Hierarchies	-	-	-	$+0.22$	-	-
ANA	Abstraction	-	-	-0.33	-	$+0.50$	$+0.20$
DAM	Encapsulation	-	$+0.25$	$+0.33$	-	-	$+0.20$
DCC	Coupling	-0.25	-0.25	-0.33	-	-0.50	-
CAM	Cohesion	$+0.25$	-	$+0.33$	$+0.12$	-	-
MOA	Composition	-	$+0.50$	-	-	-	$+0.20$
MFA	Inheritance	-	-	-	-	$+0.50$	$+0.20$
NOP	Polymorphism	-	$+0.50$	-0.33	$+0.22$	$+0.50$	$+0.20$
CIS	Messaging	$+0.50$	-	-	$+0.22$	-	-
NOM	Complexity	-	-	-0.33	-	-	-

Table 2.1: Relationships between design metrics, design properties, and quality attributes as defined by QMOOD [12]. The numbers in the quality attribute columns show the weighting factors used to calculate quality attributes from design properties.

As seen in Table 2.1, quality attributes can be either positively or negatively influenced by metric values. For example, *flexibility* is calculated as $0.25 \cdot DAM - 0.25 \cdot DCC + 0.50 \cdot MOA + 0.50 \cdot NOP$, which indicates that the DAM, MOA, and NOP metrics have a positive effect on *flexibility*, whereas DCC has a negative effect on *flexibility*. Furthermore, each metric can influence one or more quality attributes using potentially different weighting factors for each of them. For example, the ANA metric has positive weighting factors of $+0.50$ and $+0.20$ for *extendibility* and *effectiveness*, but a negative weighting factor of -0.33 for *understandability*. This illustrates clearly that, according to QMOOD, there often is a tradeoff between different quality attributes, because attempts to improve one quality attribute can cause other quality attributes to decrease at a similar rate.

To further diversify the quality analyses that can be performed in this thesis, metrics from the Chidamber and Kemerer metrics suite [20], McCabe's Cyclomatic Complexity [53], and a few additional metrics that can be calculated by the tool *CKJM extended* [43] are used as secondary measures of software quality. Even though these metrics were not intentionally designed as assessors of ISO defined software quality attributes, they are at least occasionally used in this way in existing research, as confirmed by mapping studies such as the ones conducted by Jabangwe et al. [41] in 2015 and Arvanitou et al. [10] in 2017. After all, McCabe's Cyclomatic Complexity metric can, for example, be mapped to the QMOOD defined design property *complexity*. This mapping, in turn, provides information about the quality attributes that are influenced by the metric. Other metrics can be mapped to design properties in a similar way, thus enabling their use as assessors of quality attributes as well. However, since the weighting factors that QMOOD uses might not be accurate for these metrics, it is only possible to infer either an increase or decrease of quality attributes, but no exact magnitude of the quality effect.

2.3 Effects of Design Patterns on Software Quality

Ever since software design patterns were first described by the *Gang of Four* [30] in 1994, researchers have demonstrated an ongoing interest in their development, specification, usage, and evaluation [52]. Early examples of studies investigating the quality effects of design patterns include those published by Schmidt and Stephenson [75] in 1995 and by Christensen and Ron [21] in 2000. Since then, more than 60 studies have been conducted about the quality effects of design patterns [52]. Even today, the quality evaluation sub-field of design pattern research is still seeing active contributions, with around 3-5 papers being published per year in recent years [52].

A comprehensive summary of the quality effects of design patterns was published by Ampatzoglou et al. [4] in 2013. In this systematic mapping study, the authors not only list existing primary research covering the quality effects of design patterns, but also provide a detailed overview of the quality attributes that were investigated by the identified publications. The results of this investigation are presented as a table that shows how many primary studies found either a positive or a negative effect between design patterns and different software quality attributes. A subset of these results that covers the quality effects of behavioral design patterns is shown in Table 2.2.

Design Pattern	Adapt.	Maint.	Stab.	Test.	Und.
Chain of Responsibility	++	++			+
Command	+−	++	+		−
Interpreter	−	++			+
Iterator	++	++			+
Mediator	−−	++		−	+
Memento	−−	−−			−
Observer	++	++	−	−	+−−
State	−−	++−	−		+
Strategy	+−	++	−		+
Template Method	++	++	+		−
Visitor	−−	++++++−−		−	+−−−

Table 2.2: Quality effects of behavioral design patterns as summarized by Ampatzoglou et al. [9]. Each "+" represents one study that found a positive effect, whereas each "−" represents one study that found a negative effect.

As seen in Table 2.2, observed quality effects vary considerably across design patterns and quality attributes. However, at least maintainability appears to be largely positively affected according to the covered primary studies. A more complete overview of the quality advantages, disadvantages, and trade-offs of all 23 GoF design patterns can be found in the original paper by Ampatzoglou et al. [4]. Meanwhile, a more recent discussion of quality evaluation studies is presented in the systematic review conducted by Wedyan and Abufakher [95] in 2020, which finds similar results and explicitly highlights the sometimes contradictory observations about the quality effects of design patterns.

Research methods that are commonly used to investigate the quality effects of design patterns include conceptual analyses, controlled experiments, and case studies [4]. These methods are described in more detail in the following paragraphs. The information that is thereby provided consists of a short description of each method as well as a discussion of the advantages and disadvantages that each method exhibits when used for the purpose of design pattern quality evaluation.

In **controlled experiments**, researchers ask study participants to complete specific programming tasks in equivalent pattern and non-pattern projects [95]. Pattern and non-pattern projects are then compared across metrics such as task completion times [48], correctness of implemented solutions [66], and perceived implementation difficulty [74]. Since metrics like these are particularly relevant in real work environments, this helps explain the popularity of controlled experiments. However, to enable participants to complete the given tasks in a reasonable amount of time, the used projects are usually rather small, and the tasks relatively simple. Thus, any results that are found might not be applicable to larger projects and more complex tasks [48]. Furthermore, since human participants are needed to conduct controlled experiments, it is difficult to use this research method for large-scale or long-term evaluations.

In **conceptual analyses**, quality effects of design patterns are evaluated on a purely theoretical level through a comparison of mathematical models that represent pattern and non-pattern solutions [95]. Supported by its mathematical foundation, this approach promises accurate and objective analysis results. However, it is difficult to create models for complete systems due to the rapid increase in complexity of larger models. Because of this, existing studies often only model design patterns themselves [7][37]. What is not modeled, however, are interactions between different design pattern instances, or interactions between design patterns and non-pattern classes. Thus, although existing models have proven to be useful for quality evaluations on the pattern level, quality evaluations on the system level have yet to be achieved through conceptual analyses.

In **case studies**, existing projects with varying degrees of design pattern usage intensity are compared utilizing information such as software metrics [5], version history [11], and bug tracking information [31]. The projects that are usually used for these comparisons are open-source or industrial projects of various sizes [95]. This use of real-life projects as the object of analysis ensures that the results that are found by case studies not only serve to fulfill scientific curiosity, but also provide insights that are relevant beyond academia [73]. However, a recurring shortcoming of existing studies is that they only evaluate the quality effects of design patterns on a relatively small scale, often analyzing no more than a handful of different projects [95]. Therefore, the question arises how generalizable the results of such small-scale investigations are.

To accomplish the research goal of this thesis, which is to perform a large-scale analysis of the quality effects of design patterns in existing projects, the case study method is the most appropriate one. This is partly because case studies use existing projects as their object of analysis, while conceptual analyses, for example, only perform a theoretical analysis. Furthermore, case studies can be used for a large-scale analysis much more easily than controlled experiments can, because no human study participants are needed.

2.4 Mining Software Repositories

As mentioned in the previous section, many existing case studies investigate the quality effects of design patterns on a relatively small scale (see Figure 2.3) [95]. This is not necessarily problematic on its own as far as the results of these studies are concerned. However, given their limited sample size, projects in these studies might not accurately capture the wealth of diversity that is present in software projects of different domains, sizes, runtimes, etc. Therefore, any conclusions that are drawn based on the results of small-scale studies might only apply to a small subset of similar software projects.

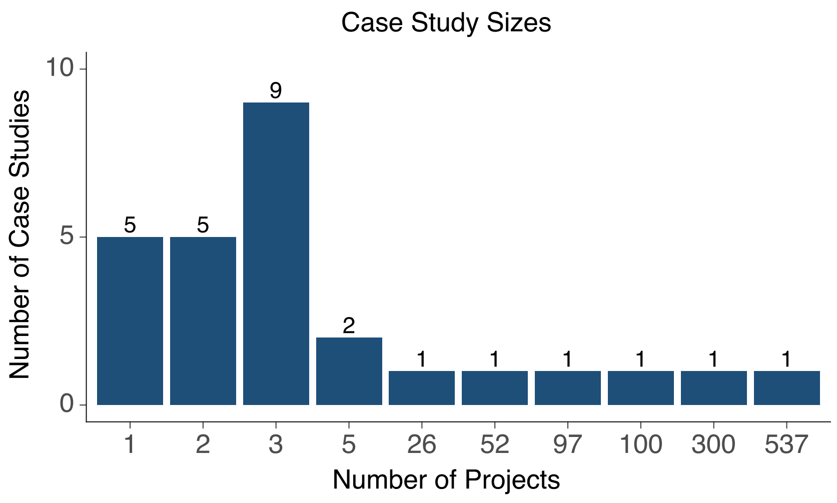

Figure 2.3: Overview of existing case study sizes as measured by the number of projects that the studies analyze [95]. Only 8 out of 27 studies (30%) cover more than 3 projects in their analysis, and even the largest study covers only 537 projects.

To achieve more generalizable results, larger-scale analyses that cover a more diverse set of projects are needed. An area of research that focuses on such large-scale analyses is *Mining Software Repositories* (MSR) [22]. The basic premise of MSR studies is to utilize the information that is available in source control systems, defect tracking systems, and archived project communication to discover interesting or actionable insights into the projects that are investigated [3][22]. Research in MSR is often supported by the use of tools that partially or fully automate the data collection and analysis process [19][34], thus enabling much larger-scale investigations than a manual process would allow.

This thesis follows the same ideas as other MSR studies. Projects to analyze are retrieved from a large software project repository that contains tens of thousands of projects. Because such a large number of projects are analyzed, the results that are found have the potential to be more generalizable than the results of smaller-scale studies. Furthermore, data collection and analysis are automated. As a result, the analyses that are conducted as part of this thesis can be performed in a reasonable timeframe even though the number of analyzed projects is two orders of magnitude higher than in the largest existing case studies that investigate the quality effects of design patterns.

Chapter 3

Methodology

As described in Section 1.3, the primary research goal of this thesis is to conduct a large-scale analysis of the quality effects of design patterns. To accomplish this, several intermediary steps have to be completed. These start with the selection of projects that should be analyzed and end with an analysis of software metrics calculation results and design pattern detection results that might provide insights into the effects that the use of design patterns has on the quality of software projects. A high-level overview of these steps is shown in Figure 3.1.

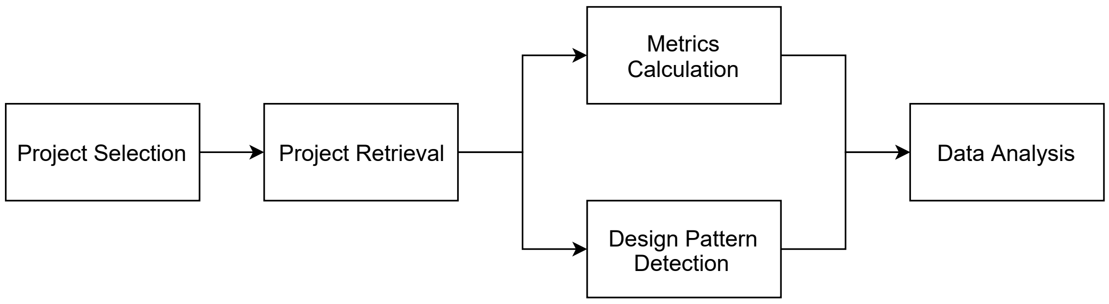

Figure 3.1: Overview of the steps that are necessary to accomplish the research goal.

The following sections provide detailed descriptions for each of these steps. Section 3.1 covers the process and criteria that were used to find and select software project repositories from which projects to analyze could be retrieved. Sections 3.2 and 3.3 describe how the tools for software metrics calculation and design pattern detection were selected. Finally, Section 3.4 describes the data collection and processing pipeline that was built, including information on both the architecture of the pipeline as well as its implementation.

3.1 Software Repository Selection

3.1.1 Selection Criteria for Software Repositories

Before any software metrics can be calculated or any design patterns can be detected, a set of projects that should be analyzed has to be defined. The selection of these projects plays a vital role because it has a significant effect on the outcome of the analysis. After all, representative and generalizable results can only be achieved if enough projects are included in the analysis and the projects themselves cover a broad range of project types, sizes, maturity levels, etc. Further technical requirements arise due to the tools that are used during software metrics calculation, design pattern detection, and analysis.

A list of primary selection criteria that were applied during project/repository selection is shown below and further explained in the following paragraphs:

1. at least 10,000 projects have to be available,
2. projects have to be implemented in Java,
3. projects' source code (i.e., *.java files) has to be available,
4. projects' bytecode (i.e., *.class files) has to be available.

Ad 1.: Since one of the main goals of this thesis is to analyze the quality effects of design patterns on a much larger scale than existing studies do, a sufficiently large number of projects have to be available in the selected repository. Among the largest existing studies so far [95] are two studies that were conducted by Ampatzoglou et al. [6] in 2015 and Hussain et al. [35] in 2017. The study by Ampatzoglou et al. is based on the Percerons repository [9] and covers 537 projects, whereas the study by Hussain et al. is based on a subset of 51 projects from the *Qualitas.class Corpus* [84]. Using a set of projects that is at least one or two orders of magnitude larger than this, i.e., on the order of thousands to tens of thousands of projects, is, therefore, necessary to achieve a sufficiently large improvement in generalizability over the state of the art.

Ad 2.: According to the TIOBE index, Java is still the most popular programming language today [87]. Prompted by this, and because a cross-language analysis of design patterns is beyond the scope of this thesis, the decision was made to cover only Java projects in the conducted analysis. This decision is very much in line with common practice in existing research about the quality effects of design patterns, which also predominantly uses Java projects as the object of analysis. However, as a consequence of this restriction, some of the results that are observed during analysis might not be applicable to other programming languages even though design patterns themselves do not have this limitation. This is further discussed as a threat to validity in Section 5.4.

Ad 3.: To ensure that only Java projects are analyzed, programming languages that are used in the retrieved projects are identified through *enry* [81], which is an open-source tool for programming language detection. Since *enry* performs programming language detection based on source files, projects' sources have to be available through the repository from which the projects are retrieved.

Ad 4.: The tools that are used for software metrics calculation and design pattern detection are *CKJM extended* [43] and *SSA* [88], respectively. Since both tools can only process Java bytecode, bytecode has to be available for all projects that are included in the analysis. Although bytecode could be generated from source code through a compilation step, it would take a considerable effort to automate this process for projects with different structures and varying compilation dependencies, as seen in similar efforts by Martins et al. [50] and Palsberg and Lopes [64]. Furthermore, compilation would lead to significantly increased processing times and is, therefore, not feasible at the required scale given the hardware resources that are available for this thesis.

Several additional properties of repositories were examined during repository search and evaluation to gauge how much effort is required to automate project retrieval and which analyses can be performed when using a specific repository. These properties include, for example, the possibility of individual project downloads, rate-limiting of downloads, the total size of projects on disk, availability of project history, and availability of issue tracking information. To what extent these properties influenced repository selection is further discussed in Section 3.1.3.

3.1.2 Available Software Repositories

A set of candidate repositories that could be used as sources of projects in this thesis was collected through a semi-structured manual search in existing software repository mining papers. Papers accepted to the *Mining Software Repositories* (MSR) conference in 2018 [67] and 2019 [68] were used as a seed set for the search. Snowballing [96] was then employed to extend the search beyond this initial set of papers, which means that both outgoing as well as incoming references of the papers were followed and examined in multiple iterations to identify additional literature that could be relevant for the stated purpose of finding software repository candidates.

Through this search process, a set of candidate repositories containing Java projects were identified. These repositories can be broadly categorized into:

- hosting platforms for open-source projects,
- artifact repositories used by dependency managers,
- curated datasets compiled specifically for project analysis.

Repositories that belong to the same category are often similar regarding their fulfillment of the defined selection criteria. Hosting platforms for open-source projects usually contain more projects than repositories belonging to the other two categories. However, they generally only provide the source code of the hosted projects, while the bytecode is missing. Artifact repositories as well as curated datasets, on the other hand, usually do provide both bytecode and source code, but tend to have fewer projects available.

An overview of the recorded properties, i.e., repositories' fulfillment of selection criteria, is shown in Table 3.1 for the most promising identified repositories from each category.

Repository	Category	Java/JVM Projects	Source Code	Bytecode
50K-C [50]	Curated Dataset	50,000 [50]	yes	yes
AndroZoo [1]	Curated Dataset	>3,000,000 [1]	no	yes
GitHub [32]	Open-Source Hosting	>1,400,000 [59]	yes	no
Maven Central [58]	Artifact Repository	>100,000 [69]	yes	yes
SourceForge [80]	Open-Source Hosting	>50,000 [59]	yes	no

Table 3.1: Fulfillment of selection criteria for the most promising identified repositories.

3.1.3 Selected Software Repositories

Based on the information that was gathered throughout repository search and evaluation, the repository that was selected for use in this thesis is the Maven Central repository, which is the default artifact repository from which the Apache Maven build tool downloads external dependencies [86]. Among the evaluated repositories, Maven Central is the repository that best fulfills the defined selection criteria. It contains a total of more than 100,000 Java projects, which is well above the stated minimum requirement of 10,000 projects, as well as both source code and bytecode for these projects, which are needed to perform metrics calculation and design pattern detection.

Beyond fulfillment of the primary selection criteria, the Maven Central repository exhibits several additional properties that are beneficial for large-scale automated processing. Most importantly, it does not impose any rate-limiting on project downloads that would artificially increase the time required to retrieve projects from it. Furthermore, projects can be downloaded from the repository individually, thereby enabling easy distribution of project retrieval as well as metrics calculation and design pattern detection across multiple processing threads or machines.

Although the Maven Central repository is particularly well-suited for automated processing, it is also rather limited in terms of additional project information that it provides. Project history is available at the level of releases, but the repository does not contain any issue tracking information or more detailed version history on the level of commits. While none of this information is required for the analyses performed as part of this thesis, the listed limitations do, at least to some degree, restrict the potential for further research that builds on top of the dataset provided by this thesis.

As seen in Table 3.1, the only repository other than the Maven Central repository that fulfills all primary selection criteria is the 50K-C repository, which is a subset of GitHub repositories for which bytecode could be successfully created through an automated compilation procedure [50]. Including projects from this repository in the analysis in addition to the projects retrieved from the Maven Central repository could potentially lead to improved analysis results (see Section 6.4, *Further Research*), but was not done as part of this thesis due to the additional effort that would be required to make the processing pipeline compatible with multiple different sources of projects.

3.2 Design Pattern Detection

3.2.1 Selection Criteria for Pattern Detection Tools

The following primary selection criteria were employed during the selection process for design pattern detection tools:

1. detection of design patterns in Java projects has to be possible,
2. command-line execution of the tool has to be possible,
3. the tool should be free to use without restrictions,
4. a sufficiently large number of design patterns should be detectable.

Ad 1.: As described in Section 3.1.1, the analyses that are conducted as part of this thesis were deliberately limited to Java projects. Because of this, any tool that does not support design pattern detection in Java projects has to be disqualified as a potential candidate without further consideration, even if the tool satisfies the remaining selection criteria better than other tools do.

Ad 2.: Due to the large number of projects that have to be analyzed, it is not feasible to manually execute the selected design pattern detection tool for each project. Therefore, all tools have to be evaluated regarding their support for automated execution. Ideally, tools should offer a command-line interface through which a design pattern detection run can be started and should output their results in a well-known and easily parsable format such as XML, JSON, or CSV, thereby enabling simple integration into an automated data collection and processing pipeline. Tools that have to be excluded based on this selection criterion are primarily ones that are implemented as Eclipse [28] plug-ins, which is a commonly used implementation type among design pattern detection tools.

Ad 3.: To improve the reproducibility of the results achieved in this thesis, any tools that are selected should be publicly available without any associated costs or usage restrictions. Even though the use of tools that do not satisfy these selection criteria wouldn't necessarily prevent attempts to reproduce the achieved results, it would certainly make such attempts more time-consuming due to the additional effort that would be required to obtain the needed tools. Furthermore, even after the tools are obtained, they would still have to be manually added to the *Qualisign* project or linked to it in some way, whereas tools that are free to use can be directly distributed with the project.

Ad 4.: Analyses that are performed as part of this thesis should allow for comparison of quality effects across different design patterns. To enable such comparisons, it is necessary that the selected design pattern detection tools can detect a sufficiently large number of design patterns. While it would be ideal to detect and analyze the quality effects of all 23 *GoF* design patterns, established design pattern detection tools generally are only able to detect a subset of them [94]. Thus, the number of detectable design patterns can be used as a deciding factor between tools that are otherwise equally well-suited for use in this thesis, favoring tools that can detect a larger number of design patterns over tools that can only detect a few of them.

Many additional factors could be taken into consideration during selection of design pattern detection tools. For example, tools could be evaluated regarding their precision and recall to reduce the risk of drawing wrong conclusions based on inaccurate pattern detection results. Additionally, performance characteristics could also be used as a primary selection criterion to exclude tools that cannot produce results at the scale required for this thesis in a reasonable timeframe. However, information about detection accuracy and performance is not provided for all available tools [62]. Furthermore, even when these characteristics are reported, measurements are often not directly comparable across tools due to the lack of an established and commonly used benchmark for design pattern detection tools [51].

3.2.2 Available Pattern Detection Tools

An overview of available design pattern detection tools and techniques has been created on several occasions by various design pattern researchers. More recent studies with this goal include the ones conducted by Dong et al. [25] in 2009, Rasool and Streitfdert [70] in 2011, Al-Obeidallah et al. [62] in 2016, and Wang et al. [94] in 2018. Each of these studies lists between 15–30 primary studies about the topic of design pattern detection, though there is some overlap between them. In addition to such studies that specifically target design pattern detection tools and techniques, references to primary research about the topic of design pattern detection can also be found regularly in systematic mapping studies that cover larger parts of the design pattern body of knowledge. For example, Mayvan et al. [52] have identified more than 100 primary studies about design pattern detection.

It should be noted, however, that not all primary studies in design pattern detection research do provide ready-to-use tools that implement the detection approaches that are described by them. Some studies only offer theoretical explanations of the suggested approaches, and others have only implemented their detection strategies in internal prototypes that are not publicly available. Additionally, some studies do contain download links for implemented tools, but the links are either dead or redirect to unrelated web pages. Thus, the total number of publicly available design pattern detection tools is much smaller than the supposed upper limit of more than 100 tools that the number of primary studies found by Mayvan et al. [52] might suggest.

A subset of publicly available design pattern detection tools is shown in Table 3.2. All of the listed tools do support design pattern detection in Java projects. Even beyond the listed subset, this selection criterion turned out to be fulfilled by a very large majority of identified design pattern detection tools. Generally speaking, most tools either support pattern detection only for Java projects or for projects implemented in a small set of different programming languages, with Java usually being one of them. Similarly, all identified tools with publicly available implementations can be used for free without any usage restrictions. Thus, none of the tools described in existing literature had to be excluded based on such restrictions, and only a few tools were excluded due to missing Java support.

Tool	Type	Year of Publication	Detectable Patterns
DP-CORE [24]	Standalone CLI	2016	6
DPF [13]	Eclipse Plug-In	2014	18
MARPLE [98]	Eclipse Plug-In	2015	5
PINOT [79]	Standalone CLI	2006	13
Sempatrec [2]	Eclipse Plug-In	2014	11
SSA [88]	Standalone CLI/GUI	2006	15

Table 3.2: Subset of publicly available design pattern detection tools that support design pattern detection in Java projects. All tools are free to use.

3.2.3 Selected Pattern Detection Tools

The design pattern detection tool that was selected for use in this thesis is *SSA*. It was first described and implemented by Tsantalis et al. [88] in 2006 and uses similarity scoring between graphs for the purpose of design pattern detection. At an abstract level, this works by representing both the design patterns that should be detected as well as the projects in which to search for the patterns as graphs. Similarities between these graphs are then calculated to identify design patterns in the projects. Since the implemented algorithm doesn't require exact matches between graphs, it is capable of detecting not only canonical versions of design patterns, but can also identify implementation variants that don't exactly match the design pattern structures as they are described in literature.

SSA fully satisfies all the primary selection criteria. For example, it is distributed as an executable JAR file that can be started through a command-line interface, thus ensuring that the execution of the tool can be automated. Additionally, results of pattern detection runs are provided in an XML format, which further aids integration into the processing pipeline of the *Qualisign* project. Since execution of *SSA* only requires a Java installation, it can be distributed with the *Qualisign* project quite easily and without complicating its setup process, because the *Qualisign* project itself already uses Java. Another advantage of *SSA* is that it implements many performance optimizations which ensure fast design pattern detection times even for larger projects. This makes *SSA* particularly suitable for large-scale analyses such as the one that is conducted in this thesis.

Among the more negative characteristics of *SSA* are its limited number of detectable design patterns and its mediocre recall (24–52%) and precision (51–80%) [15]. While 15 detectable patterns are enough to gain at least some insights into the quality effects of design patterns, more complete coverage of the full set of 23 *GoF* design patterns certainly would be beneficial, not only for the analyses that are directly conducted as part of this thesis, but also to further improve the usefulness of the dataset that is created. Similarly, higher precision and recall values would also increase the credibility of the observed results and the created dataset. However, even though *SSA* has these flaws, other suitable tools generally don't fare much better. For example, *PINOT* can detect 13 compared to *SSA*'s 15 design patterns, but has even lower recall (13–50%) and precision (9–78%) [15] and is less convenient to set up and use.

3.3 Software Metrics Calculation

3.3.1 Selection Criteria for Software Metrics Tools

The selection criteria that were used during selection of software metrics tools follow the same basic ideas as those that were used during selection of design pattern detection tools in Section 3.2.1. Selection criteria 1–3 remain unchanged, requiring tools to support metrics calculation for Java projects, to be executable through the command-line, and to be free to use without restrictions. Selection criterion 4 now requires a sufficiently large number of metrics that can be calculated, rather than a sufficiently large number of design patterns that can be detected. In addition, a new selection criterion 5 is introduced that requires metrics calculation tools to perform their calculations based on bytecode rather than source code.

The full list of primary selection criteria is shown below and explained in more detail in the following paragraphs:

1. calculation of metrics for Java projects has to be possible,
2. command-line execution of the tool has to be possible,
3. the tool should be free to use without restrictions,
4. a sufficiently large number of metrics should be available,
5. metrics calculation has to be done based on bytecode rather than source code.

Ad 1., 2., 3.: Since these selection criteria exactly match the ones that were employed for the selection of design pattern detection tools, their underlying motivations are the same as described in Section 3.2.1.

Ad 4.: A sufficiently large number of metrics has to be available through the selected metrics calculation tools to ensure that the planned analysis of the quality effects of design patterns can be conducted. Due to the wide variety of software metrics that have been proposed throughout the years [61], there is no exact upper limit to the number of metrics that a tool might calculate. Furthermore, not all existing metrics are equally well-suited as predictors of software quality attributes. However, metrics calculation tools tend to implement more popular metrics which generally have been validated more thoroughly by the research community than less popular ones [10]. Thus, even though the total number of implemented metrics is not a perfect predictor for the suitability of metrics calculation tools as far as their use in this thesis is concerned, it is still useful as a first approximation.

Ad 5.: During evaluation of tools for metrics calculation and design pattern detection, it was found to be highly advisable to either only use tools that are executed on source code or to only use tools that are executed on bytecode. This is because mapping between metrics calculation results and design pattern detection results requires much less effort if all results are generated from the same input files. If source code and bytecode are mixed, mapping between results becomes much harder, because certain language constructs (nested classes, constructors, etc.) are represented in different ways in these files. Furthermore, JAR files containing bytecode sometimes contain additional files that are added during the build process, which makes mapping between results even more difficult. Therefore, since *SAA* - the selected design pattern detection tool - only supports bytecode, the selected metrics calculation tool should also use bytecode.

Further characteristics of metrics calculation tools such as their correctness and performance were considered as potential candidates for primary selection criteria. Information on this level of detail was, however, not readily available in existing literature or through the websites of the tool creators. Because of this, these characteristics were only evaluated for a minority of identified metrics calculation tools after a preliminary selection based on the listed primary selection criteria had already taken place.

3.3.2 Available Software Metrics Tools

To find available software metrics tools, a search strategy was used that covered three different sources through which such tools should be identified. First, a literature search was conducted through Microsoft Academic [55]. Its goal was to find systematic mapping studies (SMS) and systematic literature reviews (SLR) on the topic of software metrics tools. Snowballing [96] was employed to extend the search to relevant results that didn't match the used search terms. This primary search in academic literature was then complemented with a secondary web search to potentially identify additional tools that had only been used outside of a research context and, therefore, might not have been covered by SMSs and SLRs. Finally, design pattern literature used in this thesis was investigated to assess tool popularity among design pattern researchers and to find tools that the primary and secondary searches might not have brought up.

Overall, a total of 93 unique software metrics tools were identified through this process. A large part of them came from SMSs, SLRs, and similar studies that were found through the first search in academic literature about software metrics in general and tools for software metrics calculation in particular. For example, Valenca et al. [89] list 42 tools, Kayarvizhy [45] list 23 tools, and Nuñez-Varela et al. [61] list 14 tools. Despite the broad coverage of these studies, some additional tools could be identified through the secondary web search and the final search in design pattern literature. More specifically, a website by Monperrus [57], which lists a total of 32 tools for software metrics calculation, contains several tools that had not been discovered by the initial literature search. Furthermore, *CKJM extended* [43] - an extended version of the *CKJM* [82] tool - was identified due to its use in design pattern studies, to name just two examples.

An overview of some of the more promising software metrics tools for Java is presented in Table 3.3. As shown in this table, available tools are rather diverse regarding their fulfillment of the defined selection criteria. Most tools are standalone tools that offer either a command-line interface, a graphical user interface, or both. Furthermore, a majority of the listed tools is free to use, with *CKJM*, *CKJM extended*, *Dependency Finder*, *jPeek* and *Eclipse Metrics Plugin* being open-source while *JHawk*, *SourceMonitor* and *Understand* are closed-source tools. Required input formats show an even split between bytecode and source code, and the tools can calculate an average of 31 metrics.

Tool	Type	Free to Use	Input	Metrics
CKJM [82]	Standalone CLI	Yes	Bytecode	8
CKJM extended [43]	Standalone CLI	Yes	Bytecode	19
Dependency Finder [85]	Standalone CLI/GUI	Yes	Bytecode	33
JHawk [90]	Standalone CLI/GUI	No	Source Code	86
jPeek [49]	Standalone CLI	Yes	Bytecode	18
Eclipse Metrics Plugin [93]	Eclipse Plug-In	Yes	Source Code	31
SourceMonitor [18]	Standalone GUI	Yes	Source Code	12
Understand [77]	Standalone CLI/GUI	No	Source Code	47

Table 3.3: Some of the more promising metrics calculation tools for Java projects.

3.3.3 Selected Software Metrics Tools

Among the software metrics tools that fulfill all the defined selection criteria, the one that was found to be most suitable for use in this thesis is *CKJM extended*. The tool was originally developed by Spinellis [82] as *CKJM*, primarily supporting calculation of metrics from the Chidamber and Kemerer metrics suite [20]. Jureczko and Spinellis [43] later extended the tool to also support calculation of the metrics defined in the QMOOD metrics suite [12], bringing the total number of metrics that the tool can calculate from 8 to 19 and giving the tool its new name *CKJM extended*. Apart from fully satisfying all the defined selection criteria, benefits of the tool include its good performance, which aids large-scale analysis, and its distribution as an executable JAR file, which makes the tool very easy to set up and use. Integration into the processing pipeline of the *Qualisign* project is further assisted by the fact that *CKJM extended* outputs its metrics calculation results in an XML format, thereby making the results very simple to work with.

The main reason why *Dependency Finder* [85] and *jPeek* [49] were not selected instead of *CKJM extended*, even though they also satisfy all of the primary selection criteria, is that both tools only support a rather homogeneous set of metrics. In fact, at the time of writing, *jPeek* only supports 18 different cohesion metrics. Similarly, *Dependency Finder* primarily supports basic size metrics such as the number of classes, number of methods, number of method parameters, etc. As far as this thesis is concerned, such a lack of metrics diversity would be quite problematic, because it would restrict the set of quality attributes that can be included in the software quality analysis. For example, according to the QMOOD model described by Bansiya and Davis [12], cohesion is categorized as a factor that only influences reusability, understandability, and functionality, but does not influence flexibility, extendibility or effectiveness. Thus, a more diverse set of metrics is needed to improve the coverage of analyzable software quality attributes.

The use of multiple tools for software metrics calculation was also considered. However, none of the identified tools that did fulfill the primary selection criteria offered a sufficiently large increase in metrics diversity to warrant the implementation effort required for their inclusion. Therefore, no other metrics calculation tools beyond *CKJM extended* were selected for use in this thesis.

3.4 Data Collection and Processing

3.4.1 Data Collection Pipeline

The goal of data collection is to gather metrics and design pattern data for all projects that should be analyzed and to persist this data in a format that is suitable for further processing. For this purpose, a data collection pipeline was built in Scala. It first retrieves the most recent versions of all projects that are available in the Maven Central repository. Projects are then filtered to remove ones that either are not Java projects at all or are compiled against Java versions that are not supported by the used metrics calculation and design pattern detection tools. Finally, software metrics are calculated through *CKJM extended* and design patterns are detected through *SSA* for all projects that remain after the filtering stage. The results of both tools are stored in a PostgreSQL database.

A graphical representation of the data collection pipeline that shows the three data collection stages and their sub-steps is shown in Figure 3.2. Further details about the pipeline stages and their sub-steps are explained in the following paragraphs.

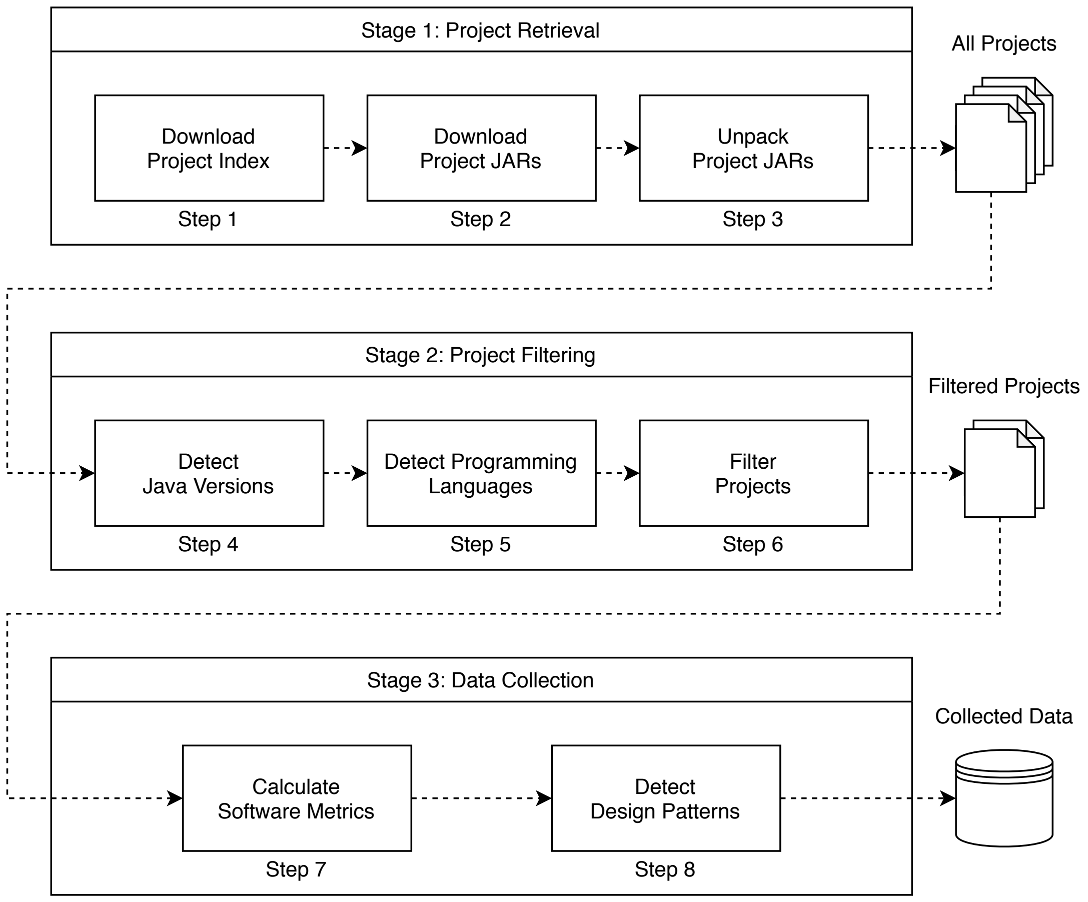

Figure 3.2: Overview of the data collection pipeline.

Stage 1 - Project Retrieval: Before any project downloads can be started, the data collection application has to identify the newest versions of all projects that are available in the Maven Central repository. This is accomplished by first downloading the project index that the Maven Central repository provides, and by then querying this index for the required meta-data which includes the groupId, artifactId, and newest version of all projects. With this meta-data, the download URLs of the projects' bytecode and source code JARs are constructed, and the JARs are downloaded. Since the tools that are used in later pipeline stages can only process the *.java and *.class files that are contained inside of the JARs, but not the JARs themselves, the final step of the project retrieval stage unpacks the JARs to reveal the needed *.java and *.class files.

Stage 2 - Project Filtering: The tools used for metrics calculation and design pattern detection can only be executed on projects compiled against Java versions up to and including Java 8. Furthermore, only projects consisting of 100% Java source code should be included in the analysis. The Maven Central repository, however, also hosts projects with newer Java versions as well as projects that are fully or partially implemented in other JVM-based languages such as Scala, Kotlin, and Groovy. To distinguish unsuitable projects from suitable ones, Java versions of the downloaded projects are read from the projects' *.class files, and used programming languages are detected through an open-source tool called *enry* [81]. A filtering step is then performed to exclude all unsuitable projects from further data collection steps. Thus, only Java projects with Java versions that are less than or equal to Java 8 are left after the filtering step.

Stage 3 - Data Collection: In the final stage of the data collection pipeline, metrics calculation and design pattern detection are performed through *CKJM extended* and *SAA*. Both tools take the bytecode of the projects that passed the filtering stage as their input and produce one XML file per project as their output. Since these XML files would be inconvenient to work with during further processing due to XML's limited query capabilities, the metrics and design pattern data is then read from the XML files and stored in a PostgreSQL database.

3.4.2 Collected Metrics and Design Pattern Data

An entity-relationship model of the collected data is shown in Figure 3.3. While its overall structure is quite simple, some elements of the model warrant further discussion.

The `fraction` attribute of the `project_languages` entity represents the fraction of code that uses a given programming language. Its calculation is based on the bytes of code that are written in each language. Since only pure Java Tprojects should be used, suitable projects must have a single project language with a `name` of "Java" and a `fraction` of 1.

The value contained in the the `java_version` attribute of the `projects` entity is the `major_version` of a project's *.class files as specified by JSR-202 [42]. Java version 8 corresponds to a `major_version` of 52. Thus, all suitable projects must have a `java_version` value that is less than or equal to 52.

Progress of the data collection pipeline is shown through the `[step_n]_status` attributes of the `projects` entity. Here, `step_n` refers to the eight pipeline steps in Figure 3.2. Values can be either 0 for pending steps, 1 for successfully completed steps, or larger than 1 for unsuccessfully completed steps. Because only projects that pass all steps successfully can be included in the analysis, processing of a project stops with the first unsuccessfully completed step.

Design pattern data is splitt across two different entities: `pattern_instances` and `pattern_roles`. This is necessary because design patterns generally have multiple participating classes. For example, each `instance` of the "Observer" `pattern` consist of a single class with a `role` of "Observer" as well as one or more classes with a `role` of "Subject". Additionally, *SSA* lists a "Notify()" role for this pattern that refers to the method through which observed changes are communicated to the subjects. The `element` attribute of a `pattern_roles` entity contains a fully-qualified class, method, or field name, i.e. `package.Class`, `package.Class::method()`, or `package.Class::field`.

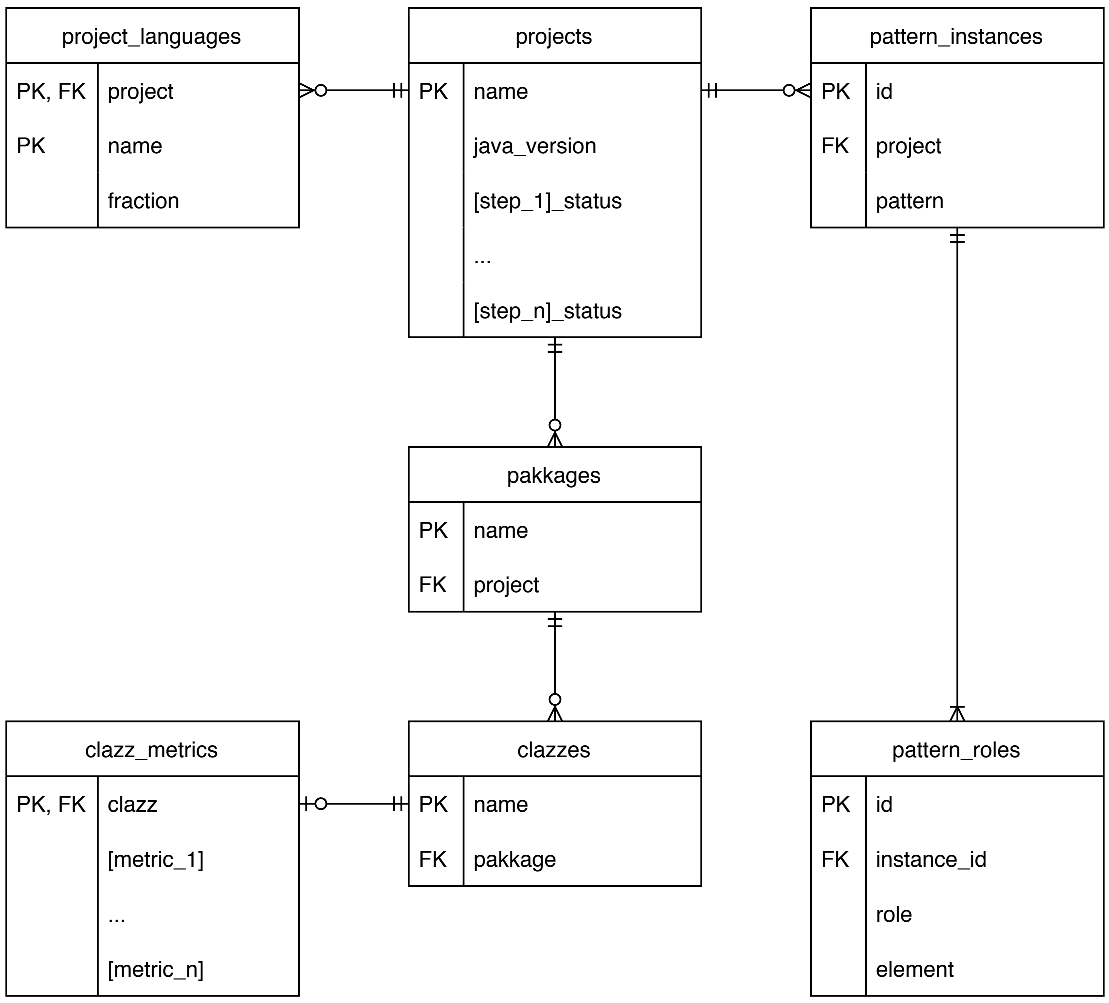

Figure 3.3: Entity-relationship model of the collected data.

3.4.3 Data Collection Issues

Overall, 34% of projects that were identified through Maven Central's project index successfully passed all steps of the data collection pipeline. More specifically, metrics and design pattern data could be collected for 89,621 out of 263,572 projects that were hosted in the Maven Central repository when the data collection pipeline was started on 2020-03-09. Table 3.4 offers a more detailed breakdown of the number of projects that remained after each of the eight data collection steps. The most important issues that occurred during data collection are explained in the following paragraphs.

#	Step	Projects	% of Total
1.	Download Project Index	263,572	100.0
2.	Download Project JARs	246,889	93.7
3.	Unpack Project JARs	244,260	92.7
4.	Detect Java Versions	215,105	81.6
5.	Detect Programming Languages	199,131	75.6
6.	Filter Projects	129,620	49.2
7.	Calculate Software Metrics	89,683	34.0
8.	Detect Design Patterns	89,621	34.0

Table 3.4: Remaining projects after each step of the data collection pipeline.

Step 2, downloading of project JARs, fails for around 17,000 projects. The primary reason for this is that projects that are hosted in the Maven Central repository are only required to provide bytecode JARs. Although many projects also provide source code JARs, some projects do not do this, causing downloads of their source code JARs to be unsuccessful.

Most of the 29,000 projects that are lost in step 4 do not contain any *.class files at all, which causes version detection to fail. Even if version detection was done through other means, no data could be extracted from these projects, since metrics calculation and design pattern detection also use *.class files as their inputs.

The 16,000 failures during step 5 are primarily caused by projects that exceed the two-minute execution time limit that is enforced independently for steps 3, 5, 7, and 8. These limits were set to prevent very large projects that cannot be processed in a reasonable timeframe from blocking the data collection pipeline.

In step 6, almost all of the 40,000 failures are caused by exceptions that are thrown by dependencies of the *CKJM extended* tool when executing the tool on the given projects. Even though these failures were investigated, no working fixes were found for them.

The remaining losses of projects can be ascribed to project filtering in step 6 and a small number of random failures such as tool crashes and Docker issues that occasionally occurred throughout the whole data collection pipeline.

3.4.4 Data Preprocessing

Once the data collection process concludes, the raw metrics and design pattern data is available in the used PostgreSQL database in a normalized format. Projects that successfully passed all steps of the data collection pipeline contain complete data, whereas all other projects contain only incomplete data. The purpose of data preprocessing is to take this raw data and to prepare it for analysis. To accomplish this, operations such as filtering of incomplete data and aggregation of metrics data on the package and project levels are applied. Furthermore, feature vectors for classes, packages, and projects are created that contain the preprocessed data in a pre-joined, denormalized format that is better suited for analysis than data in a normalized format.

An overview of the preprocessing pipeline, which is implemented as a series of SQL queries, is shown in Figure 3.4. Further details about the four stages of the pipeline are explained in the following paragraphs.

Stage 1 - General Preprocessing: In this stage, projects that successfully completed the full data collection pipeline are separated from ones with failed data collection steps. This separation is necessary, because projects that have not completed the data collection pipeline contain incomplete data, which makes them unsuitable for analysis. To accomplish the desired separation, materialized views are created for projects, packages, classes, pattern instances, pattern roles, and class metrics. These views have the same structure as the original database tables described in Section 3.4.2. However, only data from projects without data collection failures is added to them.

Stage 2 - Metrics Preprocessing: The raw metrics data that is collected by *CKJM extended* primarily contains class metrics from the Chidamber and Kemerer [20] and QMOOD [12] metrics suites. However, the QMOOD quality attributes, which can be calculated from the QMOOD metrics, are not provided by the tool. To make them available for analysis, they are calculated in this preprocessing stage and stored in a new materialized view that contains both the raw metrics as well as the calculated QMOOD quality attributes. Two additional materialized views are then created, containing aggregate data of metrics and quality attributes on the package and project levels. Depending on the specific metric, the calculated aggregate values are either sums, or averages, or both.

Stage 3 - Design Pattern Preprocessing: As seen in the entity-relationship model in Section 3.4.2, the raw design pattern data does not contain any direct relationships between pattern instances or roles on one side, and classes or packages on the other side. To enable analysis of the quality effects of design patterns on the class and package levels, these relationships, therefore, have to be identified and added during preprocessing. A retroactive mapping like this is possible, because the `element` attribute of the `pattern_roles` entity always contains at least a fully-qualified class name through which the mapping to class or package entities can be done. Results of these mapping steps are stored in a materialized view that contains all attributes from the `pattern_roles` entity, the pattern name from the `pattern_instances` entity, and foreign-key attributes that refer to the project, package, and class in which the role occurs.

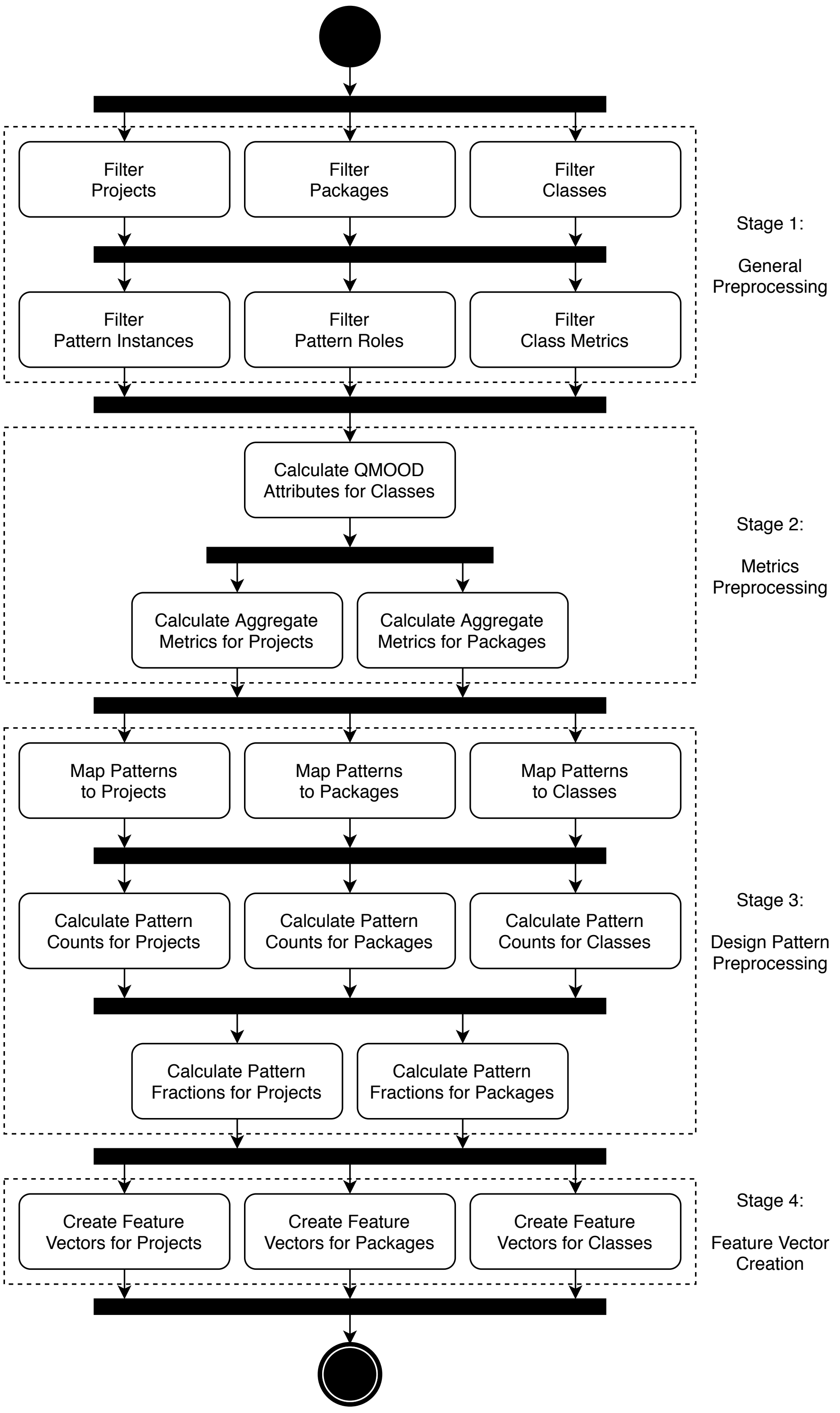
Filter Projects
Filter Packages
Filter Classes
Stage 1:
General Preprocessing
Filter Pattern Instances
Filter Pattern Roles
Filter Class Metrics
Calculate QMOOD Attributes for Classes
Stage 2:
Metrics Preprocessing
Calculate Aggregate Metrics for Projects
Calculate Aggregate Metrics for Packages
Map Patterns to Projects
Map Patterns to Packages
Map Patterns to Classes
Stage 3:
Design Pattern Preprocessing
Calculate Pattern Counts for Projects
Calculate Pattern Counts for Packages
Calculate Pattern Counts for Classes
Calculate Pattern Fractions for Projects
Calculate Pattern Fractions for Packages
Stage 4:
Feature Vector Creation
Create Feature Vectors for Projects
Create Feature Vectors for Packages
Create Feature Vectors for Classes

Figure 3.4: Overview of the preprocessing pipeline.

In addition to the mapping steps, the design pattern preprocessing stage also calculates aggregate values regarding the intensity of design pattern use on the project, package, and class levels. More specifically, it calculates how many design pattern instances are found at these levels and how large the fraction of classes is that do participate in design patterns. Pattern instance counts are calculated per project, package, and class, whereas pattern class fractions are only calculated for projects and packages. For both types of aggregate values (counts and fractions), calculations are done once for each of the 23 design patterns, and then again across all design patterns without distinguishing between their types.

Stage 4 - Feature Vector Creation: In the final stage of the preprocessing pipeline, feature vectors are created for projects, packages, and classes. Each feature vector contains all of the data of the corresponding element (project, package, or class) that is needed during analysis (see Figure 3.5). For feature vectors of projects and packages, this data consists of the pattern class fractions from stage 3 as well as the aggregate metric values from stage 2. For classes, on the other hand, the data consists of the pattern counts (rather than fractions) from stage 2 and the non-aggregated metric values. In addition to this data, the feature vectors also contain boolean values that enable easier selection of specific subsets of feature vectors during analysis. For example, `is_multi_pattern_clazz` marks classes that participate in more than one design pattern, whereas `is_in_pattern_pakkage` marks classes that are part of a package that contains at least one design pattern.

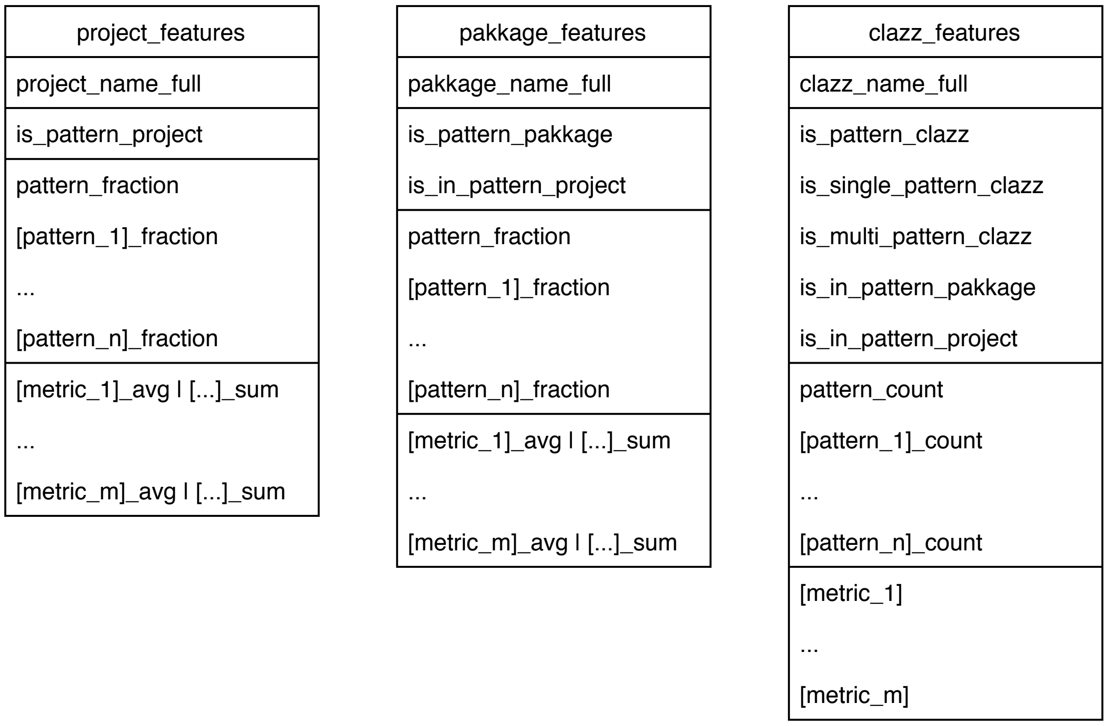

Figure 3.5: Feature vectors for projects, packages, and classes.

3.4.5 Data Analysis

The final data processing step that is conducted as part of this thesis is data analysis. The purpose of this step is to use descriptive and inferential statistics to describe and interpret the data in the created dataset. Thereby, the information that is necessary to answer the posed research questions is collected. More detailed descriptions of the way in which the analyses are conducted are presented in the following paragraphs. The results that are achieved through the data analysis are then presented in Chapter 4.

One of the main results that are produced by this thesis is the created dataset consisting of software metric and design pattern data for projects in the Maven Central repository. The first part of the analysis focuses on descriptive statistics for this dataset. These statistics are created through several Python scripts that first read the required data from the feature vectors in the database and then calculate and visualize the desired statistical information. The results that are produced by these scripts include information such as (i) the number of projects, packages, and classes in the dataset, (ii) value distributions of the collected metrics, and (iii) the number of detected design pattern instances. Results that consist of tabular data are stored as CSV files, whereas results that contain graphical information such as plots are stored as PDF files.

In the second part of the data analysis, the quality effects of design patterns are evaluated. More specifically, correlations between design pattern use and software metric values are calculated and interpreted, thus answering the research questions that are covered by this thesis. The process through which these analyses are conducted follows the same procedure that is used to create the descriptive statistics of the dataset. This means that Python scripts are used to first load the required data from the feature vectors in the database and to then perform the statistical calculations and visualizations. As before, the produced results are stored as either CSV or PDF files, depending on whether they contain tabular or graphical information.

Chapter 4

Results

The results produced by this thesis consist of two main parts. The first part is the created dataset, which contains software metric and design pattern data for projects in the Maven Central Repository. The second part consists of the results of the conducted quality analysis, which uses the dataset created in part one to evaluate the quality effects of design patterns. Both of these parts are described in detail in the following sections. Sections 4.1 to 4.3 describe the dataset itself, providing information about the collected project, software metric, and design pattern data. In Section 4.4, the results of the quality analysis are presented, covering observed correlations between software metrics on one side, and intensity of design pattern utilization on the other. These results are then employed in Section 4.5 to answer the research questions raised in Section 1.3.

4.1 Project Statistics

Through the data collection process, software metric and design pattern data is collected for 89,621 projects retrieved from the Maven Central Repository. As shown in Figure 4.1, no design patterns are detected in 58,323 (65.1%) of these projects, which are therefore classified as non-pattern projects[1]. Correspondingly, one or more design patterns are detected in each of the remaining 31,298 (34.9%) projects, which are classified as pattern projects. Since even the largest existing case studies only analyze a few hundred projects [95], the created dataset achieves a two orders of magnitude improvement in sample size compared to the state of the art.

Overall, the projects that are part of the dataset contain a total of 321,046 packages and 2,969,494 classes. Packages show a similar pattern-to-non-pattern split as projects, consisting of 71.3% non-pattern and 28.7% pattern packages. For classes, the split shifts to 88.6% non-pattern and 11.4% pattern classes. Furthermore, 3.3% of classes are multi-pattern classes that participate in more than one design pattern. Looking only at pattern projects, the percentages of pattern and multi-pattern classes rise to 15.9% and 4.6%, respectively. These results are similar to the findings of Ampatzoglou et al. [6], who observe 20.2% pattern and 4.2% multi-pattern classes in their dataset.

[1]Some projects might be incorrectly classified as (non-)pattern projects. This is because *SSA* can only detect 15 out of 23 design patterns [88] and has less than 100% precision and recall [15].

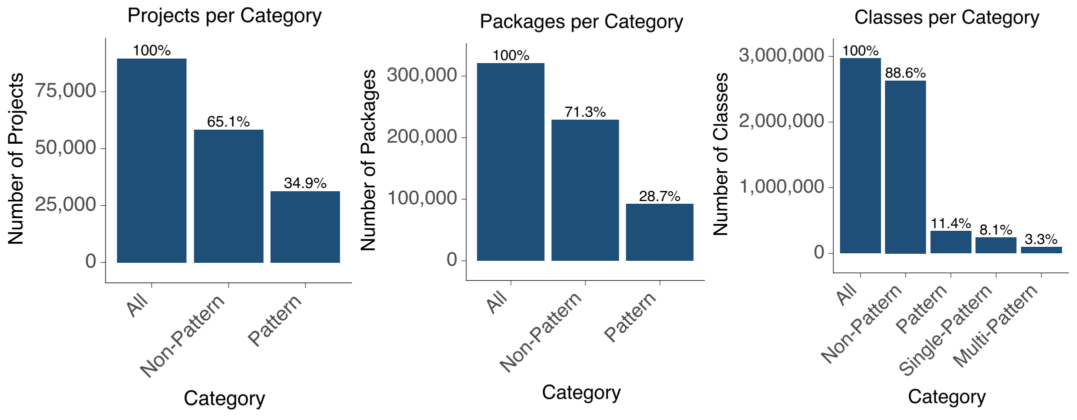

Figure 4.1: Overall number of projects, packages, and classes in the created dataset.

As shown in Figure 4.2, sizes of projects, packages, and classes are rather diverse, with pattern variants often being noticeably larger than their non-pattern counterparts. More specifically, pattern projects, packages, and classes contain a median number of 3,190, 858, and 95 lines of code, respectively. Their non-pattern counterparts, on the other hand, only contain a median number of 464, 237, and 43 lines of code each. Part of the reason for these size differences might be that many design patterns consist of multiple participating classes that exhibit at least some degree of interaction between each other. However, even among pattern classes, smaller ones do exist. For example, most of the pattern "classes" with just a single line of code are interfaces defining the method signatures of *State*, *Adapter*, and *Factory Method* design pattern instances.

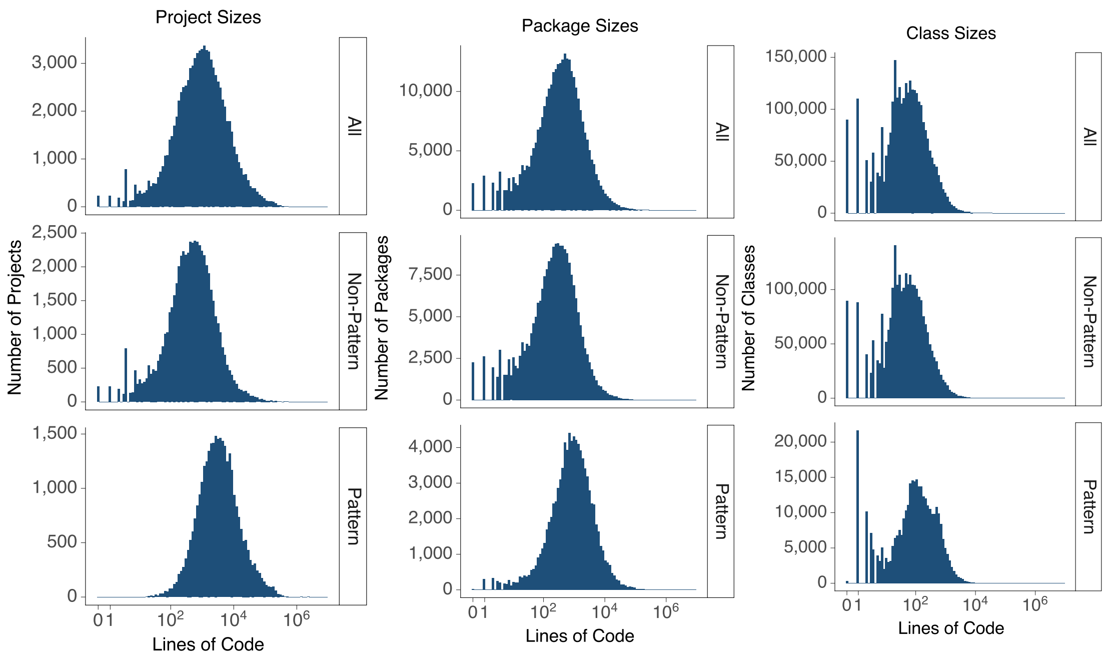

Figure 4.2: Project, package, and class sizes as measured by their lines of code.

4.2 Metric Statistics

In total, 29 software metrics are calculated for every class in the dataset using *CKJM extended* [43]. These metrics consist of six QMOOD quality attributes [12], eleven QMOOD design properties [12], and twelve additional metrics which come primarily from the Chidamber and Kemerer [20] metrics suite, henceforth referred to as *C&K+ metrics*.

The purpose of this section is to provide an overview of the commonalities and differences between the used software metrics. To accomplish this, value distributions of the metrics are first presented as histograms, showing average metric values for all projects in the dataset. These value distributions are then discussed to highlight and explain noteworthy similarities across different metrics. Finally, Spearman's rank correlation coefficients between metrics are presented as a heatmap, thus providing a statistical confirmation of the discussed similarities.

Figure 4.3 shows the value distributions of the six QMOOD quality attributes. Since all six quality attributes are calculated as weighted sums of the eleven underlying QMOOD design properties (see Section 2.2), it is expected that at least some similarities between the quality attributes exist. These similarities are especially apparent across the three quality attributes *functionality*, *reusability*, and *understandability*. The values of these attributes are heavily influenced by the number of classes in each project, either through the DSC (number of classes) or NOH (number of classes without children) property. This is because DSC and NOH are significantly larger in magnitude for most projects than the remaining QMOOD design properties that influence *functionality*, *reusability*, and *understandability* (see Figure 4.4).

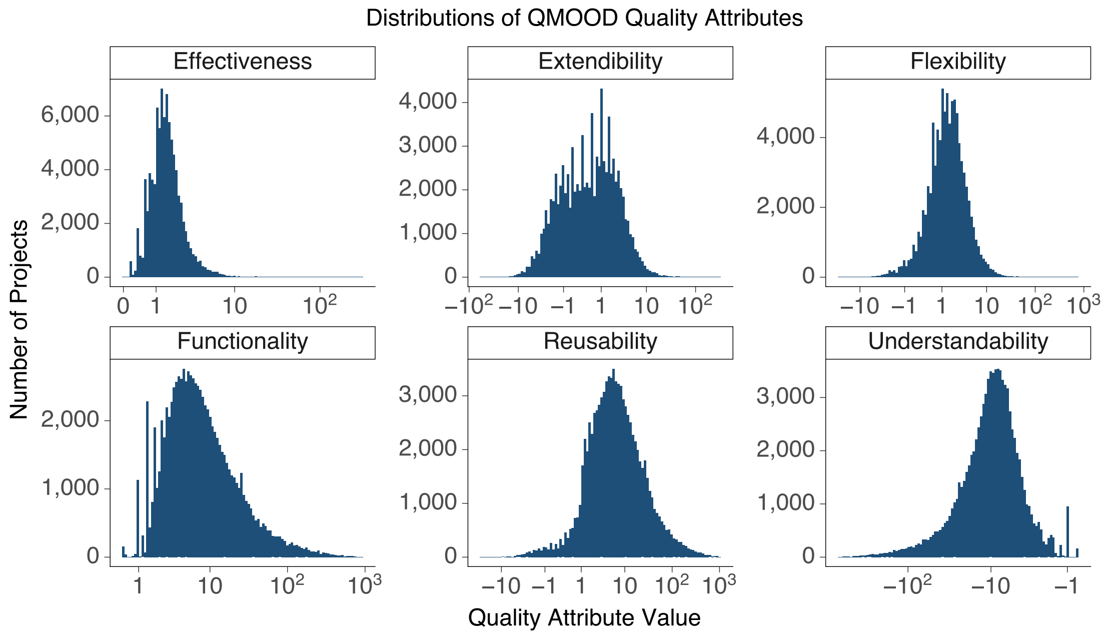

Figure 4.3: Value distributions of the measured QMOOD quality attributes.

Value distributions of the eleven QMOOD design properties are presented in Figure 4.4. Here, two different sets of properties show strong similarities.

The first set of similar properties consists of the two properties DSC and NOH. Since DSC counts the total number of classes in a project, and NOH counts the number of classes without children, similarities between these two properties are expected. After all, NOH necessarily increases at a similar rate as DSC, unless inheritance is used very heavily.

The second set of properties with a high degree of similarity consists of CIS, NOM, and NOP. As before, these similarities are expected, since all three properties count related (sub-)sets of methods. More specifically, CIS counts public methods, NOM counts all methods, and NOP counts polymorphic (i.e., public or protected) methods.

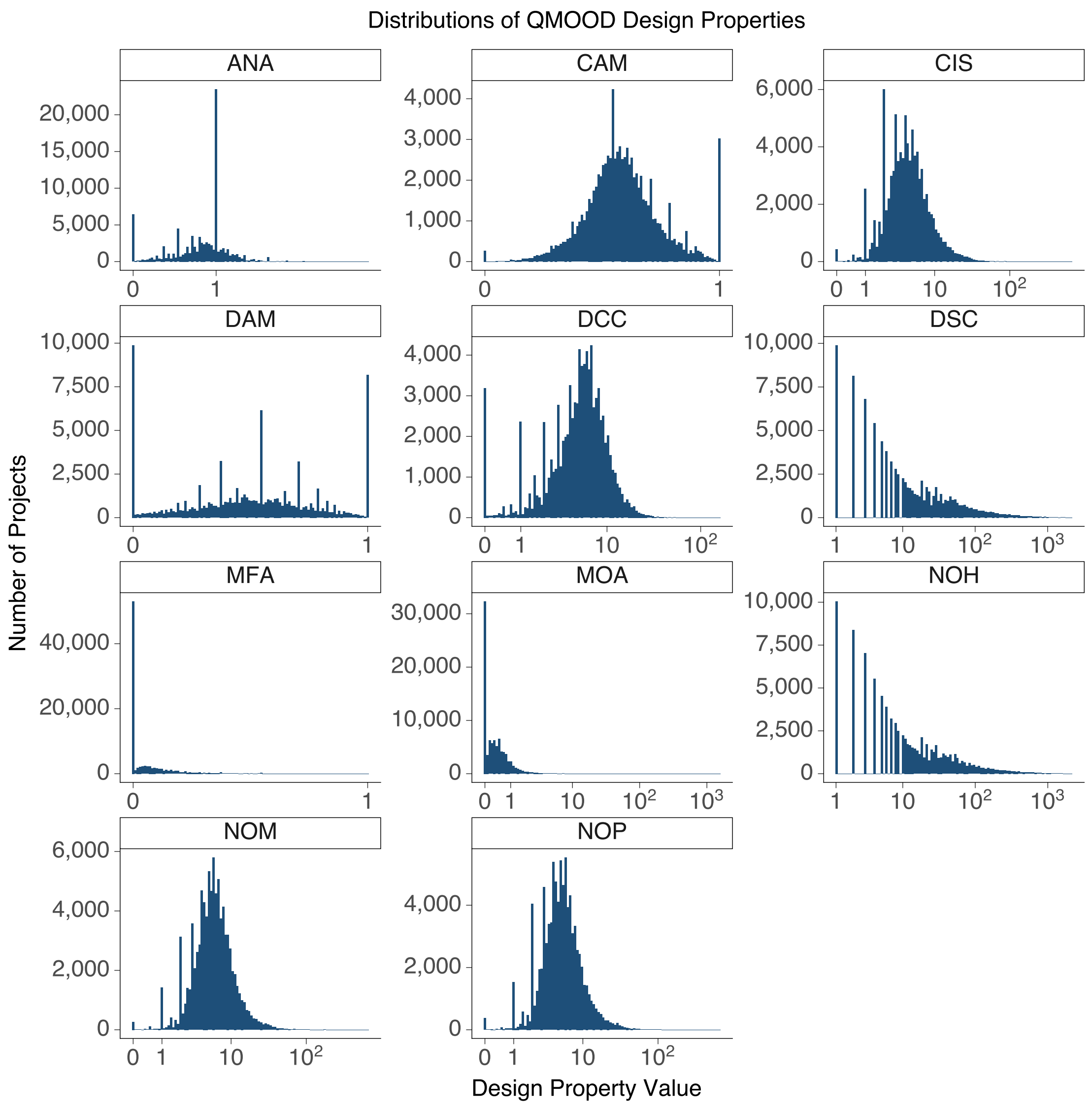

Figure 4.4: Value distributions of the measured QMOOD design properties.

Among the twelve C&K+ metrics shown in Figure 4.5, two more sets of metrics exhibit rather similar value distributions.

Here, the first set of metrics with a high degree of similarity consists of CBM and IC. Their similarity can be explained by the fact that both metrics measure coupling in inheritance hierarchies. While NOC also has a peak at the zero value, its distribution beyond the zero value is distinctly different from the distributions of CBM and IC.

The second set of similar metrics consists of LCOM, RFC, and WMC. All three of these metrics are related to the number of methods in a project. Thus, it is perhaps not too surprising that all of their distributions resemble the shape of the NOM (number of methods) histogram, which was shown in Figure 4.4.

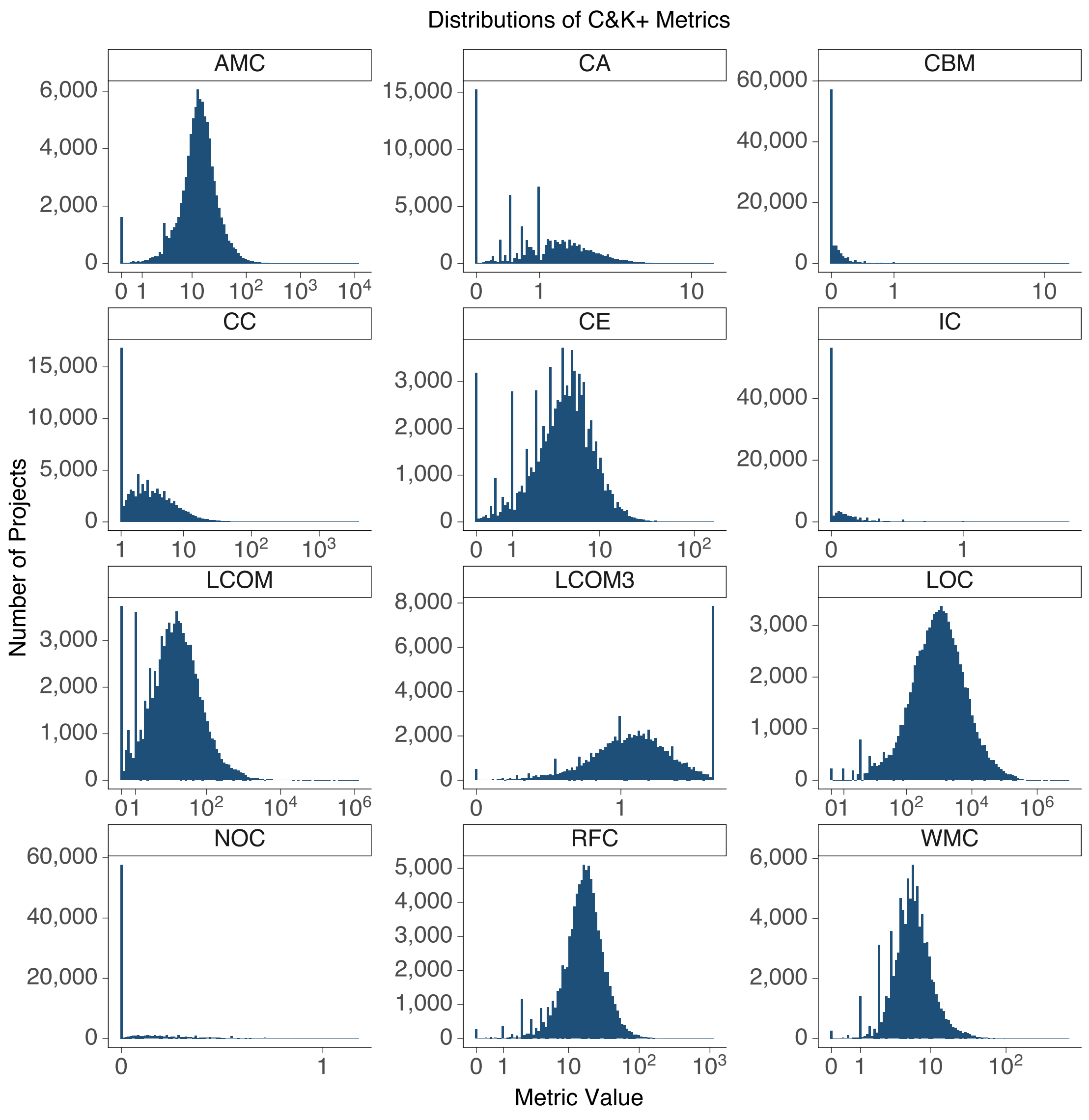

Figure 4.5: Value distributions of the measured C&K+ metrics.

A look at the Spearman correlation coefficients between the QMOOD quality attributes, QMOOD design properties, and C&K+ metrics (see Figure 4.6) confirms the visually observed similarities between the discussed metrics. Each of the five mentioned metric groups shows high correlations between its members. For example, correlation coefficients between *functionality*, *reusability*, and *understandability* all have magnitudes larger than 0.85. The remaining similarity groups show similar results. Among these groups, the lowest correlation coefficient (0.57) is observed between LCOM and RFC. All the remaining correlation coefficients of the similarity groups have values above 0.88.

Beyond the correlations within similarity groups, Figure 4.6 reveals further correlations across different metric types. For example, the design property DSC is highly correlated with the quality attributes *functionality*, *reusability*, and *understandability*. This supports the previously discussed supposition that DSC has a strong effect on these three quality attributes. Similarly, NOP is highly correlated with *effectiveness*, thus uncovering another case where a QMOOD design property strongly influences a QMOOD quality attribute.

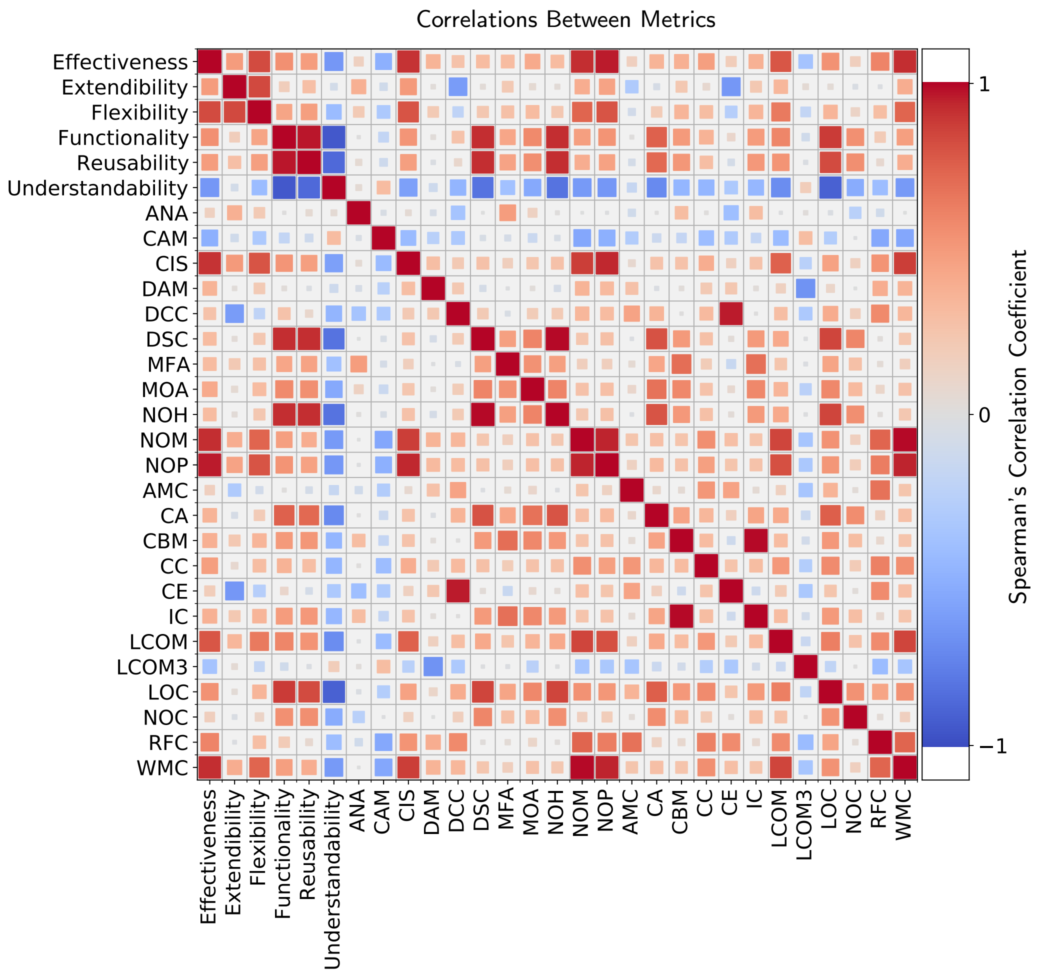

Figure 4.6: Correlations between the measured QMOOD quality attributes, QMOOD design properties, and C&K+ metrics.

4.3 Pattern Statistics

Through the use of the design pattern detection tool *SSA* [88], 15 different *GoF* design patterns are detected in the projects of the dataset. In this section, the results of this detection process are presented. First, the total number of detected pattern instances is shown for each design pattern. Then, distributions of the usage intensities of the different design patterns are presented as histograms. The usage intensity of a design pattern, in this context, refers to the fraction of classes in a project that participate in at least one instance of the given design pattern. Finally, Spearman correlation coefficients between the usage intensities of the detected design patterns are visualized as a heatmap.

Overall, 342.722 design pattern instances are detected in the projects that are part of the dataset. As shown in Figure 4.7, the number of detected instances varies significantly across different design patterns. Whereas the three most used design patterns in the dataset account for 69.8% of overall design pattern instances, the six least used design patterns only account for 2.1% of design pattern instances. In absolute numbers, this means that patterns such as *State* and *Singleton*, which each account for more than 20% of pattern instances, occur roughly twice per pattern project, on average. On the other hand, patterns such as *Composite* and *Command*, which each account for less than 0.5% of pattern instances, occur only once per 50 pattern projects, on average.

These results are consistent with the findings of Ampatzoglou et al. [6]. In their study, they also describe large usage intensity differences across design patterns and identify similar sets of patterns as most and least used. For example, the *Adapter* and *Singleton* design patterns each make up more than 20% of total design pattern instances in their dataset. *Composite* and *Observer*, on the other hand, make up less than 1% each.

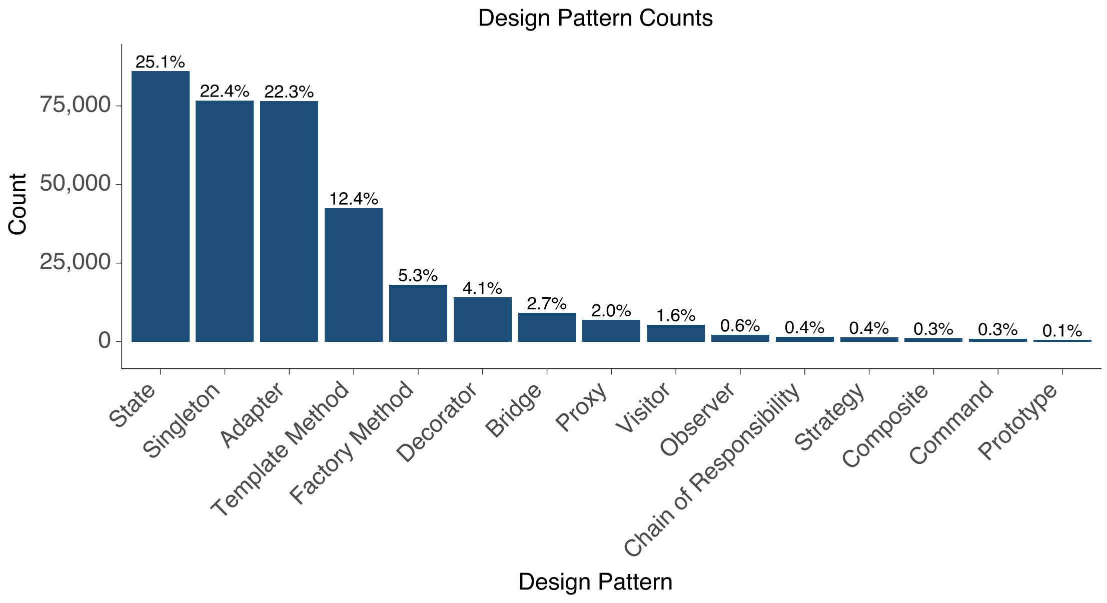

Figure 4.7: Overall number of detected design patterns in the created dataset.

A more fine-grained overview of pattern usage intensities is provided in Figure 4.8. As shown in this figure, most design patterns have usage intensities between 0 and 0.25, which means that 0-25% of classes in the respective projects are participating in each of the given design patterns. The peaks of the usage intensity distributions are generally at the lowest bucket. This indicates that even when design patterns are used, they are often only used very sparingly. The only patterns that deviate from this trend are the *State* and *Adapter* design patterns. Both of these patterns are among the top three most-used patterns, and for both of them, the usage intensity peaks are at a noticeably higher value than for the remaining patterns. Furthermore, their peaks are significantly wider, thus showing more diversity of usage intensities than most other patterns do.

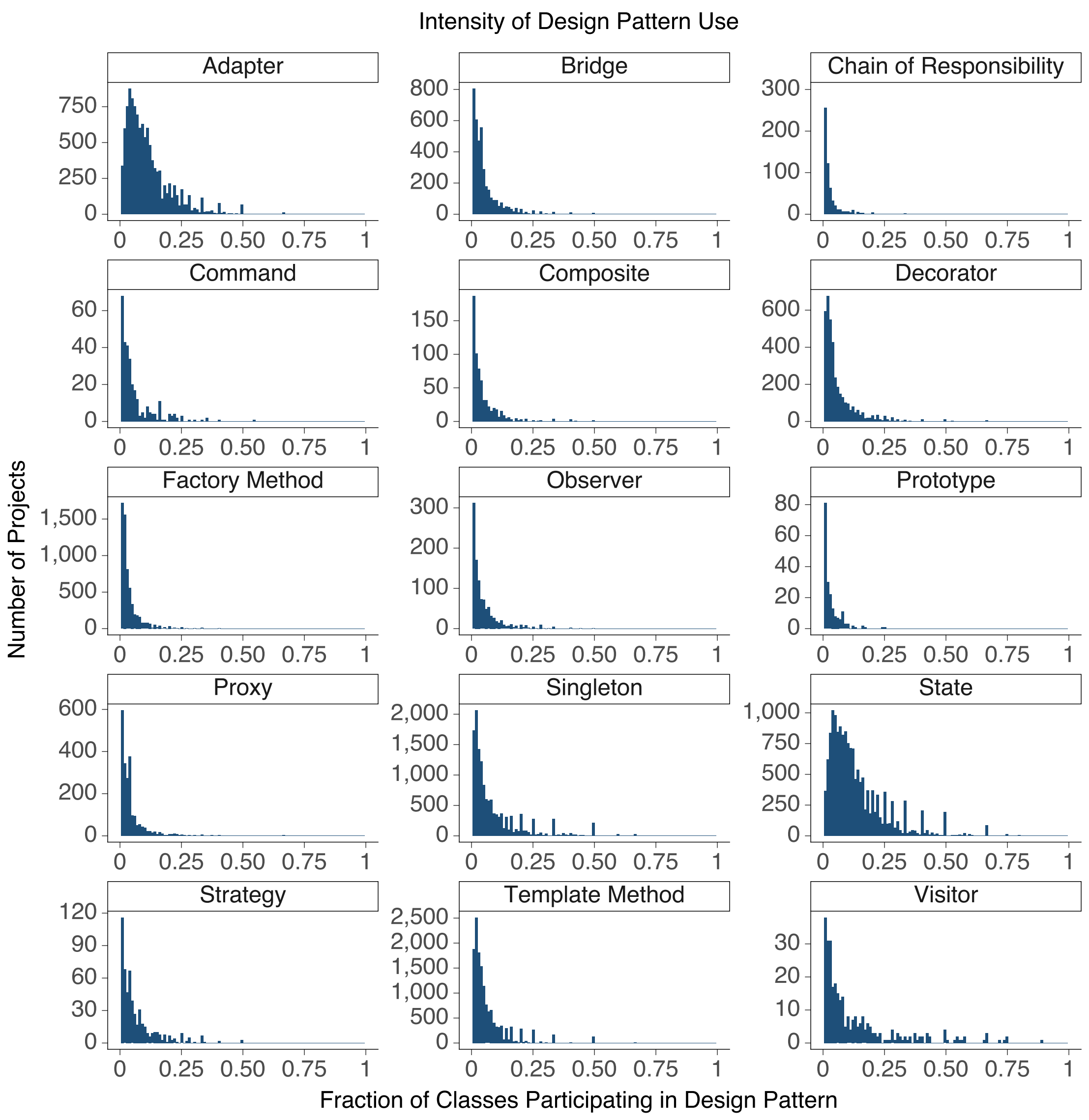

Figure 4.8: Usage intensities of the analyzed design patterns. Each histogram only shows projects that contain at least one instance of the corresponding design pattern.

Although the usage intensity distributions of many design patterns look very similar, the correlation coefficients between the usage intensities are rather low (see Figure 4.9). For example, even the highest observed correlation coefficient (*Adapter-State*) has a value of just 0.59. Furthermore, only three additional design pattern pairs have correlation coefficients above 0.4. These are the correlation coefficients between the design pattern pairs *Adapter-Factory Method* (0.46), *Adapter-Bridge* (0.42), and *Factory Method-State* (0.41). The number of design pattern pairs with even lower correlation coefficients is considerably higher. 15 pairs have correlation coefficients between 0.4 and 0.3, 13 pairs between 0.3 and 0.2, 55 pairs between 0.2 and 0.1, and 33 pairs below 0.1.

The low correlation coefficients between design pattern usage intensities might be an indicator that, in many cases, encountered design problems can be solved through the use of a single design pattern. After all, if combinations of multiple design patterns were commonly needed, correlations between design patterns should be higher. Given that different design patterns are intended to solve different design problems, this result seems to be in line with expected usage patterns of design patterns.

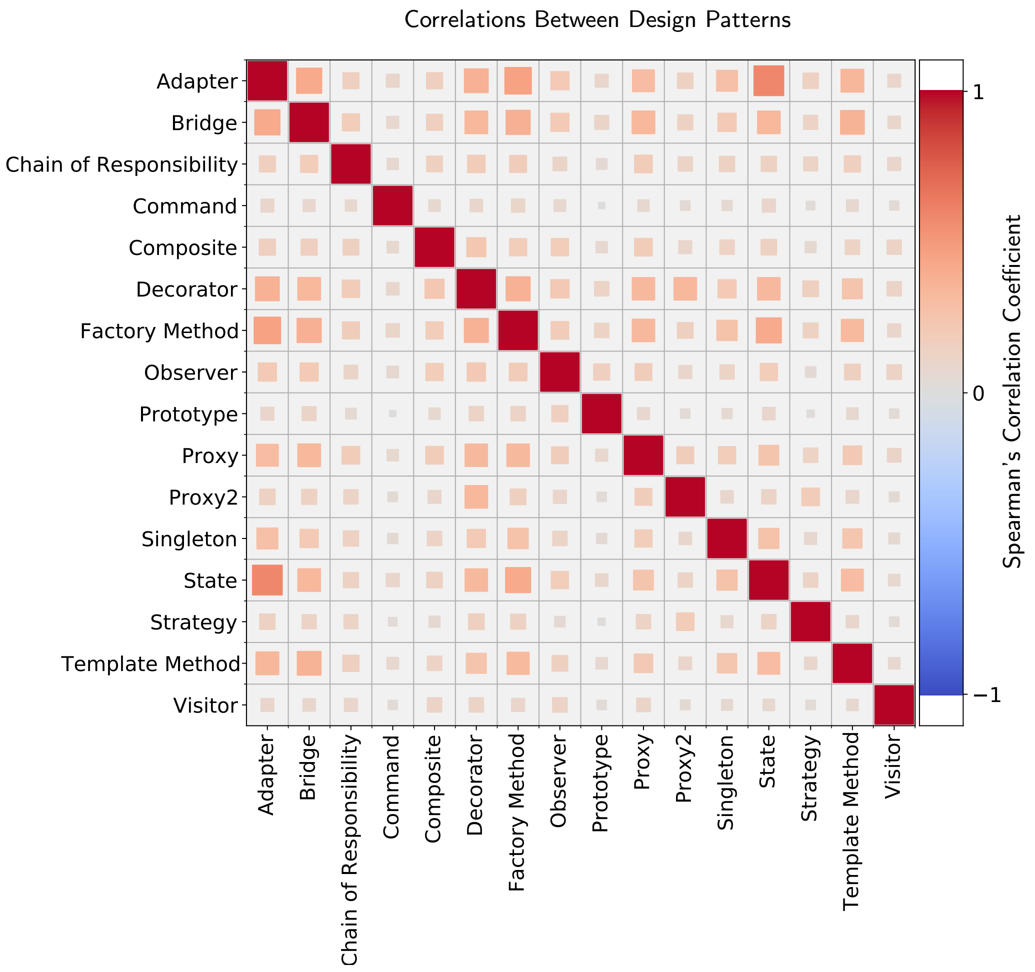

Figure 4.9: Correlations between the usage intensities of the analyzed design patterns.

4.4 Quality Analysis

To analyze the quality effects of design patterns, correlations between design pattern usage intensities and software metrics are calculated. The results of these calculations are visualized and discussed throughout the remainder of this section. For visualization purposes, heatmaps of the calculated correlation coefficients are presented. In the discussions that accompany these visualizations, the achieved results are summarized and noteworthy observations are highlighted.

Figure 4.10 shows the correlations between design pattern usage intensities and the six QMOOD quality attributes. Upon inspection of this figure, two distinct subsets of quality attributes are revealed: *effectiveness*, *extendibility*, and *flexibility* on the one hand, and *functionality*, *reusability*, and *understandability* on the other hand.

For the first subset (*effectiveness*, *extendibility*, and *flexibility*), the observed correlation coefficient magnitudes are very small for all of the analyzed design patterns. More specifically, correlations for these three quality attributes range from -0.04 (*State-extendibility*) to 0.19 (*Singleton-effectiveness*), with several pairings having correlation magnitudes of 0.00 (*Adapter-extendibility*, *Composite-extendibility*, *Command-flexibility*, etc.).

For the second subset (*functionality*, *reusability*, *understandability*), the correlation coefficients are noticably higher on average. Of the 48 pairings, a total of 4 (8%) pairings have correlation coefficient magnitudes above 0.4, 11 (23%) between 0.4–0.3, and 9 (19%) between 0.3–0.2. This leaves 24 (50%) pairings with correlation magnitudes below 0.2. It is interesting to note that the highest correlation magnitudes are observed for the design patterns that are most commonly used (*State*, *Singleton*, *Adapter*, etc.).

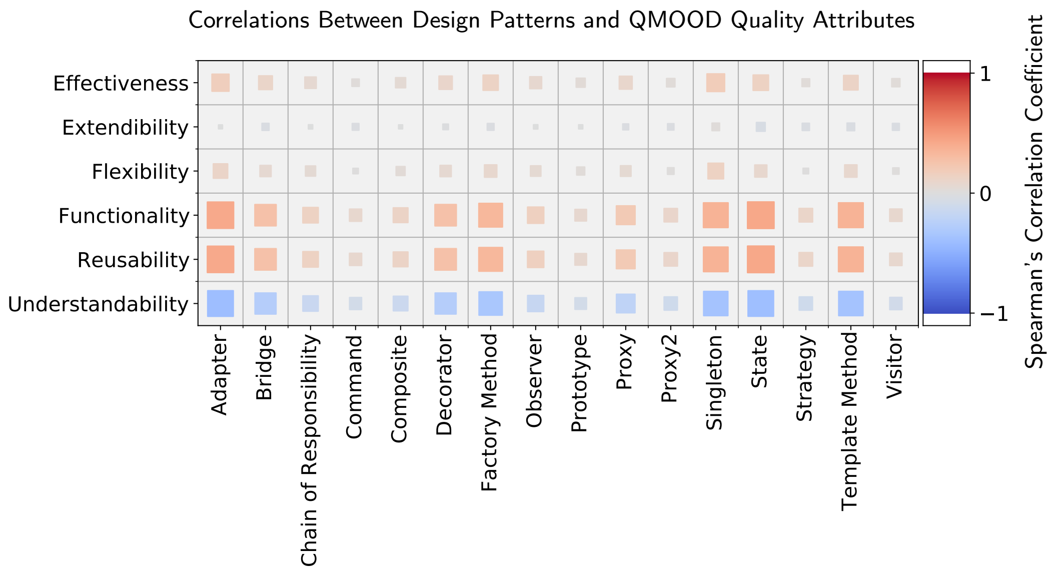

Figure 4.10: Correlations between the usage intensities of the analyzed design patterns and the measured QMOOD quality attribute values.

Since QMOOD quality attributes are calculated as weighted sums of QMOOD design properties, a lower-level understanding of the factors that influence the observed correlations in Figure 4.10 can be achieved by looking at the correlations between QMOOD design properties and design pattern usage intensities. These correlations are shown in Figure 4.11 and discussed in the following paragraphs.

Design properties with low correlations for all design patterns include ANA, CAM, CIS, DAM, DCC, NOM, and NOP. None of these properties have any correlation coefficient magnitudes that are higher than 0.2. Meanwhile, the design properties DSC, MFA, MOA, and NOH all have one or more correlation coefficients above 0.2, with a total of 4 (2x DSC, 2x NOH) correlations being higher than 0.4, and 9 (3x DSC, 3x MOA, 3x NOH) additional correlations being higher than 0.3.

The strongest correlations are observed for the design patterns that are most commonly used (*State*, *Singleton*, *Adapter*, etc.). Furthermore, the design properties with the highest correlations (DSC, NOH) are related to project size, whereas class size (CIS, NOM, NOP) and average depth of inheritance hierarchies (ANA) show very weak correlations.

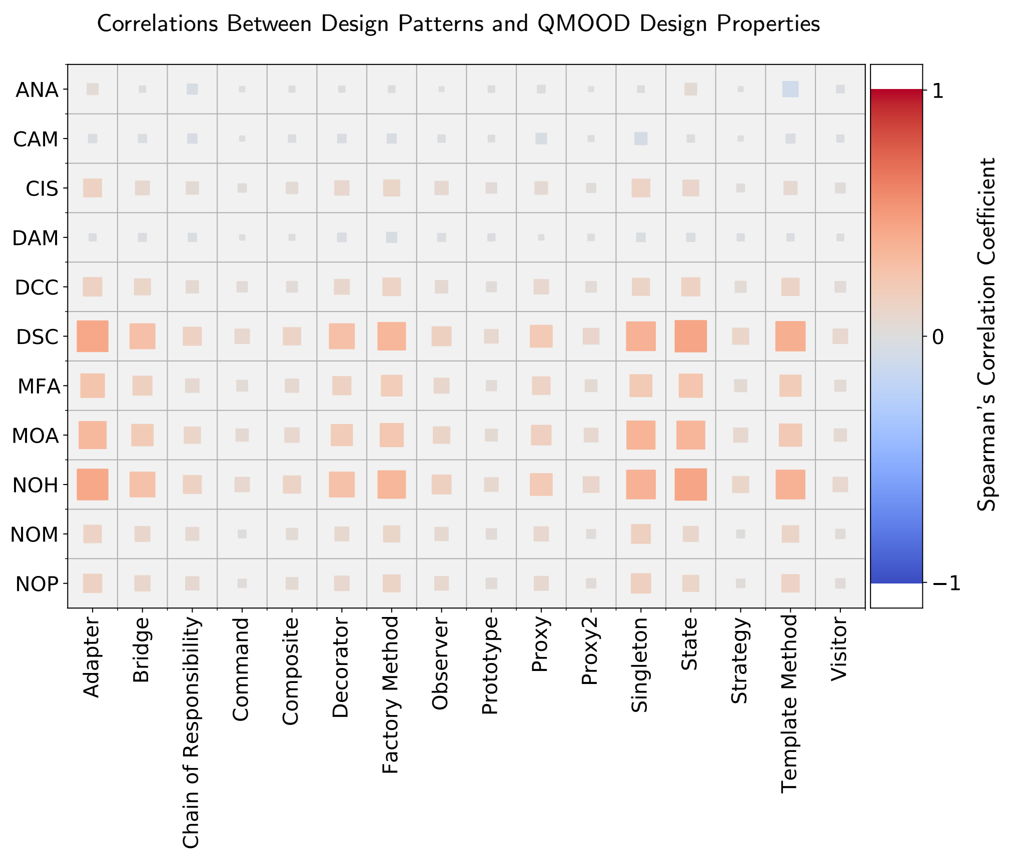

Figure 4.11: Correlations between the usage intensities of the analyzed design patterns and the measured QMOOD design property values.

The two previous parts of the quality analysis focus on metrics from the QMOOD metrics suite, i.e., QMOOD quality attributes and QMOOD design properties. To further diversify results, thus adding another perspective to the investigation, the final part of the analysis covers correlations between usage intensities of design patterns and C&K+ metrics.

As shown in Figure 4.12, the highest correlation coefficient magnitudes are yet again observed for the design patterns that are most commonly used. For example, *State*, *Singleton*, and *Adapter*, which have the highest pattern counts in the dataset, all have multiple correlation coefficients that are larger than 0.3. Less commonly occurring patterns, such as *Visitor*, *Observer*, and *Strategy*, exhibit noticeably weaker correlations, reaching correlation coefficients of 0.15 at best.

On the metrics side, the three metrics with the strongest correlations are CA (afferent coupling), LOC (lines of code), and NOC (number of children). All three of these metrics reach correlations above 0.3 for multiple analyzed design pattern usage intensities.

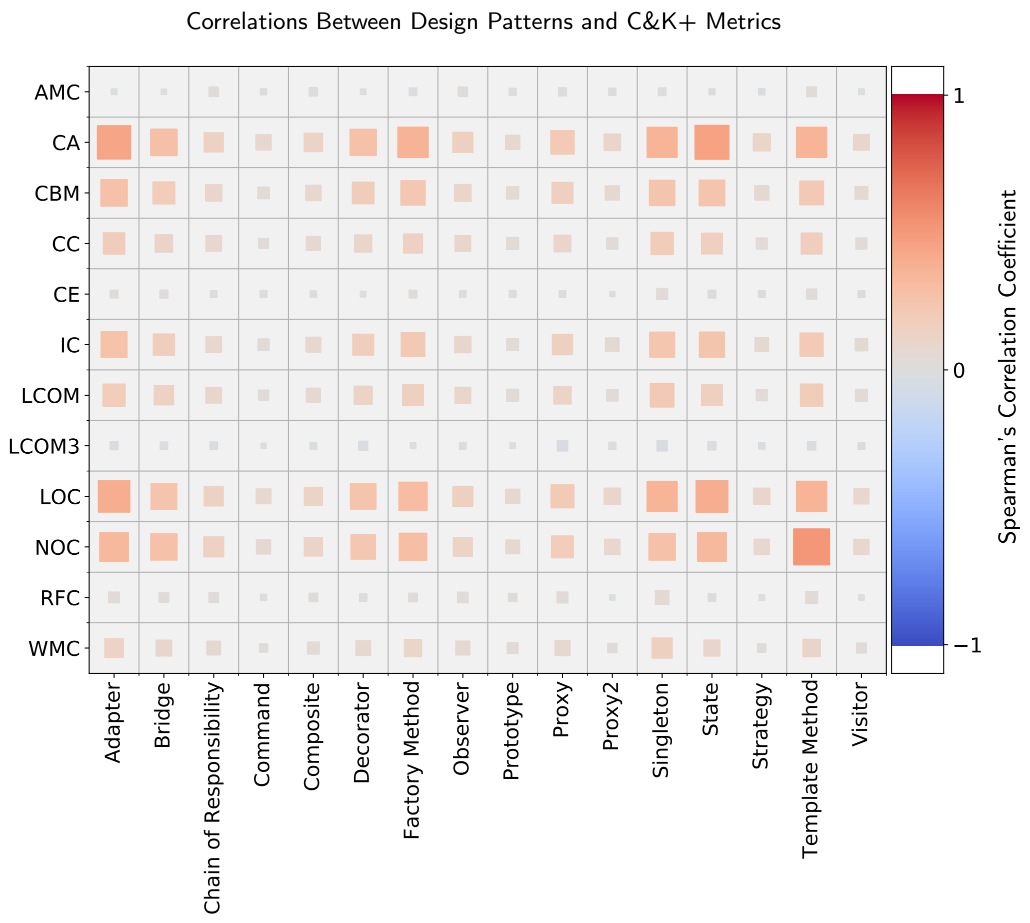

Figure 4.12: Correlations between the usage intensities of the analyzed design patterns and the measured C&K+ metric values.

4.5 Answering the Research Questions

Through the use of the quality analysis results described in the previous section, the research questions posed in Section 1.3 can be answered. Answers are first provided for the six sub-questions **RQ1.1–RQ1.6**, which cover correlations between design patterns and *individual* QMOOD quality attributes. These answers are then taken into account when answering the main research question **RQ1** - which covers correlations between design patterns and *all* QMOOD quality attributes - at the end of this section.

RQ1.1: Are there correlations between design pattern use and *effectiveness*?

According to results from the quality analysis, correlations between design pattern usage intensities and the QMOOD quality attribute *effectiveness* are very weak. More specifically, correlation coefficients range from 0.02 on the low end up to 0.19 on the high end.

None of the C&K+ metrics can be reasonably mapped to any QMOOD design properties that influence *effectiveness*. Thus, the C&K+ results don't provide any further evidence regrading correlations between design pattern usage intensities and *effectiveness*.

RQ1.2: Are there correlations between design pattern use and *extendibility*?

For the QMOOD quality attribute *extendibility*, all correlation cofficients have magnitudes below 0.04. The resulting average correlation coefficient magnitude of 0.02 makes *extendibility* the quality attribute with the weakest observed correlations in this thesis.

Looking at the C&K+ results, several metrics related to *extendibility* show comparatively strong correlations. Among the *coupling* metrics, CA has correlations up to 0.46, and IC up to 0.27. Similarly, the *inheritance* metric *NOC* has correlations up to 0.52. However, since *coupling* negatively influences *extendibility*, whereas *inheritance* positively influences it, the overall effect might be neutral given appropriate weighting factors.

RQ1.3: Are there correlations between design pattern use and *flexibility*?

As far as the results from the QMOOD quality attribute analysis are concerned, correlations for the quality attribute *flexibility* are very weak. The strongest correlations have magnitudes of only 0.14, and the weakest correlations go as low as 0.01, resulting in an average correlation coefficient magnitude of 0.06.

The C&K+ results, in contrast, suggest a moderately strong negative correlation between design patterns and *flexibility*. This is because QMOOD defines *flexibility* to be negatively influenced by *coupling*, and the two *coupling* metrics CA and IC reach correlation coefficients of up to 0.46 and 0.27, respectively. However, none of the other QMOOD design properties that influence *flexibility* are represented by the C&K+ metrics. Thus, the observed correlations might not hold if a more diverse set of metrics were used.

RQ1.4: Are there correlations between design pattern use and *functionality*?

Correlations between design pattern usage intensities and *functionality* are weak to moderate, reaching values between 0.07–0.43, and an average correlation coefficient magnitude of 0.22. Higher correlations are observed for more frequently occurring design patterns, whereas less frequently occurring design patterns show weaker correlations.

The results from the C&K+ analysis support these findings, showing moderately high correlation coefficients of up to 0.40 for the *design size* metric LOC. This matches the QMOOD design property results, which show correlations of up to 0.45 for the *design size* metric DSC. Correlation coefficients of up to 0.22 for the C&K+ *cohesion* metric LCOM further support the positive correlation with *functionality*.

RQ1.5: Are there correlations between design pattern use and *reusability*?

Correlation coefficients between design pattern usage intensities and *reusability* are between 0.08–0.43. For less commonly occurring design patterns, correlations are on the lower end of this range. Correspondingly, for more commonly occurring design patterns, correlations are on the higher end of this range.

Among the C&K+ metrics, two of seven metrics related to *reusability* show moderately strong correlations. These metrics are LOC (*design size*) with correlations between 0.07–0.40, and CA (*coupling*) with 0.07–0.46. While *design size* positively influences *reusability*, *coupling* negatively influences it. Thus, the C&K+ metrics do not provide conclusive evidence about whether the overall correlation is positive, neutral, or negative.

RQ1.6: Are there correlations between design pattern use and *understandability*?

For the QMOOD quality attribute *understandability*, the quality analysis shows moderately high negative correlations for the more commonly used design patterns, and noticeably weaker negative correlations for the less commonly used design patterns. The overall range of observed correlation coefficients is between -0.07 and -0.40.

The correlations observed during the C&K+ analysis confirm the moderately high negative correlations regarding *understandability*. LOC and CA both have multiple correlations that are stronger than 0.35, and both of the corresponding QMOOD design properties *design size* and *coupling* negatively influence *understandability* according to the definitions made by QMOOD.

RQ1: Are there correlations between design pattern use and QMOOD quality attributes?

The results of the conducted analysis show that there are weak to moderate correlations between design pattern usage intensities and QMOOD quality attributes. The strongest correlations are observed between more commonly occurring design patterns and the three quality attributes *flexibility*, *reusability*, and *understandability*. For less commonly occurring design patterns and the three quality attributes *effectiveness*, *extendibility*, and *flexibility*, the observed correlations are significantly weaker.

Chapter 5

Discussion

5.1 Interpretation of the Results

According to the results of the quality analysis, correlations between design patterns and quality attributes are strongest for the following three quality attributes:

- **functionality**, which measures how many responsibilities are fulfilled by a given set of classes and made available through the public interfaces of these classes,
- **reusability**, which measures whether it is possible to apply a given set of classes to a new problem without having to change the classes in a significant way,
- **understandability**, which measures how easy or difficult it is to fully comprehend a given set of classes and the interactions between them.

While the magnitudes of the correlations vary across design patterns, all design patterns have in common that both functionality and reusability show positive correlations, whereas understandability shows negative correlations.

If these correlations were interpreted as predictors of causal relationships between design patterns and software quality, this would mean that the use of design patterns improves functionality and reusability, but does so at the cost of understandability. Thus, this result would indicate that the use of design patterns entails a balancing act that has to weigh the importance of the positively vs. negatively affected quality attributes against each other, which matches the following advice issued by the *Gang of Four* [30]:

> "Often [design patterns] achieve flexibility [. . .] by introducing additional levels of indirection, and that can complicate a design [. . .]. A design pattern should only be applied when the flexibility it affords is actually needed."

However, a causal relationship between design patterns and the discussed software quality attributes is only one of multiple interpretations that could reasonably explain the observed correlations. This, perhaps, becomes even clearer when tracing the correlations back to the metric level. On this level, the following QMOOD and C&K+ metrics are among the most strongly correlated with design pattern usage intensities:

- DSC, which counts the number of classes in a project,
- NOH, which counts the number of classes without children in a project,
- LOC, which counts the number of lines of code in a project.

All three of these metrics are related to project size. Thus, design pattern usage intensities are apparently higher in larger projects than in smaller ones. Combined with the fact that DSC is a major influencing factor towards all three of the previously mentioned quality attributes, this observation gives rise to the following hypotheses that could explain the correlations between design pattern usage intensities and the three quality attributes *fuctionality*, *reusability*, and *understandability*.

Hypothesis 1: Design patterns cause projects to grow because they use more classes and a larger number of lines of code than alternative solutions. As a consequence, the overall size of projects with high design pattern usage intensities is comparatively large, which explains the observed correlations. Intuitively speaking, such an effect of design pattern use on project size seems plausible. After all, even the aforementioned advice by the *Gang of Four* points out that design patterns tend to introduce additional levels of indirection, which can be seen as synonymous with requiring a larger number of classes and lines of code to implement.

Hypothesis 2: Design patterns don't cause projects to grow, they just happen to be used more commonly in larger projects. This might be because design patterns provide a shared vocabulary for discussions on the design level, and thereby facilitate easier communication, which is more important in larger projects due to the potentially larger number of participating developers. Alternatively, the design problems that design patterns solve might simply be more common in larger projects. This would cause design pattern usage intensities to naturally increase as projects grow larger and start to encounter these problems more commonly.

Hypothesis 3: Larger projects don't really have higher design pattern usage intensities, but the used design pattern detection tool reports a disproportionately high number of false positives for them. This might be the case because in larger projects, more class combinations exist that look like specific design patterns. The design pattern detection tool then incorrectly classifies these classes as design patterns, even though the classes might have extensions or modifications that make them distinctly different from the canonical implementations of the design patterns, but not different enough to be considered as such by the design pattern detection tool.

Of course, any combination of these three hypotheses could also be true. For example, a combination of Hypothesis 1 and Hypothesis 2 would mean that design patterns do cause projects to grow, but projects are still more likely to encounter the design problems that design patterns solve as the projects themselves grow in size. Furthermore, there might be additional explanations for the observed correlations that are not covered by the three discussed hypotheses. In any case, further investigations are needed to conclusively prove or disprove a causal relationship between design patterns and the investigated software quality attributes.

5.2 Comparison to Existing Research

In their systematic mapping study, Wedyan and Abufakher [95] identify a total of 27 case studies that have investigated the quality effects of design patterns. Of these case studies, the ones most similar to this thesis are the studies conducted by Ampatzoglou et al. [8] in 2011, Sfetsos et al. [78] in 2014, and Hussain et al. [35] in 2017. All three of these studies use the same design pattern detection tool that is used in this thesis. Furthermore, all three studies also use QMOOD quality attributes as proxies for software quality, thus making their results highly comparable to the results of this thesis.

Even though the studies by Ampatzoglou et al., Sfetsos et al., and Hussain et al. analyze the quality effects of design patterns on a comparatively small scale, covering only 100, 26, and 51 projects, their results generally agree with those observed in this thesis. For example, the results of the three studies also show at least moderately strong correlations between design patterns and *functionality*, *reusability*, and *understandability*. Additionally, all three studies also find that correlations are stronger for more commonly used design patterns than less commonly used ones. However, Hussain et al. also observe moderately strong correlations between design patterns and *effectiveness*, *extendibility*, and *flexibility*, which is a finding that is neither supported by this thesis nor by any of the other two discussed case studies.

The quality attribute most commonly investigated by the remaining 24 case studies listed by Wedyan and Abufakher [95] is *maintainability*. Although *maintainability* is not directly measured by this thesis, *maintainability* effects of design patterns can still be estimated based on the metrics that this thesis collects. Of the collected metrics, the ones most commonly associated with *maintainability* are *LOC*, *WMC*, and *LCOM* [10]. For all three of these metrics, higher values indicate that a project is less maintainable. Thus, the maintainability effect suggested by this thesis is a negative one, because all three of the listed metrics are positively correlated with design pattern usage intensities. Among existing case studies, results regarding the effects of design patterns on *maintainability* are mixed. While some studies find a positive effect, others find a negative effect, with no clear trend towards any of the two options [95].

Looking beyond case studies into controlled experiments, the most reliable results about the quality effects of design patterns come from a multi-site joint replication study conducted by Krein et al. [48]. In this study, a controlled experiment by Prechelt et al. [66] is independently replicated four times at different universities [44, 47, 60, 65]. The results of these studies find that equivalent maintenance tasks take longer to complete in projects with design patterns than in projects without design patterns. Thus, *maintainability* appears to decrease if design patterns are involved in a project. This is consistent with the results of this thesis, which also find a negative correlation between *maintainability* and the intensity of design pattern use. It should be noted, however, that Krein et al. report that the negative effect on task completion times is reduced or even reversed for more experienced developers and developers more familiar with design patterns.

5.3 Reproducibility

According to Robles [72], there are three primary criteria that have to be fulfilled in order to ensure that an MSR case study can be replicated. These criteria are:

1. the raw data has to be publicly available,
2. the preprocessed data has to be publicly available,
3. the used tools and scripts have to be publicly available.

The following paragraphs explain how these criteria were taken into consideration when conducting the case study described by this thesis.

Ad 1.: All projects analyzed in this thesis are open-source projects that are publicly available through the Maven Central [86] repository. The specific project versions used in the analysis can be determined through the created dataset, which is publicly available on Zenodo[1] and contains the `groupId`, `artifactId`, and `version` of every included project. Thus, as long as the analyzed projects remain available through Maven Central and the created dataset remains available through Zenodo, the used projects can be uniquely identified and downloaded from Maven Central again.

Ad 2.: The preprocessed data created as part of this thesis is publicly available on Zenodo. It consists of DML dumps of project metadata, metric data, and design pattern data, as well as DDL dumps of the database schema including (materialized) views. All dumps are provided in a standard SQL format that can be imported by any SQL-compliant database, thus ensuring broad compatibility with a wide range of databases. Furthermore, the (meta-)data dumps are additionally provided in two PostgreSQL-specific formats, which enables faster import speeds if a PostgreSQL database is used.

Ad 3.: All of the tools that are used by this thesis are open-source tools that are distributed with the *Qualisign*[2] project. The *Qualisign* project itself, which performs the data collection steps of this thesis, is available as a publicly accessible project on GitHub. Similarly, the *Qualisign Analysis*[3] project, which performs the data analysis steps of this thesis, is also available as a publicly accessible project on GitHub. Dependencies of the two created projects include Scala 2.13, Python 3.8, PostgreSQL 12.2, and Docker. All of these are publicly available through the websites of their corresponding vendors.

In summary, all of the raw and preprocessed data, as well as all of the tools and scripts used in this thesis, are publicly available at the time of writing. Furthermore, all of the hosting platforms which are used (Maven Central, GitHub, and Zenodo) are explicitly intended for long-term storage of the corresponding data, tools, and scripts. Consequently, replications of the conducted case study are expected to be possible without restrictions for the foreseeable future.

[1]`https://zenodo.org/record/3731872`
[2]`https://github.com/jaichberg/qualisign`
[3]`https://github.com/jaichberg/qualisign-analysis`

5.4 Threats to Validity

In their book *Case Study Research in Software Engineering*, Runeson et al. [73] describe validity as the trustworthiness of the results achieved by a research study. Correspondingly, threats to validity are factors that call into question how trustworthy the results of a study are. The scheme that is used in this section to assess the validity of this thesis is the one described in the aforementioned book by Runeson et al. According to this scheme, validity consists of the following four aspects:

1. **construct validity** assesses whether the measurements that are conducted by a study are appropriate to answer the study's stated research questions,
2. **internal validity** assesses whether causal relationships that are examined in a study might be influenced by other factors than those that were investigated,
3. **external validity** assesses whether the results that are achieved by a study can be generalized to other cases than those that were analyzed,
4. **reliability** assesses whether exact replications of a study are possible and whether they are expected to deliver the same results as the original study.

The following subsections discuss how well these four aspects of validity are fulfilled by this thesis. Noteworthy threats to validity are highlighted and evaluated regarding their consequences for the achieved results.

5.4.1 Construct Validity

The basic idea of this thesis is to use software metrics as proxy measures for software quality to then be able to analyze relationships between design patterns and the chosen software quality proxies. As described in Chapter 2, this general idea is firmly established in design pattern research, and the use of software metrics as proxies for software quality is also commonly seen in other sub-fields of software engineering research.

A threat to construct validity that is not fully resolved is the choice of employed software metrics. Even though two popular metrics suites (QMOOD [12] and C&K [20]) are employed, use of these metrics as assessors of software quality sometimes leads to conflicting results, as discussed when answering the research questions in Section 4.5. Thus, the question arises whether any of the used metrics are flawed, and whether different metrics could provide more trustworthy results.

5.4.2 Internal Validity

Since the conducted analysis only investigates correlations, no causal relationships between design patterns and different software quality attributes can be claimed based on the results of this thesis. Nevertheless, even though no proof for a causal relationship is provided, the observed correlations at least serve as additional data points towards a better understanding of the software quality effects of design patterns.

To prove with a reasonable degree of certainty that a causal relationship between design patterns and software quality exists, one would have to (i) take a sufficiently large number of existing projects, (ii) perform refactoring operations on these projects to deliberately introduce or remove design patterns, and (iii) analyze the projects before and after each refactoring regarding changes in software quality. While such an approach might be theoretically possible, the described process appears to be difficult to perform on a large enough scale to achieve statistically significant results.

5.4.3 External Validity

One of the main motivating factors for this thesis is that the generalizability of existing studies is limited by the small number of projects investigated in these studies. This threat to external validity is improved upon in this thesis through the use of a significantly larger number of projects that are included in the analysis.

The most significant threat to external validity that remains in this thesis is that only Java projects are included in the analysis. Thus, the results achieved by this thesis cannot be generalized to programming languages other than Java, even though design patterns themselves do not exhibit this same restriction.

Another threat to external validity is that only open-source projects are analyzed. Since source code architecture and development processes in closed-source projects might be different than in open-source projects, the results of the conducted analysis might, therefore, also be different if closed-source projects were used.

A third threat to external validity is that only projects from the Maven Central repository are used. Many of these projects are libraries or frameworks that are specifically tailored to be used by other projects. As a consequence, the results of this thesis might not generalize well to non-library projects that do not need to be usable by other projects and, therefore, might have distinctly different architectures.

5.4.4 Reliability

As discussed in the reproducibility evaluation in Section 5.3, all of the data and tools that are used in this thesis are publicly available through Maven Central, GitHub, and Zenodo. Thus, exact replications of the conducted case study are possible.

However, if replications of the conducted case study were performed, the achieved results might slightly differ from the results presented in this thesis. This is because of random data retrieval and processing failures that occurred during data collection (see Section 3.4.3, *Data Collection Issues*). Despite these failures, the overall trends observed in replications of this study should be the same as those in the original study, since random failures only occurred for a small subset of projects.

Chapter 6

Conclusion

6.1 Summary

Motivated by the small scale of existing case studies, which only analyze several hundred projects at most [95], the primary goal of this thesis is to conduct a large-scale analysis of the quality effects of design patterns. To accomplish this goal, software metric and design pattern data for around 90,000 Java projects downloaded from the Maven Central Repository [86] are collected using the metrics calculation tool *CKJM extended* [43] and the design pattern detection tool *SSA* [88]. Based on the collected data, correlations between software metrics and design pattern usage intensities are calculated. Following the methodology described by the QMOOD quality model [12], the collected metrics are used as proxies for various software quality attributes. Thus, the calculated correlations provide insights into potential quality effects of design patterns.

The quality attributes analyzed by this thesis consist of the six quality attributes defined by the QMOOD quality model. According to the results of the analysis, *effectiveness*, *extendibility*, and *flexibility* are only weakly correlated with design pattern usage intensities, showing correlation coefficient magnitudes of less than 0.2 for all 15 analyzed design patterns. Correlations are noticeably stronger for *functionality*, *reusability*, and *understandability*, reaching correlation coefficient magnitudes of around 0.4 for several design patterns each, with correlations being stronger for more commonly used design patterns (*State*, *Singleton*, *Adapter*, etc.), and weaker for less commonly used ones (*Prototype*, *Command*, *Composite*, etc.). Whereas the calculated correlations suggest a positive relationship between design patterns and *functionality* as well as *reusability*, they suggest a negative relationship between design patterns and *understandability*.

Overall, the quality trends observed in this thesis are largely consistent with the results of existing studies covering the quality effects of design patterns. As in this thesis, existing case studies also find positive correlations between design patterns and the quality attributes *functionality* and *reusability*, as well as negative correlations between design patterns and *understandability*. Furthermore, the collected metrics suggest a negative correlation between design patterns and *maintainability*, which matches the results observed in existing controlled experiments.

6.2 Contribution to Design Pattern Research

This thesis makes two primary contributions to the current state of design pattern research. The first of these contributions is the created dataset, whereas the second contribution is the conducted data analysis. Both of these contributions are further discussed in the following paragraphs.

The dataset created as part of this thesis consists of design pattern and software metric data for around 90,000 Java projects from the Maven Central repository. Both the dataset itself as well as the code to recreate the dataset are freely available on Zenodo and GitHub. This provides researchers with the necessary tools to either conduct further research on the existing dataset or to modify the provided source code to to extend its capabilities. Thus, the dataset and source code enable design pattern researchers to more easily conduct additional large-scale analyses about the quality effects of design patterns on the projects that are available through the Maven Central repository.

The data analysis conducted on the created dataset serves as a proof of concept that large-scale analyses of design patterns are feasible. This, in turn, might encourage other researchers to also extend their analyses of the quality effects of design patterns to a larger scale. Thus, this thesis serves as a motivating factor that will hopefully lead to more generalizable results in design pattern research in the future. Furthermore, the results of the data analysis contribute additional evidence that suggests that correlations between design patterns and quality attributes such as *functionality*, *reusability*, and *understandability* exist.

6.3 Contribution to Software Engineering Practice

Even though the conducted case study can not conclusively prove that design patterns have an effect on software quality, the found correlations suggest that both positive effects on *functionality* and *reusability*, as well as negative effects on *understandability* might exist. This result is consistent with existing advice from the *Gang of Four*, which states that the introduction of design patterns makes a design more difficult to understand and, therefore, design patterns should only be used if they are really needed [30]. As a consequence, the findings of this thesis should motivate software engineers to more deliberately debate the advantages and disadvantages that design patterns might have before introducing them into a project.

The aforementioned advice is especially important to consider in projects with multiple participating developers. This is because existing controlled experiments have found that maintenance tasks take longer to complete in projects that use design patterns than in projects that don't use design patterns if the developers that perform the maintenance tasks are either inexperienced or not very familiar with design patterns [48]. In this context, it should be noted that many developers are probably only familiar with a small subset of commonly used design patterns (see Section 4.3 for design pattern statistics). Therefore, if less commonly used design patterns are added to a project, special care should be taken to introduce them to affected team members.

6.4 Further Research

To improve upon the results achieved by this thesis, the following additional analyses could be performed by future studies:

1. analyze multiple versions of each project,
2. analyze additional software quality attributes,
3. analyze additional large-scale repositories,
4. analyze the quality effects of design patterns on the class level.

Ad 1.: The case study conducted as part of this thesis only analyzes a single version of each of the projects covered by the analysis. Because of this, the found correlations only provide comparatively weak evidence for a causal relationship between design patterns and quality attributes. After all, projects with design patterns might have exhibited the same quality characteristics both before and after design patterns were introduced to them. Therefore, future studies could analyze multiple versions of each project to be able to more directly attribute quality increases or decreases to the introduction or removal of design patterns.

Ad 2.: The software quality analysis conducted in this thesis only covers a rather limited set of software quality attributes. The primary reason for this is that only a small number of tools could be identified that are appropriate for large-scale software quality analysis (see Sections 3.2.2 and 3.3.2). If additional tools appropriate for large-scale software quality analysis were developed, further quality attributes could be analyzed. This, in turn, would enable future studies to provide a more complete view of the quality effects of design patterns.

Ad 3.: In this thesis, around 90,000 projects from the Maven Central repository are included in the analysis. Compared to existing studies, this is a two orders of magnitude increase in the number of analyzed projects. Future studies could achieve further notable increases in case study size by including projects from different large-scale repositories in their analysis. For example, the Boa project [27] might enable analysis of around 8,000,000 projects from GitHub if compatible tools for design pattern detection and software quality analysis are developed for it. Through this, further improvements in generalizability could be achieved.

Ad 4.: This thesis only analyzes the quality effects of design patterns on the project level. To achieve more localized insights into the quality effects that design patterns have on the source code around them, future analyses could investigate design pattern's quality effects on the class level. Through this, it might be possible to more accurately judge whether a particular quality attribute is directly influenced by a design pattern, especially if relationships between pattern and non-pattern classes are investigated as potential moderating variables of class quality.

Bibliography

[1] Allix, K., Bissyande, T. F., Klein, J., and Traon, Y. L. “AndroZoo: Collecting Millions of Android Apps for the Research Community”. *Proceedings of the 13th International Conference on Mining Software Repositories*. 2016. DOI: 10.1145/2901739.2903508.

[2] Alnusair, A., Zhao, T., and Yan, G. “Rule-Based Detection of Design Patterns in Program Code”. *International Journal on Software Tools for Technology Transfer* 16.3, 2014. DOI: 10.1007/s10009-013-0292-z.

[3] Amann, S., Beyer, S., Kevic, K., and Gall, H. “Software Mining Studies: Goals, Approaches, Artifacts, and Replicability”. *Lecture Notes in Computer Science* 8987, 2015.

[4] Ampatzoglou, A., Charalampidou, S., and Stamelos, I. “Research State of the Art on GoF Design Patterns: A Mapping Study”. *Journal of Systems and Software* 86.7, 2013. DOI: 10.1016/j.jss.2013.03.063.

[5] Ampatzoglou, A. and Chatzigeorgiou, A. “Evaluation of Object-Oriented Design Patterns in Game Development”. *Information and Software Technology* 49.5, 2007. DOI: 10.1016/j.infsof.2006.07.003.

[6] Ampatzoglou, A., Chatzigeorgiou, A., Charalampidou, S., and Avgeriou, P. “The Effect of GoF Design Patterns on Stability: A Case Study”. *IEEE Transactions on Software Engineering* 41.8, 2015. DOI: 10.1109/tse.2015.2414917.

[7] Ampatzoglou, A., Frantzeskou, G., and Stamelos, I. “A Methodology to Assess the Impact of Design Patterns on Software Quality”. *Information and Software Technology* 54.4, 2012. DOI: 10.1016/j.infsof.2011.10.006.

[8] Ampatzoglou, A., Kritikos, A., Kakarontzas, G., and Stamelos, I. “An Empirical Investigation on the Reusability of Design Patterns and Software Packages”. *Journal of Systems and Software* 84.12, 2011. DOI: 10.1016/j.jss.2011.06.047.

[9] Ampatzoglou, A., Michou, O., and Stamelos, I. “Building and Mining a Repository of Design Pattern Instances: Practical and Research Benefits”. *Entertainment Computing* 4.2, 2013. DOI: 10.1016/j.entcom.2012.10.002.

[10] Arvanitou, E.-M., Ampatzoglou, A., Chatzigeorgiou, A., Galster, M., and Avgeriou, P. “A Mapping Study on Design-time Quality Attributes and Metrics”. *Journal of Systems and Software* 127, 2017. DOI: 10.1016/j.jss.2017.01.026.

[11] Aversano, L., Canfora, G., Cerulo, L., Del Grosso, C., and Di Penta, M. “An Empirical Study on the Evolution of Design Patterns”. *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. 2007. DOI: 10.1145/1287624.1287680.

[12] Bansiya, J. and Davis, C. G. “A Hierarchical Model for Object-Oriented Design Quality Assessment”. *IEEE Transactions on Software Engineering* 28.1, 2002. DOI: 10.1109/32.979986.

[13] Bernardi, M. L., Cimitile, M., and Lucca, G. D. “Design Pattern Detection Using a DSL-driven Graph Matching Approach”. *Journal of Software: Evolution and Process* 26.12, 2014. DOI: 10.1002/smr.1674.

[14] Bieman, J. M., Jain, D., and Yang, H. J. “OO Design Patterns, Design Structure, and Program Changes: An Industrial Case Study”. *Proceedings of the IEEE International Conference on Software Maintenance*. IEEE, 2001. DOI: 10.1109/icsm.2001.972775.

[15] Binun, A. and Kniesel, G. “DPJF - Design Pattern Detection with High Accuracy”. *Proceedings of the 16th European Conference on Software Maintenance and Reengineering*. IEEE, 2012. DOI: 10.1109/csmr.2012.82.

[16] Boehm, B. W., Brown, J. R., and Lipow, M. “Quantitative Evaluation of Software Quality”. *Proceedings of the 2nd International Conference on Software Engineering*. IEEE, 1978.

[17] Bourque, P. and Fairley, R. E., eds. *Guide to the Software Engineering Body of Knowledge, Version 3.0*. IEEE Computer Society, 2014.

[18] Campwood Software LLC. *SourceMonitor*. URL: http://www.campwoodsw.com/sourcemonitor.html, visited on 05/27/2020.

[19] Chaturvedi, K. K., Sing, V. B., and Singh, P. “Tools in Mining Software Repositories”. *Proceedings of the 13th International Conference on Computational Science and Its Applications*. IEEE, 2013.

[20] Chidamber, S. R. and Kemerer, C. F. “A Metrics Suite for Object Oriented Design”. *IEEE Transactions on Software Engineering* 20.6, 1994. DOI: 10.1109/32.295895.

[21] Christensen, H. B. and Ron, H. “A Case Study of Horizontal Reuse in a Project-Driven Organisation”. *Proceedings of the 7th Asia-Pacific Software Engeering Conference*. 2000. DOI: 10.1109/apsec.2000.896711.

[22] Committee, M. S. *Mining Software Repositories*. July 16, 2020. URL: http://www.msrconf.org/.

[23] Deissenboeck, F., Juergens, E., Lochmann, K., and Wagner, S. “Software Quality Models: Purposes, Usage Scenarios and Requirements”. *2009 ICSE Workshop on Software Quality*. 2009. DOI: 10.1109/wosq.2009.5071551.

[24] Diamantopoulos, T., Noutsos, A., and Symeonidis, A. “DP-CORE: A Design Pattern Detection Tool for Code Reuse”. *Proceedings of the 6th International Symposium*

on Business Modeling and Software Design. SCITEPRESS - Science, 2016. DOI: 10.5220/0006223301600167.

[25] Dong, J., Zhao, Y., and Peng, T. “A Review of Design Pattern Mining Techniques”. *International Journal of Software Engineering and Knowledge Engineering* 19.6, 2009. DOI: 10.1142/s021819400900443x.

[26] Dromey, R. G. “Cornering the Chimera”. *IEEE Software* 13.1, 1996. DOI: 10.1109/52.476284.

[27] Dyer, R., Nguyen, H. A., Rajan, H., and Nguyen, T. N. “Boa: A Language and Infrastructure for Analyzing Ultra-large-scale Software Repositories”. *Proceedings of the 2013 International Conference on Software Engineering*. IEEE, 2013. DOI: 10.1109/icse.2013.6606588.

[28] Eclipse Foundation, Inc. *Eclipse Project*. URL: https://projects.eclipse.org/projects/eclipse, visited on 07/23/2020.

[29] Fowler, M. *Patterns of Enterprise Application Architecture*. Addison Wesley, 2002.

[30] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[31] Gatrell, M. and Counsell, S. “Design Patterns and Fault-proneness a Study of Commercial C# Software”. *Proceedings of the 5th International Conference on Research Challenges in Information Science*. IEEE, 2011. DOI: 10.1109/rcis.2011.6006827.

[32] GitHub, Inc. *GitHub*. URL: https://github.com/, visited on 02/20/2020.

[33] Gravino, C., Risi, M., Scanniello, G., and Tortora, G. “Do Professional Developers Benefit from Design Pattern Documentation? A Replication in the Context of Source Code Comprehension”. *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2012. DOI: 10.1007/978-3-642-33666-9_13.

[34] Hemmati, H., Nadi, S., Baysal, O., Kononenko, O., Wang, W., Holmes, R., and Godfrey, M. W. “The MSR Cookbook: Mining a Decade of Research”. *Proceedings of the 10th Working Conference on Mining Software Repositories*. 2013.

[35] Hussain, S., Keung, J., and Khan, A. A. “The Effect of Gang-of-Four Design Patterns Usage on Design Quality Attributes”. *IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2017. DOI: 10.1109/qrs.2017.37.

[36] Hussain, S., Keung, J., Khan, A. A., and Bennin, K. E. “Correlation between the Frequent Use of Gang-of-Four Design Patterns and Structural Complexity”. *Proceedings of the 24th Asia-Pacific Software Engineering Conference*. IEEE, 2017. DOI: 10.1109/apsec.2017.25.

[37] Huston, B. “The Effects of Design Pattern Application on Metric Scores”. *Journal of Systems and Software* 58.3, 2001. DOI: 10.1016/s0164-1212(01)00043-7.

[38] ISO/IEC 25010:2011. *Systems and Software Engineering — Systems and Software Quality Requirements and Evaluation (SQuaRE) — System and Software Quality Models*. Standard. International Organization for Standardization, 2011.

[39] ISO/IEC 9126-1:2001. *Software Engineering — Product Quality — Part 1: Quality Model*. Standard. International Organization for Standardization, 2001.

[40] ISO/IEC/IEEE 24765:2017. *Systems and Software Engineering — Vocabulary*. Standard. International Organization for Standardization, 2017.

[41] Jabangwe, R., Börstler, J., Smite, D., and Wohlin, C. "Empirical Evidence on the Link between Object-oriented Measures and External Quality Attributes: A Systematic Literature Review". *Empirical Software Engineering* 20.3, 2015. DOI: `10.1007/s10664-013-9291-7`.

[42] JSR 202. *Java Class File Specification Update*. Java Specification Request. Oracle Corporation, 2006.

[43] Jureczko, M. and Spinellis, D. "Using Object-Oriented Design Metrics to Predict Software Defects". *Proceedings of the 5th International Conference on Dependability of Computer Systems*. 2010.

[44] Juristo, N. and Vegas, S. "Design Patterns in Software Maintenance: An Experiment Replication at UPM". *Proceedings of the 2nd International Workshop on Replication in Empirical Software Engineering Research*. IEEE, 2011. DOI: `10.1109/reser.2011.8`.

[45] Kayarvizhy, N. "Systematic Review of Object Oriented Metric Tools". *International Journal of Computer Applications* 135.2, 2016. DOI: `10.5120/ijca2016908269`.

[46] Khomh, F. and Gueheneuce, Y.-G. "Do Design Patterns Impact Software Quality Positively?" *Proceedings of the 12th European Conference on Software Maintenance and Reengineering*. IEEE, 2008. DOI: `10.1109/csmr.2008.4493325`.

[47] Krein, J. L., Pratt, L. J., Swenson, A. B., MacLean, A. C., Knutson, C. D., and Eggett, D. L. "Design Patterns in Software Maintenance: An Experiment Replication at Brigham Young University". *Proceedings of the 2nd International Workshop on Replication in Empirical Software Engineering Research*. IEEE, 2011. DOI: `10.1109/reser.2011.10`.

[48] Krein, J. L., Prechelt, L., Juristo, N., Nanthaamornphong, A., Carver, J. C., Vegas, S., Knutson, C. D., Seppi, K. D., and Eggett, D. L. "A Multi-Site Joint Replication of a Design Patterns Experiment Using Moderator Variables to Generalize across Contexts". *IEEE Transactions on Software Engineering* 42.4, 2016. DOI: `10.1109/tse.2015.2488625`.

[49] Lobo, P., Bugayenko, Y., Ortega, A. A., Yildirim, M., Karazhenets, S., Aristy, G., Özen, M., Lamby, P., Puzankov, N., Stepanov, D., Pavel, G., and Nizar, M. *jPeek: A Static Collector of Java Code Metrics*. URL: `https://github.com/yegor256/jpeek`, visited on 05/27/2020.

[50] Martins, P., Achar, R., and Lopes, C. V. “50K-C: A Dataset of Compilable, and Compiled, Java Projects”. *Proceedings of the 15th International Conference on Mining Software Repositories*. 2018. DOI: 10.1145/3196398.3196450.

[51] Mayvan, B. B., Rasoolzadegan, A., and Ebrahimi, A. M. “A New Benchmark for Evaluating Pattern Mining Methods Based on the Automatic Generation of Testbeds”. *Information and Software Technology* 109, 2019. DOI: 10.1016/j.infsof.2019.01.007.

[52] Mayvan, B. B., Rasoolzadegan, A., and Yazdi, Z. G. “The State of the Art on Design Patterns: A Systematic Mapping of the Literature”. *Journal of Systems and Software* 125, 2017. DOI: 10.1016/j.jss.2016.11.030.

[53] McCabe, T. J. “A Complexity Measure”. *IEEE Transactions on Software Engineering* 2.4, 1976. DOI: 10.1109/tse.1976.233837.

[54] McCall, J. A., Richards, P. K., and Walters, G. F. *Factors in Software Quality: Concept and Definitions of Software Quality*. Tech. rep. General Electric, 1977.

[55] Microsoft. *Microsoft Academic*. July 26, 2020. URL: https://academic.microsoft.com/.

[56] Miguel, J. P., Mauricio, D., and Rodriguez, G. “A Review of Software Quality Models for the Evaluation of Software Products”. *International Journal of Software Engineering & Applications* 5.6, 2014. DOI: 10.5121/ijsea.2014.5603.

[57] Monperrus, M. *List of Tools for Java Software Metrics*. URL: https://www.monperrus.net/martin/java-metrics, visited on 02/20/2020.

[58] MvnRepository. *Maven Repository: Central*. URL: https://mvnrepository.com/repos/central, visited on 02/20/2020.

[59] Nakshatri, S., Hegde, M., and Thandra, S. “Analysis of Exception Handling Patterns in Java Projects: An Empirical Study”. *Proceedings of the 13th International Conference on Mining Software Repositories*. 2016. DOI: 10.1145/2901739.2903499.

[60] Nanthaamornphong, A. and Carver, J. C. “Design Patterns in Software Maintenance: An Experiment Replication at University of Alabama”. *Proceedings of the 2nd International Workshop on Replication in Empirical Software Engineering Research*. IEEE, 2011. DOI: 10.1109/reser.2011.11.

[61] Nuñez-Varela, A. S., Pérez-Gonzalez, H. G., Martı´nez-Perez, F. E., and Soubervielle-Montalvo, C. “Source Code Metrics: A Systematic Mapping Study”. *Journal of Systems and Software* 128, 2017. DOI: 10.1016/j.jss.2017.03.044.

[62] Al-Obeidallah, M. G., Petridis, M., and Kapetanakis, S. “A Survey on Design Pattern Detection Approaches”. *International Journal of Software Engineering* 7.3, 2016.

[63] Ouhbi, S., Idri, A., Fernandez-Aleman, J. L., and Toval, A. “Software Quality Requirements: A Systematic Mapping Study”. *Proceedings of the 20th Asia-Pacific Software Engineering Conference*. Vol. 1. 2013. DOI: 10.1109/apsec.2013.40.

[64] Palsberg, J. and Lopes, C. V. “NJR: A Normalized Java Resource”. *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*. 2018. DOI: `10.1145/3236454.3236501`.

[65] Prechelt, L. and Liesenberg, M. “Design Patterns in Software Maintenance: An Experiment Replication at Freie Universität Berlin”. *Proceedings of the 2nd International Workshop on Replication in Empirical Software Engineering Research*. IEEE, 2011. DOI: `10.1109/reser.2011.12`.

[66] Prechelt, L., Unger, B., Tichy, W. F., Brössler, P., and Votta, L. G. “A Controlled Experiment in Maintenance: Comparing Design Patterns to Simpler Solutions”. *IEEE Transactions on Software Engineering* 27.12, 2001. DOI: `10.1109/32.988711`.

[67] *Proceedings of the 15th International Conference on Mining Software Repositories*. Association for Computing Machinery, 2018.

[68] *Proceedings of the 16th International Conference on Mining Software Repositories*. IEEE, 2019.

[69] Raemaekers, S., van Deursen, A., and Visser, J. “The Maven Repository Dataset of Metrics, Changes, and Dependencies”. *Proceedings of the 10th International Conference on Mining Software Repositories*. IEEE, 2013. DOI: `10.1109/msr.2013.6624031`.

[70] Rasool, G. and Streitfdert, D. “A Survey on Design Pattern Recovery Techniques”. *International Journal of Computer Science Issues* 8.6, 2011.

[71] Richardson, C. *Microservices Patterns: With Examples in Java*. Manning Publications, 2018.

[72] Robles, G. “Replicating MSR: A Study of the Potential Replicability of Papers Published in the Mining Software Repositories Proceedings”. *Proceedings of the 7th Working Conference on Mining Software Repositories*. 2010. DOI: `10.1109/MSR.2010.5463348`.

[73] Runeson, P., Host, M., Rainer, A., and Regnell, B. *Case Study Research in Software Engineering: Guidelines and Examples*. John Wiley & Sons, Inc., 2012.

[74] Scanniello, G., Gravino, C., Risi, M., Tortora, G., and Dodero, G. “Documenting Design-Pattern Instances: A Family of Experiments on Source-Code Comprehensibility”. *ACM Transactions on Software Engineering and Methodology* 24.3, 2015.

[75] Schmidt, D. C. and Stephenson, P. “Experience Using Design Patterns to Evolve Communication Software Across Diverse OS Platforms”. *Proceedings of the 9th European Conference on Object-Oriented Programming*. 1995. DOI: `10.1007/3-540-49538-x_19`.

[76] Schumacher, M., Fernandez-Buglioni, E., Hybertson, D., Buschmann, F., and Sommerlad, P. *Security Patterns: Integrating Security and Systems Engineering*. John Wiley & Sons, Inc., 2006.

[77] Scientific Toolworks, Inc. *Understand - Visualize Your Code*. URL: `https://scitools.com/`, visited on 05/27/2020.

[78] Sfetsos, P., Ampatzoglou, A., Chatzigeorgiou, A., Deligiannis, I. S., and Stamelos, I. “A Comparative Study on the Effectiveness of Patterns in Software Libraries and Standalone Applications”. *Proceedings of the 9th International Conference on the Quality of Information and Communications Technology*. 2014. DOI: 10.1109/quatic.2014.26.

[79] Shi, N. and Olsson, R. “Reverse Engineering of Design Patterns from Java Source Code”. *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2006. DOI: 10.1109/ase.2006.57.

[80] Slashdot Media. *SourceForge - Download, Develop and Publish Free Open Source Software*. URL: https://sourceforge.net/, visited on 02/20/2020.

[81] Sourced Technologies. *Enry: A Faster File Programming Language Detector*. URL: https://github.com/src-d/enry, visited on 05/21/2020.

[82] Spinellis, D. “Tool Writing: A Forgotten Art?” *IEEE Software* 22.4, 2005. DOI: 10.1109/ms.2005.111.

[83] Spinellis, D., Gousios, G., Karakoidas, V., Louridas, P., Adams, P. J., Samoladas, I., and Stamelos, I. “Evaluating the Quality of Open Source Software”. *Electronic Notes in Theoretical Computer Science* 233, 2009. DOI: 10.1016/j.entcs.2009.02.058.

[84] Terra, R., Miranda, L. F., Valente, M. T., and Bigonha, R. S. “Qualitas.class Corpus: A Compiled Version of the Qualitas Corpus”. *ACM Sigsoft Software Engineering Notes* 38.5, 2013. DOI: 10.1145/2507288.2507314.

[85] Tessier, J. *Dependency Finder: A Suite of Tools for Analyzing Compiled Java Code*. URL: https://jeantessier.github.io/dependency-finder/, visited on 05/27/2020.

[86] The Apache Software Foundation. *Apache Maven Project*. URL: https://maven.apache.org/, visited on 02/20/2020.

[87] TIOBE Software BV. *TIOBE Index*. URL: https://www.tiobe.com/tiobe-index/, visited on 02/20/2020.

[88] Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., and Halkidis, S. T. “Design Pattern Detection Using Similarity Scoring”. *IEEE Transactions on Software Engineering* 32.11, 2006. DOI: 10.1109/tse.2006.112.

[89] Valenca, K., Canedo, E. D., and Figueiredo, R. M. D. C. “Construction of a Software Measurement Tool Using Systematic Literature Review”. *Proceedings of the 2018 IEEE International Conference on Internet of Things and IEEE Green Computing and Communications and IEEE Cyber, Physical and Social Computing and IEEE Smart Data*. 2018. DOI: 10.1109/cybermatics_2018.2018.00308.

[90] Virtual Machinery. *JHawk*. URL: http://www.virtualmachinery.com/jhawkprod.htm, visited on 02/20/2020.

[91] Vokác, M., Tichy, W. F., Sjøberg, D. I. K., Arisholm, E., and Aldrin, M. “A Controlled Experiment Comparing the Maintainability of Programs Designed with and with-

out Design Patterns—A Replication in a Real Programming Environment". *Empirical Software Engineering* 9.3, 2004. DOI: 10.1023/b:emse.0000027778.69251.1f.

[92] Wahyuningrum, T. and Mustofa, K. "A Systematic Mapping Review of Software Quality Measurement: Research Trends, Model, and Method". *International Journal of Electrical & Computer Engineering* 7.5, 2017. DOI: 10.11591/ijece.v7i5.pp2847-2854.

[93] Walton, L. *Eclipse Metrics Plugin*. URL: http://eclipse-metrics.sourceforge.net/, visited on 05/27/2020.

[94] Wang, Y., Zhang, C., and Wang, F. "What Do We Know about the Tools of Detecting Design Patterns?" *2018 IEEE International Conference on Progress in Informatics and Computing (PIC)*. IEEE, 2018. DOI: 10.1109/pic.2018.8706318.

[95] Wedyan, F. and Abufakher, S. "Impact of Design Patterns on Software Quality: A Systematic Literature Review". *IET Software* 14.1, 2020. DOI: 10.1049/iet-sen.2018.5446.

[96] Wohlin, C. "Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering". *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*. ACM Press, 2014. DOI: 10.1145/2601248.2601268.

[97] Yan, M., Xia, X., Zhang, X., Xu, L., Yang, D., and Li, S. "Software Quality Assessment Model: A Systematic Mapping Study". *Science in China Series F: Information Sciences* 62.9, 2019. DOI: 10.1007/s11432-018-9608-3.

[98] Zanoni, M., Arcelli Fontana, F., Stella, F., Fontana, F. A., and Stella, F. "On Applying Machine Learning Techniques for Design Pattern Detection". *Journal of Systems and Software* 103, 2015. DOI: 10.1016/j.jss.2015.01.037.