Bachelorarbeit im Studiengang Mobile Medien

A PlantUML Based Approach for Design Conformance Checking

vorgelegt von

Pia Schilling

an der Hochschule der Medien Stuttgart

am 12. August 2024

zur Erlangung des akademischen Grades eines Bachelor of Science

Erst-Prüfer: Prof. Dr. Tobias Jordine

Zweit-Prüfer: Prof. Dr. Edmund Ihler

Ehrenwörtliche Erklärung

„Hiermit versichere ich, Pia Schilling, ehrenwörtlich, dass ich die vorliegende Bachelorarbeit mit dem Titel: „A PlantUML Based Approach for Design Conformance Checking“ selbstständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Ebenso sind alle Stellen, die mit Hilfe eines KI-basierten Schreibwerkzeugs erstellt oder überarbeitet wurden, kenntlich gemacht. Das KI-Werkzeug DeepL wurde ausschließlich zum Lektorat und als Übersetzungshilfe verwendet. Anschließend wurde sichergestellt, dass der Inhalt mit der intendierten Aussage noch übereinstimmt. Die Arbeit ist noch nicht veröffentlicht oder in anderer Form als Prüfungsleistung vorgelegt worden. Ich habe die Bedeutung der ehrenwörtlichen Versicherung und die prüfungsrechtlichen Folgen (§ 24 Abs. 2 Bachelor-SPO, § 23 Abs. 2 Master-SPO (Vollzeit)) einer unrichtigen oder unvollständigen ehrenwörtlichen Versicherung zur Kenntnis genommen.“

Stuttgart, 12.08.2024

[illegible]

Kurzfassung

Software-Erosion ist ein bekanntes und umfassend untersuchtes Phänomen, das schwerwiegende Auswirkungen auf Softwaresysteme haben kann. Daher ist es von entscheidender Bedeutung, Erosion frühzeitig zu erkennen und zu verhindern. Verschiedene Ansätze für die Entwurfskonformitätsprüfung existieren bereits, um Erosion aufzudecken, jedoch werden Systeme nach wie vor häufig manuell auf Konformität geprüft. Diese Arbeit stellt einen Ansatz für statische Entwurfskonformitätsprüfung vor, der darauf abzielt, die Nutzungshürden für Anwender möglichst gering zu halten. Dafür wurde ein prototypisches Tool entwickelt, das anhand von PlantUML-Klassendiagrammen Abweichungen zwischen Implementierung und Design erkennt. Die Evaluierung des Tools zeigte, dass der Mangel an Abstraktion in den Designmodellen die Anwendbarkeit des Tools für Entwurfskonformitätsprüfungen einschränkt. Dies ist auch teilweise auf Schwächen in der Implementierung des Tools zurückzuführen. Dennoch weist der Ansatz Potential auf, das durch die Weiterentwicklung des Prototyps und einer Erhöhung des Abstraktionsgrades der Designmodelle ausgeschöpft werden könnte.

Abstract

Software erosion is a well-known and extensively studied phenomenon that can have a serious impact on software systems. It is therefore crucial to detect and prevent erosion at an early stage. Various design conformance checking approaches already exist to detect erosion, but systems are still often checked manually for conformance. This thesis presents an approach for static design conformance checking that aims to minimise the usage hurdles for users. To this end, a prototype tool was developed that uses PlantUML class diagrams to detect deviations between implementation and design. The evaluation of the tool showed that the lack of abstraction in the design models limits the applicability of the tool for design conformance checks. This is also partly due to weaknesses in the implementation of the tool. Nevertheless, the approach has potential that could be utilised by further developing the prototype and increasing the level of abstraction of the design models.

Acknowledgements

My special thanks go to my supervisor, Prof. Dr. Tobias Jordine, for his valuable support, his competent advice and his patience. His constructive advice and high level of motivation made a crucial contribution to the success of this thesis.

I would also like to thank my second supervisor, Prof. Dr. Edmund Ihler, for the helpful feedback and interesting discussions, which have enriched my work enormously.

Contents

List of Tables

List of Figures

List of Listings

Acronyms

AI	Artificial Intelligence
ADL	Architecture Description Languages
CI/CD	Continuous Integration/Continuous Deployment
CLI	Command-line interface
DSM	Dependency Structure Matrices
GQM	Goal-Question-Metrics
IESE	Institute for Experimental Software Engineering
IDE	Integrated Development Environment
LDM	Lattix Dependency Manager
RM	Reflexion Model
SAVE	Software Architecture Visualization and Evaluation
SCQL	Source Code Query Language
SQL	Structured Query Language
UI	User Interface
UML	Unified Modeling Language

Chapter 1

Introduction

Software systems typically undergo numerous changes throughout their life cycle as defects are fixed and new features are added [2]. However, when these adaptations are made without a comprehensive understanding of the overall structure, violations of the intended design can occur [3]. Over time, these violations cause the implementation to drift away from the original design. This phenomenon, known as software design erosion, is a well-researched area [3],[4]. Due to the far-reaching consequences of system erosion, several strategies have been developed to mitigate its effects, one of which is conformance checking. This approach aims to prevent erosion by continuously tracking deviations between the intended design and the actual implementation [5].

1.1 Presentation of the Problem Statement

Several approaches for design conformance checking already exist [1], [6], [5]. However, manual reviews remain the most commonly used technique for maintaining compliance between design and implementation, as existing tools often introduce additional workload or require specialized training [7], [8]. To alleviate the burden of using tools for conformance checking, this thesis presents an approach based on the widely-used modeling notation, Unified Modeling Language (UML) [9], which can be seamlessly integrated into existing development processes.

1.2 Presentation of the Research Questions

The evaluation of a PlantUML based approach for static design conformance checking aims to answer the following research questions:

R.Q. 1: Which specific design aspects can be analyzed using the proposed approach?

R.Q. 2: What are the practical advantages and disadvantages of a PlantUML- based approach for design conformance checking in software development projects?

1.3 Presentation of the Research approach

To develop a feasible approach for static conformance checking, the first step is to identify requirements and design goals by analyzing related work and theoretical foundations. Based on these identified requirements, a prototype will be implemented. This prototype will then be evaluated to address the research questions and to assess its usefulness in real-world projects.

1.4 Presentation of the Thesis Structure

Chapter 1 provides an introduction to this thesis and its research questions, while chapter 2 presents the theoretical concepts underlying the approach discussed. Chapter 3 provides an overview of other approaches for static conformance checking. Building on the findings from these two chapters, the design of the prototype tool is outlined in Chapter 4, together with implementation details relevant for the subsequent discussion. Chapter 5 then presents the evaluation methodology and results, which are discussed and interpreted in chapter 6. Finally, chapter 7 reflects back to the original motivation for this work.

Chapter 2

Theoretical Background

This section covers the fundamental concepts related to the approach presented. It aims to provide a clearer understanding of the rationale behind the design and creation of the tool.

2.1 Software Architecture and Design

No universal definition for the term software architecture exists, which often leads to confusion [2],[10]. Hochstein created a definition based on elements from several other definitions which reads as follows:

> "Software architecture deals with the structure and interactions of a software system."[2]

While software architecture represents the high-level structure and behavior of a system [5], software design refers to the design on code level. This includes module and class design as well as function definitions [11].

2.2 Architecture and Design Erosion

By adding functionality and fixing bugs, software systems undergo a large number of changes during their life cycle. This can lead to the system-structure deviating further and further from the original design [2]. The phenomenon of divergences between the implemented and the currently desired architecture is often called architecture erosion [3],[8],[4]. The declining health of software systems has far-reaching effects [3], such as a difficult-to-understand and difficult-to-maintain system caused by the increasing complexity [4], which ultimately leads to a decrease in development speed [8].

Since this phenomenon has been a subject of research since the beginning of investigations on architectural concepts, it is known under a variety of terms such as architecture erosion, architecture decay, architecture degeneration, software deterioration, architectural degeneration, software degradation [3], software aging [12], design decay, code decay [13],[14]. Although the term architecture erosion is the most preferred one in literature [3] and was first introduced by Perry and Wolf [15] in 1992. However, the different terms point out, that the phenomenon can be found on different abstraction levels of software systems [5]. While architecture erosion describes erosion on the abstract level of software architecture, the term code decay describes deviations on the more concrete code level [3]. Nevertheless, all discussions in literature agree that software degeneration is a consequence of adaptations that violate design principles [5]. In line with the distinction between software architecture and software design in the previous section, the term design erosion is used in this thesis, as the approach presented deals with design at code level.

2.3 Conformance Checking

As Lehman already described in his "laws of software evolution" back in the mid-1970s,

> "as a program evolves, its complexity increases unless work is done to maintain or reduce it." [16]

Therefore effort is required to preserve the structure of a system [4]. There are various strategies for preventing the erosion of architecture, which can be grouped into three main categories. The first group of approaches focuses on minimizing erosion, the second aims to prevent it, and the third seeks to repair it [5]. One approach belonging to the first group is architecture compliance monitoring also known as architecture conformance checking. In this approach the implemented architecture is continuously compared to the intended architecture to detect any design violations and therefore preserving the initial design [17]. Similar to the different terms for design erosion, the various conformance checking approaches also operate at different levels of abstraction [18]. The conformity of an as-implemented architecture to the as-intended architecture can either be checked statically, without executing the code, or dynamically at runtime [1]. Since the approach presented in this thesis performs static analysis, the further considerations will be limited to static approaches. In chapter 3, the most relevant approaches will be discussed.

2.4 Documentation and System Erosion

Many causes of erosion are discussed in literature. In addition to a lack of knowledge about the overall structure of the system, the lack of or inadequate design documentation is often cited as a cause of system erosion [12]. Conversely, appropriate documentation can help to reduce erosion [5]. Architecture descriptions or design descriptions are the documents that describe the system architecture or design [19]. Parnas [12] describes documentation as written, accurate information about the system that extracts the essential information in an abstracted representation. If the documentation does not match the described system, this indicates either a faulty system, incorrect documentation or deficiencies in both areas [20]. It is therefore desirable to synchronise the design and the code [7]. There are various documentation techniques for capturing the system structure and interactions [5]. Many of them use a series of views, each dealing with a different aspect of the system structure [21]. This modularisation increases the comprehensibility of complex software systems [9].

2.5 Modelling Notations

Software architectures can be modelled informally, e.g. in simple natural language, lines and boxes or modelling languages, or formally in well-defined and precise architectural or specification languages that define their own syntax and semantics to precisely specify design decisions [22]. As the field of modelling notations has been researched for over 20 years, there are a variety of different languages [23]. Although it is often argued that the informal, incomplete, inconsistent and ambiguous nature of the UML is not expressive enough to model complex architectures [18],[24],[25], it is the most widely used modelling language among the currently existing languages [9]. It has become the de facto standard [25] and in most projects UML diagrams are the first artefacts used to represent a software architecture [26]. This is due to the fact that users prefer the low learning curve, the visuality and the general-purpose scope [22]. On the other hand, formal modeling notations such as Architecture Description Languages (ADL) are rarely used because of the steep learning-curve, the lack of knowledge among stake-holders, the lack of popularity in industry, the poor tool support and the lack of tutorials and guidance [22].

2.6 UML Diagram Types

UML offers 13 visual diagram types for specifying several important design decisions [22]. The diagram types can be classified into two main categories. The static diagrams address structural aspects like the main components and their relations, while the dynamic diagrams describe the behavior of those components [27]. Each of these diagram types addresses a distinct aspect of software development [24]. Creating multiple diagrams for different viewpoints is essential to obtain a comprehensive model of the system under design [24]. As Jaiswal [11] notes, component, deployment and package diagrams are generally found in software architecture documents. These diagrams provide an overview of the system's high-level structure. On the other hand, class, object and behavior diagrams are crucial in more detailed design documents as they present implementation details, but still at a more abstract level than the source code itself [11],[28]. Class diagrams in particular describe the static structure of a program by showing the classes of the system along with their attributes and relations among the classes [27]. This offers a clearer understanding for implementations, architectures and design choices [28]. This abstraction bridges the gap between conceptual design and actual code, allowing developers to quickly grasp the implementation without needing to read the entire codebase. A study with 109 participants from 34 countries about the practical usage of UML for different software architecture viewpoints shows, that UML class diagrams are the practitioners' top choice for the structure of a system [9]. This preference indicates the importance of class diagrams in both designing and understanding software systems.

2.7 Docs-as-Code

Creating and maintaining appropriate documentation is a challenging task for all practitioners [29]. To counteract the neglect of documentation, many organisations adopt an approach called Docs-as-Code, [30] where documentation is created and managed using the same tools as for the code. This enables seamless integration into the development workflow and thus improves collaboration between writers and developers when maintaining the documentation. The use of plain text formats such as AsciiDoc [1] enables the versioning of documentation via Git, for example. In addition, documentation related topics can be included into issue trackers, code reviews and automated testing [31].

To facilitate the adaptation of the Docs-as-Code approach, an open source project called docToolchain [2] was launched. This is a tool for creating documentation in the form of a collection of scripts that contain, for example, the documentation template arc42 [3], scripts for creating documentation microsites or for publishing in Confluence [32]. There are several keynote talks by professional developers who talk about their adaptation of the Docs-as-Code approach, such as Amazon Web Services [33] or Google [34].

2.8 Diagrams-as-Code

Tools for text-based creation of UML diagrams are advantageous for integrating diagrams into the Docs-as-Code workflow. Since diagrams undergo continuous refinement and modification during the development process [35], formats like PNGs or PDFs would not make the modifications traceable when using version control systems. In contrast, using plain text formats,

[1] `https://asciidoc.org/`
[2] `https://doctoolchain.org/docToolchain/v2.0.x/`
[3] `https://www.arc42.de/`

```
@startuml

class Car {
- engine : Engine
+ <<Create>> Car(Engine)
}

class Engine {
+ <<Create>> Engine()
}

Car "1" --> "1" Engine

@enduml
```

Listing 2.1: Plain text example PlantUML diagram

also known as Diagrams-as-Code, makes this possible. Additionally, modifications to graphical UML diagrams often require specialized graphic tools, whereas PlantUML [4] can be edited in any simple text editor, enabling seamless integration into the development process. These reasons contribute to the growing popularity of the Diagrams-as-Code approach in recent years [36].

Currently, there is only very little scientific research specifically focusing on text-based tools for UML generation, so the remainder of this section is mostly based on online articles.

There are several tools for the text-based creation of diagrams, with PlantUML being one of them. This open-source tool was originally created by Arnaud Roques with the goal to keep documentation more up to date. Over the years, it has gathered a wide community, resulting in the existence of over 80 PlantUML plugins today [37].

PlantUML uses a specially defined domain-specific language [38] that can be automatically rendered into the corresponding graphical notation. It supports nine out of the 13 UML diagram types, including Sequence Diagrams, Use Case Diagrams, Class Diagrams, Object Diagrams, Activity Diagrams, Component Diagrams, Deployment Diagrams, State Machine Diagrams, and Timing Diagrams. The code snippet in Listing 2.1 shows a basic example of the PlantUML class diagram syntax. Figure 2.1 presents the rendered form of the code snippet.

[4]https://plantuml.com/

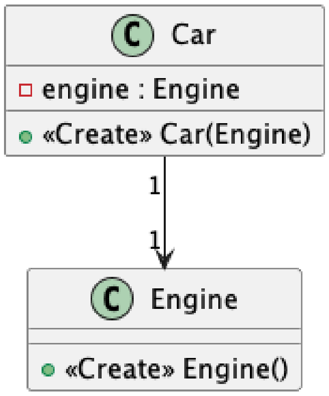


Figure 2.1: Graphical rendered example PlantUML diagram based on Listing 2.1

In addition to its Command-line interface (CLI), PlantUML offers many integrations with external tools, such as plugins for Integrated Development Environment (IDE) and text editors like IntelliJ [5], Visual Studio Code [6], and Eclipse [7]. Several online editors exist [8], [9]. Furthermore, it can be integrated into documentation formats such as AsciiDoc [10], Markdown [11], and LaTeX [12], and even used in Microsoft Word [13] or Google Docs [14] [39],[40].

Tools also exist for generating source code from PlantUML class diagrams in various programming languages like Java, C++, PHP, and SQL [41]. Additionally, diagrams can be generated from existing source code using plugins like the PlantUML Parser for Java source code [42].

Although PlantUML has been one of the most popular tools for text-based UML generation [37], Mermaid [43] has gained much popularity in recent years. This can be observed above all when comparing the GitHub statistics: As of August 2024, Mermaid has 69k stars on GitHub and 6.1k forks [44], while PlantUML only has 10k stars and about 900 forks [45]. As PlantUML has

[5] https://plugins.jetbrains.com/plugin/7017-plantuml-integration
[6] https://marketplace.visualstudio.com/items?itemName=jebbs.plantuml
[7] https://plantuml.com/de/eclipse
[8] https://plantuml.com/de/eclipse
[9] https://sujoyu.github.io/plantuml-previewer/
[10] https://asciidoc.org/
[11] https://markdown.de/
[12] https://www.latex-project.org/
[13] https://www.microsoft.com/de-de/microsoft-365/word?market=de
[14] https://www.google.de/intl/de/docs/about/

been around for much longer, it has an extensive ecosystem and external plugins offer functions for both generating source code from diagrams and creating diagrams from source code, which enables code round-tripping [36]. Code round-tripping enables generating code from models as well as automatically update the models when changes are made to the code [7]. During the research, no Mermaid-compatible plugins providing this functionality were identified.

Chapter 3

Related Work

There are various methods to control the erosion of system designs, which can be categorized based on whether they aim to minimise, prevent, or repair erosion [5]. Each of these categories is further divided into specific strategies. Architecture compliance monitoring, also known as architecture conformance checking, falls under the "minimise" category, as it seeks to reduce erosion during system development [5]. Architectural conformance can be verified either statically, without executing the code, or dynamically, at runtime [1]. Since this thesis focuses on a static design conformance approach, the following sections will explore different methods for conformance checking that are frequently discussed in the literature. For this purpose, three widely cited papers [1],[6],[5] were reviewed, each providing an overview of static architecture compliance checking approaches, to identify the most popular approaches. A common approach mentioned across all these papers is the Reflexion Model technique [1],[6],[5]. Source code query languages for conformance checking are noted in Silva and discussed in [6], while relation conformance rules and component access rules are covered only in [1]. Dependency structure matrices are exclusively described by [6]. Each of these approaches will be briefly described below.

3.1 Reflexion Models

The Reflexion Model (RM) technique was originally introduced by Murphy et al. [46] in 1995 to enhance the understanding of existing software systems by summarizing a source model from the perspective of a high-level model. By substituting the expected high-level model with an intended high-level model, RM can also be used for checking design conformance [46].

This technique requires developers to construct a high-level model, typically representing the intended or planned architecture [1], which includes the system's main components and their relationships [6]. A source code model, usually a call graph or inheritance hierarchy, has to be extracted from the implementation [46]. The declarative mapping between the source model and the high-level model must be created and maintained manually [6]. Based on the two models, a reflexion model is automatically generated, highlighting discrepancies between the models [46]. Reflexion models include convergences, divergences, and absences. Convergences mark areas in which both models agree. Divergences indicate elements present in the source model but not predicted by the high-level model, while absences point out parts predicted by the high-level model but missing from the source model [46].

The primary distinction of RM compared to other conformance checking approaches is that the construction of the idealized architecture is explicitly left to the architects, rather than being automatically generated. This approach gives architects full control over the granularity and abstraction level of the components [6].

Several adaptations of reflexion models exist, such as the inverted reflexion models citeRosik and the hierarchical reflexion models [47].

Reflexion models can identify unintended dependencies and interface misuse. They can also detect violations of architectural styles, such as layered architecture, when the architects formalize the architectural style in the high-level model.

3.2 Dependency-Structure Matrices

Dependency Structure Matrices (DSM) were originally developed to optimize product development processes [48]. Sullivan et al. [49] recognized their potential for software, especially in the evaluation of design trade-offs. Sangal et al. [48] were in 2005 the first to use DSM to manage dependencies between modules in software systems.

A DSM can summarize the dependency relationships between program elements, such as modules or classes in object-oriented systems. The rows and columns of the matrix contain these components. Dependencies between the components are indicated by an "x" or by the number of references between the components in the corresponding matrix cell [6].

To use DSM for conformance checking, design rules such as "A can use B" or "A cannot use B" can be defined for the target system. Tools such as the Lattix Dependency Manager (LDM) can then automatically detect violations of these predefined rules [6]. Violations can be displayed visually in the extracted DSM. The DSM can also be reorganized by applying partitioning algorithms, which can reveal the "most used" modules [5].

The main advantage of DSM is their simplicity, as they provide a compact and effective abstraction for the visualization of architectures. However, DSM are limited in their expressive power as they can only indicate the presence or absence of dependencies [18]. Despite this limitation, Passos et al. [6] describe DSM as a useful abstraction for visualizing software architectures, although the language of design rules remains somewhat limited [6].

3.3 Source Code Query Languages

A Source Code Query Language (SCQL) is typically used to enforce coding conventions, identify bugs, compute software metrics, and detect refactoring opportunities [50]. However, they can also be applied to check architectural conformance. Passos et al. [6] present an approach where a specific SCQL is utilized. This involves defining a set of predicates in a Structured Query Language (SQL)-like syntax, which are then used to determine whether architectural constraints have been violated in the source code.

SCQL does not require architects to define high-level models, eliminating the need for model construction. Although this approach sacrifices abstraction, it has a high level of expressiveness, which enables a detailed analysis of the source code [18].

3.4 Relation Conformance Rules and Component Access Rules

Conformance rules for relationships make it possible to define permissible and impermissible relationships between components. These rules can detect defects similar to those identified by reflection models, but with the advantage of automatic mapping. A key advantage of conformance rules for relationships is their ability to take external libraries or third-party software into account, which can detect problems such as the misuse of User Interface (UI) libraries by components that are not related to the user interface [1].

Component access rules on the other hand, allow the specification of ports, through which components are allowed to interact. The main purpose of these rules is to detect violations of information hiding [1].

3.5 Summary

As can be seen, there are many different approaches to static design conformance checking. They differ both in the level of abstraction of the model and the types of defects that can be detected. Since none of these approaches recognizes all possible violations, Knodel et al. [1] recommend a goal-oriented approach to select the most suitable approach for the respective application scenario.

Chapter 4

arcucheck - Design and Implementation

The design of the prototype tool named "arcucheck" which is based on the findings from chapter 2 and 3 is presented in this chapter. It also includes the description of implementation details that are relevant for understanding the subsequent discussions.

4.1 Goal

The aim of the presented tool *arcucheck* is to evaluate the usefulness of using a PlantUML-based approach for static design conformance checking. Therefore, it is implemented as a prototype to evaluate its limitations and potential for further development.

The tool should draw attention to deviations between the implementation and the planned design by warning if any deviations are detected. This should mainly motivate the developers to keep the documentation and especially the design diagrams consistent with the implementation. Since poorly maintained documentation is one of the main causes of system erosion, a stronger focus on documentation could help reduce erosion.

Of course, the design diagrams could also be automatically adapted after each code adaptation and transferred to the documentation. This would keep usage costs even lower than simply warning of deviations. But then deviations are covered up and the system design moves further and further away from the original design without anyone noticing. If its only warned about deviations without automatically adapting the design or the code, it is up to the developers themselves to decide whether they want to make adjustments to the divergent implementation or they want to adapt the design to the implementation. In this way, the developers are sensitised to the parts of the system where deviations frequently occur and if, for example, it would make sense to adapt the design.

4.2 Presentation of the Design Decisions

As already mentioned in chapter 1, the aim is to implement a tool for checking design conformity that minimises the hurdle for use. This can be achieved by minimising the effort for the user. Developers are identified as the primary target group since they are the ones who need to be most aware of existing deviations, as they are responsible for adjusting the code accordingly. Seamless integration into existing development processes help reduce effort for usage. Eliminating the need for training to use the tool is also helpful and can be achieved by using widely used modeling notations to specify the intended design. The following sections outline the design decisions made, to address these requirements and explain the rationale behind each decision.

4.2.1 Determination of the Integration Strategy

To integrate the tool into existing development processes, it could either be developed as an IDE plugin or structured to allow integration with Continuous Integration/Continuous Deployment (CI/CD) pipelines. Both would allow automated, regularly conformance checks. Since achieving pipeline integration compatibility is more straightforward, this approach was chosen for the prototype implementation. To facilitate easy integration into pipelines, the tool should provide a CLI.

4.2.2 Determination of the Output Format

The different approaches for conformance checking presented in chapter 3 produce various types of outputs. Dependency Structure Matrices are, as the name suggests, produce matrix-based representations. Reflexion models, relation conformance rules, and component access rules, for example, can be integrated into the Software Architecture Visualization and Evaluation (SAVE) Eclipse plugin developed by Fraunhofer Institute for Experimental Software Engineering (IESE), which generates graphical representations with icons indicating deviations. While CI/CD pipelines are capable of creating artifacts in almost any format, this would limit the tool to producing graphical outputs as well. However, since this is a prototypical implementation, a simple plain text output in the form of warnings to the console was found to be the easiest to implement while still effectively presenting details about detected deviations.

4.2.3 Determination of the Design Model Format

In order to detect any discrepancies between the design and implementation, the design must be compared with the implementation. This requires a suitable design model that can then be compared with a model representing the implementation.

Among the several existing modelling notations the most preferred one should be chosen since one of the main goals of the tool presented is to minimise the usage effort. Requiring users to learn an unfamiliar, rather complex modelling notation to be able to use the tool would therefore run counter to this goal. Based on the findings on modelling notations in section 2.5, UML was found to be the most suitable.

As many companies adopt the Docs-as-Code approach to address the neglect of documentation by reducing its maintenance costs, it would be advantageous for the tool to align with this approach. Consequently, text-based UML diagrams are preferable. This also includes the ability to version control the design models. PlantUML offers a broad ecosystem with extensive tool support and plugins that facilitate integration into existing development processes, as well as the support for code round-tripping. This is why the approach presented in this paper is based on PlantUML.

4.2.4 Determination of the UML Diagram Type

Since the tool's goal is to warn about deviations of the structure, structure UML diagram types were investigated. Available structure diagram types are Class Diagram, Object Diagram, Component Diagram, Composite Structure Diagram, Package Diagram, and Deployment Diagram [51]. Since the deployment diagram type illustrates the nodes in a distributed system [52] it is not suitable for this approach. Similarly, object diagrams, which capture a snapshot of the system at a specific moment in time, are also unsuitable since this approach is static and does not involve code execution [52]. This leaves class diagrams, component diagrams, composite structure diagrams, and package diagrams as options. Among these, class diagrams are the preferred choice of practitioners [9], aligning with the design goal of adopting familiar modeling notations. Therefore, the approach presented will utilize PlantUML class diagrams.

4.2.5 Determination of the Warning Structure

A meaningful message about the detected deviation should be provided to achieve a high expressiveness. The warnings should contain the previous presented categorisation, meaning every warning should contain the deviation subject, deviation type and deviation level. Additionally, it should point out the affected source code file and design diagram file. The warning messages are based on research about error warnings, ensuring that messages are constructed for humans, not tools, using understandable natural language. Therefore each warning contains a message describing the cause of the deviation. Since the developers should decide themselves how to treat the detected deviations, no instructions for fixing the warning are provided unlike it is recommended by guidelines about error messages [53]. Each detected deviation is described by a single warning, output below each other to the console. At the top of the single warnings, some basic statistics are provided including for example the amount of overall detected deviations per deviation level. The defined structure for warnings cam be seen in Listing 4.1.

```
Detected deviation <title>:
Level: <deviation level>
Subject: <deviation subject>
Type: <deviation type>
Cause: <message>
Affected design diagram: <path to affected .puml file>
Affected implementation at: <path to affected .java file>
```

Listing 4.1: Defined structure of a warning for a detected deviation

4.3 Deviation Analysis

Based on the capabilities of the design models in form of PlantUML class diagrams, it was analysed, what types of deviations can be detected. In addition, it was investigated how other conformity checking approaches categorise the detected deviations. Finally, it was analyzed which specific deviations could potentially be detected when comparing UML diagrams with their implementation. The result of this investigation is a set of lists that contain categorized deviations.

4.3.1 Identification of Deviation Subjects

In order to determine possible deviation areas, the components of a UML class diagram were first analysed. Based on Genero [54] the following components were identified:

- Packages
- Classes
- Each class has attributes and operations
- Attributes have their names and types
- Operations have their signatures, including their names, parameter definitions and return type
- Relationships: Association, Aggregation, Generalization, and Dependencies

To identify all potential deviations, these components were adapted. To enable a more detailed analysis, interfaces and constructors have been added. Attributes and operations have been transferred into fields and methods respectively, which is more in line with the terms used in object-oriented programming languages. The below list contains the final potential so-called deviation subjects, which form the basis for the implementation of deviation detection. It must be noted, that the term “subject” is used in the sense of “object”. The term “deviation object” was avoided due to potential confusion in an object-oriented environment. Identifying these subjects is crucial, as they form the foundation for exploring all potential deviations. The subjects define “where” a deviation can occur. The next step is to investigate the types of differences that can exist between the implementation subjects and design subjects.

- Package
- Class
- Interface
- Method
- Constructor
- Field
- Relation

4.3.2 Identification of Deviation Types

After analyzing the constructs, referred to as subjects, where deviations can occur, the next step is to investigate the types of differences that can arise between the implementation and UML subjects.

As Knodel et al. [1] describe in their paper, the three approaches analysed lead to each relationship between two components being assigned one of the following types: Convergence,

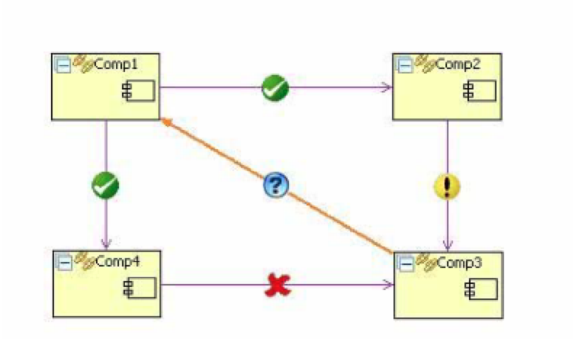


Figure 4.1: Graphical representation of typed relations,
Source: adapted from [1]

Divergence and Absence. Convergences describe correctly implemented relations, absences describe relations that are intended but not implemented and divergences describe relations that are not allowed or incorrectly implemented [1]. Figure 4.1 shows a graphical representation of these types. Green check marks indicate a convergence, black exclamation marks indicate divergences, absences are marked with a red cross and blue question marks indicate that more then one of these types apply [1]. As can be seen, these approaches are limited to analysing the relationships between components. Prujit et al. [55] present in their work deviation types that include property rule types in addition to relationship rule types. The relationship rule types are similar to those specified by Knodel. The property rule types include rules that constrain the elements in a module, including naming, visibility, responsibility, inheritance and facade conventions.

Based on Knodel's and Prujit's categorisation of relationship and property deviations, three deviation types are introduced. **Absent** deviations describe properties or relations that are missing in the implementation but are expected by the design. **Unexpected** describes the opposite case: properties or relations are present in the implementation that are not expected by the design. **Misimplemented** describes incorrectly implemented properties or relations. Misimplemented includes for example the property rule type visibility by Prujit et al. It should be noted that the term 'misimplemented' is not an official term and cannot be found in dictionaries. Since the deviation types will be used in the implementation, a single-word term is preferable to a phrase like "wrongly implemented" or "incorrect implementation". Therefore, it was decided to use this unofficial term.

4.3.3 Identification of Deviation Levels

Classes can exist on different levels of meaning in a model. This includes the analysis, design and implementation level [52]. As different use cases may be interested in different levels, the deviations identified are categorised based on their level of abstraction. **Macro-level** deviations relate to the higher design level, while **micro-level** relate to deviations at the lower level of implementation details. This enables filtering, if the details of the implementation are not of interest.

4.3.4 Identification of Potential Deviations for Systematic Analysis

The identification of potential deviation aims to provide a base for the systematic evaluation of the deviation detection capabilities of the presented approach. Each detected deviation will be assigned a deviation level, a deviation type and a deviation subject. This is intended to provide the structured classification of detected deviations.

The tables A.1, A.2, A.3, A.4, A.5, A.6 and A.7 present all potential deviations identifiable when comparing UML diagrams with their implementations based on the results from the previous sections, that were found, clustered by deviation subject. It is important to note that this list does not imply that the approach presented can detect all these deviations. Chapter 5 will illustrate the deviations, that can be detected, along with the reasons for any that can not be detected.

The deviations of the misimplemented type are further subdivided. These subtypes represent the reasons why a deviation could be labelled as misimplemented if the UML subject is examined in isolation. Isolated examination means the examination of the subjects without taking relations into account. Some of these deviations are ambiguous in their classification. For example, if a class is missing methods, the class would be labelled as misimplemented. In addition, these missing methods would also belong in the table of 'potential method deviations' as absent methods. This duplication must be resolved in the implementation. However, as this table is intended to be implementation-independent, the duplicates are listed anyway.

Deviations that relate to implementation details of classes and interfaces are assigned to the micro level, while all other deviations are assigned to the macro level.

4.4 Implementation

The following sections outline the most important implementation details of arcucheck that are relevant for understanding further analyses. For further details, see the source code in the GitLab repository [1] also listed in [56].

4.4.1 Presentation of the Technology Stack

The programming language Kotlin was used to implement arcucheck. This is due to the fact that Java is well known to the tool developer and Kotlin offers some useful functions on top of Java. Furhtermore, Kotlin's interoperability with Java makes it possible to include Java source code, which was necessary for the integration of the external PlantUMLParser plugin [2], which is described in section 4.4.4. The PlantUMLParser library is the reason why arcucheck operates on Java source code, as it extracts PlantUML diagrams from Java files. By integrating a different parser library, arcucheck could also process other languages, given that its remaining implementation is language-independent. Picocli [3] is an open-source framework for creating command line applications that can run on the JVM. Therefore, it supports Kotlin among other JVM languages. It simplified the development of the required CLI for arcucheck since it

[1] https://gitlab.mi.hdm-stuttgart.de/ps149/arcucheck
[2] hhttps://plugins.jetbrains.com/plugin/15524-plantuml-parser
[3] https://picocli.info/

provides functionalities like argument parsing, help and error messages out of the box. The Koin [4] framework was used for dependency injection.

4.4.2 Presentation of the Program Sequence

The basic procedure of arcucheck is as follows:

1. The source code is parsed into PlantUML diagrams.
2. PlantUML diagrams representing the design are compared with the PlantUML diagrams representing the implementation.
3. All deviations found are collected and output to the user.

The sequence diagram 4.2 shows a more granular program flow of arcucheck. Details of the processing within the classes are omitted for reasons of clarity and relevance. The following sections describe selected implementation details of the core classes `FileHandler`, `CodeParser`, `PUMLMapper` and `PUMLComparator` and `ResultPrinter`.

4.4.3 `FileHandler`: Resolving Design-to-Implementation Mapping

As can be seen in Figure 4.2, the user only enters the path to the directory that contains the documentation for the system to be checked. All .puml files are automatically extracted and processed. As already mentioned, the mapping between the design and the associated code part is not a trivial task and therefore difficult to automate, which is why it must be carried out manually by the users. To the mapping process at least simple and organized, the associated code part for each diagram must simply be specified as a comment at the beginning of each PlantUML diagram in the form of a path to the code part. A code part can be a single .java file or a package that may contain sub-packages. The paths to the source code files are then automatically extracted and processed together with the design diagrams. Listing 4.2 shows an example of this code path annotation.

[4] `https://insert-koin.io/`

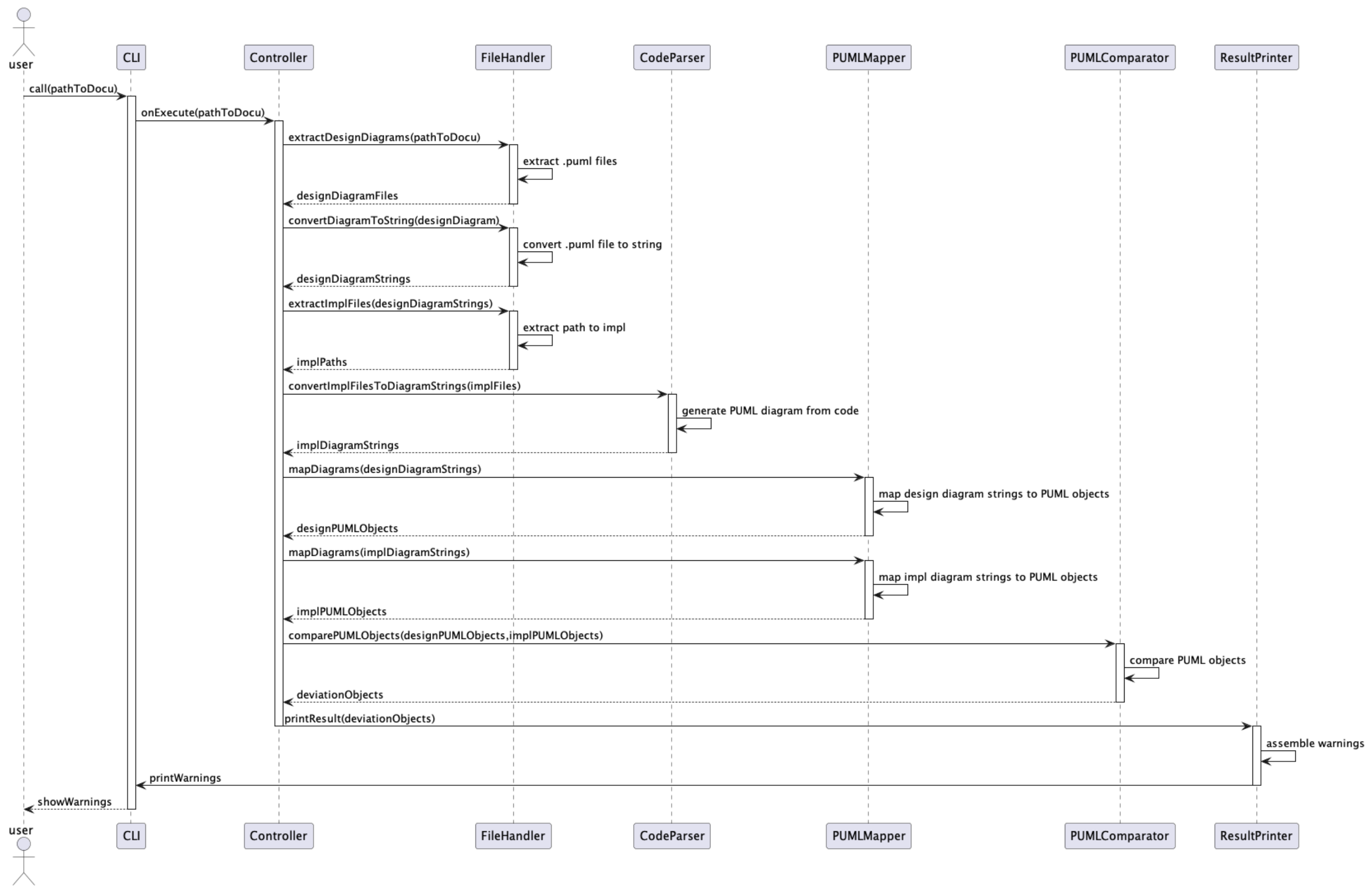

user
CLI
Controller
FileHandler
CodeParser
PUMLMapper
PUMLComparator
ResultPrinter
call(pathToDocu)
onExecute(pathToDocu)
extractDesignDiagrams(pathToDocu)
extract .puml files
designDiagramFiles
convertDiagramToString(designDiagram)
convert .puml file to string
designDiagramStrings
extractImplFiles(designDiagramStrings)
extract path to impl
implPaths
convertImplFilesToDiagramStrings(implFiles)
generate PUML diagram from code
implDiagramStrings
mapDiagrams(designDiagramStrings)
map design diagram strings to PUML objects
designPUMLObjects
mapDiagrams(implDiagramStrings)
map impl diagram strings to PUML objects
implPUMLObjects
comparePUMLObjects(designPUMLObjects,implPUMLObjects)
compare PUML objects
deviationObjects
printResult(deviationObjects)
assemble warnings
printWarnings
showWarnings

Figure 4.2: Simplified sequence diagram describing the basic program flow of arcucheck

```
'implementation_path=[project/arcudoc/de.hdm_stuttgart.login/src/main/java/
    de/hdm_stuttgart/login/]

@startuml

'PlantUML diagram content omitted for reasons of relevance

@enduml
```

Listing 4.2: Example of the code path annotation in a PlantUML diagram

4.4.4 `CodeParser`: Implementation-to-PlantUML Parsing

Since the implementation is entered as paths to source code files, it must be parsed into a PlantUML diagram first in order to be able to compare it with the PlantUML design diagrams. An open-source plugin for IntelliJ called PlantUMLParser [42] was used to automatically generate the PlantUML diagrams. Since the source code of the plugin is available on GitHub, its core functionality could be integrated into arcucheck without having to use the plugin user interface. The parser is therefore called directly from the Kotlin codebase. During the development of arcucheck, it was discovered that the PlantUMLParser does not extract some properties from the code and are consequently not displayed in the generated PlantUML diagrams. This mainly concerns associations between classes. The shortcomings resulting from this with regard to the overall functionality of the tool are explained in more detail in the evaluation chapter.

4.4.5 `PUMLMapper`: PlantUML Model-to-Object Mapping

Several approaches were considered for comparing the design PlantUML diagram to the auto generated PlantUML diagram representing the implementation. One option was to use a diff-based approach to compare plain text, which would have made it challenging to categorize the differences.

The chosen and ultimately implemented approach was to convert the PlantUML models into Kotlin objects and then compare these objects. To achieve this, a hierarchy of data classes representing the PlantUML diagrams were created. To simplify further discussions, the data classes are referred to as "PUML classes" in the following sections.

The top-level data class called "PUMLDiagram" can be seen in Listing 4.3. It consists of sub PUML classes, which in turn consist of further sub PUML classes. The hierarchy continues until the properties can be represented by primitive data types, like represented by the PUMLField data class in Listing 4.4.

The Kotlin objects are created by extracting the properties from the PlantUML diagrams using regular expressions. Regex offers the option of grouping the extracted characters. For

```
data class PUMLDiagram(
    val sourcePath: String,
    val classes: List<PUMLClass>,
    val interfaces: List<PUMLInterface>,
    val relations: List<PUMLRelation>
)
```

Listing 4.3: PUMLDiagram data class structure

```
data class PUMLField(
    val name: String,
    val dataType: String,
    val visibility: Visibility, //enum constant for the visibiltiy
    val isStatic: Boolean
)
```

Listing 4.4: PUMLField data class structure

example, method signatures can be grouped into character groups for the method visibility, method return type, method name and method parameters. The character groups can then be accessed separately and be used to easily build the PUML class object from the previous described section. An example for a regular expression used, is presented in Listing 4.5.

```
/**
 * Pattern to extract method character groups from a PlantUML class string
 * - group 0: whole method signature
 * - group 1: method visibility
 * - group 2: isStatic or isAbstract or nothing
 * - group 4: method return type
 * - group 5: method name
 * - group 6: method parameters
 */
 const val EXTRACT_METHOD = """([+\-#~])?\s*(\{(?:static|abstract)?})?\s
    *((\w+(?:<[\w<>]+>)?)\s+(\w+)\((.*?)\))"""
```

Listing 4.5: Example regular expression with groups

```kotlin
data class Deviation(
    val level: DeviationLevel,
    val subjectType: DeviationSubjectType,
    val deviationType: DeviationType,
    val affectedClassesNames: List<String>,
    val title: String,
    val description: String,
    val affectedDesignDiagramPath: String,
    val affectedImplementationPath: String,
)
```

Listing 4.6: Deviation data class structure

```
Detected deviation "Unexpected class":
Level: MAKRO
Subject: CLASS
Type: UNEXPECTED
Cause:  Class "ClassOccurrenceB" found in "testInput.klasses.a.
   classOccurrence" is present in the implementation, but not expected
   according to the design.
Affected design diagram: src/test/kotlin/testInput/klasses/b/
   classOccurrence/ClassOccurrence.puml
Affected implementation at: src/test/kotlin/testInput/klasses/a/
   classOccurrence/ClassOccurrenceB/ClassOccurrenceB.java
```

Listing 4.7: Example of a warning for a detected deviation

4.4.6 `PUMLComparator`: Comparison of PUML objects

Mapping the input models to the PUML objects simplifies the comparison process. The design PUML objects are compared to the implementation PUML objects. When discrepancies are detected, they are further investigated to identify the exact cause. Possible causes of discrepancies are listed in the tables A.1 - A.7. Based on the identified causes and additional information, deviation objects are created. These deviation objects are then collected for further processing. The information that a deviation object contains is shown in Listing 4.6.

4.4.7 `ResultPrinter`: Warning Assembling and Output

The deviation objects then enable a convenient compilation of the warnings with the structure shown in 4.2.5, which are then output to the user. An example warning is shown in Listing 4.7

Chapter 5

Evaluation

The evaluation of the tool aims to determine which design aspects can be analyzed using the presented approach. Additionally, this evaluation will explore the practical advantages and limitations of the tool.

5.1 Methodology

To explore which design aspects can be analyzed, an automated unit test will be created for each potential deviation presented in 4.3.4. The results of all unit tests will be summarised in a test report to enable systematic analysis. The report will be presented in section 5.3.

Practical limitations and advantages will be investigated by applying the presented approach to a test project. The application will then be evaluated with evaluation criteria, introduced by Knodel et al. [1]. This aims to provide insights into the practical capabilities and shortcomings of the approach, as well as a comparison to other conformance-checking methods. The test project used is called "arcudoc" and was developed four semesters ago as part of the Software Development Three course. It contains six Java modules, and a PlantUML design diagram was subsequently created for each module in order to be able to test the approach. Given the large number of classes, the PlantUML diagrams were autogenerated, leading to a lack of associations between the classes. To address this issue, the PlantUML diagram for one module was manually adjusted to include the missing associations. Adjusting only one module was deemed sufficient, as the approach application should mainly investigate other aspects than the detectable deviations as they will be tested separately with the methodology presented in the section 5.2. The source code of the test project arcudoc, as well as the PlantUML diagrams and the CI/CD pipeline setup can be found in this GitLab repository [1] also listed in [57].

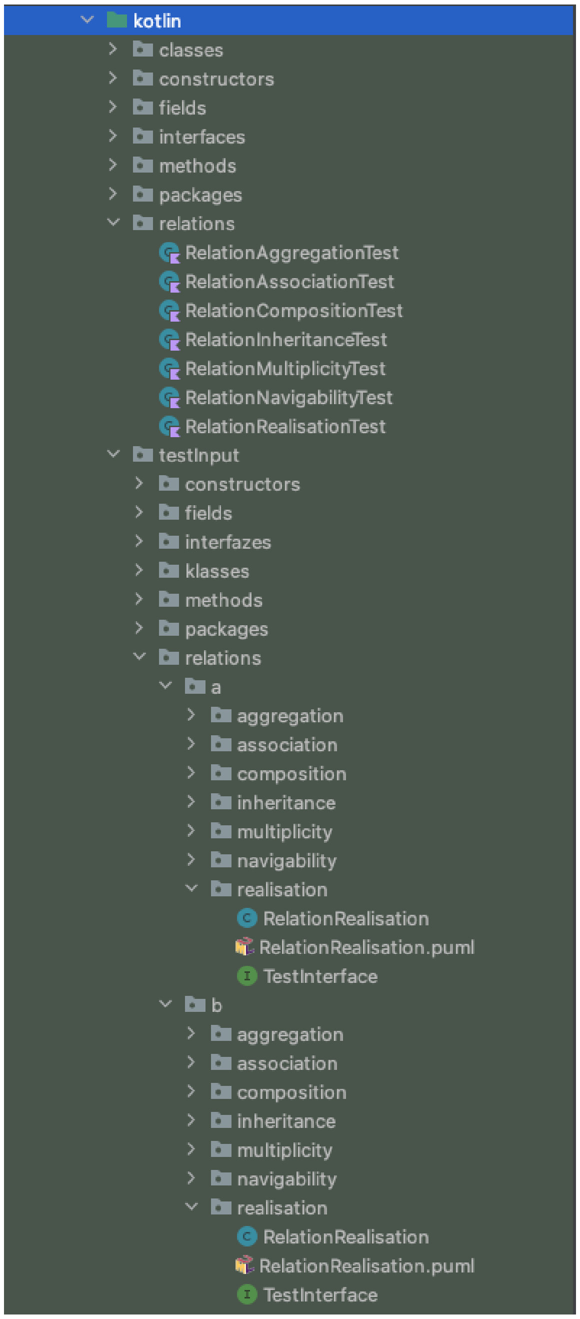
5.2 Presentation of the Unit Test Design

For each potential deviation listed in the section 4.3.4, a corresponding unit test class is created. Appendix B contains tables showing which deviation is covered by which test class. Each test class includes two test cases: one for convergences and one for divergences. The convergence test ensures that no deviations are detected when the design and implementation align, while the divergence test ensures that deviations are detected when the design and implementation do not match.

A separate test input is created for each test class. There are two packages, `a` and `b`, each containing a Java class and a PlantUML diagram, which serve as the program inputs. These classes and diagrams are minimal, focusing on a single design aspect whenever possible. Within each package, the class and diagram are convergent, meaning the implementation matches the design. However, there are intentional differences between the test pairs in package a and package codeb. This means that when comparing Java class `a` with design diagram `b`, divergences should be detected. To enhance understanding, the test design will be illustrated using an example test case.

As shown in Figure 5.1, the test input for relation deviation testing consists of sub packages `a` and `b`. Each sub package contains a test Java class and a corresponding PlantUML design diagram. For relation testing, an additional class or interface was created to serve as the relation class. All classes and design diagrams are identically named, differing only in their package containment. This naming consistency is essential to prevent unintended deviations from being detected. For most test cases, the testing of package containment was disabled

[1] https://gitlab.com/piamarie/arcudoc



```
kotlin
    classes
    constructors
    fields
    interfaces
    methods
    packages
    relations
        RelationAggregationTest
        RelationAssociationTest
        RelationCompositionTest
        RelationInheritanceTest
        RelationMultiplicityTest
        RelationNavigabilityTest
        RelationRealisationTest
    testInput
        constructors
        fields
        interfazes
        klasses
        methods
        packages
        relations
            a
                aggregation
                association
                composition
                inheritance
                multiplicity
                navigability
                realisation
                    RelationRealisation
                    RelationRealisation.puml
                    TestInterface
            b
                aggregation
                association
                composition
                inheritance
                multiplicity
                navigability
                realisation
                    RelationRealisation
                    RelationRealisation.puml
                    TestInterface
```

Figure 5.1: Screenshot of the package structure for the test classes and test inputs

```java
public class RelationRealisation implements TestInterface {
}
```

Listing 5.1: Example test input Java class `RelationRealisation` of package a

```java
public class RelationRealisation {
}
```

Listing 5.2: Example test input Java class `RelationRealisation` of package b

to avoid package-related deviations. The test input classes for testing relations of the type realisation, the classes are structured as can be seen in the Listings 5.1 and 5.2:

The design PlantUML diagrams look like this:

In package `b`, the realisation relation between the test interface `TestInterface` and the test class `RelationRealisation` is missing. Therefore, when comparing design diagram `a` with implementation `b`, a deviation indicating an "absent relation of type realisation" should be detected. Conversely, when comparing design diagram `b` with implementation `a`, a deviation indicating an "unexpected relation of type realisation" should be detected. However, comparing implementation `a` with design `a`, and implementation `b` with design `b`, should not result in any deviations.

The two test functions are therefore structured as can be seen in the Listings 5.5 and 5.6, with the setup details omitted for relevance:

All other test classes and test inputs are structured in the same way as the one just described. Based on all unit test cases, a test report is created that contains information about failed and passed tests. Failed tests indicate that the tool is not able to recognise the tested deviation type.

```
@startuml
interface TestInterface{}
class RelationRealisation{}

TestInterface  <|.. RelationRealisation
@enduml
```

Listing 5.3: Example test input PlantUML code `RelationRealisation` of package a

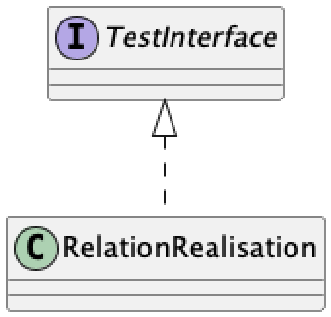


Figure 5.2: Rendered PlantUML diagram of Listing 5.3

```
@startuml
interface TestInterface{}
class RelationRealisation{}
@enduml
```

Listing 5.4: Example test input PlantUML code `RelationRealisation` of package b

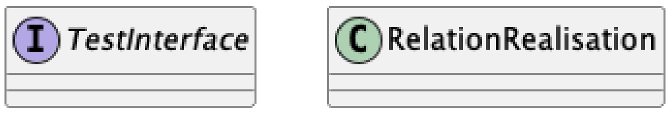


Figure 5.3: Rendered PlantUML diagram of Listing 5.4

```
@Test
  fun
 divergentRealisationRelation_reportsDeviation_ofTypeUnexpectedAbsent() {

      val resultDeviationsA = controller.onExecuteCommandTest(implClassA,
  designClassB, TEST)
      val resultDeviationsB = controller.onExecuteCommandTest(implClassB,
  designClassA, TEST)

      assert(resultDeviationsA.size == 1)
      assert(resultDeviationsB.size == 1)

      val resultDeviationA = resultDeviationsA[0]
      val resultDeviationB = resultDeviationsB[0]

      assert(resultDeviationA.deviationType == DeviationType.UNEXPECTED)
      assert(resultDeviationA.subjectType == DeviationSubjectType.
 RELATION)
      assert(resultDeviationA.level == DeviationLevel.MAKRO)
      assert(resultDeviationA.affectedClassesNames.contains(testClassName
 ))
      assert(resultDeviationA.description.contains("REALISATION"))

      assert(resultDeviationB.deviationType == DeviationType.ABSENT)
      assert(resultDeviationB.subjectType == DeviationSubjectType.
 RELATION)
      assert(resultDeviationB.level == DeviationLevel.MAKRO)
      assert(resultDeviationB.affectedClassesNames.contains(testClassName
 ))
      assert(resultDeviationA.description.contains("REALISATION"))

  }
```

Listing 5.5: RelationRealisation divergence test case

```
@Test
  fun convergentRealisationRelation_reportsNoDeviation() {
      assertEquals(emptyList(), controller.onExecuteCommandTest(
 implClassA, designClassA, TEST))
      assertEquals(emptyList(), controller.onExecuteCommandTest(
 implClassB, designClassB, TEST))
  }
```

Listing 5.6: RelationRealisation convergence test case

A distinction can be made between failed convergence test cases and failed divergence test cases. Convergence test cases that fail indicate a deficiency in the creation of the PlantUML or Kotlin models, as the models, which should normally be exactly the same, are not. Divergence test cases that fail indicate a deficiency in the comparison of the models. The test report therefore serves as a fine-grained report on the tool's capabilities and shortcomings. It should be noted that one test case per possible deviation does not cover edge cases. However, for the purpose of testing the basic capabilities and in view of the large number of test cases required anyway, this was deemed sufficient for the prototype implementation. The test report is presented and discussed in the next chapter. All unit tests can be viewed in the GitLab repository [56].

5.3 Presentation of the Unit Test Results

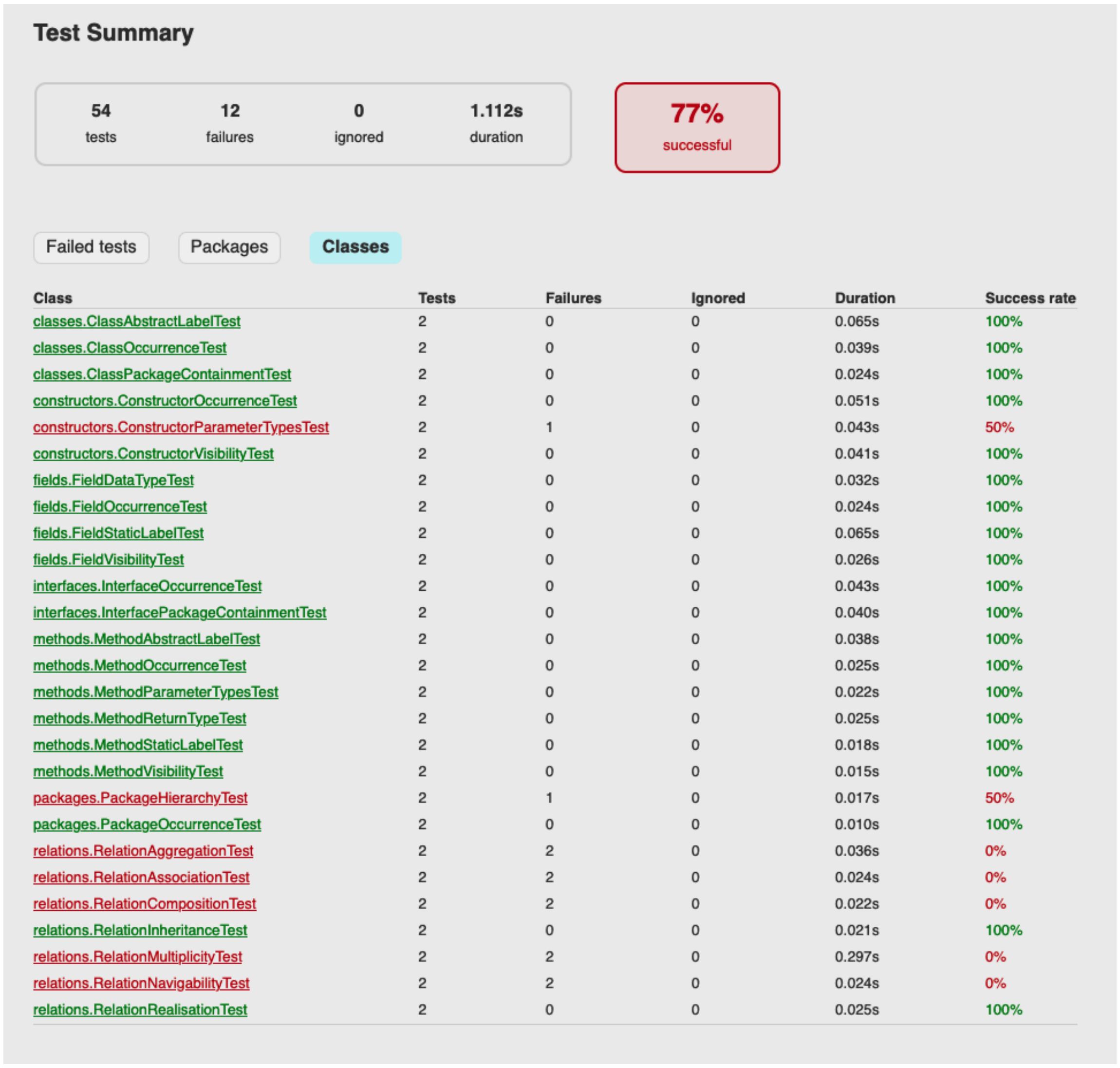
The presented test report was auto generated by Gradle [2]. To retrieve the test report, the command `./gradlew test` must to be executed in the root of the project directory of arcucheck, that is located in this GitLab repository [56].

The screenshot from the test report in Figure 5.4 shows that a total of 27 test classes were created, leading to 54 individual tests, aimed at testing all the potential deviations identified in section 4.3.4. Out of these, 12 tests failed, resulting in a success rate of 77%. This indicates that the tool is capable of testing 77% of the potential deviations.

The test cases can be divided into two categories: 26 out of 54 focused on macro-level deviations, while the remaining 28 targeted micro-level deviations. The success rate can also be analyzed separately for each deviation level. Of the 26 macro-level test cases, 11 failed, giving a success rate of 58%. In contrast, only 1 of the 28 micro-level test cases failed, resulting in a success rate of 96%.

These numbers, illustrated in Figure 5.5, show a clear trend that the tool is more effective at detecting deviations at micro-level then detecting deviations on macro-level. The primary issue in checking macro-level deviations lies in testing relations, as 10 out of the 11 failed macro level

[2] `https://docs.gradle.org/current/dsl/org.gradle.api.tasks.testing.TestReport.html`



Test Summary

54	12	0	1.112s
tests	failures	ignored	duration

77%
successful

Class	Tests	Failures	Ignored	Duration	Success rate
classes.ClassAbstractLabelTest	2	0	0	0.065s	100%
classes.ClassOccurrenceTest	2	0	0	0.039s	100%
classes.ClassPackageContainmentTest	2	0	0	0.024s	100%
constructors.ConstructorOccurrenceTest	2	0	0	0.051s	100%
constructors.ConstructorParameterTypesTest	2	1	0	0.043s	50%
constructors.ConstructorVisibilityTest	2	0	0	0.041s	100%
fields.FieldDataTypeTest	2	0	0	0.032s	100%
fields.FieldOccurrenceTest	2	0	0	0.024s	100%
fields.FieldStaticLabelTest	2	0	0	0.065s	100%
fields.FieldVisibilityTest	2	0	0	0.026s	100%
interfaces.InterfaceOccurrenceTest	2	0	0	0.043s	100%
interfaces.InterfacePackageContainmentTest	2	0	0	0.040s	100%
methods.MethodAbstractLabelTest	2	0	0	0.038s	100%
methods.MethodOccurrenceTest	2	0	0	0.025s	100%
methods.MethodParameterTypesTest	2	0	0	0.022s	100%
methods.MethodReturnTypeTest	2	0	0	0.025s	100%
methods.MethodStaticLabelTest	2	0	0	0.018s	100%
methods.MethodVisibilityTest	2	0	0	0.015s	100%
packages.PackageHierarchyTest	2	1	0	0.017s	50%
packages.PackageOccurrenceTest	2	0	0	0.010s	100%
relations.RelationAggregationTest	2	2	0	0.036s	0%
relations.RelationAssociationTest	2	2	0	0.024s	0%
relations.RelationCompositionTest	2	2	0	0.022s	0%
relations.RelationInheritanceTest	2	0	0	0.021s	100%
relations.RelationMultiplicityTest	2	2	0	0.297s	0%
relations.RelationNavigabilityTest	2	2	0	0.024s	0%
relations.RelationRealisationTest	2	0	0	0.025s	100%

Figure 5.4: Screenshot of the test report - summary

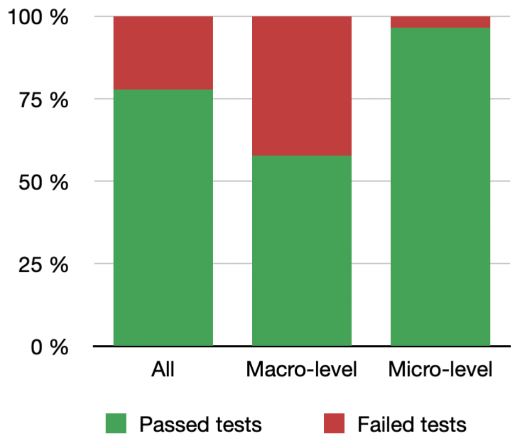


Figure 5.5: Bar chart of failed and passed tests categorised by deviation level

tests were related to this aspect, which results in a success rate of only 28%, as can be seen in Figure 5.6.

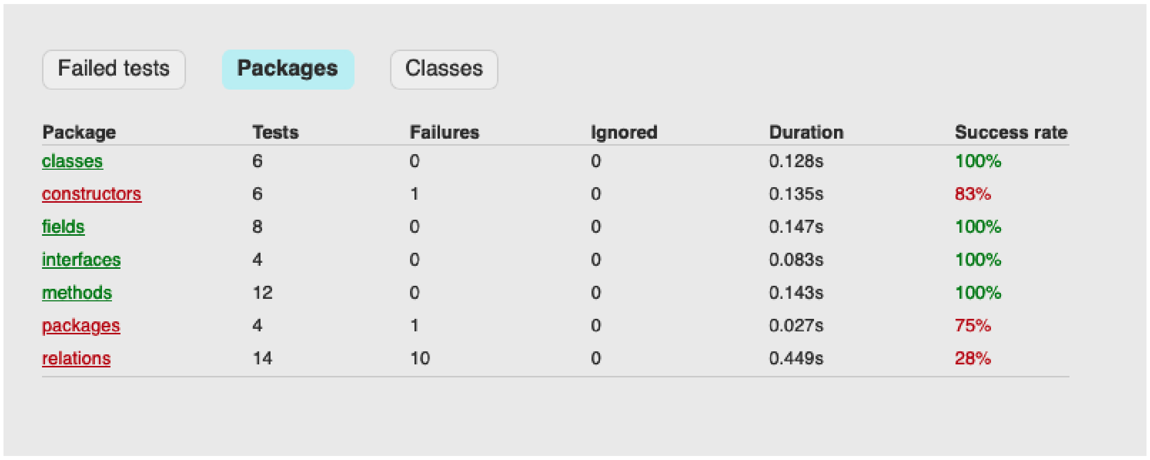
The screenshot of the failed test methods of Figure 5.7, shows that also the convergence tests do not pass for most relation related tests, which indicates a major deficiency not only in the model comparison, but also in the model mapping of relations.

5.3.1 Presentation of the Causes for Failed Tests

The reasons why tests fail and therefore the tool not beeing able to check the tested type of deviation can be divided into three categories: Defects in the PlantUML parsing from the code, defects due to ambiguities and defects based on missing implementations.

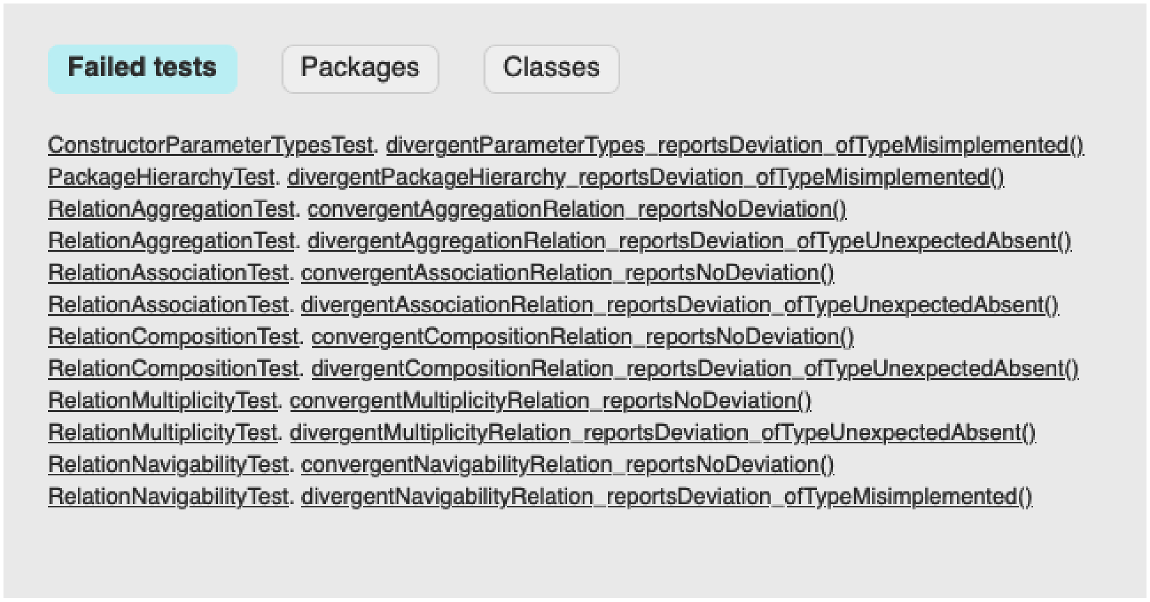
Cause: Code to PlantUML Parsing

The primary issue, responsible for most of the failures, lies in shortcomings within the code-to-PlantUML parsing process. All relation-related failures stem from the fact that the external



Package	Tests	Failures	Ignored	Duration	Success rate
classes	6	0	0	0.128s	100%
constructors	6	1	0	0.135s	83%
fields	8	0	0	0.147s	100%
interfaces	4	0	0	0.083s	100%
methods	12	0	0	0.143s	100%
packages	4	1	0	0.027s	75%
relations	14	10	0	0.449s	28%

Figure 5.6: Screenshot of the test report - categorised by deviation subject



ConstructorParameterTypesTest. divergentParameterTypes_reportsDeviation_ofTypeMisimplemented()
PackageHierarchyTest. divergentPackageHierarchy_reportsDeviation_ofTypeMisimplemented()
RelationAggregationTest. convergentAggregationRelation_reportsNoDeviation()
RelationAggregationTest. divergentAggregationRelation_reportsDeviation_ofTypeUnexpectedAbsent()
RelationAssociationTest. convergentAssociationRelation_reportsNoDeviation()
RelationAssociationTest. divergentAssociationRelation_reportsDeviation_ofTypeUnexpectedAbsent()
RelationCompositionTest. convergentCompositionRelation_reportsNoDeviation()
RelationCompositionTest. divergentCompositionRelation_reportsDeviation_ofTypeUnexpectedAbsent()
RelationMultiplicityTest. convergentMultiplicityRelation_reportsNoDeviation()
RelationMultiplicityTest. divergentMultiplicityRelation_reportsDeviation_ofTypeUnexpectedAbsent()
RelationNavigabilityTest. convergentNavigabilityRelation_reportsNoDeviation()
RelationNavigabilityTest. divergentNavigabilityRelation_reportsDeviation_ofTypeMisimplemented()

Figure 5.7: Screenshot of the test report - overview of failed test methods

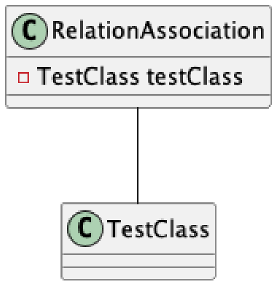


Figure 5.8: Intended PlantUML diagram for the given code of Listing 5.7

library, PlantUMLParser, is unable to extract associations from the code. This includes specific types like composition, aggregation as well as the navigability and the multiplicity.

As a result, when generating PlantUML diagrams from code that contains class relationships beyond simple interface implementation or class inheritance, those relations are completely omitted. This is also why the convergence tests fail. The PlantUML design model contains the association and the PlantUML implementation model does not contain the association, which results in divergences.

Figure 5.8 shows the expected PlantUML diagram, that should be generated from the code snippet of Listing 5.7. Figure 5.9 shows the actual diagram produced, that demonstrates that associations are not extracted at all.

```java
public class RelationAssociation {
    private TestClass testClass;
}

public class TestClass {
}
```

Listing 5.7: Minimal Java code snippet representing a simple association

The difficulty in extracting association relationships compared to inheritance and realization relationships from Java source code might explain this issue. Section 6.4 briefly describes two strategies for extracting associations.

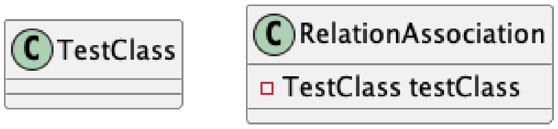


Figure 5.9: Actual generated PlantUML diagram for the given code of Listing 5.7

Cause: Ambiguities

The test `divergentParameterTypes_reportsDeviation_ofTypeMisimplemented()` in the test class `ConstructorParameterTypesTest` failed due to ambiguity. Constructors are identified solely by their visibility and parameter types, which makes it challenging to determine whether a constructor is completely absent or if the parameter types are incorrect. In the current implementation, deviations in parameter types lead to deviations being categorized as either an "unexpected" or "absent" constructor although the constructor might be existent, with different parameter types. This ambiguity is difficult to resolve. However, since the warning descriptions about constructor deviations already include parameter types, such as *"Constructor with the parameters [String, String] in the class 'ConstructorOccurrence' is not expected according to the design but is present in the implementation"*, these warnings might be sufficient for the user. Therefore, a complete resolution of this issue may not be crucial.

Similar ambiguity issues can arise when two classes or interfaces share the same name. Currently, if a class cannot be found in the expected package, the tool searches other packages for a class with the same name. If found, it outputs a warning including the description: *"According to the design, the Class "ClassPackageContainment" is expected in the package "testInput.klasses.b", but it is absent in the implementation. Instead, a class with the same name is found in the package "testInput.klasses.a". It may be in the wrong package."* The phrase *"it may be in the wrong package"* reflects the uncertainty because the scanned package might contain a class with the same name that is not the one originally sought.

Cause: Missing Implementation

Finally, some shortcomings are due to the fact that some aspects have not been implemented in

```
    @Test
    fun convergentMultiplicityRelation_reportsNoDeviation() {
        /**
         * Multiplicity is neither extracted from the implementation nor from the design, so it is ignored in both models,
         * resulting in convergent models. Since the test aims to verify multiplicity, it cannot be considered passed
         * in any case because this aspect was overlooked.
         */
        assert(false)
    }
```

Listing 5.8: Modified test case in `RelationMultiplicityTest.kt` to prevent result distortion

the prototypical development. The extraction of multiplicity from PlantUML design diagrams was not implemented because, as previously described, associations and therefore also multiplicities cannot be extracted from the code. Therefore, extracting multiplicity from design diagrams would not have resulted in any additional functionality, as the implementation models would always lack multiplicity. However, not extracting multiplicities from the design diagrams means that the convergence test between code and design for the models is passed, as both models ignore the multiplicity and therefore match. To solve this problem, the convergence test was modified as shown in Listing 5.8 to prevent distortion of the results.
In addition, the package hierarchy test failed because the package hierarchy check was not implemented for the prototype. Packages with a deviating hierarchy are directly marked as missing or unexpected. No further investigation is carried out to determine whether only one sub-package is incorrectly placed.

Furthermore, the tool does not currently support overloaded methods. While this does not cause a test to fail because edge cases have not been tested, it should still be mentioned. The reason for this is that the current tool implementation stores methods in maps that use the method name as the key. Duplicate keys, i.e. duplicate method names, are therefore not permitted. However, this could be changed by using the entire method signature as the key.

5.4 Unit Test Summary

While association relations cannot be analysed other relations including inheritance and realisation can be effectively examined. Missing classes, interfaces and packages as well as wrong package containment's can reliably be detected as well as missing and unexpected methods, fields and constructors. Furthermore deviations in visibilities, data types, return types and parameter types of class members can effectively be detected.

The systematic testing of arcucheck's deviation detection capabilities therefore demonstrated that it performs well in identifying implementation detail deviations. However, it struggles with detecting macro-level deviations, primarily due to its inability to extract association relations. The implications of this limitation for the overall usefulness of the tool will be discussed in the next chapter.

5.5 Presentation of the Evaluation Criteria

To determine which evaluation criteria are suitable for assessing the presented approach, papers evaluating static conformance checking methods were analysed. Passos et al. [6] outline four criteria: Expressiveness, Abstraction Level, Ease of Application, and Architecture Reasoning and Discovery. In contrast, Knodel et al. [1] propose 13 criteria dimensions for comparing the three investigated architecture conformance checking approaches. Given their finer granularity and the opportunity for a more detailed analysis, the 13 criteria were chosen over the four.

Like stated by Knodel et al. [1], the criteria were created with guidance of the Goal-Question-Metrics (GQM). The GQM approach is a measurement mechanism for feedback and evaluation. It therefore enables the determination of strengths and weaknesses as well as the evaluation of the quality of certain processes and products [58]. The adaptation of the evaluation dimensions presented by Knodel et al. [1] not only has the advantage that the evaluation criteria were created and refined by a group of experts from the IESE, but also that the approach presented could be categorized between the approaches discussed by Knodel et al. [1]. The 13 defined dimensions are presented below. It should be emphasised that these have been adopted exactly as they are presented in [1], even if their descriptions have sometimes been shortened.

- **Input:** the artifacts that are necessary to apply the approach
- **Involved stakeholders:** persons that are involved in the evaluation as well as their tasks
- **Manual steps:** steps that must be carried out manually to apply the approach
- **Evaluation performance:** the time required for evaluation result computation
- **Defect types:** classes of defects the evaluation approach is able to detect (unintended dependencies, context exploration, broken information hiding, misusage of interfaces, misuse of patterns, or violation of architectural styles)
- **Probability of false positives:** likelihood for false positives
- **Maintainability:** robustness of the approach with respect to code evolution like modification, addition and removal of code entities
- **Transferability:** describes how the artifacts created for the evaluation of a system can be reused for the evaluation of a new version of the same system, a variant of the same system, the system after restructuring or a different system
- **Scalability:** in which degree the approach scales up to handle large systems
- **Ease of application:** subjective user experience in intuitiveness, need of iterative refinements, and learning curve
- **Multiple View Support:** how the approach is able to handle multiple overlapping or conflicting static views of the same system
- **Restructuring Scenarios:** describes the support of conducting "what-if" analyses on the actual system to explore how architectural compliance might be affected by different structural modifications

- **Decision-making support:** how the approach supports decision making, trade-off analysis, and scenario-specific analysis when reasoning about architectural or non-functional qualities

For the evaluation, each criterion will be applied to the presented approach. It should be noted in advance that the criteria were initially introduced in order to evaluate approaches for checking architectural conformity. Since the arcucheck works at the design level, a lower level of abstraction, a comparison of some criteria may not be meaningful or even impossible. Further details on this will become apparent in the application of the criteria and the discussion of the results in the next chapter.

5.6 Presentation of the Evaluation Criteria Results

In the following sections, the application of arcucheck to the test project is analyzed using the criteria presented above.

5.6.1 Input

To apply the presented approach, the necessary artifacts include the Java source code and PlantUML files (`.puml`) with PlantUML class diagrams of the system to be tested. The inclusion of the `.puml` files in the plain text-based documentation (Docs-as-Code) is intended but not technically required.

The automatic extraction of all design diagrams from a directory possibly containing also other files than PlantUML files proved to be very useful. For the test project, it significantly simplified the integration of arcucheck into the GitLab pipeline for automated processing as all `.puml` were automatically extracted from the specified documentation directory that also contains Markdown (`.md`) files. Since the architecture documentation for arcudoc is written in Markdown, integrating PlantUML diagrams in the documentation text has proven to be more challenging than anticipated. While Markdown allows for the direct embedding of PlantUML code within the `.md` files, it does not support the inclusion of external `.puml` files. This limitation is problematic, as arcucheck relies on PlantUML diagrams being saved in separate `.puml` files. However, if AsciiDoc was used as the documentation format, it would be possible to include `.puml` files within the documentation text, as demonstrated in the architecture documentation of arcucheck, which utilizes the AsciiDoc format. The documentation can also be found in the GitLab repository in the "docs" directory [56].

5.6.2 Manual Steps

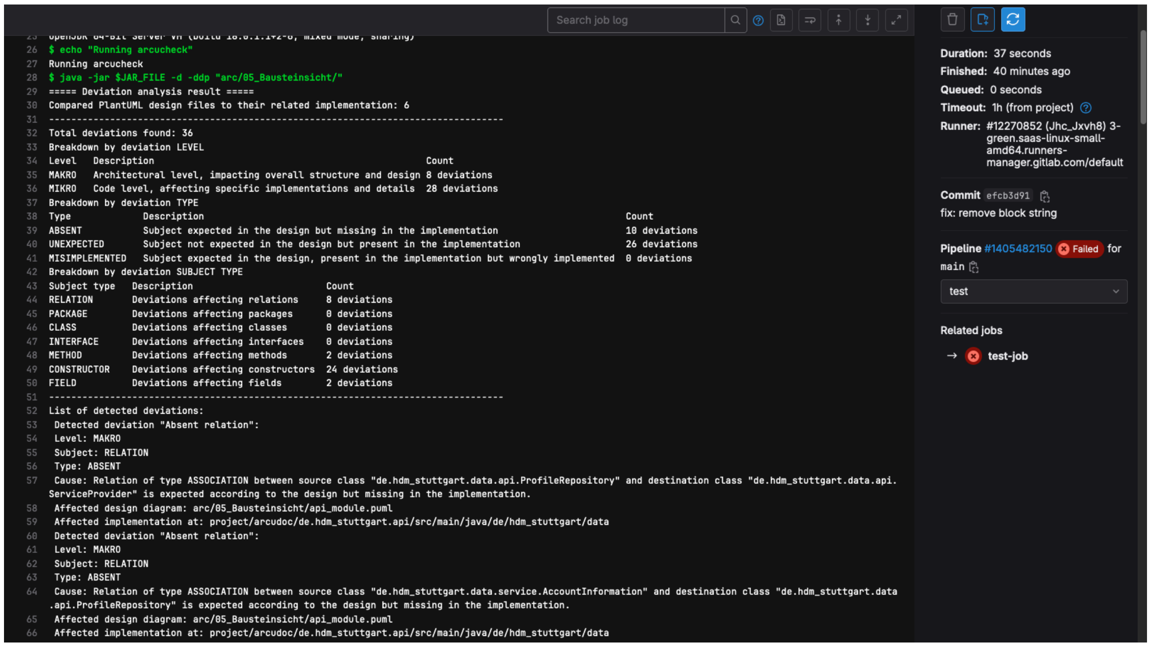
Required manual steps in order to apply the approach presented are the creation of the design model in form of PlantUML diagrams, the design-to-code mapping and the reviewing of output warnings.

As the application of arcucheck to this test project has shown, the creation of detailed PlantUML diagrams for an entire system is laborious and very time-consuming, why even the diagrams for the test project have been partly auto generated. Considering that systems can be much larger and more complex than arcudoc, the manual creation of the detailed class diagrams is almost impossible. In addition, needing to specify the path to the related code for each diagram adds extra work, since every diagram file has to be modified individually.

Once the design diagrams are specified, the manual steps can be reduced to reviewing the results. As illustrated in Figure 5.10, the pipeline of the test project triggered by code changes will automatically fail if any deviations are detected. Consequently, there is no need to start the process manually or check for deviations, as this is can be handled automatically.

5.6.3 Involved Stakeholders

A person is needed to create design models in the form of PlantUML diagrams and to add the design-to-code mapping. Additionally, someone is required to review the output warnings and address them. The review is typically carried out by developers, as the tool should be integrated into their development process. The developers are responsible for adapting the implementation



```
OpenJDK 64-Bit Server VM (build 18.0.1.1+2-6, mixed mode, sharing)
$ echo "Running arcucheck"
Running arcucheck
$ java -jar $JAR_FILE -d -ddp "arc/05_Bausteinsicht/"
===== Deviation analysis result =====
Compared PlantUML design files to their related implementation: 6
---------------------------------------------------------------------------------
Total deviations found: 36
Breakdown by deviation LEVEL
Level   Description                                                Count
MAKRO   Architectural level, impacting overall structure and design 8 deviations
MIKRO   Code level, affecting specific implementations and details  28 deviations
Breakdown by deviation TYPE
Type             Description                                                           Count
ABSENT           Subject expected in the design but missing in the implementation      10 deviations
UNEXPECTED       Subject not expected in the design but present in the implementation  26 deviations
MISIMPLEMENTED   Subject expected in the design, present in the implementation but wrongly implemented  0 deviations
Breakdown by deviation SUBJECT TYPE
Subject type   Description                      Count
RELATION       Deviations affecting relations     8 deviations
PACKAGE        Deviations affecting packages      0 deviations
CLASS          Deviations affecting classes       0 deviations
INTERFACE      Deviations affecting interfaces    0 deviations
METHOD         Deviations affecting methods       2 deviations
CONSTRUCTOR    Deviations affecting constructors  24 deviations
FIELD          Deviations affecting fields        2 deviations
---------------------------------------------------------------------------------
List of detected deviations:
 Detected deviation "Absent relation":
 Level: MAKRO
 Subject: RELATION
 Type: ABSENT
 Cause: Relation of type ASSOCIATION between source class "de.hdm_stuttgart.data.api.ProfileRepository" and destination class "de.hdm_stuttgart.data.api.
ServiceProvider" is expected according to the design but missing in the implementation.
 Affected design diagram: arc/05_Bausteinsicht/api_module.puml
 Affected implementation at: project/arcudoc/de.hdm_stuttgart.api/src/main/java/de/hdm_stuttgart/data
 Detected deviation "Absent relation":
 Level: MAKRO
 Subject: RELATION
 Type: ABSENT
 Cause: Relation of type ASSOCIATION between source class "de.hdm_stuttgart.data.service.AccountInformation" and destination class "de.hdm_stuttgart.data
.api.ProfileRepository" is expected according to the design but missing in the implementation.
 Affected design diagram: arc/05_Bausteinsicht/api_module.puml
 Affected implementation at: project/arcudoc/de.hdm_stuttgart.api/src/main/java/de/hdm_stuttgart/data
```

Duration: 37 seconds
Finished: 40 minutes ago
Queued: 0 seconds
Timeout: 1h (from project)
Runner: #12270852 (Jhc_Jxvh8) 3-green.saas-linux-small-amd64.runners-manager.gitlab.com/default

Commit `efcb3d91`
fix: remove block string

Pipeline #1405482150 Failed for `main`

Figure 5.10: Screenshot of the failed CI/CD pipeline in the test project

to the design by modifying the source code accordingly.

High-level system design is generally handled by architects. However, since this approach involves a low level of abstraction, architects may not be well-suited to handle the detailed design work. This could lead to a situation where no stakeholders feel responsible for creating the detailed class diagrams and adding the design-to-code mapping. The lack of abstraction in the detailed design process introduces challenges, particularly in the overall usage and integration process.

Since arcucheck was developed and tested by a single person, all roles were handled by that one person. As a result, the assumptions regarding ambiguous task distribution cannot be verified or proven false through the test project.

5.6.4 Evaluation Performance

The test project arcudoc consists of 97 Java classes, covered by six PlantUML diagrams. The execution time was first measured locally on an Apple M1 Max (3.2 GHz, 32 GB RAM, macOS 14.5) using IntelliJ IDEA 2024.1.4 (Ultimate Edition). Three test runs yielded execution times of 574, 559, and 552 milliseconds, resulting in an average of approximately 562 milliseconds, during which 36 deviations were detected.

Additionally, the GitLab pipeline, which includes a single stage that runs arcucheck on the same project and identifies the same deviations, had an average execution time of 32 seconds. Notably, the CI/CD job for arcucheck itself was marked with a duration of 0 seconds, indicating that arcucheck only contributes milliseconds to the overall pipeline execution time. Of course, execution times vary depending on the project size and available hardware resources.

These values are particularly interesting when compared with the evaluation performance of the other approaches identified by Knodel et al. [1], that is reported as "less than 5 minutes for all systems" [1].

5.6.5 Defect Types

The detectable errors have been dealt with in detail in 5.3. They were therefore not tested again with the test project. To enable a comparison with the other approaches, the violation categories introduced by Knodel et al. [1] are presented, together with an assessment of whether these categories are addressed.

Unintended Dependencies

The approach presented cannot identify dependency relationships, and as a result, unintended dependencies cannot be detected.

Context Exploration

External libraries and components are not included in the testing, so any defects related to them cannot be detected.

Broken Information Hiding

It is not clearly defined what Knodel et al. [1] understand by "information hiding." The approach presented can identify incorrect visibilities of methods and fields, as well as field types and method return types. This allows for the detection of concrete types being used instead of abstractions like interfaces or superclasses, as well as the identification of unauthorized public fields.

Misuse of Interface

It is also not clearly defined what exactly is meant by "interface misuse". The approach presented is able to detect unexpected and missing interface implementations, which likely addresses part of this defect type.

Misuse of Patterns

Not supported.

Violation of Architectural Styles

Not supported.

5.6.6 Probability of False Positives

The probability of warnings being issued when none should be issued is low. The convergence tests were developed to ensure that no warnings are issued if the design and implementation match. These tests primarily check for false positives. The probability of no warnings being issued when ones should be issued is also low. This was tested by the divergence test cases.

Tests failing are due to missing functionality and not other potential problems therefore it is predictable. If these specific problems were fixed, the probability of false positives would be minimal.

5.6.7 Maintainability

Modifications, additions and removals of code entities directly affect the results and have a significant impact. Even a minor change, such as changing a single character in a field name in the class implementation, requires adjustments to the design model. This indicates a very low level of robustness. This problem is also due to the tight coupling between design and implementation and therefore the lack of abstraction of the design.

5.6.8 Transferability

For the same reasons that affect maintainability, transferability is also very low. See 5.6.7.

5.6.9 Scalability

The scalability of the approach presented is limited. In large systems, the number of output warnings could be enormous, as even minor deviations trigger warnings. To improve this, it could be advantageous to limit the warnings to a macro-level, which is possible due to the implemented deviation categorisation.

5.6.10 Ease of Application

The simplicity of the application would have to be tested with users in order to be able to make well-founded statements. At this stage, only assumptions can be made based on the experiences of test application.

Learning Curve

The learning curve might not be steep as the tool has been designed to have a high probability that users are already familiar with the formats of the required artefacts. This increases the probability that no specialized training is needed in order to use the tool. UML was chosen because it is the most widely used modelling language, and PlantUML was chosen because of its popularity as a text-based UML tool. For more information, see Chapter 1 and Chapter 3.

However, this does not guarantee that all users are familiar with UML and PlantUML. Since UML and PlantUML are widely known, there are many resources that should make learning UML easier compared to a custom ADL. Overall the learning curve is assumed to be low.

Intuitiveness

Given that the approach uses a CLI, which is common in the software development toolkit, it should be intuitive for most developers. Additionally, the tool provides error messages for incorrect user entries and includes a help menu for further guidance. Furthermore, once the tool is integrated into the CI/CD pipeline, users will not need to perform any additional work. The output, in the form of warnings, is structured similarly to standard error messages, making it familiar and understandable for developers who encounter such messages regularly in their work. Overall, the intuitiveness of the tool is expected to be high since it can be applied without requiring any special training.

Iterative Refinements

This criterion evaluates whether the approach supports starting with a minimal design model and iteratively adding more design rules to simplify the usage process. Arcucheck facilitates this by allowing users to input individual PlantUML diagrams rather than requiring the entire documentation directory. This flexibility enables testing the tool with minimal input.

Based on the results related to the learning curve, intuitiveness, and iterative refinements, it can be inferred that the overall ease of use is high.

5.6.11 Multiple View Support

The approach presented supports only class diagrams, so multiple views are not supported.

5.6.12 Restructing Scenarios

This criterion examines the extent to which it is possible to explore different structural adaptations, of a system and assess their conformity with the architectural design. The "what-if" analyses can be conducted by inserting fictitious components into the design and then checking their conformity with the implementation. Progress towards the restructuring goal can then also be monitored.

In principle, the presented approach allows the addition of fictitious components as two independent models are compared. A fictitious class or interface could simply be added in the design. However, since adjustments in either model immediately lead to deviations, trying out different structural adjustments is always associated with a deviation. Therefore this has no real benefit, as the same result is always achieved: an absent implementation. This is also a deficit caused by a lack of abstraction. Nevertheless, it can be useful in the development process as it allows you to track the restructuring process by checking how many subjects like classes, interfaces and methods are missing in the implementation.

5.6.13 Decision-making-support

The approach presented does not support any decision making processes.

5.7 Evaluation Criteria Summary

Using arcucheck for test project and evaluating it using the criteria, provided valuable insights into its practical advantages and limitations. One significant drawback is the extensive manual effort required to specify detailed PlantUML class diagrams. The assumption that these diagrams already exist because they were created during the design process, as originally intended, proves to be unrealistic, as real projects are rarely specified in such detail.

Furthermore, the tool currently exhibits weaknesses in maintainability, transferability, and scalability, largely due to the lack of abstraction in the design model. These issues will be explored further in the next chapter. Decision-making support and multiple view support were not a requirement and are therefore not implemented. Further investigation would be needed to determine the feasibility of incorporating these features.

On the other hand, arcucheck performs well in terms of ease of use and evaluation performance, making it advantageous for integration into GitLab pipelines. As it turns out, compatibility with the docs-as-code approach depends on the text file format chosen for the documentation: AsciiDoc works well, while Markdown limits the possibilities. The identified support for restructuring scenarios, thanks to the use of two independent models, could be beneficial for monitoring development processes.

The use of arcucheck in a real project additionally revealed that the lack of support for enums leads to undefined behavior. This limitation is the reason why many deviations are detected, even though the design and implementation of all modules, except for the API module, should align perfectly.

When comparing the evaluation results to the approaches investigated by Knodel et al. [1], it is clear that their approaches are significantly more advanced. Not only do they require less manual effort, but their functionality for detecting deviations is also superior. Additionally, these approaches operate at a higher level of abstraction, which likely contributes to their better overall performance.

5.8 Summary

The systematic unit tests and the use of arcucheck in a test project as well as the application of evaluation criteria provided valuable insights into the capabilities and limitations of the prototype. The implications of these findings for the overall usefulness of the tool and also for the implementation-independent approach of using PlantUML diagrams for conformance checking are discussed in the next chapter.

Chapter 6

Discussion

In the previous chapters, interesting insights were gained into the functionalities of the implemented prototype and the potential of a PlantUML-based approach to static design conformance checking. In this chapter, these insights are discussed and form the basis for answering the research questions. In addition, various aspects for further research are highlighted.

6.1 Summary of the Results

The prototypical implementation of arcucheck has demonstrated that it is generally feasible to use PlantUML diagrams to detect deviations between design and implementation. The comparison process has proven to be reliable, consistently returning the expected results. While certain types of deviations, such as association relations, cannot be detected due to arcucheck's inability to extract them from the source code, other types of deviation can reliably be detected. A further analysis showed that arcucheck performs better in detecting deviations at the micro level, i.e. implementation details of classes, than in detecting deviations at the macro level, which primarily includes relationships.

The prototype has both practical advantages and disadvantages. On the positive side, it is easy to use and can be automated efficiently. The text-based nature of the design models makes them integrable into the docs-as-code workflow, although this depends on the text file format used for the documentation. The warnings generated are expressive, as they describe the location and exact cause of the deviations detected and contain a classification that enables filtering.

On the downside, creating many detailed class diagrams requires a significant amount of effort, making the process labor-intensive, although PlantUML offers straightforward syntax. Furthermore, this method is not salable for large projects, as even minor deviations can result in warnings. This leads to a potential overload of warnings, which are hard to review and maybe cover up the warnings about critical violations. It is important to note that not all deviations are necessarily problematic or indicate the erosion of a system [3].

6.2 Interpretation of the Results

The reliable results of the model comparison indicate that the use of PlantUML for modeling, along with the implementation strategy of mapping PlantUML diagrams to a hierarchy of Kotlin objects, was a good choice. It was initially assumed that the PlantUML specification might have limitations that would prevent the representation of certain constructs. However, this assumption proved to be unfounded, as everything needed could indeed be specified with PlantUML.

The immediate output of warnings when small changes are made to either model suggests a tight coupling between the implementation and design models, indicating that there is almost no abstraction in the design. Inconsistencies to a certain degree are an expected side effect of the evolution of software [18]. The flexibility required for the evolution of software systems is not guaranteed by the tool.

This tight coupling has implications for the tool's effectiveness. The fact that arcucheck primarily detects deviations at the implementation detail level, rather than at the level of class relations, suggests that it cannot capture the full complexity of software designs. Several factors contribute to this limitation. Class diagrams lack abstraction, and reverse-engineered diagrams often have additional shortcomings, as noted by Gue et al. [28]. This results in a lack of abstraction in the model and limitations in the generated diagrams from the code, such as the inability to extract associations. This ultimately leads to a incomplete representation of the design.

Ignoring the facts that such detailed design specifications are unrealistic and implementation details are often irrelevant, arcucheck is able to show the degree of conformance of design and implementation, apart from associations. However, these facts cannot be ignored when checking applicability to real systems, which means that arcucheck is currently not suitable for checking design conformance. However, it could be useful in environments where implementation details need to be examined, such as coding style checkers.

In summary, the initial expectations for arcucheck were not met, primarily due to an underestimation of the importance of abstraction in software design.

6.3 Limitations of the Prototype

Arcucheck was implemented to investigate the feasibility of a PlantUML based approach for static design conformance checking. While several shortcomings were identified during the evaluation, it is important to note that some of these limitations, mainly concerning associations, are specific to arcucheck's particular implementation, rather than inherent flaws in the abstract approach itself. This distinction suggests that with further development and refinement of the prototype, the use of PlantUML for static design conformance could be proven useful.

6.4 Areas for Future Work

Some of the limitations mentioned are rooted in the specific implementation of arcucheck, and further tool development could enhance its feasibility. One key area for improvement is the extraction of associations from the source code. This could be done programmatically, by analyzing the fields in Java source files and comparing their data types to all available classes and then add identified associations to the models afterwards. Additionally, method bodies would need to be analyzed to identify temporary uses of other classes. Another possible approach to extract associations could involve using Artificial Intelligence (AI). For example, a brief investigation involved providing a Java source code example to ChatGPT [1] with the prompt, *"How does the PlantUML diagram for the following classes look?"* Without additional specifications, associations, including navigability, were included in the generated result. This AI-based strategy appears promising and could significantly benefit the tool by incorporating associations.

The main drawbacks of the tool are related to the lack of abstraction in the model. To address this, it could be worthwhile to explore how the level of abstraction might be increased. One potential approach is to use a different type of diagram, such as component diagrams, instead of class diagrams. This would require a major restructuring of the prototype as well as exploration, how component diagrams could be extracted from the source code.

A potential feature for the tool is the analysis of naming convention compliance. To enable this, users would need to define naming guidelines separately. This functionality would allow for the verification and assurance of consistent naming practices across the system.

6.5 Answering of the Research Questions

The following provides brief answers to the initially presented research questions. The preceding sections offer a detailed analysis that serves as the basis for these responses.

R.Q. 1: Which specific design aspects can be analyzed using the proposed approach?

Arcucheck can detect absent, unexpected, and incorrectly implemented classes, interfaces, packages, methods, constructors, and fields. It can also identify absent and unexpected inheritance and realisation relations. A detailed analysis is provided in section 5.3. A table of detectable deviations, categorized by subject, is available in Appendix B. A distinction must be made between the prototype arcucheck and the abstract approach: The approach has the potential to detect association deviations and also to cover naming convention checks if the

[1] https://chatgpt.com/

input model in the form of clas diagrams is not changed. Choosing a different type of diagram would, for example, enable an analysis at a higher level of abstraction.

> *R.Q. 2: What are the practical advantages and disadvantages of a PlantUML- based approach for design conformance checking in software development projects?*

The practical advantages include ease of use and the possibility of integration into development processes. Disadvantages include scalability, maintainability and transferability as well as the considerable manual effort required to create the design models. Details can be found in 5.6 and the preceding sections.

Chapter 7

Conclusion

The motivation for this thesis was to develop an approach for static design conformance that minimizes the burden of use by integrating seamlessly into existing development processes and utilizing familiar modeling notations. The analysis of related theoretical backgrounds led to the design of a PlantUML-based approach, using class diagrams as design models. A prototypical tool was developed based on the identified requirements and evaluated in terms of its practical advantages, disadvantages, and its ability to analyze specific design aspects. The findings indicate that while the approach is generally viable, the implementation has shortcomings, primarily due to the lack of abstraction in the class diagrams used as input models. Therefore using class diagrams was a flawed design decision, which led to the tool's lack of effectiveness. However, the design goals of using widely adopted modeling notations for design models and achieving seamless integration into development processes were met. Further research is needed to determine whether the limitations can be effectively addressed to make a PlantUML-based approach viable for real-world systems, which is not yet the case.

Bibliography

[1] J. Knodel and D. Popescu, "A Comparison of Static Architecture Compliance Checking Approaches," in *2007 Working IEEE/IFIP Conference on Software Architecture (WICSA'07)*, pp. 12–21, IEEE Computer Society, 2007.

[2] L. Hochstein and M. Lindvall, "Combating architectural degeneration: A survey," *Information and Software Technology*, vol. 47, pp. 643–656, 7 2005.

[3] R. Li, P. Liang, M. Soliman, and P. Avgeriou, "Understanding software architecture erosion: A systematic mapping study," 3 2022.

[4] M. Lindvall, R. Tesoriero, and P. Costa, "Avoiding Architectural Degeneration: An Evaluation Process for Software Architecture," tech. rep., Fraunhofer Center Maryland for Experimental Software Engineering, 2002.

[5] L. De Silva and D. Balasubramaniam, "Controlling software architecture erosion: A survey," *Journal of Systems and Software*, vol. 85, pp. 132–151, 1 2012.

[6] L. Passos, R. Terra, M. T. Valente, R. Diniz, and N. Das Chagas Mendonça, "Static architecture-conformance checking: An illustrative overview," *IEEE Software*, vol. 27, pp. 82–89, 9 2010.

[7] A. Nugroho and M. Chaudron, "A Survey of the Practice of Design – Code Correspondence amongst Professional Software Engineers," in *First International Symposium on Empirical*

Software Engineering and Measurement, pp. 467–469, Institute of Electrical and Electronics Engineers (IEEE), 4 2008.

[8] R. Li, P. Liang, M. Soliman, and P. Avgeriou, "Understanding Architecture Erosion: The Practitioners' Perceptive," 3 2021.

[9] M. Ozkaya and F. Erata, "A survey on the practical use of UML for different software architecture viewpoints," *Information and Software Technology*, vol. 121, 5 2020.

[10] D. Garlan, "Software Architecture," tech. rep., School of Computer Science Carnegie Mellon University, 2008.

[11] M. Jaiswal, "Software Architecture and Software Design," *International Research Journal of Engineering and Technology*, 2019.

[12] D. L. Parnas, "Software aging," in *Proceedings - International Conference on Software Engineering*, pp. 279–287, Publ by IEEE, 1994.

[13] A. B. Math, E. B. Allen, and B. J. Williams, "Assessing Code Decay: A Data-driven Approach," tech. rep., Computer Science Information Systems Northwest Missouri State University, 2015.

[14] R. Lammel, R. Oliveto, and R. Robbes, "Empirical evidence of code decay: A }systematic mapping study.," tech. rep., Department of Computer Science and Engineering Mississippi State University, 2013.

[15] D. E. Perry and A. L. Wolf, "Foundations for the Study of Software Architecture," tech. rep., ATT Bell Laboratories, 1992.

[16] M. M. Lehman, "Laws of Software Evolution Revisited," tech. rep., Department of Computing Imperial College, 1996.

[17] A. Caracciolo, M. F. Lungu, and O. Nierstrasz, "A Unified Approach to Architecture Conformance Checking," tech. rep., Software Composition Group, University of Bern, 2015.

[18] N. J. Karsidi, *Managing Software Design Erosion with Design Conformance Checking*. PhD thesis, Delft University of Technology, 2012.

[19] M. Maier, E. David, and R. Hilliard, "Software architecture: Introducing ieee standard 1471," 2001.

[20] D. L. Parnas, "Precise Documentation: The Key To Better Software," tech. rep., Middle Road Software, Inc., 2010.

[21] P. Clements, D. Garlan, R. Little, R. Nord, and J. Stafford, "Documenting Software Architectures: Views and Beyond," tech. rep., Carnegie Mellon University, 2003.

[22] M. Ozkaya, "Do the informal & formal software modeling notations satisfy practitioners for software architecture modeling?," *Information and Software Technology*, vol. 95, pp. 15–33, 3 2018.

[23] M. Ozkaya, "The analysis of architectural languages for the needs of practitioners," *Software - Practice and Experience*, vol. 48, pp. 985–1018, 5 2018.

[24] G. Reggio, M. Cerioli, and E. Astesiano, "Towards a Rigorous Semantics of UML Supporting Its Multiview Approach," tech. rep., DISI - Universit'a di Genova (Italy), 2001.

[25] M. Petre, "UML in practice," tech. rep., Centre for Research in Computing The Open University Milton Keynes, 2013.

[26] H. Osman hosman, I. R. Dave Stikkolorum drstikko, I. Arjan van Zadelhoff avzadelh, and I. R. Michel Chaudron chaudron, "UML Class Diagram Simplification: What is in the developer's mind?," tech. rep., Leiden Institute of Advanced Computer Science, Leiden University, 2012.

[27] M. Claypool, M. Brambilla, J. Cabot, and M. Wimmer, *TBD, Series Editor C M & Model-Driven Software Engineering in Practice*. Morgan Claypool Publishers, 2012.

[28] Y.-G. Guéhéneuc, "A Systematic Study of UML Class Diagram Constituents for their Abstract and Precise Recovery," tech. rep., D´epartement d'informatique et de recherche op´erationnelle Universit´e de Montr´eal, 2004.

[29] G. Garousi, V. Garousi-Yusifolu, G. Ruhe, J. Zhi, M. Moussavi, and B. Smith, "Usage and usefulness of technical software documentation: An industrial case study," in *Information and Software Technology*, vol. 57, pp. 664–682, Elsevier B.V., 2015.

[30] R. Thomchick, "Improving access to API documentation for developers with Docs-as-Code-as-a-service," *Proceedings of the Association for Information Science and Technology*, vol. 55, pp. 908–910, 1 2018.

[31] E. Holscher, "Docs as Code — Write the Docs." https://www.writethedocs.org/guide/docs-as-code/[Online], 2023. (accessed: 19/07/2024).

[32] "docToolchain [online]." https://doctoolchain.org/docToolchain/v2.0.x/index.html. (accessed: 17/07/2024).

[33] W. the Docs, "Marcia Riefer Johnston & Dave May - One AWS team's move to docs as code - YouTube [Online]." https://www.youtube.com/watch?v=Cxuo3udElcE, 2022. (accessed:19/07/2024).

[34] N. D. Video, " Documentation, Disrupted: How Two Technical Writers Changed Google Engineering Culture - YouTube [Online]." https://www.youtube.com/watch?v=EnB8GtPuauw, 2015. (accessed:19/07/2024).

[35] C. Lange and M. Chaudron, "In practice: Uml software architecture and design description," tech. rep., Eindhoven University of Technology, 2006.

[36] J. Cabot, “From Text to Models: A Comprehensive Guide to Textual Modeling and Diagrams as Code Tools in 2024 [online].” https://modeling-languages.com/text-uml-tools-complete-list/, 2024. (accessed: 16/07/2024).

[37] J. Cabot, “A coffee with Arnaud Roques (creator of PlantUML) [Online].” https://modeling-languages.com/interview-plantuml/, 2016. (accessed: 18/07/2024).

[38] M. Trifan, B. Ionescu, and D. Ionescu, “A Combined Finite State Machine and PlantUML Approach to Machine Learning Applications,” in *SACI 2023 - IEEE 17th International Symposium on Applied Computational Intelligence and Informatics, Proceedings*, pp. 631–636, Institute of Electrical and Electronics Engineers Inc., 2023.

[39] PlantUML, “Open-Source-Tool, das einfache Textbeschreibungen verwendet UML-Diagramme zu zeichnen.[online].” https://plantuml.com/de/. accessed: 19/07/2024.

[40] J. Cabot, “An ecosystem of tools around PlantUML to render textual UML diagrams anywhere you want (updated) [online].” https://modeling-languages.com/plantuml-textual-uml-online/[Online], 2017. (accessed: 18/07/2024).

[41] ussi Vatjus-Anttila, “GitHub - jupe/puml2code: PlantUML code generator[Online].” https://github.com/jupe/puml2code, 2021. (accessed: 14/07/2024).

[42] PlantUMLParser, “PlantUML Parser Plugin for JetBrains IDEs | JetBrains Marketplace[Online].” https://plugins.jetbrains.com/plugin/15524-plantuml-parser. accessed: 19/07/2024.

[43] mermaid.js, “Mermaid | Diagramming and charting tool [Online].” https://mermaid.js.org//. (accessed: 19/07/2024).

[44] mermaid js, “GitHub - mermaid-js/mermaid: Generation of diagrams like flowcharts or sequence diagrams from text in a similar manner as markdown [Online].” https://github.com/mermaid-js/mermaid, 2024. (accessed:19/07/2024).

[45] J. Vatjus-Anttila, “GitHub - plantuml/plantuml: Generate diagrams from textual description.” https://github.com/plantuml/plantuml, 2022. (accessed: 14/07/2024).

[46] G. C. Murphy and D. Notkin, “Software Reflexion Models: Bridging the Gap between Source and High-Level Models,” tech. rep., Dept. of Conputer Science Engineering University of Washington, 1995.

[47] R. Koschke and D. Simon, “Hierarchical reflexion models,” in *Proceedings - Working Conference on Reverse Engineering, WCRE*, vol. 2003-January, pp. 36–45, IEEE Computer Society, 2003.

[48] N. Sangal, E. Jordan, V. Sinha, and D. Jackson, “Using Dependency Models to Manage Complex Software Architecture,” tech. rep., 2005.

[49] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen, “The Structure and Value of Modularity in Software Design,” tech. rep., 2001.

[50] M. W. Godfrey, C. Tudor, and G. Gˆırba, “Query Technologies and Applications for Program Comprehension,” *In Proceedings of ICPC*, pp. 285–288, 2008.

[51] UML.org, “What is UML | Unified Modeling Language[Online].” https://www.uml.org/. (accessed:09/08/2024).

[52] J. Rumbaugh, I. Jacobson, and G. Booch, *The unified modeling language reference manual*. Addison-Wesley, 1999.

[53] T. Barik, *Error Messages as Rational Reconstructions*. PhD thesis, North Carolina State University, 2018.

[54] M. Genero and M. Piattini, “A Survey of Metrics for UML Class Diagrams,” Tech. Rep. 9, 2005.

[55] L. Pruijt, C. Köppe, J. M. van der Werf, and S. Brinkkemper, "The accuracy of dependency analysis in static architecture compliance checking," in *Software - Practice and Experience*, vol. 47, pp. 273–309, John Wiley and Sons Ltd, 2 2017.

[56] P. Schilling, "Schilling Pia / arcucheck · GitLab [Online]." https://gitlab.mi.hdm-stuttgart.de/ps149/arcucheck, 2024. (accessed:08/08/2024).

[57] P. Schilling, "Pia Schilling / arcudoc · GitLab[online]." https://gitlab.com/piamarie/arcudoc, (2022). (accessed: 07/08/2024).

[58] V. R. Basili, G. Caldiera, and H. D. Rombach, "THE GOAL QUESTION METRIC APPROACH," tech. rep., Institute for Advanced Computer Studies Department of Computer Science University Of Maryland, 1994.

Appendices

Appendix A

Potential Deviations

Relations			
Type	Description	Level	Sub type
Absent association relation	Relation of type association is expected according to the design but missing in the implementation	Macro	Aggregation, Composition
Unexpected association relation	Relation of type association is not expected according to the design but present in the implementation	Macro	Aggregation, Composition
Absent inheritance relation	Relation of type inheritance is expected according to the design but missing in the implementation	Macro	
Unexpected inheritance relation	Relation of type inheritance is not expected according to the design but present in the implementation	Macro	
Absent realisation relation	Relation of type realisation is expected according to the design but missing in the implementation	Macro	
Unexpected realisation relation	Relation of type realisation is not expected according to the design but present in the implementation	Macro	
Misimplemented association	Association is expected and present but implemented incorrectly	Macro	Incorrect navigability
			Incorrect multiplicity

Table A.1: Potential relation deviations

Packages			
Type	Description	Level	Sub type
Absent package	Package is expected according to the design but missing in the implementation	Macro	
Unexpected package	Package is not expected according to the design but present in the implementation	Macro	
Misimplemented package	Package is expected and present but implemented incorrectly	Macro	Incorrect package hierarchy
			Unexpected/missing package members

Table A.2: Potential package deviations

Interfaces			
Type	Description	Level	Sub type
Absent interface	Interface is expected according to the design but missing in the implementation	Macro	
Unexpected interface	Interface is not expected according to the design but present in the implementation	Macro	
Misimplemented interface	Interface is expected and present but implemented incorrectly	Macro	Incorrect package containment
		Micro	Unexpected/missing methods in interface
			Unexpected/missing constants in interface

Table A.3: Potential interface deviations

Classes			
Type	Description	Level	Sub type
Absent class	Class is expected according to the design but missing in the implementation	Macro	
Unexpected class	Class is not expected according to the design but present in the implementation	Macro	
Misimplemented class	Class is expected and present but implemented incorrectly	Macro	Incorrect package containment
		Micro	Unexpected/missing abstract label
			Unexpected/missing methods in class
			Unexpected/missing fields in class
			Unexpected/missing constructors in class

Table A.4: Potential class deviations

Methods			
Type	Description	Level	Sub type
Absent method	Method is expected according to the design but missing in the implementation	Micro	
Unexpected method	Method is not expected according to the design but present in the implementation	Micro	
Misimplemented method	Method is expected and present but implemented incorrectly	Micro	Incorrect method return type
			Incorrect method visibility
			Incorrect method parameter types
			Unexpected/missing abstract label
			Unexpected/missing static label

Table A.5: Potential method deviations

Constructors			
Type	Description	Level	Sub type
Absent constructor	Constructor is expected according to the design but missing in the implementation	Micro	
Unexpected constructor	Constructor is not expected according to the design but present in the implementation	Micro	
Misimplemented constructor	Constructor is expected and present but implemented incorrectly	Micro	Incorrect constructor visibility
			Incorrect constructor parameter types

Table A.6: Potential constructor deviations

Fields			
Type	Description	Level	Sub type
Absent field	Field is expected according to the design but missing in the implementation	Micro	
Unexpected field	Field is not expected according to the design but present in the implementation	Micro	
Misimplemented field	Field is expected and present but implemented incorrectly	Micro	Incorrect field visibility
			Incorrect field data type
			Unexpected/missing static label

Table A.7: Potential field deviations

Appendix B

Mapping of Potential Deviations to Unit Test Classes

Relations			
Potential deviation	Test class	Convergence test passes	Divergence test passes
Absent/unexpected association relation	RelationAssociationTest.kt	false	false
Absent/unexpected composition relation	RelationCompositionTest.kt	false	false
Absent/unexpected aggregation relation	RelationAggregationTest.kt	false	false
Absent/unexpected inheritance relation	RelationInheritanceTest.kt	true	true
Absent/unexpected realisation relation	RelationRealisationTest.kt	true	true
Misimplemented navigability	RelationNavigabilityTest.kt	false	false
Misimplemented multiplicity	RelationMultiplicityTest.kt	false	false

Table B.1: Test classes for potential relation deviations

Packages			
Potential deviation	Test class	Convergence test passes	Divergence test passes
Absent/unexpected package	PackageOccurrenceTest.kt	true	true
Misimplemented package hierarchy	PackageHierarchyTest.kt	true	false
Misimplementaiton: Absent/unexpected package members	Covered through ClassPackageContainmentTest.kt and InterfacePackageContainmentTest.kt	true	true

Table B.2: Test classes for potential package deviations

Interfaces			
Potential deviation	Test class	Convergence test passes	Divergence test passes
Absent/unexpected interface	InterfaceOccurrenceTest.kt	true	true
Misimplemented package containment	InterfacePackageContainmentTest.kt	true	true
Misimplementation: Absent/unexpected methods in interface	Covered through MethodOccurrenceTest.kt	true	true
Misimplementation: Absent/unexpected constants in interface	Covered through FieldOccurrenceTest.kt	true	true

Table B.3: Test classes for potential interface deviations

Classes			
Potential deviation	Test class	Convergence test passes	Divergence test passes
Absent/unexpected class	ClassOccurrenceTest.kt	true	true
Misimplemented package containment	ClassPackageContainmentTest.kt	true	true
Misimplemented abstract label	ClassAbstractLabelTest.kt	true	true
Misimplementation: Absent/unexpected method in class	Covered through MethodOccurrenceTest.kt	true	true
Misimplementation: Absent/unexpected field in class	Covered through FieldOccurrenceTest.kt	true	true
Misimplementation: Absent/unexpected constructor in class	Covered through ConstructorOccurrenceTest.kt	true	true

Table B.4: Test classes for potential interface deviations

Methods			
Potential deviation	Test class	Convergence test passes	Divergence test passes
Absent/unexpected method	MethodOccurrenceTest.kt	true	true
Misimplemented return type	MethodReturnTypeTest.kt	true	true
Misimplemented visibility	MethodVisibilityTest.kt	true	true
Misimplemented parameter types	MethodParameterTypesTest.kt	true	true
Misimplemented abstract label	MethodAbstractLabelTest.kt	true	true
Misimplemented static label	MethodStaticLabelTest.kt	true	true

Table B.5: Test classes for potential method deviations

Constructors			
Potential deviation	Test class	Convergence test passes	Divergence test passes
Absent/unexpected constructor	ConstructorOccurrenceTest.kt	true	true
Misimplemented visibility	ConstructorVisibilityTest.kt	true	true
Misimplemented parameter types	ConstructorParameterTypesTest.kt	true	false

Table B.6: Test classes for potential constructor deviations

Fields			
Potential deviation	Test class	Convergence test passes	Divergence test passes
Absent/unexpected field	FieldOccurrenceTest.kt	true	true
Misimplemented data type	FieldDataTypeTest.kt	true	true
Misimplemented visibility	FieldVisibilityTest.kt	true	true
Misimplemented static label	FieldStaticLabelTest.kt	true	true

Table B.7: Test classes for potential field deviations