TU WIEN Informatics

An open-source tool for detecting violations of object-oriented design principles in Java

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Roland Oberweger, BSc
Matrikelnummer 01225446

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Franz Puntigam

Wien, 16. April 2020

Roland Oberweger	Franz Puntigam

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

An open-source tool for detecting violations of object-oriented design principles in Java

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Roland Oberweger, BSc
Registration Number 01225446

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Franz Puntigam

Vienna, 16th April, 2020

Roland Oberweger	Franz Puntigam

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Roland Oberweger, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 16. April 2020

Roland Oberweger

Acknowledgements

I want to thank my advisor, professor Franz Puntigam, for his extensive feedback to improve my thesis. My study colleagues I want to thank for their support during my study and the great time we had together. Thank-you to my friends and family, especially my parents who made my study possible. Thanks to my dogs for keeping me company and enduring all that rubber ducking. Last but not least, I want to thank the love of my life for all her love and support.

Kurzfassung

Designprinzipien helfen Entwicklern beim Erstellen von Designs, welche leicht implementiert und gewartet werden können. Designprinzipien sind aber nur Heuristiken. Eine Verletzung eines Designprinzips zu finden hat zwei Vorteile. Erstens hat eine Designprinzip-Verletzung, im Gegensatz zu einer Metrik, eine genaue Stelle im Programmcode. Zweitens ist das Designprinzip selbst schon die Anleitung, um das Problem zu beheben. Normalerweise wird die Einhaltung von Designprinzipien manuell überprüft, weil es wenig bis gar keine Toolunterstützung gibt, vor allem im Open-Source Bereich. Verfügbare Tools konzentrieren sich auf Metriken, häufige Programmierfehler und ein paar wenige Best-Practices, bringen diese aber nie in Verbindung mit Designprinzipien. Die meiste bestehende Literatur im Bereich der Designqualität und ihrer Messbarkeit fokussiert sich auf Metriken, aber es wurde auch schon daran geforscht Verletzungen von Designprinzipien zu finden. Designprinzipien können in Design-Best-Practices zerlegt werden, welche konkret genug sind, damit ihre Verletzungen mittels statischer Code-Analyse gefunden werden können.

In dieser Arbeit analysieren wir die Erkennbarkeit von Verletzungen von 23 Designprinzipien. Wir schauen uns alle Prinzipien an die bei einer Websuche aufscheinen und häufig diskutiert werden. Weil Designprinzipien vage sind, beschreiben wir sie genau. Für manche Designprinzipien gibt es bereits Erkennungsstrategien in der Literatur, für die restlichen haben wir selbst welche entwickelt. Um die Erkennungsstrategien zu evaluieren führen wir ein statisches Code-Analyse-Tool ein, welches die Erkennungsstrategien implementiert. Das neue Tool lassen wir über Open-Source Projekte laufen und analysieren das Ergebnis, um die Eignung der Erkennungsstrategien und das Tool selbst zu bewerten.

Unsere Evaluierung legt nahe, dass Verletzungen von 13 der analysierten Designprinzipien automatisch erkannt werden können. Wir finden heraus, dass Verletzungen von sieben Designprinzipien grundsätzlich nicht automatisch erkennbar sind. Bei zwei Prinzipien waren wir von ihrer automatischen Erkennbarkeit überzeugt, müssen diese Annahme aber aufgrund unserer Evaluierung wieder verwerfen. Für ein Designprinzip haben wir zu wenig Information um eine sichere Aussage treffen zu können. Wir denken die automatische Erkennung von Verletzungen von Designprinzipien ist hilfreich, um die Designqualität eines Projekts zu bestimmen. Die Anzahl an gefundenen Verletzungen ist ein Indikator für die Gesamtqualität. Weiters denken wir, dass die Typen von Verletzungen Aufschluss darüber geben, welche Designaspekte vernachlässigt wurden.

Abstract

Design principles help developers to create designs which are easy to implement and maintain but they are only heuristics. Finding a violation of a design principle has two benefits. First, unlike a metric a design principle violation has a location in the source code. Second, the the design principle itself is already the guidance on how to fix the problem. Usually, compliance with design principles is reviewed manually because there exists little to none tool support, especially in the open-source field. Available static code analyzers concentrate on metrics, common programming flaws and some design best practices but do not bring them into relation with design principles. Most existing research in the field of design quality and its measurement is focused on metrics but some research was already conducted on how to find violations of design principles. Design principles can be broken down into design best practices which in turn are concrete enough for static code analyzers to detect violations of them.

In this thesis, we analyze the automatic detectability of violations of 23 design principles. We take all design principles into account which come up via a web search and are frequently discussed. Because design principles are vague we describe them in detail. For some principles there are already detection strategies described in the literature, for others we come up with strategies ourselves. To evaluate the detection strategies we introduce a static code analyzer that implements the found strategies. We run the new tool against open-source projects and analyze the output to assess the appropriateness of the detection strategies and the tool itself.

Our evaluation suggests the automatic detectability of violations of 13 of the analyzed principles. We find violations of seven design principles are inherently not automatically detectable. Two of the principles we thought to be automatically detectable, but had to reject our assumption after the evaluation. For one principle we do not have enough information to make an informed statement. We find the automatic detection of violations of design principles useful to assess the quality of software projects because the amount of violations is an indicator for the overall design quality. We also think the types of violations give hints about the design aspects that got neglected.

Contents

CHAPTER 1

Introduction

1.1 Problem Description

Object-oriented design principles, such as the Single Responsibility Principle described by Robert C. Martin [29], help developers to create designs that are easy to implement and maintain, but they are more like heuristics than concrete rules [9]. Usually compliance with these design principles is reviewed manually, but research on how to measure object-oriented design was done since Chidamber and Kemerer published an article [7] for a metric suite in 1994 [4]. Such metric suites can indicate design flaws, but they do not lead developers to the cause of the problem [27]. Plösch et al. state that the adherence of design principles can be verified by checking for violations of design best practices. Design best practices like AvoidPublicInstanceVariables are more fine grained than design principles and can be checked with static code analyzers [37].

To prove this concept, Plösch et al. built an automatic code analyzing tool called MUSE [38] that can check for violations of design best practices. The result can be imported into "Software Product Quality Reporter" (SPQR) [13], which is used to build design quality models and can be used to map violations of design best practices to violations of design principles [3]. They only provide design best practices for ten design principles, although there are many more principles available and used in practice [3]. Furthermore, MUSE under the hood uses the commercial tool Understand[1] and MUSE and SPQR are not available publicly. To get violations of design principles one has to use all three, Understand, MUSE and a configured instance of SPQR.

The established open-source static code analyzers for Java, namely Checkstyle[2], Spot-

[1] https://scitools.com/
[2] https://checkstyle.org/

Bugs[3], PMD[4] and plugins for SonarQube like SonarJava[5], use a mixture of metrics and code smell detection. What they do not do, is linking the problems they find to design principles.

Therefore, there currently does not exist an open-source tool that out of the box explicitly warns developers about possible design principle violations in Java. The only tool that gives such warnings is MUSE in combination with SPQR which only covers ten principles and is not available as open-source.

1.2 Expected Results

The expected result is an open-source tool which can identify violations of object-oriented design principles in Java code. The tool will cover all commonly known principles, i.e., those principles that come up via Google and Google Scholar. It will be integrable into SonarQube, either by being a SonarQube plugin or by integrating into SpotBugs, respectively PMD, which both are available as SonarQube plugins. An integration with SonarQube is intended, because it already has the means to manage reported issues, e.g., mark them as false-positives.
To achieve this goal two research questions have to be answered:

RQ 1: Which violations of object-oriented design principles for Java can be detected automatically, and how can we do that?

Not all design principles mentioned in literature might be automatically detectable. First, existing design principles have to be collected. Only principles which come up via a web search and are frequently discussed will be considered. For each design principle it has to be evaluated if and how a violation of it can be detected automatically.

RQ 2: How useful is the automatic detection (according to RQ1) to assess the quality of a software project?

The methods obtained by answering RQ1 will be implemented in an open-source tool. Since design principles are just heuristics, the tool to be built is only expected to report possible candidates for violations. The decision, whether the reported violation is indeed existent or if the source code does not infringe any design principles, has to be made by a developer. To not waste the developers time, the tool needs to be evaluated to what degree it can detect violations of design principles and how many false-positives it produces.

In summary, the expected results are (1) a list of object-oriented design principles whose violations are automatically detectable in Java, (2) the description of the detection

[3] https://spotbugs.github.io/
[4] https://pmd.github.io/
[5] https://www.sonarsource.com/products/codeanalyzers/sonarjava.html

strategy for each item on the list and (3) an open-source tool that detects violations of design principles in Java out of the box and integrates with SonarQube. As a side product (4), a list of design principles whose violations cannot be detected automatically will be created.

1.3 Methodological Approach

First, a literature research will be conducted on several topics. Research on object-oriented programming will provide information about the object-oriented programming concepts that object-oriented design principles are about. Software design quality will also be researched to get information about what design means, why design quality matters, how it can be measured and to introduce terms like bad smell, design best practice and design principle. The next topic to research are design principles themselves, i.e., which do exist and how violations of each one can be detected.

With the information gathered with the literature research, the tool to automatically detect design principle violations can be build. For this the above mentioned existing open-source tools have to be assessed, as to which is best suited to integrate the new tool with, i.e., which makes the implementation of the new rules the easiest. This will be done by reading the documentation and, if not decidable by the documentation, writing small prototypes in each existing tool. To check the functionality of the new tool, examples will be written that violate design principles on purpose.

The evaluation of the tool will be based on the self written examples and by running the tool against open-source Java projects. Results of the tool will be checked at least at a sample basis depending on the amount of found possible violations.

1.4 Structure of the Thesis

Chapter 2 introduces the object-oriented programming paradigm and its concepts. Design principles make statements about the usage of these concepts, so each concept will be described shortly.

In Chapter 3 software design quality is described. This includes an introduction and a section about the relevance of design quality. Furthermore, in this chapter the terms used later on in the thesis, e.g., design best practice and design principle, are defined. How design quality can be measured is the last part and will describe different approaches to measuring design quality.

Chapter 4 contains a listing of design principles whose violations are automatically detectable. For each principle there is a description and the detection strategy for violations.

Chapter 5 lists those design principles whose violations are not automatically detectable. This principles are also described and the reason for the non-detectability is given.

Chapter 6 describes the implementation and evaluation of the new tool. First, the selection process of the base tool is discussed. Up next are selected implementation details of the new tool followed by the evaluation.

CHAPTER 2

Object-Oriented Programming

2.1 Introduction

Object-oriented programming is a programming paradigm initially used by SIMULA, a simulation language created in the 1960s. The first substantial implementation of object-orientation was SMALLTALK, developed in the 1970s. Other examples for old object-oriented programming languages are C++ and Eiffel [47][52]. Newer object-oriented languages include Java, C# and Python.

Encapsulation is important in software development because it is necessary to decompose systems into smaller units that are simpler to develop and maintain. Object-oriented languages do this by letting developers encapsulate procedures and local state in objects, rather than having programs and data separated from one another. Each object provides a service specified by its interface that other objects can use, i.e., send a message to the object. Callers only know the interface, the implementation is abstracted and can be changed independently. Object types can inherit from other object types and objects of such subtypes can be used wherever an object of supertype is expected. This implies that the receiver of a message can in general only be determined at runtime. [35][46][47][52]

In the following, the most important object-oriented concepts are briefly and superficially described. For example, there are single-inheritance and multiple-inheritance, but the chapter about inheritance will only explain single-inheritance because only the basic concept gets explained. Since this thesis concentrates on Java the concepts will only be explained in the way that Java uses them.

2.2 Concepts

2.2.1 Class

Classes are templates that are used to create one or more similar objects. Apple would be a class and apple-1 and apple-2 are objects, i.e., instances of that class. Apple is also the type of apple-1 and apple-2. Classes encapsulate data and procedures because they have visible operations, but also hidden instance variables and hidden methods. At runtime, a class gives the information how objects behave to incoming messages. During development it provides the developer with an interface that tells her or him how to interact with objects of that class. [1][35][47][52]

2.2.2 Object

Objects only exist at runtime. They are individual, identifiable instances of classes and therefore have the layout depicted by their classes. Each object has its own instance variables, but can receive the same messages as other objects of the same class. The state of an object is usually only accessible for the objects operations. The visible operations of an object are those of its interface. [1][35][46][52]

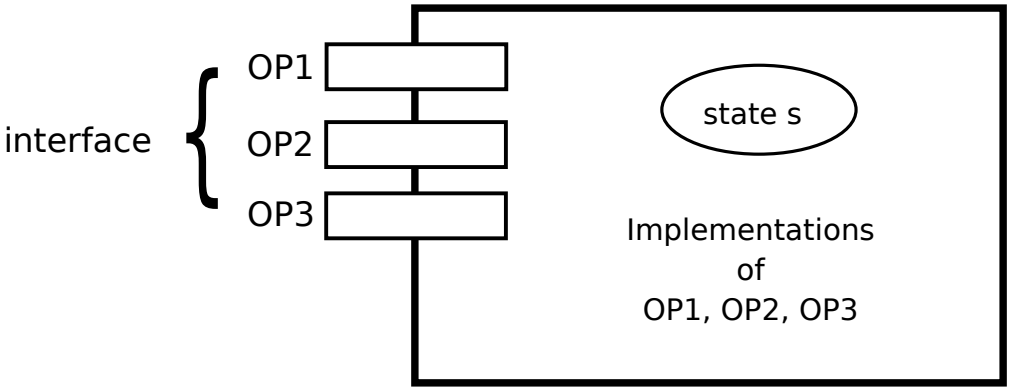

Figure 2.1: Structure of an object [52]

2.2.3 Subtyping

Classes can be subtypes of other classes. Class A is a subtype of class B if its interface makes it acceptable in contexts expecting type B. Class A might add additional functionality to its interface. A being a subtype of B implies, that an object of type A can be used where B is expected. [35][46]

2.2.4 Inheritance

Inheritance allows a class to be based on another class and inherit its behavior. Inheritance is therefore primarily a reuse technique. Subclasses might add new operations and instance

variables to provide more functionality than the superclass, but can also be allowed to override inherited operations. Many object-oriented languages tie subtyping and inheritance together, i.e., a subclass is always also a subtype. [1][46][52]

2.2.5 Polymorphism

Polymorphism means "having or assuming different forms". In object-oriented programming it is the ability of objects of different classes to handle the same message, but with different behavior. A message that can be sent to an object of a supertype can also be sent to objects of its subtypes. This enables reusability, because at compile-time a supertype can be specified, but the implementation (in the subtype) to be used, i.e., the executed behavior, is in general only decided at runtime. [1][35][47]

2.2.6 Abstraction

Users of an object should not make assumptions about its implementation. A class represents a simplified model, hiding irrelevant details from the user. As long as the external interface does not change, the implementation details can be changed at will. [1][46][47]

2.2.7 Encapsulation

Objects keep data and functions together and may hide them from users. The details of an object's implementation are kept secret, only an external interface is visible for the outside world. Therefore, implementation details can be changed without breaking the contract of the interface. [1][46]

2.3 Java

In Java, classes are defined using the `class` keyword. Classes can contain members (data) and methods, but at least have to have one constructor, a special function that is used to instantiate classes. Creating objects is done with the `new` keyword.

The visibility of members and methods can be restricted. For example, the `private` modifier limits the visibility to the class itself, the `protected` modifier constrains the visibility to the class itself and subclasses of it. Therefore, different levels of information hiding are possible.

A class can inherit from only one other class with the `extends` keyword. The subclass may override methods and add additional methods and members. It can only use members and methods of its superclass if the visibility modifier allows it. Subclasses are also always subtypes as far as the compiler can decide. Another way to use subtyping in Java is the utilization of interfaces. Interfaces are defined with the `interface` keyword and typically only contain method stubs, i.e., signatures without implementation. Since Java 8 it is possible to provide default implementations for methods. An interface can inherit

from multiple other interfaces. Making a class a subtype of an interface is done with the `implements` keyword, whereby a class can be the subtype of multiple interfaces.

Variables are typed and may hold objects of the specified type or subtypes of it, which enables polymorphism. Abstraction can, for example, be achieved by using interfaces or abstract classes which are partly implemented classes that can leave parts of the implementation to their subclasses. [61]

CHAPTER 3

Software Design Quality

3.1 Introduction

Software design quality is not about the quality of the product itself, but rather the quality of the code behind it. Human activities are error prone and software design is no exception. Perfect software design might therefore not be achievable and flaws in the design may have a strong impact on quality attributes like flexibility and maintainability. Today there are many legacy object-oriented systems that are monolithic, inflexible and hard to maintain. Just using an object-oriented language is not enough to produce good software. The development has to be done with design quality in mind, i.e., design rules and practices should be applied. [27]

3.2 Quality attributes and properties

Jagdish and Carl defined six quality attributes for their QMOOD metric. They are based on the ISO 9126 attributes functionality, reliability, efficiency, usability, maintainability and portability. Reliability and usability were left out because they have more to do with implementation rather than design. Portability was replaced by extendibility, efficiency with effectiveness and maintainability with understandability, to better suite design characteristics. Reusability and flexibility were added as important attributes of object-oriented design. Table 3.1 provides an overview of the quality attributes. [20]

Quality attributes are not directly measurable because they are too abstract. Design properties on the other hand are tangible concepts that can be observed directly by looking at the structure, relationships and functionality of attributes, methods and classes. The design properties abstraction, encapsulation, coupling, cohesion, complexity, design size, messaging, composition, inheritance, polymorphism and class hierarchies are often used as indicators of design quality. Table 3.2 gives the definitions of the design properties. [20]

Quality Attribute	Definition
Reusability	The design can be applied to new problems without significant effort.
Flexibility	Changes can be incorporated into the design. The design can be adapted to provide functionally related capabilities.
Understandability	The design can be easily learned and comprehended. Correlates directly with the complexity of the design structure.
Functionality	The responsibilities of classes of a design, which are provided through their public interfaces.
Extendibility	New requirements can be incorporated into the existing design.
Effectiveness	The design's ability to achieve the demanded functionality and behavior by using object-oriented concepts and techniques.

Table 3.1: Quality attributes (based on [20])

3.3 Relevance of design quality

Software is subject to constant change. Modern development approaches prefer short development cycles and raise the need for high maintainability and extensibility. Higher design quality means code that is easier to understand, maintain and extend. Improvements to design quality can be seen as an investment because they take up resources now, but reduce costs in the future. Such improvements are most profitable when they are done as early as possible. [5][44][50]

Trading in design quality for development speed is also known as technical debt. There is no problem in disregarding design quality for some time, as long as the design is improved later on and the technical debt is paid back. Stalling the payback for to long is what might lead to unproportional costs in the future. [5][8]

Experienced developers rate the importance of design quality higher than novices. Overall design quality is seen as relevant, but not as much as functional correctness. The time spent on design quality is cut by commercial time pressure. This pressure indicates that actions concerning design quality need to fit efficiently and effectively into the development process. This can be aided with appropriate tools especially because experience is a key factor for design quality which novices do not have. Therefore inexperienced developers are seen as main contributers to poor quality design. Tools are useful to point out areas of the code that may need design improvements. To achieve design quality developers also rely on Clean Code [29] practices and the related SOLID Principles. [50][54]

Design Property	Definition
Design size	The number of classes in a design.
Hierarchies	The number of non-inherited classes that have children.
Abstraction	A measurement of the generalization-specialization aspect, indicated by the depth of hierarchies.
Encapsulation	A measurement of the enclosing of data and behavior in classes, i.e., attributes and methods are declared private to prevent access from the outside.
Coupling	Measures the number of objects an object has to communicate with to function properly.
Cohesion	Measures the relatedness of methods and attributes in a class.
Composition	Measures the aggregation relationships of a design.
Inheritance	A measure of the "is-a" relationship between classes. Correlates to the depth of inheritance hierarchies, but focuses on the reuse of functionality.
Polymorphism	The ability to substitute objects with matching interfaces for one another at runtime. Measured by the amount of methods whose implementation is chosen on runtime.
Messaging	The number of public methods that are available to other classes.
Complexity	A measure of the difficulty to comprehend the internal and external structure and the relationships of classes.

Table 3.2: Design properties (based on [20])

3.4 Important terminology

3.4.1 Bad smells

Bad smell is a term coined by Martin Fowler and Kent Beck [10]. Smells are symptoms of code decay or other problems with the code quality that can lead to maintenance issues. The description of a bad smell contains a description of the symptom, i.e., what the code currently looks like and possible refactorings that can eliminate the smell. An example for a smell is "Duplicated Code". The symptom is code that does (nearly) the same in multiple places. To improve the code, one can, for example, use the refactorings

"Extract Method" or "Extract Class". [34][54]

Manually finding bad smells based on textual descriptions is time-consuming and error-prone. Thus there exist a variety of static code analyzers that can detect bad smells. Smells are just indicators for possible problems. Bad smells reported by tools still need human evaluation. Code generators might produce smelly code, but that might be ok if it does not have to be maintained by humans. [34]

3.4.2 Design best practices

Design best practices are heuristics that shall prevent developers from design pitfalls. In contrast to bad smells, which have a negative view and therefore describe already existing problems, design best practices have a positive view. Their descriptions are preventative, i.e., they describe what should be done to prevent bad smells. Design best practices make quality issues more tangible for developers because they help to communicate quality properties. Violations of design best practices can be used to assess the design quality. More violations equal lower design quality and vice versa. [2][37]

Sticking to the bad smells example, there is the design best practice AvoidDuplicates. It just states that source code should not be duplicated.

Brauer et al. conducted a survey with 214 participants on the importance of design best practices. They concluded that the practices AvoidDuplicates, AvoidUsingSubtypesInSupertypes, AvoidPackageCycles, AvoidCommandsInQueryMethods and AvoidPublicFields are rated as very important. On the not so important end of the spectrum are for example AvoidManyTinyMethods and AvoidProtectedFields. In summary, 26 of the 49 design best practices in question were rated as important. [2]

3.4.3 Design principles

Design principles are standards that are used to organize the structural components of software design. Applying design principles has multiple benefits. They help in building a common basis of architectural knowledge, help at designing large scale systems and protect beginners from pitfalls. [15]

Design principles exist on two abstraction layers. There are coarse-grained and fine-grained design principles. Each coarse-grained principle is one of the four traits of good object-oriented design, which are low coupling, high cohesion, moderate complexity and proper encapsulation. These principles are so abstract that they are difficult to apply in practice. [37][42]

Fine-grained design principles on the other hand are concrete enough to guide developers to build, understand and maintain high-quality software systems. An example is the Single Responsibility Principle, which is one of the five SOLID principles by Martin [29]. The SOLID principles are a part of a much bigger catalog of design principles. [29][43]

Plösch et al. conducted a survey about the importance of different design principles to the practitioners. The highest ranking ones are Single Responsibility Principle, Separation of Concern Principle, Information Hiding Principle, Don't Repeat Yourself Principle and Open Closed Principle.

Design principles relate with design best practices in the way, that best practices ensure the adherence of design principles. Figure 3.1 gives an overview of the relation of bad smells, design best practices and design principles. [37]

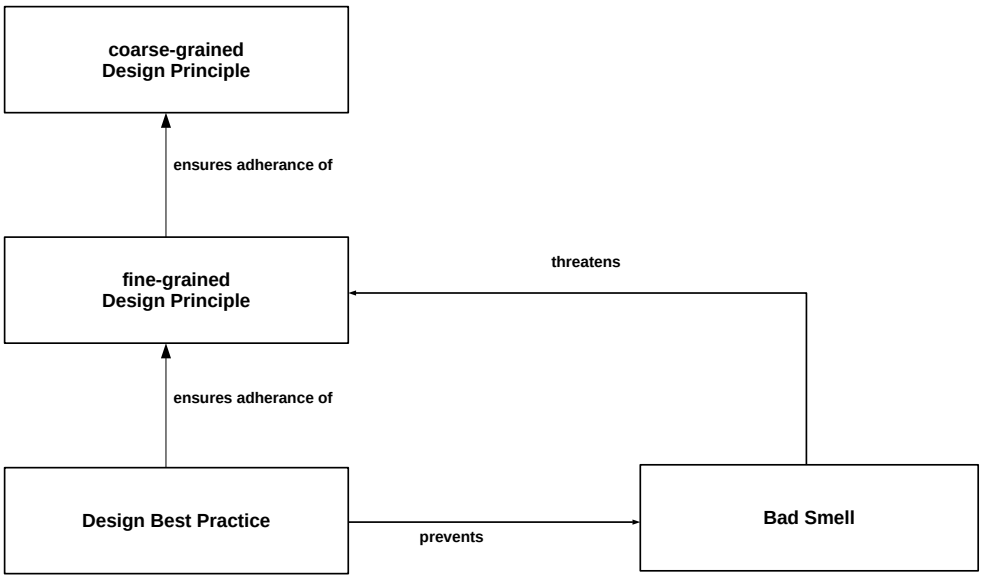

Figure 3.1: Relationship between bad smells, design best practices and design principles (based on [37])

3.5 Measuring design quality

Measuring design quality makes it possible to identify problems and trends. Further, it enables the comparison of products and processes. Even though there is no single definition of quality, which also means there is no simple measure of design quality that is acceptable to everybody, having quality defined in a measurable way makes it easier to discuss viewpoints. [12]

3.5.1 Approaches

Manual

The manual approach is an activity executed by a human which is based on guidelines and expertise. The analyst searches for bad smells in the system under investigation. The outcome is mainly based on her or his expertise making the result rather unpredictable. This approach is expensive, unrepeatable and non-scalable. On the upside, a human

manually assessing a design can incorporate domain specific aspects and design decisions, for example based on framework usage, into the evaluation. [3][26][42]

Metric-based

The metric-based approach uses software to calculate metric values that represent design aspects of a system. An early and very influential metric suite was published by Chidamber and Kemerer. This metric suite consists of six metrics, for example "Number of Children (NOC)", which reports the number of immediate subclasses of a class. Unfortunately, a single metric in isolation makes it hard to find the underlying design flaw and does not imply any possible improvements. To mitigate this weaknesses, Marinescu combined multiple metrics to so called detection strategies. This approach makes it easier to locate design flaws. Since metrics are just numbers there need to be thresholds that indicate problems. Improper adjustment of these thresholds leads to unrecognized flaws and incorrect suspects, i.e., false-negatives and false-positives. Additionally, metrics cannot express design flaws that are on a semantic level, for example, flaws related to naming. [3][6][7][27][34]

Rule-based

The rule-based approach uses software to search for violations of design best practices in software artifacts. Such artifacts can be for example UML class diagrams, documentation or source code. Violations of rules can be exactly located. Furthermore, each rule, i.e., each design best practice, has a name, a description and an improvement advice, making it easier for developers to deal with found violations. [3]

CHAPTER 4

Detectable Design Principles

In this chapter we deal with all design principles whose violations can be automatically detected with a static code analyzer. To be discussed in this thesis, a design principle has to be published in literature and/or has to be frequently discussed on the internet. Hence, design principles only proposed by someone on the internet without any discussions are not qualified. Also, principles that are published in a single book or paper but are not taken up by others or discussed further are also not considered. To find relevant design principles Google and Google Scholar were used. The websites Principles Of Object Oriented Design [66], Principles Wiki [67] and Prinzipien der Softwaretechnik [69] proved themselves as very helpful. The lists on these websites were used as a starting point for further research. According to our research we found violations of the following design principles to be automatically detectable:

- Single Responsibility Principle (SRP)
- Open Closed Principle (OCP)
- Interface Segregation Principle (ISP)
- Information Hiding (IH)
- Law of Demeter (LOD)
- Keep It Simple Stupid (KISS)
- Don't Repeat Yourself (DRY)
- Information Expert (IE)
- Single Choice Principle (SCP)

- Program to an Interface, not an Implementation (PINI)
- Favor Composition over Inheritance (FCOI)
- Common Reuse Principle (CRP)
- Acyclic Dependencies Principle (ADP)
- Stable Dependencies Principle (SDP)
- Stable Abstractions Principle (SAP)
- Option Operand Principle (OOP)

4.1 Single Responsibility Principle (SRP)

Variants

- One Responsibility Rule
- Separation of Concerns
- Curly's Law
- Do One Thing

Description

The Single Responsibility Principle states that a class should only have one reason to change. This principle aims to make the code more cohesive. The idea is that a change should impact as few dependencies[1] as possible. Having a class with more than one responsibility makes the class more likely to change. A class with a single responsibility is also easier to comprehend which is very important taken into account how often code is read. A bad example would be the following Employee class:

```
class Employee
{
    ...
    public Money calculatePay (...) ...
    public void saveToDatabase(...) ...
    public String reportHours(...) ...
    ...
}
```

[1] Dependency and dependencies are synonymously used with dependence and dependences in this thesis.

This class would be changed if the requirements for the payment calculation, for the database storage or for the format of the hours would change. This class currently has multiple responsibilities. A change to the database storage mechanism might influence the format of the hours by accident. By delegating the responsibilities to other classes a change has less influence. Just introducing these classes would look like this:

```
class Employee
{
    ...
    private PayrollCalculator pc;
    private DatabaseStorer ds;
    private HoursFormatter hf;
    ...
    public Money calculatePay(...)
    {
        return pc.calculatePay(this,...)
    }
    public void saveToDatabase(...)
    {
        ds.saveToDatabase(this,...)
    }
    public String reportHours(...)
    {
        return hf.reportHours(this,...)
    }
    ...
}
```

One step further, one can move the three methods to different classes which only have the responsibility of managing the interaction.

```
class PayrollEmployee
{
    ...
    private Employee emp;
    private PayRoll pr;
    ...
    public Money calculatePay(...)
}
```

```
class DataBaseEmployee {
    ...
    private Employee emp;
    private Database db;
    ...
    public void saveToDatabase(...)
}
```

Making such separations is only advisable if the system really changes in this separate ways. If a change to the payment calculation always comes hand in hand with a change of the database storage mechanism, then those two would not benefit from applying the Single Responsibility Principle since it would unnecessarily increase the complexity. [15][19][29]

Relation to other principles

Encapsulate the Concept that Varies (ECV): Both principles aim at keeping changes local by encapsulating code that changes together.

Violation detection strategy

Design best practices: AvoidNonCohesiveImplementations

A violation of the Single Responsibility Principle might exist if the class is not very cohesive. This can be a warning sign because parts of the class do not interact with each other and should therefore be separated into multiple classes. A commonly used metric for cohesion is LCOM4 [17] which results in a value higher than 1 for non-cohesive classes. It gives the number of disconnected method sets whereby connections are method calls and the use of the same instance variable. [3][19]

Bräuer[3] proposed a second best practice named CheckUnsuitableFunctionalityOfClass. He says, if clients typically only use portions of the classes interface, i.e., its public methods, the class might provide different functionality to different clients. We think this is a sound best practice for humans, but can be problematic if implemented with a static code analyzer. Classes can offer nearly the same functionality in slight variations, i.e., methods that do nearly the same. Those methods will commonly not be used together by clients. A counterexample is a class which follows the Option Operand Principle and has a method with a boolean option split into two methods. The class would be flagged as violating the Single Responsible Principle because those two methods will probably not be used both by a single client. To prevent such false-positives we think this best practice should not be implemented.

Origin

The Single Responsibility Principle is one of the **S**OLID principles by Martin published in [29].

4.2 Open Closed Principle (OCP)

Variants

-

Description

Meyer defined the Open Closed Principle over modules that should be open for extension but closed to change. The solution for him was to use inheritance. The here more relevant definition comes from Martin. He says that software entities (classes, methods, ...) should be open for extension but closed for modification. This sounds like a contradiction at first but still is achievable. [29][32]

If a single change creates a ripple of changes in dependent entities the design is too rigid. The design was not open for extension. Open for extension here means that the behavior of the entity can be changed to fulfill new requirements. Closed for modification means the change in behavior should not need a change in the already working source code. It should be possible to fulfill the new requirements by adding new code.[29]

Taking the example from [29] we have two shapes, circle and rectangle, and want to draw a list of shapes somewhere. A solution violating the Open Closed Principle would look like this:

```
public class Circle
{
    public void draw(){...}
}
public class Rectangle
{
    public void draw(){...}
}
```

```
public void drawShapes(Collection<Object> shapes)
{
    for(Object shape : shapes)
    {
        if(shape instanceof Rectangle)
        {
            ((Rectangle) shape).draw();
        }
        else if(shape instanceof Circle)
        {
            ((Circle) shape).draw();
        }
    }
}
```

Adding the shape triangle to this code would need a change of the drawShapes() method. Most likely, this is not the only code piece that handles shapes so all places with such if/else cascades need to be changed. To adhere to the Open Closed Principle an abstraction[2] can be introduced.

```
public interface Shape
{
    public void draw();
}
public class Circle implements Shape
{
    @Override
    public void draw(){...}
}
public class Rectangle implements Shape
{
    @Override
    public void draw(){...}
}
public class Triangle implements Shape
{
    @Override
    public void draw(){...}
}
```

[2] Abstraction here and later in the thesis is used as a synonym for supertype which can be an interface or a class (an abstract class).

```
public void drawShapes(Collection<Shape> shapes)
{
    for(Shape shape : shapes)
    {
        shape.draw();
    }
}
```

The drawShapes() method is now open for extension because new shapes can also be drawn with it, but closed for modification because a new shape requires no change of the method. The new requirement is implemented somewhere with new code that produces triangles and provides them to drawShapes().

Relation to other principles

Encapsulate the Concept that Varies (ECV): The Open Closed Principle aims to encapsulate the concept that varies in an abstraction. Clients only use the abstraction, the variations, i.e., the implementations are hidden from clients.

Dependency Inversion Principle (DIP) and Liskov Substitution Principle (LSP): These principles should also be considered when applying the Open Closed Principle because these two give advice on the use of abstractions.

Violation detection strategy

Design best practices: DontReturnMutableField, UseAbstractionAsParameterType, UseAbstractionAsReturnType, AvoidPublicFields, AvoidPublicStaticFields, AvoidProtectedFields, AvoidRuntimeTypeIdentification

Detecting a violation of the Open Closed Principle is not straightforward because it has a lot to do with predicting future requirements and protecting the software entity from change resulting from that new requirements. Still, some precautions can be taken. Methods should not return mutable fields and use abstractions as parameter and return types. The first protects against changes needed because callers start manipulating the return value. The second leaves the method open for extension as some new requirement might already be covered by the abstraction. Further, public fields, static and non-static, as well as protected fields should be avoided. A class that has public or protected fields might need to change because others depending on these fields are changing, which goes against the closeness of the Open Closed Principle. Last but not least, runtime type identification should be avoided. As shown in the example above, such checks might need to be changed when a new requirement comes in. [3]

Origin

The Open Closed Principle is originally from Meyer[32], but is also one of the SOLID principles by Martin published in [29].

4.3 Interface Segregation Principle (ISP)

Variants

-

Description

The Interface Segregation Principle demands that clients should not be forced to depend on methods they do not use. An interface that has groups of methods that are used by different clients leads to multiple problems. One comes for classes that implement such fat interfaces. [29]

In the following example the Mammal interface is a fat interface, that has unnecessary methods for some implementations.

```
public interface Mammal
{
    void breath();
    void fly();
}
public class Tiger implements Mammal
{
    @Override
    public void breath(){/*code*/}
    @Override
    public void fly(){throw new NotImplementedException();}
}
```

The Tiger class is required to provide a stub implementation for the fly method. This is unnecessary code that needs to be maintained. If the fly methods signature changes for some reason, Tiger would also need to be changed, although it cannot fly anyways. Additional behavior added to the Mammal interface that the Tiger is not capable of also leads to more changes and more stub methods in the Tiger class. Following the Interface Segregation Principle, the Mammal interface needs to be split up.

```
public interface Mammal
{
    void breath();
}
public interface FlyingMammal extends Mammal
{
    void fly();
}
public class Tiger implements Mammal
{
    @Override
    public void breath(){/*code*/}
}
public class Bat implements FlyingMammal
{
    @Override
    public void breath(){/*code*/}
    @Override
    public void fly(){/*code*/}
}
```

The Tiger now only has the methods that its clients really need. Also, changes to the flying behavior only affects the Bat, but not the Tiger.

Calling clients benefit because in the first version the Tiger has a fly method that only fails at runtime. This might be obvious in this example, but in a real code base this could lead to confusion and unintended behavior. Clients also might need to be adapted or at least recompiled because a part of an interface they do not use is changed. A smaller interface isolates the clients from each other. Changes to one interface cannot affect clients of another interface. [29]

Relation to other principles

Single Responsibility Principle (SRP): Entities that have only one responsibility will also have smaller interfaces, increasing the chance that a client does not depend on more than he needs. Nevertheless, classes that adhere to the Single Responsibility Principle can still violate the Interface Segregation Principle because some clients do not need the whole interface.

Information Hiding: Breaking the interface of a class into smaller pieces enables information hiding because clients only know about the part of the whole interface that they need to know.

Program to an Interface, not an Implementation: Having small interfaces defined for clients facilitates the use of interfaces.

Violation detection strategy

Design best practices: AvoidStubForInheritedMethod

For the Interface Segregation Principle Bräuer[3] again proposed the CheckUnsuitableFunctionalityOfClass best practice. If clients of an interface commonly consume only a part of the interface, this might indicate that the interface is too fat and needs to be split up. Again we think this is a valid best practice to be used by humans, but not by static code analyzers for the same reasons depicted in the Single Responsibility Principle chapter. Methods that provide nearly the same functionality with different options will frequently not be used together by a single client. Having a separate interface for each client would make the system overly complex to maintain and is not viable. Thus, in our opinion this best practice would produce unjustifiable false-positives.

An implementable warning sign are classes that do not provide an implementation for a method of an interface they implement. Not providing an implementation here means an empty method body or a body with a single statement which throws an exception. [3][29]

Origin

The Interface Segregation Principle is one of the SOLID principles by Martin published in [29].

4.4 Information Hiding (IH)

Variants

-

Description

Information Hiding is a principle that says to hide implementation details as much as possible. If there is information in a class A that B does not need to know to use A, then it should be hidden. This ensures that changes to A that are not modifying its interface stay local, they do not cause ripples in the system. Changes to the hidden internals cannot affect B because B does not even know about them. [36].

It is to note that encapsulation is not the same as information hiding. Encapsulation means combining data and functions but they are not necessarily hidden. So encapsulation enables information hiding but does not enforce it. [51]

To hide information one should use the lowest possible access modifier for variables and methods.

Access to attributes should be prohibited and if necessary only be possible through methods. As Meyer describes in his Uniform Access Principle[32], all data accesses should happen in a uniform way, i.e., only through methods. This way, the source of the

requested value is hidden. What can be a field now may need to be calculated on request in the future which only a method can do.

Fitting abstractions will not let others modify their state directly through setter methods. Because they can change the internal state the setters should contain checks to prevent an invalid state. Leaking internals can be prevented by copying objects before returning them or making them immutable.

Relation to other principles

Information Segregation Principle: Breaking the interface of a class into smaller pieces enables information hiding since clients only know about the part of the whole interface that they need to know.

Law of Demeter (LOD) and Information Expert (IE): Both principles also try to hide implementation details and keep clients from depending on them.

Violation detection strategy

Design best practices: AvoidPublicFields, AvoidProtectedFields, AvoidSettersForHeavilyUsedFields, DontReturnMutableField, UseAbstractionAsReturnType

The most massive violation of the Information Hiding Principle are public fields. Classes that have public fields lost the control over their state and can be changed from the outside at any time in unpredictable ways. Protected fields have the same problem. Setting heavily used fields from the outside should not be possible at all. Heavily used is pretty subjective and should be configurable as the percentage of methods which use the specific field. Internals that leak through return values are also a violation. Returning a mutable field of the class by reference makes it de facto public and enables arbitrary changes to the internal state. The use of concrete types as return types may also needlessly leak information about the internals of a class. [3]

Origin

Information Hiding was first described by Parnas in [36].

4.5 Law of Demeter (LOD)

Variants

- Principle of Least Knowledge
- Don't Talk to Strangers

Description

The Law of Demeter as described by Lieberherr, Holland, and Riel[23] states that a method M of class C should only call the following methods:

1. Methods directly in its class C
2. Methods of fields of class C
3. Methods of parameters of M
4. Methods of objects created by M

It is not allowed to call the method of an object returned by neither of the above method calls. This principle aims to prevent objects knowing too much about the internals of other classes. Calling a method on the result of a method call means the caller needs to know about the internals of the result. This also aids testability since only the direct objects from above need to be mocked, but not results of calls to them. A study also found a correlation between violations of the Law of Demeter and the bug proneness of projects. [63][14]

The following code piece illustrates all of the allowed calls from above (referenced by their number) and illegal calls.

```
public class House
{
    private Set<Person> persons = new HashSet<>();
    private Floor floor;
    private boolean ownerIsHome;

    public void openDoor(){}
    public void closeDoor(){}
```

```
    public void letInPersons(Persons persons)
    {
        // Case 1: Method of the same class
        openDoor();
        // Case 3: Method of a parameter
        Iterator<Person> it = persons.iterator();
        // Illegal case A
        while(it.hasNext())
        {
            // Illegal case A
            Person p = it.next();
            // Case 2: Method of a field
            this.persons.add(p);
            // Illegal case B
            if(p.getBelongings().getHouse().equals(this))
            {
                ownerIsHome = true;
            }
        }
        closeDoor();
        Dirt dirt = new Dirt();
        // Case 4: Method of locally created object
        dirt.stickTo(floor);
    }
}
```

Case B is the one that the Law of Demeter tries to prevent. House is now coupled to the Belongings class and knows about the internals of Persons, Person and Belongings. To adhere to the Law of Demeter, a new method needs to be provided by Persons because this is the only object letInPersons() may send messages too.

```
...
if(persons.anyIsLivingIn(this))
{
    ownerIsHome = true;
}
...
```

With this function call House is only coupled to Persons and knows nothing about its internals. On the other hand Persons has to provide a specialized method. Following the Law of Demeter leads to more methods.

Case A is in conflict with the Law of Demeter but is also a very common construct in Java since Iterators can only be used this way. One could refactor the code in a way

that the Persons object handles the entrance but this would mean two more methods. Persons needs to provide a method to give the House to and House needs a method to add a single Person.

```
...
persons.enter(this);
...
house.addPerson(person);
...
```

Adherence to the principle can lead to more complex code than necessary. This is the reason why the Law of Demeter is controversial. To obey the law, a high number of delegate methods may be needed that do nothing than provide the methods of internal objects to the outside. [53]

The extreme case are objects that only hold data, so called plain old java objects or POJOs. These objects provide no action, data can only be set and retrieved. Such POJOs can also be nested.

```
public class Car
{
    private SteeringWheel steeringWheel;
    private Engine engine;
    ... more fields
    ... getters and setters
}
public class Engine
{
    private IgnitionPlug ignitionPlug;
    ....
}
```

If a caller now wants to get the production date of the ignitionPlug, it has to chain some getter calls.

```
car.getEngine().getIgnitionPlug().getProductionDate()
```

Applying the Law of Demeter means to provide a method in car that returns that date directly.

```
car.getIgnitionPlugProductionDate()
```

Such delegate methods would be needed for every nested getter method which bloats the interface of Car. Thus it was proposed to not apply the Law of Demeter on POJOs but only on objects that are containing behavior. [64]

Other exceptions are in our opinion the builder pattern and APIs with a functional style, like the Stream API. Those use method call chains for simpler use and readability, making the principle not applicable for them.

Relation to other principles

Information Hiding (IH): Both principles try to hide implementation details and keep clients from depending on them.

Information Expert (IE): Information Expert advocates to have active objects that are asked to do a task rather than having passive objects that get queried for information and maybe updated. Objects should not reach through to strangers but only communicate with the information expert.

Violation detection strategy

Design best practices: AvoidChainedMethodCalls

Detecting violations of the original Law of Demeter would be straightforward because all chained method calls are forbidden. Masking the chained calls with local variables is also detectable, but as already described there are common exceptions from the Law of Demeter. We think that for a human the principle is a good guide to avoid unnecessarily nested calls, but a static code analyzer does not have the same context at hand. Simply flagging all chained calls would lead to an unreasonable number of false-positives. POJOs can be detected by their structure because their methods only set or return fields. In our opinion chained calls to POJOs should be legal to minimize the number of false-positives and special cases like Iterators, Streams and the builder pattern should also be excluded from the rule.

Origin

The Law of Demeter was first described by Lieberherr, Holland, and Riel [23].

4.6 Keep It Simple Stupid (KISS)

Variants

- Rule of Simplicity

Description

Keep It Simple Stupid means to find the simplest solution that works. Developers can be tempted to write unnecessarily complex code to prove their ability to handle this complexity. It might also be the case that they want to write future proof code. This leads them to the usage of hard to understand language features like inheritance and polymorphism. The solution they built might never be needed because one cannot look into the future, but its now implemented that way and harder to understand and maintain in the future. The fault might not even be with developers, but with product managers who want features implemented which will never get used. These also can influence the maintainability of the overall code base. [39]

The principle advices to avoid complex language features but only if they are not the easiest solution. If inheritance and polymorphism are the right tool for the job they should by used by all means. [40]

As Keep It Simple Stupid is a very general principle the ways to adhere to the principle are also general. One should try to avoid complex language features. Premature optimization can result in more complex code although a simple solution might also have worked. Large and complex code blocks are also to be avoided. Long methods and many control statements are harder to understand and might be avoided with more classes and methods. [33][40]

Relation to other principles

-

Violation detection strategy

Design best practices: AvoidLongMethods, AvoidComplexMethods

As already stated, Keep It Simple Stupid is a general principle. Therefore, there is no straightforward way to detect violations, i.e., overly complex solutions. A static code analyzer does not have enough context to decide if a problem could, for example, also have been solved without the use of inheritance. If a specific code piece needs to be optimized or not and if a feature is needed at all are not decidable for a static code analyzer.

We think the only interpretation of Keep It Simple Stupid that a static code analyzer can check for are hard to read code constructs like long methods or a high cyclomatic complexity of methods. Both decrease the readability and are therefore violating the principle.

Long method means the number of lines exceeds a predefined threshold. This threshold is opinion based since there is no number at which readability decreases drastically. The maximum number of lines should be set by the developers of a project according to their shared opinion. Books and discussions on the internet [74][58][28][31] recommend sizes of

around 20 to 30 lines but others advocate for 100 to 200 lines. A higher number than 200 was not recommended by anyone so this can be taken as a defensive default value.

Cyclomatic complexity is a metric that measures the complexity of a program. It was developed by McCabe [30] and calculates the number of linearly independent paths through a program. Sequential statements do not increase the cyclomatic complexity but control statements like if and while do. Figure 4.1 shows the complexities of common control statements whereby e is the number of edges, n is the number of nodes and p the number of connected components (in these cases always 1). In the original paper 10 was stated as the upper bound at which methods become more difficult to comprehend. McConnell [31] has the same opinion and therefore it can be chosen as a default value to indicate problematic methods.

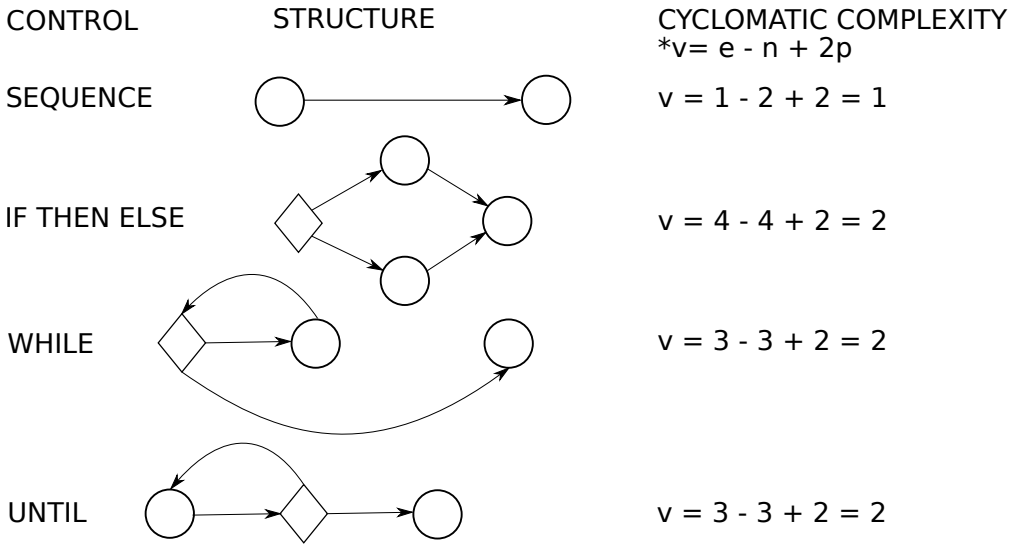

Figure 4.1: Cyclomatic complexities (based on [30])

Origin

The principle is reported to first be stated by Kelly Johnson, an engineer working for Lockheed and building military aircrafts. [41]

4.7 Don't Repeat Yourself (DRY)

Variants

- Single Point of Truth
- Single Source of Truth

Description

> "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system." [18]

Requirements of software are constantly changing and so the software has to change too. Having duplications in the code, i.e., the same knowledge or behavior is implemented multiple times though doing (nearly) the same, leads to more effort. Every change has to be performed on every duplication. Forgetting to change a duplication can create contradictions. The Don't Repeat Yourself principle states to avoid duplications, but only have one implementation. Thus, only one part of the code has to be changed and no other part can be forgotten. [18]

Some degree of duplication is unavoidable because documentation is always a duplication of what is written down in code. This duplication should be minimized as much as possible. To mitigate the problems that come with duplication the Self Documentation Principle[32] suggests to put documentation next to the code, i.e., it should be part of the module. In Java this can be done with Javadoc and inline comments. The spacial proximity increases the chance of consistency, i.e., if code is changed, the documentation is also updated.

Hunt and Thomas[18] describe four reasons for duplicated code:

- **Imposed duplication:** During development different representations of the same information are needed. For example a Java class that matches the corresponding database table. Such duplications may be avoided with code generators that build the different representations from a single source. Documentation can also be a duplication of the code. Therefore it should not document what the code does, but give higher-level information, e.g., why the code does this.
- **Inadvertent duplication:** Mistakes in the design can lead to duplication. An example are classes with unnormalized data. A Line class should not have a length attribute besides a start and an end point, because the length can always be calculated. A separate field may be forgotten to update and contradict the other information. Having a field for performance reasons is valid, but should be hidden behind a method.
- **Impatient duplication:** Time pressure and laziness can tempt a developer to just copy an existing code part and make the small needed changes instead of extracting the shared functionality. This saves a short time now but may cost a lot more in the future. This sort of duplication can easily be detected and prevented, but needs discipline and resources to spend up front.
- **Interdeveloper duplication:** Multiple developers working on the same project simultaneous can implement the same behavior at the same time without knowing

about the other implementation. Such duplications can be undetected for years because nobody reads both pieces. Technical leads may realize the similarity in tasks and bring attention to the developers so they talk with each other. Another way is to appoint responsibilities, i.e., some developers know how to implement something and the already existing tools for the task. This information can also be provided in a wiki or some other form of shared knowledge space.

To prevent duplication of code, the existing code has to be easy to reuse. Documentation aids the reusability since it tells other developers when and how to use existing code. Public interfaces, classes, enums, annotations and methods should be documented. Big chunks of inline comments should be avoided on the other hand because they might only tell what the code is doing, which is duplication. [3][18]

Relation to other principles

Single Choice Principle (SCP) and Encapsulate the Concept that Varies (ECV): These two principles support the Don't Repeat Yourself principle because both try to keep a list of choices in only one place. This prevents duplicated code that decides between the possibilities in multiple places.

Violation detection strategy

Design best practices: AvoidDuplication, DocumentPublicInterfacesClassesEnumsAnd Annotations, DocumentPublicMethods, AvoidMassiveInlineComments

An obvious violation of Don't Repeat Yourself is duplicated code. A number of lines that do exactly the same thing should not occur in the system. To avoid the creation of too many methods, there should be a threshold for the number of lines that are duplicated since two lines that are copied might be more understandable and easier maintainable than a separate method in another class. [73]

As described the Don't Repeat Yourself principle is not only about having no code duplication, but also about helping to avoid duplication. We therefore think that insufficient documentation is also a violation. Interfaces, public classes, enums, annotations and public methods should all have Javadoc attached, so another developer can use them easier. Inline comments however should not be needed on more than a quarter of the statements.

Origin

The Don't Repeat Yourself principle was first described in [18] by Hunt and Thomas.

4.8 Information Expert (IE)

Variants

- Tell Don't Ask
- Do It Myself

Description

The Information Expert principle gives advise about the responsibilities of objects. It says the responsibility for a task should be with the object which has the largest subset of the required information. In practice, it can be used to find the place to put a needed method. Following this principle leads to lower coupling because the information expert does not need to ask other objects for information or at least the minimal amount of other objects since it has the most information at hand. If a responsibility is not given to the information expert, the now responsible object has to ask for information. Therefore, also the variant Tell Don't Ask. [22]

The following example from [22] demonstrates the use of Information Expert. There are already three classes Sale, SaleLineItem and ProductDescription.

```
class Sale
{
    private List<SaleLineItem> items;
    ...
}
class SaleLineItem
{
    private ProductDescription product;
    private int quantity;
    ...
}
class ProductDescription
{
    private double price;
    private String name;
    ...
}
```

Currently, there is no method to get the total amount of a sale and it should be added. To calculate the total for each sold item, the quantity and the price are needed. According to Information Expert, the class to give the responsibility to is the one with the most information. Because sale already has the list of sold items it is the expert and getTotal()

should be added to it. The first solution looks like this, but violates the Information Expert principle.

```
class Sale {
    private List<SaleLineItem> items;
    public double getTotal() {
        double total = 0.0;
        for(SaleLineItem item: items) {
            ProductDescription product =
                item.getProductDescription();
            total += item.getQuantity() * product.getPrice();
        }
        return total;
    }
}
```

This works, but Sale is not the expert in calculating the subtotal per item. SaleLineItem has more information since it already knows about the quantity so it should be responsible for the subtotal.

```
class Sale
{
    private List<SaleLineItem> items;
    public double getTotal()
    {
        double total = 0.0;
        for(SaleLineItem item: items)
        {
            total += item.getSubtotal();
        }
        return total;
    }
}
class SaleLineItem
{
    private ProductDescription product;
    private int quantity;
    public double getSubtotal()
    {
        return quantity * product.getPrice();
    }
}
```

In this solution every object is the information expert because ProductDescription is the expert for the price. In contrast to the first solution, Sale is not coupled to ProductDescription.

There are situations where applying Information Expert would lead to higher coupling and lower cohesion. Many applications need some sort of persistence. For example we want to save a Dog object. According to Information Expert, Dog would be the information expert for saving itself because it knows everything about itself. Giving the responsibility for saving to another object means it has to ask for all the information stored in Dog. In this case the latter is preferable because otherwise Dog would need information about the database schema and database connections and so on. This lowers the cohesion of Dog and couples it to database related classes which might even be on another application layer. [22]

Relation to other principles

Law of Demeter: By applying the Information Expert principle, obeying the Law of Demeter gets easier because one does not have to ask for information in many places which includes strangers.

Violation detection strategy

Design best practices: AvoidGetAndSet, AvoidUnnecessaryInformationTransfer

We think a violation might exist if for an attribute first the getter and then the setter method is called. This means the calling object asks for information, performs a task with it and stores the (probably) changed information back. In this case the called object might be the information expert and the task should be performed in the called object, not the calling object.

Calling multiple getter methods of a single object can in our opinion also be an indication of a violation. The called object is the expert, otherwise the getter calls would not be needed and it is probably able to perform the needed task itself.

Also, calling multiple getter methods of different objects in one place means the calling object gathers a lot of information to do something with it. In this case there are multiple partial information experts. The principle says to give the responsibility to the partial expert with the most information. If there is someone with more information than others the second violation case would alert because multiple getter calls go to one object. If the information is split equally, the responsibility should be, according to the principle, given in such a way that lowest coupling and highest cohesion are the goal. This type of violation is hard to detect in a helpful way. The static code analyzer could only check for multiple getter calls but cannot evaluate if this is the right spot for them. Therefore, we believe it would likely produce a high number of false-positives and due to that this case should not be enforced with a tool.

Origin

The Information Expert principle originates from Larmans book "Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development" [22].

4.9 Single Choice Principle (SCP)

Variants

-

Description

> "Whenever a software system must support a set of alternatives, one and only one module in the system should know their exhaustive list."[32]

Having to decide between multiple alternatives is common in software systems. The following code pieces are an example of such a case differentiation over a known set of values.

```
enum EDepartment
{
    SALES,
    HR;
}
...
switch (department)
{
    case HR:
        new HumanRessources().getNumberOfEmployees();
        break;
    case SALES:
        new Sales().getEmployees().size();
        break;
    default:
        ...
}
...
```

Such switch statements (or their corresponding if/else versions) can be distributed over the source code. If a new department has to be added all of these switch statements have to be changed. The compiler will not notice a problem and so some differentiation might

be forgotten, leading to unindented behavior. The new department would fall into the default case or if there isn't one nothing would be executed in this case. The problem will only show during runtime. Following the Single Choice Principle, there would only be one instance of such a case distinction and this is the only one that knows all possible alternatives. Adding a new department requires only changes in one place, not many. [32]

Now that there exists a single point of choice, clients need to make use of it. A possible way is offered with polymorphism and dynamic binding. Polymorphism enables the clients to rely on abstractions, not knowing the concrete type they are working with at runtime. Dynamic binding ensures the method of the runtime type is called. In the following example, a Department interface is introduced to enable polymorphism. A method decides which implementation should get used and the client uses polymorphism and dynamic binding to avoid having a case differentiation. [32]

```
interface Deparment
{
    int getNumberOfEmployees();
}
...
//Single point of choice
Department getDepartment(EDepartment department)
{
    switch (department)
    {
        case HR:
            return new HumanRessources();
        case SALES:
            return new Sales();
        default:
            throw new IllegalArgumentException(
                "Department unknown");
    }
}
...
//Client that needs no case differentiation
getDepartment(department).getNumberOfEmployees();
...
```

The mapping from enum value to Department subtype can also be done in EDepartment, making a switch statement or such completely unnecessary.

```
enum EDepartment
{
    SALES
    {
        @Override
        Department getDepartment() {
            return new Sales();
        }
    },
    HR
    {
        @Override
        Department getDepartment() {
            return new HumanRessources();
        }
    };
    abstract Department getDepartment();
}
```

Relation to other principles

Don't Repeat Yourself (DRY): The Single Choice Principle prevents duplication of code that decides between alternatives, therefore it supports the Don't Repeat Yourself principle.

Violation detection strategy

Design best practices: AvoidRuntimeTypeIdentification, AvoidCaseDistinctionOverEnums

If the distinction is done over runtime types, it can be detected by looking for symptoms like getClass() calls and the usage of the instanceof operator. The usage of runtime type identification should not be necessary at all.

Differentiation over enum values is of course allowed to be used, but according to the Single Choice Principle only one module, so in the case of Java only one class, should contain it. If a client needs to decide something based on an enum value, it should ask the single point of choice to do it for it. In Java switch statements and if/else cascades can be used to discriminate between cases. The use of a single if statement is an exception that is allowed in our opinion because only a specific value is considered. If there are two or more cases we guess a violation might be present.

Sometimes int values or the like or strings are used as cases. In our opinion such distinctions over non-enum values cannot be recognized as violations because these types are used for all kinds of things. Flagging every usage of case distinctions over such types might lead to many false-positives. Enums on the other hand are designed to represent a finite list of alternatives.

Origin

The Single Choice Principle was introduced by Meyer [32].

4.10 Program to an Interface, not an Implementation (PINI)

Variants

-

Description

Program to an interface, not an implementation means that the static type of variables should never be a concrete implementation. Fields, parameters and local variables should have an interface or an abstract class as their static type. Following this advice brings two benefits. First, clients don't know about the objects they use. Second, clients don't know about the classes that define those objects. Clients are therefore only bound to the interfaces, respectively the abstract classes. Interfaces and abstract classes are said to be more stable than concrete implementations, which mitigates ripple effects if implementations are changed. The use of interfaces can also restrict the access of clients to the concrete implementation. An implementation might have more public methods than the interface it implements. Clients that rely on interfaces only have access to the methods defined in the interface. [11][48][49]

Relation to other principles

Open Closed Principle (OCP): Using abstractions supports the idea of closeness against modification because future behavior changes might already be covered by the used abstractions.

Dependency Inversion Principle (DIP): If abstractions are used as much as possible, high-level modules will use abstractions too, which is one of the goals of the Dependency Inversion Principle.

Violation detection strategy

Design best practices: UseInterfaceIfPossible, ProvideAbstractionForClass

According to the principle the types of fields, variables, parameters and return types should all be abstractions, not concrete classes. A type that is not an interface or abstract class is therefore a violation of the principle. [3]

If it is necessary for every class to have an abstraction, is an open discussion. We think adding abstractions for classes that are nonvolatile or which only have (and most likely will have) one implementation would not bring any benefit though it increases the

complexity of the system. To mitigate false positives in such cases, a type declaration should in our opinion only be flagged if a usable abstraction already exists for the concrete class.

To enable the former, every public class has to provide an abstraction. A class that does not inherit from an abstract class or implements an interface is therefore a violation of the principle [3]. This again goes against the opinion of many developers, but this rule can just be disabled by them.

Origin

"Program to an interface, not an implementation" originates from "Design Patterns: Elements of Reusable Object-Oriented Software" by Gamma et al. [11].

4.11 Favor Composition over Inheritance (FCOI)

Variants

- Composite Reuse Principle

Description

To reuse code in an object-oriented system two techniques can be used, inheritance and composition. Reuse by inheritance means, the superclass has functionality that a subclass also uses. Internals of the superclass can be visible to the subclass. Reuse by composition works by composing complex behavior through the use of other objects. The complex object delegates tasks to other objects. It can only use public methods, internals stay hidden. [11]

If inheritance is used, the subclass can modify the inherited behavior by overriding methods, which is an advantage. The disadvantages are the reason this principle exits. Because inheritance is defined at compile time, the behavior cannot be changed at runtime, which is possible with composition. The biggest drawback is the tight coupling between superclass and subclass. Subclasses have access to details of the parents implementation, which breaks the encapsulation of the parent. Changes to the superclass have a high chance of causing changes to its subclasses. Introducing a subclass can also require changes to the future parent if the parents implementation is not fully appropriate. [11][45]

Favoring composition over inheritance aids encapsulation and helps to keep classes small and focused. Especially class hierarchies are prevented from overgrowing. The drawbacks are more objects and more interrelationships between them. [11]

A common example for a misused inheritance is a Stack class that inherits from ArrayList:

```
class Stack<T> extends ArrayList<T>
{
    public void push(T value)
    {
        add(value);
    }
    public T pop()
    {
        return remove(size() - 1);
    }
}
```

To keep the example short the class is incomplete and lacks exceptions, but it works for valid usage. Implementation was fast and simple, other useful methods like size() are inherited from ArrayList and work out of the box. The problem is that Stack now really is a List and a client can use it as such. Nothing prevents a client from calling remove() directly on an element in the middle of the Stack or manipulating it in other ways. The way elements are stored cannot be changed easily in the future because this would break inheritance. Any client that is relying on Stack being a List would not compile anymore. An implementation based on composition would look like this:

```
class Stack<T>
{
    private List<T> stack = new ArrayList<>();

    public void push(T value)
    {
        stack.add(value);
    }
    public T pop()
    {
        return stack.remove(stack.size() - 1);
    }
    public long size()
    {
        return stack.size();
    }
}
```

The interface of Stack now only has the methods it should have. Direct access or manipulation of elements in the middle of the stack is not possible. The implementation of Stack can be changed without a client noticing because the decision to use ArrayList is

encapsulated. The drawbacks are forward methods like the size() method. Such forward methods are common when composition is used.

Relation to other principles

Liskov Substitution Principle (LSP): Favoring composition over inheritance prevents subtyping that is only done to achieve code reuse, but actually violates the Liskov Substitution Principle.

Violation detection strategy

Design best practices: UseCompositionNotInheritance, CheckUnusedSupertype, OnlyInheritFromAbstractClasses

Based on [3] we think a violation may exist if a subclass only uses public functionality of the superclass and does not override any method. This case might not be a proper is-a relationship, thus inheritance is unnecessary. The stack example above is such a case. In the first implementation variant, Stack only makes use of public methods of ArrayList, which is why composition can be used and should be favored. If a subclass overrides a method, it most likely will also be used as a subtype of the superclass. There can be false-positives if a subclass fulfills the criteria for a violation but a subtype relationship is still desired.

Client usage of a subclass can also be an indicator. If clients only use methods of the subclass, but not the superclass, there might be a problem in the hierarchy. The subclass probably does not have an is-a relationship with the superclass and should use composition instead. [3]

To mitigate the tight coupling and consequential ripple effects by changes, classes should only inherit from abstract classes. Those provide none or little implementation, making changes less likely. [11]

Origin

Favor composition over inheritance is one of the principles introduced by "Design Patterns: Elements of Reusable Object-Oriented Software" [11].

4.12 Common Reuse Principle (CRP)

Variants

-

Description

> "The classes in a component are reused together. If you reuse one of the classes in a component, you reuse them all." [29]

Component here is synonymous with package. If a class reuses a class from another package, the whole package has to be available. Therefore, the using class is also dependent on the whole package. Reused classes seldom are standalone, which reinforces the argument. Either multiple reusable classes are used by a client, which should all be in the same package according to the principle, or the reused class uses other non-public classes in the package. The negative interpretation of the principle says that a dependency on all classes of a package should be required. If the reuse of a class is possible without the usage of other classes in the package, they should not be in the package in the first place. Classes in the package should be inseparable. Depending on one class, but none of the others should not be possible. [21][29]

Relation to other principles

Common Closure Principle (CCP): The Common Reuse Principle and the Common Closure Principle both are package cohesion principles and can contradict each other. Classes that are used together might not change together and vice versa.

Violation detection strategy

Design best practices: CheckUnsuitableFunctionalityOfPackage

We think violations of the Common Reuse Principle can be detected by looking at the usage of a package by clients. If clients commonly only use a part of the public classes, the principle might be violated because the classes do not have to be used together and should be split up in more packages.

Origin

The Common Reuse Principle is one of the cohesion package principles by Martin published in [29].

4.13 Acyclic Dependencies Principle (ADP)

Variants

-

Description

> "Allow no cycles in the component dependency graph." [29]

Component here is synonymous with package. When a class in package A uses a class in package B, package A becomes dependent on package B. These usages form a directed graph. The principle now states that this graph should be acyclic. Having cycles in the dependency graph leads to problems. [29]

Direct cycles between two packages can be detected easily. The problem here is tight coupling. Because the packages are so tightly coupled, they could be one, but this violates the Common Closure Principle. Releasing the packages also becomes a problem. If a dependency is acyclic, the depended on package is standalone. Developers can work on it independently from other packages. When they release a new version, the dependent package can incorporate the changes. If the two packages are depending on each other, it is not clear were to start. Development has to be conducted on both at the same time and the package developers have to interact with each other. [21]

Since dependencies are transitive, bigger cycles can occur. In the example in Figure 4.2 the red line is the problem which creates the cycle. If the red dependency was not there, ErrorHandler is standalone, it depends on nothing. If the red dependency is added, ErrorHandler now depends on every other component because UserInterface does. To break the cycle two strategies can be used.
One is the Dependency Inversion Principle. ErrorHandler can provide an interface that the class in UserInterface inherits, which reverses the dependency. ErrorHandler can be released independently again because in the interface it specified what it needs and which implementation is used is only determined at runtime.
The second option is the introduction of a new component that both ErrorHandler and UserInterface depend on. ErrorHandler now also has a dependency, but there is no cycle that makes it depend on every other component transitively because of UserInterface. [29]

Relation to other principles

Stable Dependencies Principle (SDP) / Stable Abstractions Principle (SAP): All three principles are concerned with the coupling between packages and should therefore be applied together.

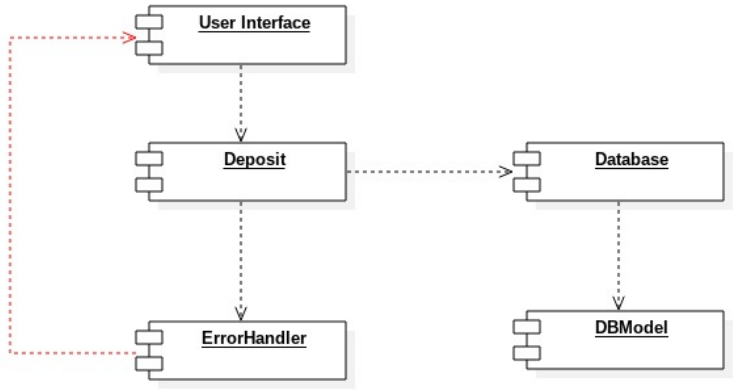

Figure 4.2: Dependency graph [75]

Violation detection strategy

Design best practices: AvoidPackageCycles

Violations are cycles in the dependency graph, so the whole dependency graph has to be built and checked for cycles.

Origin

The Acyclic Dependencies Principle is one of the coupling package principles by Martin published in [29].

4.14 Stable Dependencies Principle (SDP)

Variants

-

Description

> "Depend in the direction of stability." [29]

In a software system there are packages that are stable and some which are unstable. Stable packages are hard to change, unstable packages are easier to modify. The Stable Dependencies Principle decides the stability of a package based on its incoming and outgoing dependencies. [21][29]

A package that only has incoming dependencies, i.e., other packages depend on it, is as stable as possible. Figure 4.3 depicts such a case. First, changes to the stable package x

would create ripple effects in the many dependent packages. Second, because the stable package does not depend on any other package, nothing can force it to change. [21][29]

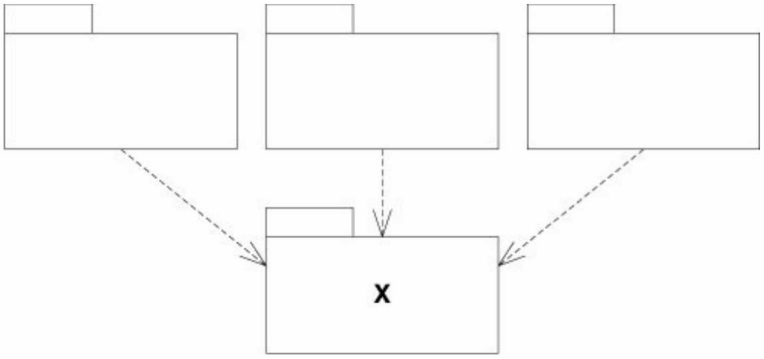

Figure 4.3: Stable package [29]

Figure 4.4 shows the most unstable package possible. It only has outgoing dependencies. It can be changed at any time because nothing depends on it. Furthermore, changes to the packages it depends on may require changes to it, too. To be clear, unstable is not bad per se. A fully stable system would be very hard to change. Unstable components enable easy changeability of systems. Therefore, some packages need to be flexible and unstable. [21][29]

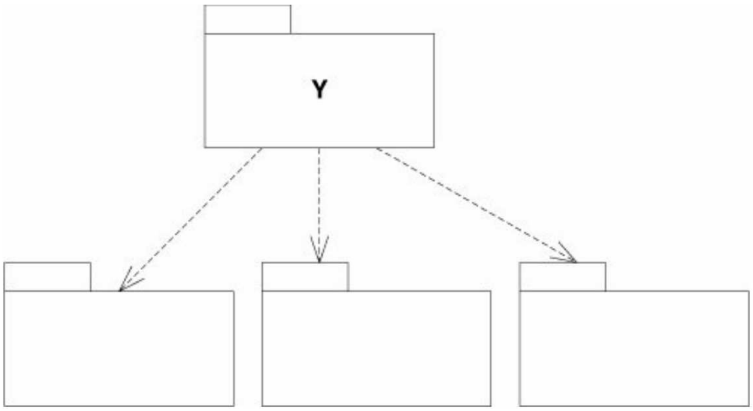

Figure 4.4: Unstable package [29]

However, the Stable Dependencies Principle says that packages should not depend on more unstable packages. Such a dependency would make the unstable package harder to change. The unstable package is designed to be changed, so it will most likely need to get changed in the future. These changes may now require changes to stable packages that depend on the unstable one. To prevent such problems, every dependency should be a conscious decision. [21][29]

Martin also defined a metric to measure the stability of a metric based on its dependencies. $I = \frac{Ce}{Ca+Ce}$, whereby I is the instability, Ca is the number of classes outside this package that depend on a class inside the package and Ce is the number of classes inside the package that depend on classes outside the package. Therefore an I = 0 has no outgoing dependencies and marks the most stable package possible. An I = 1, would only have outgoing dependencies, making it the most unstable package possible. The Stable Dependencies Principle in this terms says, if a package A depends on package B, I(A) should be higher than I(B), i.e., I(A) > I(B). [21][29]

Relation to other principles

Acyclic Dependencies Principle (ADP) and Stable Abstractions Principle (SAP): All three principles are concerned with the coupling between packages and should therefore be applied together.

Violation detection strategy

Design best practices: DependInTheDirectionOfStability

To find violations of the Stable Dependencies Principle, the instability metric has to be calculated for every package. Having the metric values at hand for each dependency of packages in the system, the metric values have to be compared. If a package A depends on a package B with a higher instability metric, i.e., I(A) < I(B), a violation has been found.

Origin

The Stable Dependencies Principle is one of the coupling package principles by Martin published in [29].

4.15 Stable Abstractions Principle (SAP)

Variants

-

Description

> "A component should be as abstract as it is stable." [29]

Component here is synonymous with package. Because of the Stable Dependencies Principle dependencies run in the direction of stability. The Stable Abstractions Principle adds that stable packages should be abstract and unstable packages concrete. Stable packages need to be abstract to keep up the extensibility. Unstable packages should be concrete because concrete code is easier to change. [21][29]

In a well designed system the following two statements [21] hold:

- More stable packages containing a higher number of abstract classes or interfaces should be heavily depended upon.
- Less stable packages containing a higher number of concrete classes should not be heavily depended upon.

Abstraction can be measured with a simple relation metric. $A = \frac{Na}{Nc}$, whereby A is the abstraction level, Na the number of abstract classes and interfaces and Nc the overall number of classes and interfaces in the package. A value of A = 1 means that there are no concrete classes. A value of A = 0 implies the lack of abstractions in the package. Values around A = 0.5 are to be avoided since these packages are neither abstract nor concrete. [21][29]

Because the principle reasons over stability and abstraction and there is also the instability metric, the relation between both values can be determined for a package. Figure 4.5 shows this relation. Packages should reside in proximity to the "Main Sequence". Especially the zones around (0,1) and (1,0) are worth striving for. Packages in these zones are either abstract and stable or concrete and instable. The other two corners of the diagram should be avoided. Packages in the "Zone of Pain" are stable but concrete. They are not extensible because of the lack of abstraction, but also hard to change because they are stable. There are valid exceptions because some concrete classes offer utility functionality that is not subjected to permanent change. Classes that represent database schemas also fall into this zone.
The "Zone of Uselessness" holds packages which are abstract and unstable. Unstable means the package does not have many dependents which an abstract package should have. [29]

Relation to other principles

Acyclic Dependencies Principle (ADP) and Stable Dependencies Principle (SDP): All three principles are concerned with the coupling between packages and should therefore be applied together.

Violation detection strategy

Design best practices: PackagesShouldBeAsAbstractAsStable

To find violations of the Stable Dependencies Principle, the instability metric and the abstraction metric have to be calculated for every package. With those the distance to the "Main Sequence" can be calculated: $D = |A + I - 1|$. [29]

As a human reviewer, the goal is to have packages as close as possible to the two end points of the "Main Sequence", i.e., (0,1) and (1,0). The evaluation of a deviation from those points and the "Main Sequence" can be done case by case. A tool only has the

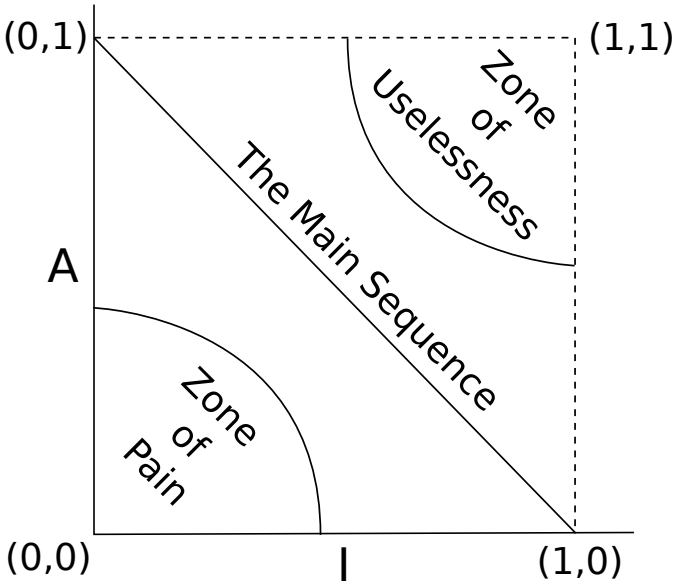

Figure 4.5: Stability abstraction relation [29]

calculated distance at hand, so it needs a threshold for violations. Since no value was proposed for the threshold it has to be determined. Figure 4.6 shows the values of the the distance function. We think a threshold of $D < 0.7$ seems to be suitable because it excludes the "Zone of Pain" and the "Zone of Uselessness" but does not restrict values in the desired corners which should keep the number of false-positives low.

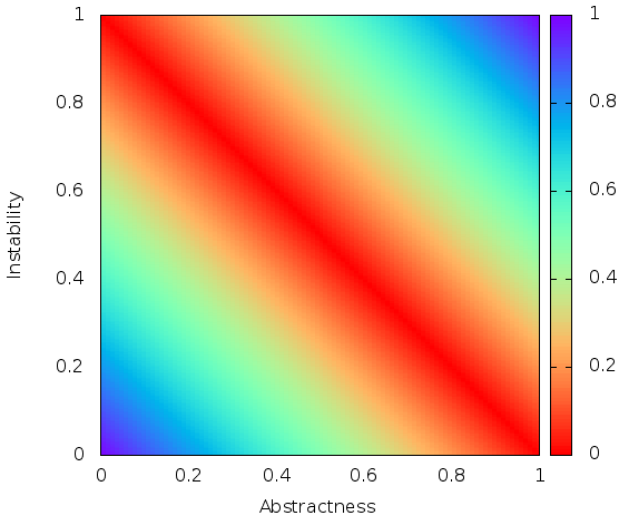

Figure 4.6: Distance function [55]

Origin

The Stable Abstractions Principle is one of the coupling package principles by Martin published in [29].

4.16 Option Operand Principle (OOP)

Variants

-

Description

> "The arguments of a routine should only include operands (no options)." [32]

An operand is a value that the function will operate upon. An option on the other hand is a mode of operation. Operands tend to stay the same but options are likely to be added or removed. The distinction of both is based on the possibility to find a default value for the argument. If a client hypothetically would not provide a value for an argument and it is possible to find a reasonable default value, the argument is an option. [32]

In the following code piece dividend and divisor are operands, there are no default values for them. The function cannot operate without them. roundingMethod is an option because one of the rounding methods can be chosen as the default.

```
public int divide(
    int dividend,
    int divisor,
    ERounding roundingMethod)
{
    double result = ((double) dividend) / divisor;
    if(roundingMethod == ERounding.ceiling)
    {
        return (int) Math.ceil(result);
    }
    else if (roundingMethod == ERounding.floor)
    {
        return (int) Math.floor(result);
    }
    else
    {
        return (int) Math.round(result);
    }
}
```

A violation of the Option Operand Principle, i.e., a method that has an option as a parameter, can be resolved in two ways. Either it is possible to split the method into multiple methods, each for one possible value of the option. This approach is especially useful for boolean options because just two methods are needed. Or, make the option a field of the object and make it modifiable with an extra method. In the setter method the option becomes an operand because the setter operates upon it. [32]

For the example from above one could introduce three different methods divideCeil(int dividend, int divisor), divideFloor(int dividend, int divisor) and divideRound(int dividend, int divisor). The option is now in the name of the method. The second way for refactoring creates divide(int dividend, int divisor), which only has the two operands and roundingMethods becomes a field set to a default value.

```
private ERounding roundingMethod = ERounding.ceiling;

public void setRoundingMethod(ERounding roundingMethod) {
    this.roundingMethod = roundingMethod;
}

public int divide(int dividend, int divisor)
{
    double result = ((double) dividend) / divisor;
    if(roundingMethod == ERounding.ceiling)
    {
        return (int) Math.ceil(result);
    }
    else if (roundingMethod == ERounding.floor)
    {
        return (int) Math.floor(result);
    }
    else
    {
        return (int) Math.round(result);
    }
}
```

Adhering to the principle has multiple benefits. As already mentioned, options are likely to be added or removed which would be a change to the methods signature, breaking existing client code or would require the introduction of an overloaded method. If options are fields, not parameters, changes to them will not have such a huge impact on clients or not impact them at all. A new option can have the current solution as the default value, keeping everything the same for existing clients and new clients can configure the new option via a new setter method.

Having defaults also means that options only have to be set if they diverge from the norm. For the most common cases the default values might be right, making client code simpler. Novices can use classes easier without thinking about advanced options but sophisticated users can change options as they please. Last but not least, options as fields are persistent in the object, therefore multiple consecutive method calls can use the same options. [32]

Relation to other principles

-

Violation detection strategy

Design best practices: AvoidOptionParameters

We think detecting violations of the Option Operand Principle is limited because static code analyzers can in general not distinguish between operands and options. To detect an option, it would have to be an argument that is used in a condition and a reasonable default value must exist for it. If a reasonable default value exists, is not decidable for the analyzer, leaving the usage in a condition as the only thing an analyzer can check. Now the problem is that operands are also used in conditions since the behavior of the method can depend on the value of an operand, which nonetheless does not make it an option.

Only if a boolean parameter is used in a condition, it is in our opinion very likely that it is an option because it splits the code into two paths and decides the mode of operation. One of the paths can be selected as the default, so there always exists a reasonable default value.

Origin

The Option Operand Principle is described in [32] by Meyer.

CHAPTER 5

Undetectable Design Principles

In this chapter we describe those design principles whose violations cannot be detected with a static code analyzer. These principles also fall under the selection criteria stated in the previous chapter. We believe violations of the following design principles cannot be detected by a static code analyzer:

- Liskov Substitution Principle (LSP)
- Dependency Inversion Principle (DIP)
- Single Level of Abstraction (SLA)
- Common Closure Principle (CCP)
- Integration Operation Segregation Principle (IOSP)
- Command Query Separation (CQS)
- Encapsulate the Concept that Varies (ECV)

5.1 Liskov Substitution Principle (LSP)

Variants

-

Description

Liskov originally defined the principle as follows:

> "If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T." [24]

Martin put it in easier terms:

> "Subtypes must be substitutable for their base types." [29]

According to the principle, subtypes should behave as the supertype is expected to. A method m can declare the supertype T as the type of a parameter p, but at runtime a subtype S might be passed to m. The developer that codes method m only knows about the behavior that T promises. If S has a different behavior, then m will malfunction because the developer did not plan for it. Promised behavior in this case are preconditions, postconditions, invariants and history constraints. Preconditions are conditions that the caller has to ensure. Breaking them might lead to unexpected behavior. Postconditions are ensured by the called object, they are promises that the caller can expect to hold after the call. Invariants hold true all the time (except over defined intervals during method execution). History constraints (server- and client-controlled) restrict the way an objects state may change. For example a counter variable may only be increased one by one (server-controller) and some methods may only be called in a specific order (client-controlled). Subtypes may only weaken preconditions and strengthen postconditions and invariants. Server-controlled history constraints can be limited further in subtypes, client-controlled history constraints can be extended. [24][25][29]

An example for a violation of the Liskov Substitution Principle is a Square that is a subtype of Rectangle (adapted from [29]). Mathematically this relation would be correct because a square is a rectangle. But, the behavior of these two objects in code is different and this principle is about behavior. The Rectangle class already provides behavior that is common for rectangles. It has a height and a width and provides a function to calculate the area.

```
public class Rectangle
{
    private double width;
    private double height;

    public void setWidth(double width)
    {
        this.width = width;
    }
```

```
    public void setHeight(double height)
    {
        this.height = height;
    }

    public double area()
    {
        return width * height;
    }
}
```

To make the Square class work as a subclass of Rectangle the setter methods have to be overridden since width and height of a square are always the same. Each setter therefore has to set both attributes.

```
public class Square extends Rectangle
{
    @Override
    public void setWidth(double value) {
        super.setWidth(value);
        super.setHeight(value);
    }

    @Override
    public void setHeight(double value) {
        super.setWidth(value);
        super.setHeight(value);
    }
}
```

The Square class is now valid for itself. A caller that works with a Square reference will not experience any problems but this class still violates the Liskov Substitution Principle. The problem is that it behaves differently than the Rectangle class, more precisely it weakened the postconditions of the setter methods. The original setWidth() of Rectangle ensures that width is set to the new value but height stays the same. The overridden method in Square does not ensure the latter. A caller might have the following code:

```
void g(Rectangle r)
{
    r.setWidth(5);
    r.setHeight(4);
    if(r.area() != 20)
        throw new Exception("Bad area!");
}
```

The developer of g() worked with the pre- and postconditions of Rectangle. Therefore, the exception should never be thrown but if a Square object is passed to g() the exception will be thrown. This function is an example in which Square is not substitutable for Rectangle, so Square cannot be a subtype of Rectangle.

Relation to other principles

-

Violation detection strategy

We think violations of the Liskov Substitution Principle are not detectable with a static code analyzer in Java because preconditions, postconditions invariants and history constraints are not part of the language. There is no way to programatically state these conditions and let them be checked. Only preconditions are sometimes checked in the form of guard clauses, e.g., checks for null values, but those are simple if statements that are part of the normal program code. These checks are furthermore often not equivalent to the preconditions. If at all, preconditions, postconditions and invariants are stated in comments which clearly cannot be processed with a tool. Therefore, it cannot be checked if subtypes weaken or strengthen the conditions, which is needed to detect violations.

Origin

The Liskov Substitution Principle is named after Barbara Liskov who described it in [24]. It is also one of the SOLID principles by Martin published in [29].

5.2 Dependency Inversion Principle (DIP)

Variants

-

Description

> "A. High-level modules should not depend on low-level modules. Both should depend on abstractions.

> B. Abstractions should not depend upon details. Details should depend upon abstractions." [29]

A common way to write software is to implement low-level modules that can be reused and reusing them in high-level modules. Often the low-level modules are hardwired into the high-level modules. This approach prevents the reuse of high-level modules because the low-level modules cannot be exchanged. To make the low-level modules exchangeable, high-level modules should depend on abstractions, not concrete implementations. [29]

A naive interpretation of the principle would be that one just has to use abstractions everywhere. But, this leaves out an important part of the Dependency Inversion Principle which is ownership. If abstractions are used but they are owned by the implementations of the abstractions, changes in the implementations can break through the abstractions. The principle tells us to invert the ownership, so the client has control over the abstractions. [29]

The benefit can be shown with a simple example, taken from [29]. In the first version there is a Button class that can turn a Lamp on or off. The Button class has a poll method that somehow is executed. When receiving the poll message Button decides whether the Lamp has to be turned on or off.

```
public class Button
{
    private Lamp lamp;
    public void poll()
    {
        if(/*some condition*/)
        {
            lamp.turnOn();
        }
        else
        {
            lamp.turnOff;
        }
    }
}
```

In this implementation Button directly depends upon Lamp and can be therefore only be used to control Lamps, but nothing else. Button is not reusable to control other devices. To fix this, an abstraction has to be introduced. A button can be used to switch devices on or off, so SwitchableDevice is a good candidate for an abstraction.

```
public interface SwitchableDevice
{
    void turnOn();
    void turnOff();
}

public class Button
{
    private SwitchableDevice device;
    public void poll()
    {
        if(/*some condition*/)
        {
            device.turnOn();
        }
        else
        {
            device.turnOff;
        }
    }
}
```

Now Lamp has to implement the SwitchableDevice interface, so Button only has to depend upon SwitchableDevice. It is now reusable with every implementation of a SwitchableDevice. It is important to note that the ownership of the interface has to be with Button, not Lamp or with none of both. If Lamp had the control over SwitchableDevice, changes to the Lamp can influence Button, which should not happen because Button is the high-level module here. In the best case SwitchableDevice is independent of both. This enables Buttons to control any SwitchableDevice but also enables SwitchableDevices to be controlled by anything that can work with SwitchableDevices.

Relation to other principles

Program to an interface, not an implementation (PINI): If a lot of abstractions are used, it becomes easier to apply the Dependency Inversion Principle because high-level modules already depend upon abstractions. Though, only programming against abstractions is not enough since the Dependency Inversion Principle also specifies the ownership of the abstractions.

Open Closed Principle (OCP): Making high-level modules depend upon abstractions also makes them open for extension but closed for modification, which is the proposition of the Open Closed Principle.

Violation detection strategy

Violations of the Dependency Inversion Principle are in our opinion not detectable by a static code analyzer. Looking for the usage of concrete types alone would be the same naive approach already described above. Since nonvolatile classes may be used directly, an abstraction is not needed and this would create many false-positives. The second problem is the inversion part. To fully cover the principle, the ownership of abstractions would need to be determined, which a tool cannot do. The placement of the classes in the packages would be the best indicator for ownership, but is not reliable enough. Abstractions and implementations could be in separate packages, but still belong together. Last, but not least, used libraries do provide their abstractions themselves. The ownership of the abstraction cannot be with the client of the library but this would still be a violation since the ownership resides on the wrong side, namely with the provider of the implementations. Since the usage of libraries is also common practice there would be too many false-positives.

Origin

The Dependency Inversion Principle is one of the SOLID principles by Martin published in [29].

5.3 Single Level of Abstraction (SLA)

Variants

-

Description

The Single Level of Abstraction principle says that statements in a method should all act on the same level of abstraction. High level statements should not be mixed with low level statements. [28]

```
File writeUserFromDatabaseToFile(String id, String filePath)
{
    User user = getUserFromDatabase(id);

    File file = new File(filePath);
    try {
        BufferedWriter writer = new BufferedWriter(new FileWriter(file));
        writer.write(user.getName());
        writer.newLine();
        writer.write(user.getAddress());
        writer.close();
    } catch (IOException e) {
        ...
    }

    return file;
}
```

The getUserFromDatabase() method is on a higher level than the writing to the file. The lines to write the user are a separate block that should be extracted to a method that is on the same level. getUserFromDatabase() only states what is done but not how. The writing on the other hand is very explicit. Extracting it to writeUserToFile() hides the how and emphasizes the what.

```
File writeUserFromDatabaseToFile(String id, String filePath)
{
    User user = getUserFromDatabase(id);
    return writeUserToFile(filePath, user);
}
```

Relation to other principles

Violation detection strategy

We believe detecting violations of this principle is not possible with a static code analyzer because it cannot decide at which abstraction level a particular class or function sits. A human sees a meaning behind the used abstractions and has a mind map to classify abstractions. This information is not accessible for a tool. The examples mentioned in [71] are useful for developers but implemented in a static code analyzer they would create an unjustifiable amount of false-positives. The first example are loop bodies containing multiple statements that could be extracted to a method. A human could tell if the body is indeed working on another abstraction level as the surrounding code but a tool cannot. The second example are code blocks in a function, formatted with an empty line and often a comment before each block. Those surely are candidates to be extracted in separate

methods where the comment becomes the method name. However, the existence of such blocks does not mean that the code is not on the same abstraction level. Each block is on a higher level but the contained code might be on the same. Therefore, reporting violations based on these signs would not be helpful.

Origin

The Single Level of Abstraction principle is a rule of Martin published in [28].

5.4 Common Closure Principle (CCP)

Variants

-

Description

> "The classes in a component should be closed together against the same kinds of changes. A change that affects a component affects all the classes in that component and no other components." [29]

Component here is synonymous with package. The Single Responsibility Principle states for classes that they should only have one reason to change and the Common Closure Principle states the same for packages. Classes that are anticipated to be changed together if a requirement changes or is added should reside in the same package. If a change only affects one package, other packages stay untouched and the work required to release and redeploy a new version is reduced. [21][29]

Relation to other principles

Common Reuse Principle (CRP): The Common Reuse Principle and the Common Closure Principle both are package cohesion principles and can contradict each other. Classes that are used together might not change together and vice versa.

Violation detection strategy

We think violations of the Common Closure Principle cannot be detected with a static code analyzer. A violation would exist if classes that might change in the future are not in the same package. This kind of foresight is not possible at all for a tool and even developers can only guess the future.

Origin

The Common Closure Principle is one of the cohesion package principles by Martin published in [29].

5.5 Integration Operation Segregation Principle (IOSP)

Variants

-

Description

According to the Integration Operation Segregation Principle there are two kinds of methods: integrations and operations. Integrations do not contain any logic, logic is only allowed to occur in operational methods. Integration methods may call other integrations and operations but only one kind, otherwise integration and logic would be mixed. Operation methods are the leafs of the dependency tree as depicted in Figure 5.1. As can also be seen in the figure, operation methods do not depend on each other, so they depend on nothing. This contradicts the common schema in which method calls are hardwired in other operational methods. This leads to functional dependencies. The dependencies have to be injected into methods with logic, which makes testing cumbersome because fake objects are needed to isolate a test case. Following the Integration Operation Segregation Principle leads to integration methods that are less error-prone since they only call some other methods without any control statements. The produced operational methods do not have dependencies and can therefore be tested autonomously. [70][68]

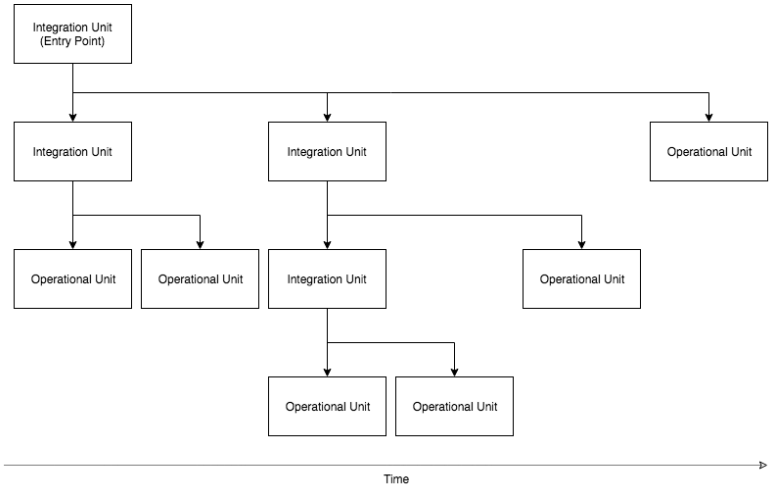

Figure 5.1: Integration and operational methods [70]

In the following example getUsersWithBirthdayToday() mixes the logic of getting the users, with the integration of the later method calls. This makes the code harder to understand because it mixes levels of abstraction. filterByBirthdayToday() also has a violation because it is an operational method, but knows about another operational method, namely today(). It depends on today() and in a test today cannot be set arbitrarily. today() could be injected in some way into filterByBirthdayToday(), making

a mock possible, but the easiest way is to just provide the data since this removes the functional dependency.

```
Set<User> getUsersWithBirthdayToday()
{
    DataTable rawData = new SQL("SELECT * FROM users;")
        .execute();
    Set<User> users = convertToUsers(rawData);
    Set<User> usersWithBirthday = filterByBirthdayToday(users);
    return usersWithBirthday;
}

private Set<User> filterByBirthdayToday(Set<User> users) {
    Date today = today();
    return users.stream()
        .filter(user -> user.getBirthday().equals(today))
        .collect(Collectors.toSet());
}
```

Transforming the code according to the Integration Operation Segregation Principle requires two things. First the SQL select has to be moved to an operational method. Second, the call to today() has to be moved out of filterByBirthdayToday().

```
Set<User> getUsersWithBirthdayToday()
{
    DataTable rawData = getAllUsers();
    Set<User> users = convertToUsers(rawData);
    Date today = today();
    Set<User> usersWithBirthday = filterByDate(users, today);
    return usersWithBirthday;
}

private Set<User> filterByDate(Set<User> users, Date date) {
    return users.stream()
        .filter(user -> user.getBirthday().equals(date))
        .collect(Collectors.toSet());
}
```

getUsersWithBirthdayToday() now only integrates. It can easily be checked with a review, because it is easy to read since all the details are hidden. A test case only has to check if all necessary steps are executed but can ignore details. filterByDate() can be tested independently.

Relation to other principles

-

Violation detection strategy

Detecting violations of this principle with a static code analyzer is in our opinion not possible. The analyzer would have to be able to distinguish between integration methods and operational methods, but as can be seen in the example above both kinds look the same to an analyzer. The kind of a method is based on semantics so only a human can decide it with confidence.

Additionally, the author of the principle acknowledges the need for exceptions of it [62]. According to him, sparse use of control statements like an if or the use of for-each loops is permitted in integration methods if it aids the readability, but does not lead to details leaking into the integration. Such exceptions cannot be built into a static code analyzer that needs clear rules to work properly.

Origin

The author of the Integration Operation Segregation Principle is Ralf Westphal. It was first mentioned by him in [68].

5.6 Command Query Separation (CQS)

Variants

-

Description

> "Functions should not produce abstract side effects." [32]

According to the Command Query Separation principle, methods should be either a command or a query, but not both at the same time. Functions, as in the quote, are queries. Queries return a value, but do not change the observable state of an object, i.e., they do not have abstract side effects. This means they can be called multiple times in a row and will always return the same result. In other words, the question does not change the answer. Commands change the state but do not return a value. [57][32]

Henney [16] brings the (shortened) Iterator interface of Java as an example:

```
public interface Iterator<E>
{
    boolean hasNext();
    E next();
    void remove();
}
```

The method hasNext() is a query. It will not change the state of the iterator, i.e., it can be called multiple times and will always return the same element. remove() is a command. Indicated by the return type void, it does not return anything, but removing an element is changing the state. next() is a problematic mix of query and command. It does return the element the iterator is currently pointing at but also has the side effect of advancing the pointer. Calling next() multiple times in a row will return another element each time because of the side effect. [16]

Having side effects in queries makes reasoning about programs harder. A query following the definition of the principle keeps referential transparency sound. Referential transparency means, an expression can be replaced by its value without changing the behavior of the program. In the following example the method n() has a side effect. A call to a newly created object will always return 1 for n(), but 1 and n() are not interchangeable because multiple calls to n() will change the result. Especially n() * 2 would be 2, but n() + n() would be 3. While reading program code such side effects can easily be overlooked. [32]

```
class Number
{
    private int value = 0;
    public int n()
    {
        value++;
        return value;
    }
}

Number number = new Number();
number.n() // == 1
number.n() // == 2
```

It appears that keeping queries completely side effect free makes code easier to read and use. Queries can be used in any order and multiple times in succession. Commands often have temporal constraints, so they need to be called in a specific order. On the other hand, having only queries without any side effects would be too restrictive because there are legitimate side effects. That is the reason for differentiating between concrete

side effects and abstract side effects. Concrete side effects, as defined by Meyer, are assignments to attributes and calls of commands. Abstract side effects are concrete side effects that change the publicly observable state. Side effects that only change the internal state do not change the outcome of public queries, meaning that those side effects are not observable for a client and thus not forbidden. Only side effects that interfere with public queries are a problem. Caching is an example for such useful side effects.

```
class Users
{
    private int cachedId = -1;
    private User userCache;
    public User getUser(int id)
    {
        if(cachedId != id)
        {
            cachedId = id;
            userCache = getUserFromDB(id);
        }
        return userCache;
    }
}
```

getUser() is a query, but it also changes the internal state because it updates cachedId and userCache. However, these side effects are not abstract because for a client the state change is transparent. Multiple consecutive calls to getUser() will return the same result, but the performance of the method is increased since roundtrips to the database are prevented. Fowler [57] also explicitly states that he likes to adhere to the Command Query Separation principle but is ready to break it if it aids the program code. He brings the pop() method of a stack as an example, that both returns (query) the latest element on the stack, but at the same time removes it (command). Splitting such a method in two methods can be more cumbersome than breaking the principle intentionally.

Relation to other principles

-

Violation detection strategy

We think the meat of the principle that queries should not produce abstract side effects cannot be checked with a static code analyzer. In theory a query could be detected because it is a method with a return type that is not void. The problem is the detection of abstract side effects. To distinguish between concrete side effects (that are detectable) and abstract ones, the analyzer would have to find out if the state change in the query

does affect any query in the class. Taking the Users class from above as an example, the analyzer could recognize getUser() as a query and updates to chachedId and userCache, so it detected concrete side effects. Now it has to check if the side effects are abstract side effects. It can now recognize that the used attributes affect the outcome of getUser(), which is a public query. Tough, raising a violation here would be a false-positive because the cache mechanism is not an abstract side effect, but the analyzer cannot reliably determine how the internal state influences the outcome of getUser().

The other aspect of the principle, namely that commands are not allowed to return something, is also not detectable in our opinion. A command could be detected by looking for concrete side effects, which are detectable if other commands are assumed to have return type void. So if a method changes any attribute or calls a method that does have void as return type, it is detected as a command. If the method itself has different return type as void, a violation would have been found. The problem is the same as before. Queries that do not have non-abstract side effects look like commands with a return value to a static code analyzer. Because it cannot distinguish between concrete and abstract side effects a detection is not possible.

Origin

The Command Query Separation principle is originally from Meyer[32].

5.7 Encapsulate the Concept that Varies (ECV)

Variants

-

Description

The advice of this principle to encapsulate the concept that varies works for two different aspects of software development. The first is code that needs to change during maintenance. If some code is expected to be changed in the future, it should be encapsulated to keep the needed changes as local as possible. With proper encapsulation the change might be done in one place only.
The second aspect is code that needs to change during runtime. Using abstractions, the code to execute can be chosen at runtime by using a different implementation. The abstraction encapsulates the concept that varies, whereby the variations are hidden from the user. New variants can be added by creating new implementations of the abstraction. Existing code does not have to be changed. [11][40]

Relation to other principles

Single Responsibility Principle (SRP): A responsibility can be seen as a concept that varies over time, so both the Single Responsibility Principle and the Encapsulate

the Concept that Varies principle recommend to encapsulate this concept.

Open Closed Principle (OCP): To apply the Open Closed Principle, abstractions have to be used, that encapsulate the varying behavior of the implementations. Encapsulate the Concept that Varies has the same goal.

Single Choice Principle (SCP): The Single Choice Principle encapsulates the decision between multiple alternatives. Because such lists of alternatives tend to change, the Encapsulate the Concept that Varies principle would also suggest to encapsulate the decision because it is a varying concept.

Don't Repeat Yourself (DRY): If the concept that varies is encapsulated, it cannot be repeated over and over in the source code, which prevents duplication.

Violation detection strategy

Detecting violations of the principle is in our opinion not possible for neither of the two aspects. A static code analyzer cannot presume which code might need changes in the future. Developers can have guesses based on experience and planning but an automatic detection is impossible. For the second aspect, the tool would need to find scattered code pieces that are variants of the same concept. For example, there would be case distinctions over authentication methods in multiple places, i.e., if or switch statements that decide which method to use. This would be a violation because authentication methods are prone to change and should be encapsulated in one place. To flag this as a violation, the static code analyzer would need to make sense of the case distinction, which it cannot do.

Origin

The principle, to encapsulate the concept that varies is described in "Design Patterns: Elements of Reusable Object-Oriented Software" [11].

CHAPTER 6

Tool Implementation

6.1 Selection of base-tool

Because integration with SonarQube is a requirement for the tool it can only be implemented in limited ways. To keep the implementation as simple as possible it should use an already existing SonarQube plugin that provides means to traverse Java source code. Plugins that can be used are SonarJava[1] (the official Java plugin for SonarQube), SpotBugs[2], PMD[3] and Checkstyle[4]. All this options provide the possibility to add custom rules. The rules for all options are written using the visitor pattern, i.e., we fill in callbacks for nodes in the syntax tree, check for a condition and maybe report a problem.

Checkstyle does fulfill the requirements from above, i.e., we can write custom rules filling in a visitor pattern, but it comes with limitations that render it useless for the detection of violations of design principles. Checkstyle does not provide information about the full inheritance hierarchy. This means, it is for example not possible to check if some class inherited from another. [56]

SpotBugs requires the SonarJava plugin but does not provide a simpler way to write custom rules than the SonarJava plugin itself. To minimize the requirements to run the tool SpotBugs is eliminated as an option. [59]

PMD also is not simpler to use than SonarJava. Furthermore, custom rules for SonarQube can be packed in a standalone JAR-file, whereas the custom rules for PMD need to be packed with the PMD plugin for SonarQube. [65][60][72]

To summarize, SonarJava is picked because it needs the least dependencies, the new custom rules can be distributed as a single JAR-file that only needs to be put into the

[1] https://www.sonarsource.com/products/codeanalyzers/sonarjava.html
[2] https://github.com/spotbugs/sonar-findbugs
[3] https://github.com/jensgerdes/sonar-pmd
[4] https://github.com/checkstyle/sonar-checkstyle

plugins directory of SonarQube and it is the official plugin for Java code analyzing in SonarQube. [72]

6.2 Implementation details

6.2.1 Overview

The tool is implemented as a plugin for SonarJava. An example project is provided by SonarJava which already has the build process configured via Maven[5]. The resulting jar archive has to be placed in the plugin folder of the SonarQube instance. SonarQube automatically detects the plugin.

For writing rules the class IssuableSubscriptionVisitor is the starting point. It provides the means to visit nodes of the syntax tree and to report issues. Typically one would write an automatic test for a rule. SonarJava provides helper classes to run rules against self written Java files that are annotated with the comments "// Noncompliant" and "// Compliant". The helper classes check if all expected issues were reported.

To represent the method of detection by using design best practices, an interface BestPractice was created. For each best practice an instance of BestPractice was written which handles the detection of violations. Because multiple design principles can use the same best practice the class DesignPrincipleRule was introduced which also extends IssuableSubscriptionVisitor. For each best practice that a design principle uses there exists an instance of DesignPrincipleRule. The following code pieces show a best practice and a design principle rule.

[5] https://docs.sonarqube.org/display/PLUG/Writing+Custom+Java+Rules+101

```
public class AvoidPublicFields implements BestPractice {
    @Override
    public List<Tree.Kind> nodesToVisit() {
        return ImmutableList.of(Tree.Kind.CLASS);
    }

    @Override
    public Stream<? extends Tree> checkViolation(
        JavaFileScannerContext context, Tree toCheck) {
        return Utils.getFieldsOfClass((ClassTree) toCheck)
            .filter(field -> field.symbol().isPublic())
            .filter(field ->
                !(field.symbol().isFinal()
                && field.symbol().isStatic()))
            .map(field ->
                new Violation(field, "Public field used."));
    }
}

@Rule(
    key = "IHAvoidPublicFieldCheck",
    name = "Information Hiding Principle violation. " +
        "Public fields should not be used.",
    priority = Priority.INFO,
    tags = {"design"})
public class AvoidPublicFieldsCheck extends DesignPrincipleRule {

    @Override
    public BestPractice bestPractice() {
        return new AvoidPublicFields();
    }
}
```

The best practice specifies to visit only classes, so checkViolation() will only be called for classes and the parameter toCheck is always instance of ClassTree. checkViolation() searches for public fields and returns all found fields, so DesignPrincpleRule can report issues for them, with the message specified in AvoidPublicFieldsCheck.

How rules can be written in general is sufficiently good documented, but how the syntax tree is composed can only be guessed since there is little documentation about it. The most help are the existing rules for SonarJava available in their Github repository[6]. Another way to find the needed part of the syntax tree is writing examples and looking

[6]https://github.com/SonarSource/sonar-java

at the tree in the debugger. After some time we got the hang of it because often the same methods are used.

Next up are implementation details for selected design principles.

6.2.2 Keep It Simple Stupid (KISS)

Because the acceptable length for methods is a subjective value we implemented it as a rule parameter with a default value of 200 lines. Cyclomatic complexity is already provided by the SonarJava API and is only checked against the maximum value which can also be configured.

6.2.3 Don't Repeat Yourself (DRY)

The detection of duplicate code is a separate feature of SonarQube so we did not implement it again for the tool.

6.2.4 Single Choice Principle (SCP)

We implemented AvoidCaseDistinctionOverEnums in a way that every case distinction is flagged as a violation even if there is only one in the whole project. The way SonarJava analyzes files makes this the easiest implementation, but it also makes sense from the point of view that even a single case distinction should be avoided and the decision moved into the enum itself. Case distinctions are switch expressions or if/else cascades. We do not count single if expression as a problem since it only targets a single case, which is not as problematic as a distinction over all or most values of an enum.

6.2.5 Program to an Interface, not an Implementation (PINI)

For the best practice UseInterfaceIfPossible we filtered "usable abstractions" by a blacklist because, e.g., Serializable or Comparable are abstractions, but mostly not used as static types for parameters or return values.

6.2.6 Favor Composition over Inheritance (FCOI)

We implemented CheckUnusedSupertype per client, i.e., if a single client only uses methods of the declared subtype, but none of the supertype, the subtype relation is marked as a possible violation. Other clients that do use supertype methods do not prevent the violation.

6.2.7 Package principles (CRP, ADP, SDP, SAP)

As it turned out, the way SonarJava analyzes files is on a file per file basis. This means it is only possible to get information about other types declared in a class, but it is not possible to get the usages of a method for example, i.e., places were it is called. This kind of information is needed to detect violations of package principles. Therefore, we had

to implement a ProjectSensor which has access to all Java files of the analyzed project but not the syntax trees that SonarJava usually provides. So we used another library, namely JavaParser[7], to get the needed information. If this problem would have shown up earlier in the implementation process we probably could have implemented the whole plugin with JavaParser but the package principles unfortunately were the last ones. The decision to use SonarJava as the base tool was still the right choice because the other options operate in the same manner, i.e., on a file per file basis and SonarJava is always available on SonarQube.

[7] https://javaparser.org/

CHAPTER 7

Evaluation

7.1 Overview

During the development of the tool we used small self written examples to test the functionality. This process is also described in Chapter 6.2.1. Those tests prove the ability of the tool to find intended violations of design principles but to assess its usefulness for real projects it has to be evaluated against real projects. Therefore, we selected two open-source projects based on their popularity, constant development in the last years and size. We searched GitHub[1] for Java projects sorted by the number of stars which are an indicator for popularity. We determined if a project was constantly developed over the last years by looking at its commit history. In terms of size we were looking for projects that span multiple packages with multiple classes in each, so we might find violations of package principles. The upper limit regarding size was the number of violations found by the tool because the evaluation has to be done manually. According to these criteria we chose MPAndroidChart[2] and fastjson[3]. MPAndroidChart contains 17,000 lines of code in 159 files which hold 183 classes, interfaces and enums. fastjson contains 40,000 lines of code in 176 files which hold 227 classes, interfaces and enums.

With the way we evaluated the tool we were only able to recognize false-positives but not false-negatives. To minimize false-negatives we already tried to cover, as well as possible for us, all aspects of the design principles with design best practices. The best practices we researched and came up with are additionally specified in a way that they only find candidates, which have to be reviewed by a developer. This means, they are designed to rather produce false-positives than false-negatives. Furthermore, since design principles are only heuristics, a complete coverage of them with a tool is impossible in our opinion. Because of these reasons we decided to concentrate our evaluation on

[1] https://github.com/
[2] https://github.com/PhilJay/MPAndroidChart
[3] https://github.com/alibaba/fastjson

false-positives and leave the evaluation of false-negatives to future research endeavors. Finding false-negatives for example does not have much overhead if the tool is actively used in a project and reviews are conducted, which can reveal design flaws that the tool did not report.

Running our tool against the two projects we found the violations depicted in Table 7.1 on page 87 and Table 7.2 on page 88.

The majority of violations in both projects is caused by a lack of documentation (DRY), the usage of concrete types (OCP, PINI) and the usage of public and protected fields (IH). The high amount of violations of AvoidRuntimeTypeIdentification in fastjson is due to its nature of being a JSON serializer/deserializer. The library has to accept objects of type Object and gather the needed type information itself.
The only best practices for which the tool could not find any violations are those of the principle Information Expert. All other best practices were violated at least once in one of the two projects.

7.2 Design principles in detail

7.2.1 Don't Repeat Yourself (DRY)

Detecting missing Javadoc works reliably and we think it gives a good impression if a project is trying to provide Javadoc or not. fastjson provides very little Javadoc which can indicate a conscious decision against it. MPAndroidChart however only lacks Javadoc on some parts of the code which implies a problem with the consistency of the documentation. The tool does help here to find places were Javadoc is missing.
We have to emphasize that only missing Javadoc is detected. Javadoc that only consists of useless information in regards to implementation, for example a Copyright notice, is treated as existing Javadoc which distorts the results.

The detection of too many inline comments delivers mixed results. On the one hand problematic comments are pointed out which in our opinion can be deleted after adapting the code. Commented out code could be deleted because it should also exist in the version control system. Comments which describe code blocks could be deleted after extracting the block into a method with a expressive name. Comments that describe every line could be deleted after renaming variables and also extracting code into methods to make the intention of the code clearer. Comments that are used to provide the reader the meaning of an integer, for example in a switch clause, could be deleted after the integer is refactored to an enum.
On the other hand we think many comments marked as violation are meaningful and needed. This means the tool delivers many false-positives for AvoidMassiveInlineComments. Users of the tool might have to change the allowed amount of comments to their needs but we still think this is a useful rule because developers have to think twice over the comments and can always dismiss the violation in SonarQube later.

As already stated before SonarQube has a built in code duplication detection which assesses fastjson with 14.7% and MPAndroidChart with 4.0% duplication. In the web user interface of SonarQube users are able to see which lines of code are duplicated and where.

7.2.2 Open Closed Principle (OCP)

The detection of public, public static and protected fields, the detection of runtime type identification and the detection of mutable class members that are returned by methods works reliably. Projects like fastjson that have to rely on runtime type identification can always turn off AvoidRuntimeTypeIdentification in SonarQube. We think these rules are useful.

Detecting concrete types as parameter or return type turned out to be error-prone. Although the tool found plenty of violations we were not able to approve a single one. The vast majority of alternatives (supertypes) the tool suggested are types that only make sense in specific contexts. In MPAndroidChart for example there is a type Poolable which is inherited by a lot of other classes to make them able to be stored in a Pool. The tool suggests Poolable for every parameter type and return type that inherits from Poolable. Even if the tool would be made configurable so these types can be ignored there would still be too many false-positives. The tool currently only searches for supertypes of the declared type but does not evaluate whether it can really be used or not, i.e., if using the suggested supertype would break the existing code. To mitigate these problems the algorithm could be made more sophisticated. For parameters for example it could rule out supertypes which do not provide the methods called on the parameter object.The problem with such a solution would be that it only assesses the current code but would not force the developer to think about the type decision and try to solve the problem with a higher abstraction. UseAbstractionAsParameterType could still be beneficial if it only suggests supertypes that can be used in the existing code if evaluations against other projects show a benefit. Based on the evaluation of MPAndroidChart and fastjson we suggest to remove UseAbstractionAsParameterType and UseAbstractionAsReturnType completely.

7.2.3 Program to an Interface, not an Implementation (PINI)

Detecting concrete classes that do not have any supertypes works as expected. A problem are classes like the before described Poolable in MPAndroidChart which are inherited by many classes. This kind of supertypes suppress violations of ProvideAbstractionForClass but we would not consider them real abstractions because they are more like indicators that enable a specific functionality. An enhancement for the tool would be to make classes configurable that are ignored as supertypes.

UseInterfaceIfPossible in general has the same problems as UseAbstractionAsParameterType and UseAbstractionAsReturnType. The vast majority of violations are false-positives because the tool does not check whether using the suggested type would break

the existing code. Unlike with the other two best practices we were able to find some valid violations of UseInterfaceIfPossible. In a few cases concrete types of collections, e.g., ArrayList or HashSet, were used as declared types of variables. From these cases we only consider those as problematic were the variable is a class member. Variables with concrete declared types in short methods are no problem in our opinion. Summarized, to be more beneficial we think the tool should only check for class members with concrete types and should only suggest supertypes that work with the existing code.

7.2.4 Information Hiding (IH)

As described before the detection of public, public static and protected fields and the detection of mutable class members that are returned by methods work reliably and we think they are useful.

The only best practice that is unique for Information Hiding is AvoidSettersForHeavilyUsedFields. Based on the violations found in the two projects we think AvoidSettersForHeavilyUsedFields can only report violation candidates, i.e., it may report false-positives. In our opinion, in some cases the current code is easier to maintain than having to inject the value of the field through the constructor and making the passed object immutable. Immutability is required because otherwise the injector would still be able to manipulate the passed object from the outside. Nevertheless we think the rule is helpful because it forces the developer to reflect the necessity of the setter and think about alternatives.

7.2.5 Single Choice Principle (SCP)

Detecting runtime type identification works reliably as described before. Finding case distinctions over enums does deliver valid violations but also produces false-positives. The valid violations can either be resolved by moving code into the enum itself or by avoiding code duplication through the use of methods. There are two types of false-positives. Some case distinctions have to stay were they are because otherwise the Single Responsibility Principle would be violated, e.g., in MPAndroidChart different chart types calculate the position of the legend based on the legends orientation and alignment which are enums. By moving the calculation into the enums they would have to know about the different kinds of charts which only moves the problem but does not resolve it. The second type of false-positives are enums that do not represent alternatives but rather possibilities. fastjson makes heavy use of enums that represent features which can be enabled. The code that gets executed if a feature is enabled can also not be moved due to the Single Responsibility Principle. Nevertheless, because of the found valid violations we still think AvoidCaseDistinctionOverEnums is a useful best practice.

7.2.6 Keep It Simple Stupid (KISS)

The tool is able to find long and complex methods as expected. With a threshold for cyclomatic complexity of ten, all found violations look complicated enough to us to

deserve a refactoring. Teams with other opinions can configure the threshold.

7.2.7 Single Responsibility Principle (SRP)

Finding non cohesive classes is currently error-prone but we still think it is beneficial. We were able to identify classes that have more than one responsibility, e.g., classes in fastjson that can serialize and deserialize despite those two methods having no shared members. The most false-positives are POJOs because they are inherently non cohesive. Each member builds a group with its corresponding getter and setter methods. Other false-positives emerge because the tool currently ignores type hierarchies when grouping together connected members and methods. Template methods, which are methods that need to be implemented in the subclass, appear to be standalone in the subclass despite being used with other members and methods in the superclass. To enhance the tool the relationship of members and methods in superclasses has to be considered and POJOs should be ignored completely.

As a side effect, dead code, e.g., unused fields and methods as well as empty methods, is reported as non cohesive because dead code does not connect with any other members and methods. Although it is not the main goal of the implementation of this principle we still think it is helpful.

7.2.8 Favor Composition over Inheritance (FCOI)

Of the three best practices used for this principle we can only approve UseCompositionNotInheritance as a strong indicator for a violation. In our opinion all violations reported by it can be resolved by using composition. The other two best practices produced many false-positive cases for which we could not find a way to solve them without inheritance. For example OnlyInheritFromAbstractClasses raises violations for exceptions that inherit from other exceptions which is common practice. Many libraries, the JDK included, only provide concrete classes that have to be inherited to use them. We think OnlyInheritFromAbstractClasses and CheckUnusedSuperType should be removed as standalone best practices but be added as an extension to UseCompositionNotInheritance. If UseCompositionNotInheritance finds a problem the other two best practices can be used to emphasize the violation.

7.2.9 Acyclic Dependencies Principle (ADP)

Detecting package cycles works reliably. A purely cosmetic enhancement would be to also display the classes that are part of the cycle not only the packages. Currently a user would have to use external tools to find the involved classes.

7.2.10 Option Operand Principle (OOP)

As described the tool can only detect boolean option parameters and this works as expected. We found two false-positives in fastjson because one deserialization class has

two methods which accept boolean values and write strings according to the value. This behavior looks like an option to the tool but we do not consider it a problem because this usage of a boolean parameter seems like a specialty of libraries like fastjson to us. Two violations in MPAndroidChart are debatable because boolean parameters here are used to enable/disable a feature. The enabling/disabling is executed through other methods, not by setting a boolean member. We would argue that it is a question of style whether someone prefers one method setXEnabled() or two methods enableX()/disableX() and therefore we also do not consider these two violations real false-positives.

7.2.11 Interface Segregation Principle (ISP)

All reported violations of not implemented inherited methods are correct. Either they are completely empty or just throw an exception like an UnsupportedOperationException which indicates that this method is not implemented.

7.2.12 Law of Demeter (LOD)

After evaluating the Law of Demeter against the two projects we think this principle should be removed from the tool because the automatic violation detection is too error-prone. Some false-positives arise if collections are used since it is forbidden to call a method of an object which was fetched from a collection. Resolving some other violations would lead to more complex code in our opinion. We think none of the found violations justify the automatic detection of this principle.

7.2.13 Stable Dependencies Principle (SDP)

We were able to identify stable packages which depend on much more instable packages with the tool. The detection works as expected but the results have to be assessed by a developer because currently even a difference in instability of 0.01 is a violation. Humans have to decide whether this is really a problem. The tool could also have a configurable threshold for the difference in our opinion to minimize false-positives.

7.2.14 Common Reuse Principle (CRP)

The tool is able to find classes which are used separated from other classes in the same package. For example, for MPAndroidChart classes are currently grouped into packages by their role, i.e., there are packages named charts, renderer and datasets. Based on the found violations though we think the classes should be grouped by chart type (line, bar, pie) to conform better with the Common Reuse Principle. This result is only based on the class usage inside the library itself. We are aware that for libraries like fastjson and MPAndroidChart the way clients use the classes can be completely different from internal usage. Still, the tool showed its capability to detect violations of the principle.

7.2.15 Stable Abstractions Principle (SAP)

The tool identified packages in both projects which are stable but concrete. Therefore, we conclude that the tool reliably detects violations of the Stable Abstractions Principle.

7.2.16 Information Expert (IE)

Because our tool did not find any violations of the Information Expert principle we were not able to evaluate if the automatic detection of this principle makes sense and is beneficial.

7.3 Summary

Based on our evaluation we think the tool is able to detect violations of the following principles reliably: Keep It Simple Stupid (KISS), Option Operand Principle (OOP), Interface Segregation Principle (ISP), Acyclic Dependencies Principle (ADP), Common Reuse Principle (CRP), Stable Dependencies Principle (SDP), Stable Abstractions Principle (SAP)

For the principles Don't Repeat Yourself (DRY), Information Hiding (IH) and Single Choice Principle (SCP) the tool is able to find violations but also reports false-positives. Nonetheless, we think the automatic detection of violations of these three principles is helpful.

In its current state the tool showed the potential to detect violations of the principles Program to an Interface, not an Implementation (PINI), Single Responsibility Principle (SRP) and Favor Composition over Inheritance (FCOI). Fundamentally, violations are found but the tool reports many false-positives. We already described possible enhancements to minimize the false-positive rate above and think that with them the automatic detection of these principles can be useful.

For the Open Closed Principle (OCP) the in our opinion important best practices UseAbstractionAsParameterType and UseAbstractionAsReturnType are error-prone and based on the evaluation we recommend to not use them. There are still best practices left that are able to find violations but we do not think they are covering the main proposition of the principle. The left over best practices are also included in other principles so we currently do not think the automatic detection of violations of the Open Closed Principle is beneficial.

Automatically detecting violations of the Law of Demeter (LOD) proofed itself to be very error-prone in the evaluation and because of that we do not think that it is useful.

The tool did not find any violations of the Information Expert (IE) principle, therefore we are not able to make any statements about the usefulness of its automatic violation detection.

The tool showed its ability to find design problems and report their location in the source code. Because each problem is based on a design principle the solution to fix the design flaw is to not violate the principle. From a developers perspective problems are delivered with a possible solution and we think this makes the tool useful.

In contrast to SonarJava, Checkstyle and SpotBugs the tool is focused on design. All three other analyzers cover some parts of the tool, i.e., SpotBugs detects circular dependencies but on a class level, Checkstyle detects missing Javadoc, too high cyclomatic complexity, too open visibility and long methods and SonarJava detects too high cyclomatic complexity, missing Javadoc, long methods and empty methods. This intersection exists but the found violations are not connected to design principles. The tool we implemented is able to detect far more design flaws and connects them to design principles, giving the developer more context for an existing problem.

7.4 Research Questions Revisited

RQ 1: Which violations of object-oriented design principles for Java can be detected automatically, and how can we do that?

We found violations of the following principles to be detectable automatically by a static code analyzer:

- Keep It Simple Stupid (KISS)
- Option Operand Principle (OOP)
- Interface Segregation Principle (ISP)
- Acyclic Dependencies Principle (ADP)
- Common Reuse Principle (CRP)
- Stable Dependencies Principle (SDP)
- Stable Abstractions Principle (SAP)
- Don't Repeat Yourself (DRY)
- Information Hiding (IH)
- Single Choice Principle (SDP)
- Program to an Interface, not an Implementation (PINI)
- Single Responsibility Principle (SRP)
- Favor Composition over Inheritance (FCOI)

Violations of the these principles are in our opinion inherently not detectable by a static code analyzer:

- Liskov Substitution Principle (LSP)
- Dependency Inversion Principle (DIP)
- Single Level of Abstraction (SLA)
- Common Closure Principle (CCP)
- Integration Operation Segregation Principle (IOSP)
- Command Query Separation (CQS)
- Encapsulate the Concept that Varies (ECV)

Based on the evaluation of our tool we had to reject our assumption that violations of the Open Closed Principle (OCP) and the Law of Demeter (LOD) can be detected automatically because our tool produced too many false-positives for these principles. We think that violations of the Information Expert (IE) principle can be detected automatically but our tool did not find any violations for the two projects we ran it against. Therefore, we can not give any informed statement about its automatic detectability.

To find violations of design principles we broke them down into design best practices because design principles are too abstract. Each design best practice covers an aspect of the design principle. If a best practice is violated, then the design principle, which the best practice is a part of, is also violated. Detailed descriptions of the detection strategies can be found in Chapter 4. To detect violations of best practices the abstract syntax tree of the code has to be scanned for problematic patterns. Our tool for example looks for missing Javadoc (DocumentPublicMethods) or switch expressions over an enum type (AvoidCaseDistinctionOverEnums).

RQ 2: How useful is the automatic detection (according to RQ1) to assess the quality of a software project?

With our tool we were able to detect violations of 13 design principles. Each reported violation is a design flaw but the severity of the flaw has to be assessed by a developer. We think, missing Javadoc for example can be fixed much easier than problematic inheritance hierarchies that should be replaced by composition. The number of found violations nonetheless is in our opinion an indicator for the design quality of a project. The type of found violations also gives information about the kind of problems the project has. For example, many Information Hiding violations indicate a very open design in terms of visibility. In such a project, changing the internals of objects can be hard. Many violations of the Single Choice Principle may mean the developers have insufficient knowledge or experience in object-oriented programming. Violations of the Interface

Segregation Principle and the Favor Composition over Inheritance principle suggest problems in the inheritance and type hierarchies. If violations of Don't Repeat Yourself, Single Responsibility Principle, Keep It Simple Stupid or Option Operand Principle are found, then the code might be hard to read. Many violations of package principles may indicate that the way classes are grouped into packages was not, or insufficiently, thought through.

Design Principle		Design best practice		
Name	Violations	Name	Violations	False-positives
PINI	744	UseInterfaceIfPossible	730	● ● ●
		ProvideAbstractionForClass	14	○ ○ ○
DRY	691	DocumentPublicMethods	592	○ ○ ○
		AvoidMassiveInlineComments	52	● ● ○
		DocumentPublicInterfacesClassesEnums...	47	○ ○ ○
OCP	636	AvoidProtectedField	272	○ ○ ○
		UseAbstractionAsParameterType	148	● ● ●
		UseAbstractionsAsReturnType	126	● ● ●
		AvoidPublicField	40	○ ○ ○
		DontReturnMutableField	29	○ ○ ○
		AvoidRuntimeTypeIdentification	20	○ ○ ○
		AvoidPublicStaticField	1	○ ○ ○
IH	471	AvoidProtectedField	272	○ ○ ○
		UseAbstractionsAsReturnType	126	● ● ●
		AvoidPublicField	40	○ ○ ○
		DontReturnMutableField	29	○ ○ ○
		AvoidSettersForHeavilyUsedFields	4	● ○ ○
SRP	84	AvoidNonCohesiveImplementations	84	● ● ○
SCP	62	AvoidCaseDistinctionOverEnums	42	● ○ ○
		AvoidRuntimeTypeIdentification	20	○ ○ ○
KISS	42	AvoidComplexMethods	42	○ ○ ○
		AvoidLongMethods	0	○ ○ ○
ADP	36	AvoidPackageCycles	36	○ ○ ○
FCOI	29	OnlyInheritFromAbstractClasses	22	● ● ●
		UseCompositionNotInheritance	6	○ ○ ○
		CheckUnusedSuperType	1	● ● ●
ISP	21	AvoidStubForInheritedMethod	21	○ ○ ○
OOP	15	AvoidOptionParameters	15	● ○ ○
SDP	9	DependInTheDirectionOfStability	9	○ ○ ○
CRP	5	CheckUnsuitableFunctionalityOfPackage	5	○ ○ ○
LOD	3	AvoidChainedMethodCalls	3	● ● ●
SAP	2	PackagesShouldBeAsAbstractAsStable	2	○ ○ ○
IE	0	AvoidGetAndSet	0	-
		AvoidUnnecessaryInformationTransfer	0	-

Table 7.1: Violations in MPAndroidChart
○ ○ ○ = none, ● ○ ○ = few, ● ● ○ = many, ● ● ● = a great many

Design Principle		Design best practice		
Name	Violations	Name	Violations	False-positives
DRY	1260	DocumentPublicMethods	1069	○ ○ ○
		AvoidMassiveInlineComments	119	● ● ○
		DocumentPublicInterfacesClassesEnums...	72	○ ○ ○
OCP	1167	AvoidRuntimeTypeIdentification	655	○ ○ ○
		UseAbstractionAsParameterType	209	● ● ●
		AvoidProtectedField	104	○ ○ ○
		AvoidPublicField	100	○ ○ ○
		UseAbstractionsAsReturnType	61	● ● ●
		AvoidPublicStaticField	24	○ ○ ○
		DontReturnMutableField	14	○ ○ ○
PINI	1014	UseInterfaceIfPossible	978	● ● ●
		ProvideAbstractionForClass	36	○ ○ ○
SCP	691	AvoidRuntimeTypeIdentification	655	○ ○ ○
		AvoidCaseDistinctionOverEnums	36	● ● ○
IH	287	AvoidProtectedField	104	○ ○ ○
		AvoidPublicField	100	○ ○ ○
		UseAbstractionsAsReturnType	61	● ● ●
		DontReturnMutableField	14	○ ○ ○
		AvoidSettersForHeavilyUsedFields	8	● ○ ○
KISS	234	AvoidComplexMethods	222	○ ○ ○
		AvoidLongMethods	12	○ ○ ○
SRP	120	AvoidNonCohesiveImplementations	120	● ● ○
FCOI	46	CheckUnusedSuperType	34	● ● ●
		OnlyInheritFromAbstractClasses	9	● ● ●
		UseCompositionNotInheritance	3	○ ○ ○
OOP	29	AvoidOptionParameters	29	● ○ ○
LOD	26	AvoidChainedMethodCalls	26	● ● ●
ADP	21	AvoidPackageCycles	21	○ ○ ○
SDP	10	DependInTheDirectionOfStability	10	○ ○ ○
ISP	9	AvoidStubForInheritedMethod	9	○ ○ ○
CRP	3	CheckUnsuitableFunctionalityOfPackage	3	○ ○ ○
SAP	3	PackagesShouldBeAsAbstractAsStable	3	○ ○ ○
IE	0	AvoidGetAndSet	0	-
		AvoidUnnecessaryInformationTransfer	0	-

Table 7.2: Violations in fastjson
○ ○ ○ = none, ● ○ ○ = few, ● ● ○ = many, ● ● ● = a great many

CHAPTER 8

Conclusion

After summarizing concepts of object-oriented programming we gave an overview of software design quality. We explained why design quality is important, how it can be measured and defined terms like design principle and design best practice. To answer the research questions we researched design principles and determined whether violations of them could be automatically detectable. To proof the assumed detectability we implemented a static code analyzer and evaluated it against two open source projects.

We identified 13 principles whose violations can be found by a static code analyzer. Violations of seven of the investigated design principles are in our opinion inherently not detectable. For two principles we had to reject our assumption that violations of them can be detected automatically, because the evaluation of our tool showed that the detection of them is too unreliable. For one principle we did not get enough information in our evaluation to make an informed statement about its automatic detectability.

We think that the number of found violations by our tool is an indicator for a projects design quality. The type of violations also gives insight about the design aspects that need to be improved.

Future Work For the future we think it would be beneficial to perform a survey, like Bräuer [3] did, to get more opinions on the appropriateness of the best practices we came up with for the design principles. This survey should evaluate whether or not the best practices cover all aspects of the design principles and might provide input for new best practices. To further improve our tool it should be run against more projects to get more information about the appropriateness of our rules. We especially need more information for the Information Expert principle because we did not get any in our evaluation. Also, more information is needed for the Open Closed Principle and the Law of Demeter since further evaluation might prove their automatic detection useful, contrary to our evaluation result. Another way to improve the tool would be its continuous use in an active project. This could bring up more false-positives that have to be fixed but more

importantly false-negatives could be found. If a developer finds a design flaw that the tool did not report they found a false-negative and the tool has to be changed to cover this case too, if possible. Finding false-negatives is labor-intensive in an evaluation but does not have much overhead during project development if reviews are performed.

APPENDIX A

Appendix

A.1 Design best practices

This is a list of all design best practices in the thesis together with the pages they appear on. Further descriptions and sources can be found on the referenced pages.

AvoidCaseDistinctionOverEnums

Case distinctions in the form of switch expressions or if/else cascades should be avoided, because changes to the enum may need follow up changes to every one of these case distinctions. 39

AvoidChainedMethodCalls

Calling methods on the results of method calls should be avoided. Such chains lead to tight coupling with the result types. The nearer type becomes more difficult to change, because his internals are used. 29

AvoidComplexMethods

Methods should not be too complex in terms of cyclomatic complexity, as a high cognitive demand reduces the readability. 30

AvoidDuplication

Code (and knowledge in general) should not be duplicated. 33

AvoidGetAndSet

Instead of getting a value or multiple values of an object and setting some value on it afterwards, the object should be asked to make the update, because it is the information expert. 36

AvoidLongMethods

Long methods reduce the understandability and should therefore be avoided, whereby the acceptable length is subjective. 30

AvoidMassiveInlineComments

If a method needs a lot of inline comments, it might need refactoring to make the code more clear without the comments. Comments can become out of sync with the implementation and should be avoided whenever possible. 33

AvoidNonCohesiveImplementations

Non cohesive implementations have multiple responsibilities, hence also multiple reasons to change, making them more fragile. Therefore they should be split up. 18

AvoidOptionParameters

Parameters that only influence the behavior of a method should be transformed to option fields. In the case of boolean parameters the method can also be split into two methods. Option parameters complicate the method signature, decreasing readability and are not reusable, contrary to fields. 53

AvoidPackageCycles

The usage graph of packages should not have any cycles because the release order of packages becomes unclear. 46

AvoidProtectedFields

Protected fields open up the internals of a class for its subclasses, making it harder to change in the future. 21, 25

AvoidPublicFields

Public fields leak the internals of a class completely, making changes to it complicated. 21, 25

AvoidPublicStaticFields

Public static fields create a tight coupling between clients and the field, making changes to the class more difficult. 21

AvoidRuntimeTypeIdentification

Runtime type identification is not future proof against new types and poses a risk for future changes. 21, 39

AvoidSettersForHeavilyUsedFields

If a class heavily uses a field internally, it should not be setable from the outside. Internals can be changed much easier, if they cannot be manipulated from the outside. 25

AvoidStubForInheritedMethod

A subtype that overrides a method but does not provide a meaningful implementation (empty body or only throws exception) is an indicator for a invalid subtype relation. 24

AvoidUnnecessaryInformationTransfer

If multiple getter calls to an object are made, information is transferred that the object is the expert on. Whatever task the information is for, the object is probably better suited to execute it, because it already has the information. 36

CheckUnsuitableFunctionalityOfPackage

Classes of a package should have to be used together. If clients can use a part of a package alone it might have to be split up. 44

CheckUnusedSupertype

If clients only use methods of the subtype, not the supertype, the subtype relation might be invalid and only exists for code reuse in the subtype. 43

DependInTheDirectionOfStability

Pacakges should only depend in the direction of stability. 48

DocumentPublicInterfacesClassesEnumsAndAnnotations

Documentation on public classes, interfaces, enums and annotations simplify their reuse, because developers can easier understand when and how to use them. 33

DocumentPublicMethods

Documentation on public methods aids their reuse, because developers can easier understand when and how to use them. 33

DontReturnMutableField

If a mutable field is returned by a method, the client can manipulate the state of an (maybe even private) field. The object lost control over its own state. 21, 25

OnlyInheritFromAbstractClasses

Abstract classes are more stable than concrete ones, making ripple effects less likely to occur when the superclass is changed. 43

PackagesShouldBeAsAbstractAsStable

Packages that are stable should also be abstract but packages that are unstable should be concrete. 49

ProvideAbstractionForClass

For every class there should be an abstraction, so clients do not need to couple with concrete classes. 40

UseAbstractionAsParameterType

Using abstractions as parameter types makes methods open for future subtypes, without needing to change their source code. 21

UseAbstractionAsReturnType

Using abstractions as return types keeps the internals of methods hidden, making changes to the implementation easier. 21, 25

UseCompositionNotInheritance

Inheritance should not be used for code reuse, composition is the way to go. Inheritance for code reuse might leak implementation details or create invalid and complicated type hierarchies that might be hard to entangle in the future. 43

UseInterfaceIfPossible

If an abstraction exists for a type it should be used, to decouple the client from the concrete implementation. 40

Bibliography

[1] Deborah J. Armstrong. "The quarks of object-oriented development". In: *Communications of the ACM* 49.2 (2006), pp. 123–128. ISSN: 00010782. DOI: 10.1145/1113034.1113040.

[2] Johannes Brauer et al. "A survey on the importance of object-oriented design best practices". In: *Proceedings - 43rd Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2017* (2017), pp. 27–34. DOI: 10.1109/SEAA.2017.14.

[3] Johannes Bräuer. "Measuring and Assessing Object-oriented Design Principles". PhD. Thesis. Universität Linz, 2017.

[4] Johannes Bräuer et al. "Measuring object-oriented design principles: The results of focus group-based research". In: *Journal of Systems and Software* 140 (2018), pp. 74–90. ISSN: 01641212. DOI: 10.1016/j.jss.2018.03.002.

[5] Johannes Bräuer et al. "On the suitability of a portfolio-based design improvement approach". In: *Proceedings - 2018 IEEE 18th International Conference on Software Quality, Reliability, and Security, QRS 2018* (2018), pp. 249–256. DOI: 10.1109/QRS.2018.00038.

[6] E Capra, C Francalanci, and F Merlo. "Software design quality and development effort: an empirical study on the role of governance in Open Source projects". In: *IEEE Transaction on Software Engineering* 34.6 (2008), pp. 765–782.

[7] S. R. Chidamber and C. F. Kemerer. "A metrics suite for object oriented design". In: *IEEE Transactions on Software Engineering* 20.6 (June 1994), pp. 476–493. ISSN: 0098-5589. DOI: 10.1109/32.295895.

[8] Ward Cunningham. "The WyCash portfolio management system". In: *ACM SIGPLAN OOPS Messenger* 4.2 (2007), pp. 29–30. ISSN: 10556400. DOI: 10.1145/157710.157715.

[9] John Dooley. "Object-Oriented Design Principles". In: *Software Development and Professional Practice*. Berkeley, CA: Apress, 2011, pp. 115–136. ISBN: 978-1-4302-3802-7. DOI: 10.1007/978-1-4302-3802-7_10.

[10] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999. ISBN: 0-201-48567-2.

[11] E. Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Pearson Education, 1994. ISBN: 9780321700698.

[12] Puneet Kumar Goyal and Gamini Joshi. "QMOOD metric sets to assess quality of Java program". In: *Proceedings of the 2014 International Conference on Issues and Challenges in Intelligent Computing Techniques, ICICT 2014* (2014), pp. 520–533. DOI: 10.1109/ICICICT.2014.6781337.

[13] H. Gruber et al. "Tool Support for ISO 14598 based code quality assessments". In: *6th International Conference on the Quality of Information and Communications Technology (QUATIC 2007)*. Sept. 2007, pp. 21–29. DOI: 10.1109/QUATIC.2007.11.

[14] Yi Guo et al. "An empirical validation of the benefits of adhering to the law of demeter". In: *Proceedings - Working Conference on Reverse Engineering, WCRE* (2011), pp. 239–243. ISSN: 10951350. DOI: 10.1109/WCRE.2011.36.

[15] Wang Haoyu and Zhou Haili. "Basic design principles in software engineering". In: *Proceedings - 4th International Conference on Computational and Information Sciences, ICCIS 2012* (2012), pp. 1251–1254. DOI: 10.1109/ICCIS.2012.91.

[16] Kevlin Henney. "A tale of two patterns". In: *Java Report* 5.12 (2000), pp. 84–88.

[17] Martin Hitz and Behzad Montazeri. "Measuring Coupling and Cohesion In Object-Oriented Systems". In: *Angewandte Informatik* 50 (1995), pp. 1–10.

[18] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN: 0-201-61622-X.

[19] Habib Izadkhah and Maryam Hooshyar. "Class Cohesion Metrics for Software Engineering: A Critical Review". In: *The Computer Science Journal of Moldova* 25 (2017), pp. 44–74.

[20] Bansiya Jagdish and G Davis Carl. "A Hierarchical Model for Object-Oriented Design Quality Assessment". In: *IEEE Transactions on Software Engineering* 28.1 (2002), pp. 4–17. DOI: http://dx.doi.org/10.1109/32.979986.

[21] Kirk Knoernschild. *Java Design: Objects, UML, and Process*. Pearson Education, 2001. ISBN: 0201750449.

[22] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2004. ISBN: 0131489062.

[23] K. Lieberherr, I. Holland, and A. Riel. "Object-oriented programming: An objective sense of style". In: *ACM SIGPLAN Notices* 23.11 (1988), pp. 323–334. ISSN: 15581160. DOI: 10.1145/62084.62113.

[24] Barbara Liskov. “Keynote Address - Data Abstraction and Hierarchy”. In: *Addendum to the Proceedings on Object-oriented Programming Systems, Languages and Applications (Addendum)*. OOPSLA ’87. Orlando, Florida, USA: ACM, 1987, pp. 17–34. ISBN: 0-89791-266-7. DOI: 10.1145/62138.62141.

[25] Barbara H. Liskov and Jeannette M. Wing. “A Behavioral Notion of Subtyping”. In: *ACM Trans. Program. Lang. Syst.* 16.6 (Nov. 1994), pp. 1811–1841. ISSN: 0164-0925. DOI: 10.1145/197320.197383. URL: https://doi.org/10.1145/197320.197383.

[26] Radu Marinescu. “Detecting Design Flaws via Metrics in Object-Oriented Systems A Metrics-Based Approach for Problem Detection”. In: *International Conference and Technology of Object-Oriented Languages and Systems (TOOLS)* (2001), pp. 173–182.

[27] Radu Marinescu. “Detection strategies: Metrics-based rules for detecting design flaws”. In: *IEEE International Conference on Software Maintenance, ICSM* (2004), pp. 350–359. ISSN: 1063-6773. DOI: 10.1109/ICSM.2004.1357820.

[28] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2008. ISBN: 0132350882, 9780132350884.

[29] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, 2003. ISBN: 0135974445.

[30] T. J. McCabe. “A Complexity Measure”. In: *IEEE Transactions on Software Engineering* SE-2.4 (Dec. 1976), pp. 308–320. ISSN: 0098-5589. DOI: 10.1109/TSE.1976.233837.

[31] Steve McConnell. *Code Complete, Second Edition*. Redmond, WA, USA: Microsoft Press, 2004. ISBN: 0735619670, 9780735619678.

[32] Bertrand Meyer. *Object-Oriented Software Construction*. 1st. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1988. ISBN: 0136290493.

[33] Subhas C. Misra and Virendra C. Bhavsar. “Relationships Between Selected Software Measures and Latent Bug-Density: Guidelines for Improving Quality”. In: *Computational Science and Its Applications — ICCSA 2003*. Ed. by Vipin Kumar et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 724–732. ISBN: 978-3-540-44839-6.

[34] Naouel Moha et al. “DECOR: A method for the specification and detection of code and design smells”. In: *IEEE Transactions on Software Engineering* 36.1 (2010), pp. 20–36. ISSN: 00985589. DOI: 10.1109/TSE.2009.50.

[35] Oscar Nierstrasz. “Object-oriented Concepts, Databases, and Applications”. In: ed. by Won Kim and F. H. Lochovsky. New York, NY, USA: ACM, 1989. Chap. A Survey of Object-oriented Concepts, pp. 3–21. ISBN: 0-201-14410-7. DOI: 10.1145/63320.66468.

[36] D. L. Parnas. “On the Criteria to Be Used in Decomposing Systems into Modules”. In: *Commun. ACM* 15.12 (Dec. 1972), pp. 1053–1058. ISSN: 0001-0782. DOI: 10.1145/361598.361623.

[37] Reinhold Plösch et al. “Measuring, Assessing and Improving Software Quality based on Object-Oriented Design Principles”. In: *Open Computer Science* 6.1 (2016), pp. 187–207. DOI: 10.1515/comp-2016-0016.

[38] Reinhold Plösch et al. “MUSE: A Framework for Measuring Object-Oriented Design Quality.” In: *The Journal of Object Technology* 15.4 (2016), 2:1. ISSN: 1660-1769. DOI: 10.5381/jot.2016.15.4.a2.

[39] Eric S. Raymond. *The Art of UNIX Programming*. Pearson Education, 2003, p. 17. ISBN: 0131429019.

[40] Christian Rehn. “A Principle Language for Object-Oriented Design”. Master thesis. Technische Universität Kaiserslautern, 2013.

[41] Ben R Rich. *Clarence Leonard (Kelly) Johnson 1910–1990: A Biographical Memoir*. Washington D.C., 1995.

[42] Ganesh Samarthyam et al. “MIDAS: A design quality assessment method for industrial software”. In: *Proceedings - International Conference on Software Engineering* (2013), pp. 911–920. ISSN: 02705257. DOI: 10.1109/ICSE.2013.6606640.

[43] Tushar Sharma, Ganesh Samarthyam, and Girish Suryanarayana. “Applying Design Principles in Practice”. In: *Proceedings of the 8th India Software Engineering Conference*. ISEC ’15. Bangalore, India: ACM, 2015, pp. 200–201. ISBN: 978-1-4503-3432-7. DOI: 10.1145/2723742.2723764.

[44] Sandra A. Slaughter, Donald E. Harter, and Mayuram S. Krishnan. “Evaluating the cost of software quality”. In: *Communications of the ACM* 41.8 (2002), pp. 67–73. ISSN: 00010782. DOI: 10.1145/280324.280335.

[45] Alan Snyder. “Encapsulation and Inheritance in Object-oriented Programming Languages”. In: *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*. OOPSLA ’86. Portland, Oregon, USA: ACM, 1986, pp. 38–45. ISBN: 0-89791-204-7. DOI: 10.1145/28697.28702.

[46] Alan Snyder. “Encapsulation and inheritance in object-oriented programming languages”. In: *ACM SIGPLAN Notices* 21.11 (2005), pp. 38–45. ISSN: 03621340. DOI: 10.1145/960112.28702.

[47] Mark Stefik and Daniel G. Bobrow. “Object-Oriented Programming: Themes and Variations”. In: *The AI MagazineAI Magazine* 6.4 (1985), pp. 40–62. ISSN: 0738-4602. DOI: 10.1609/aimag.v6i4.508.

[48] Friedrich Steimann. “Role = Interface - A merger of concepts”. In: *Journal of Object-Oriented Programming* 14 (2001), pp. 23–32.

[49] Friedrich Steimann and Philip Mayer. “Patterns of interface-based programming”. In: *Journal of Object Technology* 4 (2005), pp. 75–94.

[50] Jamie Stevenson and Murray Wood. "Recognising object-oriented software design quality: a practitioner-based questionnaire survey". In: *Software Quality Journal* 26.2 (2018), pp. 321–365. ISSN: 15731367. DOI: 10.1007/s11219-017-9364-8.

[51] L. Tahvildari and A. Singh. "Categorization of object-oriented software metrics". In: *2000 Canadian Conference on Electrical and Computer Engineering. Conference Proceedings. Navigating to a New Era (Cat. No.00TH8492)*. Vol. 1. May 2000, 235–239 vol.1. DOI: 10.1109/CCECE.2000.849705.

[52] Peter Wegner. "Concepts and Paradigms of Object-oriented Programming". In: *SIGPLAN OOPS Mess.* 1.1 (Aug. 1990), pp. 7–87. ISSN: 1055-6400. DOI: 10.1145/382192.383004.

[53] Rebecca Wirfs-Brock and Brian Wilkerson. "Object-oriented design: A responsibility-driven approach". In: *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 1989* 503 (1989), pp. 71–75. DOI: 10.1145/74877.74885.

[54] Aiko Yamashita and Leon Moonen. "Do developers care about code smells? An exploratory survey". In: *Proceedings - Working Conference on Reverse Engineering, WCRE* (2013), pp. 242–251. ISSN: 10951350. DOI: 10.1109/WCRE.2013.6671299.

Web references

[55] *Alternative distance function for Stable Abstractions Principle*. accessed 3. October 2019. URL: http://www.lexicalscope.com/blog/2012/10/31/alternative-distance-function-for-stable-abstractions-principle/.

[56] *Checkstyle documentation for writing custom rules*. accessed 7. July 2019. URL: https://checkstyle.sourceforge.io/writingchecks.html (visited on 07/08/2019).

[57] *CommandQuerySeparation*. accessed 8. October 2019. URL: https://martinfowler.com/bliki/CommandQuerySeparation.html.

[58] *Function Length*. accessed 30. August 2019. URL: https://martinfowler.com/bliki/FunctionLength.html.

[59] *Github sonar-findbugs*. accessed 8. July 2019. URL: https://github.com/spotbugs/sonar-findbugs.

[60] *Github sonar-pmd*. accessed 8. July 2019. URL: https://github.com/jensgerdes/sonar-pmd.

[61] *Java language specification*. accessed 9. April 2019. URL: https://docs.oracle.com/javase/specs/jls/se12/html/index.html.

[62] *Kontrollstrukturen in der Integration*. accessed 4. October 2019. URL: https://ccd-school.de/2017/02/kontrollstrukturen-in-der-integration/.

[63] *Law Of Demeter Makes Unit Tests Easier*. accessed 23. August 2019. URL: http://wiki.c2.com/?LawOfDemeterMakesUnitTestsEasier.

[64] *Law Of Demeter Revisited*. accessed 23. August 2019. URL: http://wiki.c2.com/?LawOfDemeterRevisited.

[65] *PMD documentation for writing custom rules*. accessed 8. July 2019. URL: https://pmd.github.io/latest/pmd_userdocs_extending_writing_pmd_rules.html.

[66] *Principles Of Object Oriented Design*. accessed 23. August 2019. URL: http://wiki.c2.com/?PrinciplesOfObjectOrientedDesign.

[67] *Principles Wiki*. accessed 23. August 2019. URL: http://www.principles-wiki.net/principles:start.

[68] *Prinzip der gegenseitigen Nichtbeachtung*. accessed 4. October 2019. URL: https://blog.ralfw.de/2012/12/prinzip-der-gegenseitigen-nichtbeachtung.html.

[69] *Prinzipien der Softwaretechnik*. accessed 23. August 2019. URL: http://prinzipien-der-softwaretechnik.blogspot.com/.

[70] *Refactoring Patterns - The Integration Operation Segregation Principal*. accessed 4. October 2019. URL: https://www.deepakrb.com/blog/integration-operation-principal-en20180422/.

[71] *Single Level of Abstraction (SLA)*. accessed 25. September 2019. URL: http://www.principles-wiki.net/principles:single_level_of_abstraction.

[72] *SonarJava documentation for writing custom rules*. accessed 8. July 2019. URL: https://docs.sonarqube.org/display/PLUG/Writing+Custom+Java+Rules+101.

[73] *SonarQube Metric Definitions*. accessed 9. September 2019. URL: https://docs.sonarqube.org/display/SONARqube71/Metric+Definitions.

[74] *What is the ideal length of a method for you?* accessed 30. August 2019. URL: https://softwareengineering.stackexchange.com/questions/133404/what-is-the-ideal-length-of-a-method-for-you.

[75] *Why component dependency cycle is bad*. accessed 27. September 2019. URL: https://www.lvguowei.me/post/why-component-dependency-cycle-is-bad/.