Abstract

Patterns are a source of knowledge when architecting software systems. They provide abstract and time-tested solutions that show how a system should be structured to achieve needed qualities. However, when developing software there is a chance that small mistakes are introduced in the source code. Over time, these mistakes can accumulate and break the structure of the pattern and its qualities are lost. There are methods that can help find such errors, but none of these provide a pattern abstraction. In this work, we describe a method that raises the level of abstraction from checking individual dependencies to checking key dependencies in the pattern. We implement our method, apply it to check the Model-View-Controller pattern. We show that the method can find architectural problems in real source code and examine how removal of detected erosions affects the source code.

We conducted an experiment in a software project setting to determine if using the method affects the number of architectural problems. Some project teams were randomly assigned to use a software service that automated our method. It checked how well their implementation conformed to Model-View-Controller every time they updated the source code. The experiment showed that developers that used the tool had significantly fewer detected architectural problems during the course of the project.

Our method makes conformance checking easier to use. This might help increase the adoption of conformance checking in industry.
Acknowledgements

First and foremost I would like to thank my supervisors Morgan Ericsson and Anna Wingkvist for their endless effort of pushing me forward and challenging me. I would also like to show my appreciation to Dean Staffan Carius, Head of Department Jesper Andersson and Professor Welf Löve, without who’s support my studies towards a PhD would not have been possible. A special thank you goes to Daniel Toll, as we started this journey together and we will finish it together. I value the discussions, ideas and you being present on a day-to-day basis. I also extend my gratitude to all colleagues who’s work have been affected by my absence from teaching. You are truly amazing!

Finally a heartfelt thank you to my loving wife and children, you make it all worthwhile.
# Contents

## 1 Introduction

1.1 Software Evolution ................................................. 2  
1.2 Software Architecture ........................................... 2  
  1.2.1 Architectural Patterns ................................. 3  
  1.2.2 Software Architecture Erosion ...................... 3  
1.3 Architecture Conformance Checking ....................... 4  
1.4 Researcher’s Background and Perspective ............... 5  
1.5 Research Questions ............................................ 6  
1.6 Thesis Organization ............................................ 6

## 2 Erosion in the MVC Pattern

2.1 Background ...................................................... 9  
2.2 Method .......................................................... 11  
2.3 Results .......................................................... 11  
  2.3.1 Time Breaker and Frontline ............................ 12  
  2.3.2 TWTPB .................................................... 13  
  2.3.3 Hero and Gears of Love ................................. 14  
2.4 Summary .......................................................... 15

## 3 SACC and the MVC Pattern

3.1 Background ...................................................... 17  
  3.1.1 Dependency Graphs ...................................... 19  
  3.1.2 Software Inspections .................................... 19  
  3.1.3 Reflexion Models ........................................ 20  
  3.1.4 Dependency Rules ........................................ 21  
  3.1.5 Dependency Structure Matrix ........................... 22  
3.2 Method .......................................................... 22  
3.3 Comparison of Reflexion Models and Dependency Rules 24  
3.4 A Method for Pattern-based SACC ......................... 26  
  3.4.1 Hypothetical Scenarios .................................. 28  
3.5 Summary .......................................................... 35

## 4 Evaluation of SACC of the MVC Pattern

4.1 Method .......................................................... 37  
  4.1.1 Manual Inspection ....................................... 38  
  4.1.2 Method Execution ....................................... 38
Chapter 1

Introduction

Software is intangible, pliable and flexible. This softness is one of the fundamental differences between software and other artifacts engineered by humans [McC01; Bas93; Bro87]. However, while softness is an inherent trait, it is not always easy to change and evolve software as planned. As time progresses, many small errors, mistakes, and deviations in development accumulate and can cause needed changes to be hard and time consuming [KLMN06; Kno11; BK07]. In this state the software is no longer soft and cannot be evolved efficiently. The principles governing softwares’ intended ability to evolve is an architectural trait [Ieeb; PW92]. When mistakes are allowed to accumulate in an uncontrolled way and ultimately destroy the structure of the intended architecture, the evolvability of the system is lost and the architecture is said to have eroded [DSB12; Mer10].

We need efficient and effective methods and tools to find these mistakes throughout the life cycle of software systems to control architectural erosion in software development. The overall goal of this research is to help find such mistakes by providing developers with a method to statically check how well an implementation conforms to the specified architecture. There exist such methods, but these can be difficult to use and are not well-adopted in industry [Pru15; DSB12; GGIW14]. There are indications that industry would benefit from higher level prepackaged solutions. Our general idea is to base a method for static architecture conformance checking on architectural patterns and allow the method to be parameterized to adapt to specific circumstances. We want to raise the level of abstraction from a specific technique (e.g., Reflection Models or Dependency Rules) to reasoning about how a pattern can be used in Static Architecture Conformance Checking (SACC) and what its most important constructs are. Our method should value simplicity and abstraction over completeness and detail.

In this chapter we proceed to introduce software evolution and its impact on a software system. We show that software architecture and architectural patterns is a common solution to the challenge of keeping the software in an easily evolvable state. We discuss how architectural patterns can erode during implementation, and discuss how static architecture conformance checking can
be used to gain insight into this erosion.

1.1 Software Evolution

Over the lifetime of many software systems, requirements and system environment change in unforeseen ways [Mer10]. One example of this in recent years is how users require systems or parts of systems to be available on smart phones. Smart phones are often radically different platforms from both technical and user point of view. The system (in whole or part) needs to be able to adapt to this new environment and migrate to a new technical platform. Software evolution addresses the challenges of software adapting to changes in the real world [LR03; Som04]. Changes are not easy to foresee during initial inception and design, so software must be made flexible.

In agile development philosophies the ability to keep evolving software is important to be able to rapidly respond to changes [Men08] and evolvability plays an increasingly important role in a society where software is ubiquitous [LR03]. Software platforms such as the World Wide Web, gaming consoles, and embedded systems in cars put increasing demands on software evolvability as they offer ways to update the end product via software to increase value and longevity [JB11]. There is no need to distribute software on a physical media anymore, software updates can be pushed to users automatically or at their demand as computation and data reside in the cloud. Software is expected to be softer than ever before.

Lehman discovered that proprietary systems need continuous adaptation to not become increasingly less satisfactory [Leh79]. They formulated eight laws of software evolution [Leh79; YM13; LR03]. Of these, Declining Quality and Increased Complexity are the most important for this thesis. As a system evolves the quality will decline and the complexity will increase if work is not explicitly done to manage these problems.

1.2 Software Architecture

Software architecture is a large topic and there are many definitions\(^1\) [Bas12]. Most definitions have at their core the goal of managing the complexity of software development by facilitating reasoning about a system’s structures on an abstract level. It affects and is influenced by a broad range of activities in software development from requirements to testing, deployment, and project management. Software architecture is about making important early decisions in development as informed as possible, trying to find where to make trade-offs, where requirements contradict, and to find balance [Bas12]. For example the choice to make a web based system that has the advantage of a familiar end-user experience, but can be vulnerable to security issues in many of its sub-systems such as web-browsers and server software; or how to structure the

\(^1\)SEI list of definitions available at http://www.sei.cmu.edu/architecture/start/glossary/published.cfm
source code of a game to facilitate multi-platform development while providing reuse.

Software architecture often involves stakeholders from many disciplines, from hardware to software and management. The consequence is that a system’s architecture needs to cover many viewpoints and there is a need for many different but related structures. There are numerous such views mentioned in the literature [CGB+02; Kru95]. In the context of this work we focus on the structural architecture [Kno11], how the software source code is divided and related to achieve desired qualities. The structural architecture can be thought of as clusters of source code elements. In each cluster, these elements are tightly coupled to achieve some higher level responsibility (e.g., visualization of a user interface). Elements in clusters sometimes depend on elements in other clusters to fulfill their responsibilities (e.g., the user interface needs some information to visualize). Structural architecture is about formalizing the clusters, making them more distinct, and defining the allowed and disallowed dependencies between them [Kno11; Bas12]. We call such a well-defined and high-level cluster an architectural element. The idea is to manage complexity and enable reasoning about qualities such as changeability, reuse, ease of development, and division of labor, before the architectural elements are implemented by source code elements. For the remainder of this work, the term architecture refers to the structural software architecture. For an object-oriented system, the structural architecture is often documented using UML component, package, class, and interaction diagrams [Bas12].

1.2.1 Architectural Patterns

To support architectural design decisions, the architect has a body of knowledge in the form of own and peers’ experiences. This experience is often available in the form of patterns [BHS07]. A pattern is a named, abstract problem and solution pair [BHS07; GHJV95]. The abstract problem is the generalized context the pattern operates in, the situations in which it can be used, and what the consequences of these situations are. The abstract solution is the general structure of a solution to the problem and the consequences of applying this solution. The solution is often given as a set of structural diagrams that describe the architectural elements, the division of responsibility, and rules of dependencies between the elements. This abstract structure is realized (i.e., made more specific) differently in different environments. This might give rise to pattern variations, where a variation is more specialized towards a certain domain or technological situation. For example, the Model-View-ViewModel is a variation of the Model-View-Controller pattern. Patterns are considered as a strategy to minimize software architecture erosion since they are time tested solutions to enable evolution and thus minimize the risk of erosion [DSB12].

1.2.2 Software Architecture Erosion

The initial architectural decisions often have a great impact on the development project and as the software evolves—and complexity increases—the structure
of the initial architecture is lost (it has eroded) if there is no activity to maintain it. The initial trade-offs are no longer valid so decisions based on them can no longer be trusted. The architecture loses its value, both in terms of documentation (for high level decision making) and for the practical implementation of features. For example, if an architecture that facilitates evolution of the user interfaces erodes, it becomes harder than expected to change or add user interface features. The initial advantage of the architecture is lost.

It is also harder for (new) developers to understand the implementation compared to the intended (as documented) architecture. The map and reality do not match up any more. Software architecture erosion has been researched under many names, such as software erosion [Dal09], architectural degeneration [HL05], architectural decay [RSN09], software entropy [Jac92], and technical debt [KNO12; Cum93]. At a lower level of erosion, we find related concepts such as design erosion [VGB02], design decay [IB07], code decay [EGK+01], and code smells [VEM02]. The boundaries between high and low level erosion is floating, but the underlying notion is that erosion that breaks an intended design (documented or not) is introduced as a consequence of software evolution.

There are approaches to handle software architecture erosion. These can be classified as strategies to minimize, prevent, or remove erosion [DSB12]. To fully control erosion, methods from all three categories are needed. We will focus on detection of points of architectural erosion and how removal of detected erosions affects the source code.

### 1.3 Architecture Conformance Checking

Existing erosion can be detected by checking that an implementation conforms to the intended architecture. This process of checking that the system’s actual, as implemented, architecture does not violate its intended architecture is called Architecture Conformance Checking (ACC). ACC is a process-oriented strategy to minimize erosions [DSB12]. Such checks should preferably be done throughout the development and should thus not incur high costs (in terms of effort needed). In this sense, the introduction of an automatic detection method with prompt feedback to developers can be seen as an erosion prevention measure as this enables learning and developers can avoid introducing erosion [Kno11].

Here we focus on Static ACC (SACC), i.e., conformance checking without running the system. We limit the scope of the research to only check source code entities (classes, types, etc.) against the intended architecture. SACC is an active research area and there are techniques, methods and tools that can be used. However, despite active research in SACC, industry adoption is poor [Pru15; DSB12; GGIW14]. So, in practice, many implementations are likely to violate the intended architecture resulting in systems that are hard to evolve. We argue that a higher level of abstraction (architectural pattern level) can provide a benefit in terms of easy adoption and integration in the software build process (continuous integration and deployment).
1.4 Researcher’s Background and Perspective

Since I was first introduced to the concepts of designing software I have been fascinated by the feeling I get when I find something that is well designed. The software is pliable to the changes needed and complex evolution that seem impossible is handled gracefully. I have always hunted this elusive feeling in my own designs and this is one of the driving forces in my own pursuit of software engineering. I think many software engineers can appreciate this. The opposite is also true for me; working on a badly designed (or eroded) piece of software is hard and cumbersome. This is demotivating. Many times I tend to start over, from scratch, rather than to live with (my) bad decisions or refactor them. This is probably a luxury of not being a professional developer (anymore). Possibly this is also connected to my fascination with game development, it is not for “real” (though it can certainly be big business). I find that games need to be softer to catch the elusive requirement of “fun”, than a business system resting on more solid foundations of functional requirements. As such computer games are a good, albeit extreme, test bed for software evolution.

I have developed games professionally while at Massive Entertainment (currently UBI-Soft Massive), mobile and PC-division, and semi-professionally in my own company Spell of Play Studios (mostly active between 2005–2010). The projects introduced in Chapter 2 are from this period. The development team felt good progress when it understood more and more of the architecture used. In this sense, this research is an attempt at more objectively capturing the essence of this feeling. It also becomes an introspective journey for me. What do I consider good design? Looking back at old projects and understanding them is not an easy task. Things you once were proud of suddenly seem less than good in the light of new insights. As a researcher, I want to know more, dig deeper, and really understand what in software elicits this feeling in me. Is it just a feeling of “the next thing” that feels good, abandoning the old and starting with the new? Or is it actual progress, where the state of things actually improves? If so, how can we measure this improvement, can we see it in some other way than just a feeling of “this is better” or “I like this more”? My first attempt at this was focused on code metrics—a story that is not told in this thesis—that essentially failed. We could not find any good way of comparing metrics over projects that seemed sound enough for practical use [ELO+13]. It also became very complicated to use these models in practice, nothing I could see a future developer actually doing on a regular basis.

Knowing my own world view and biases in software design is also important in my role as a teacher. I want to provide my students with a broad array of design principles as there is no way for me to teach them the specifics of every single design decision they will face. My goal is to make them realize that they do make design decisions and that it will eventually have an impact on the system, however small they are. This work is an attempt to make these small decisions visible at a larger scale and provide a simple method to find out if they are acceptable or not. During this research there have been times where the general feeling of our method has been that it has no real value. It is just
some other technique rehashed, offering less detail and less value. However, the importance and value of abstraction and simplicity has crystallized more and more.

1.5 Research Questions

To minimize erosion software architectures are often based on patterns [DSB12]. Software needs to evolve and the architecture erodes (pattern is violated). To avoid this, care must be taken to check that the implementation (source code) conforms to the intended architecture (pattern). We have developed a conformance checking method based on an architectural pattern. We evaluate this method by posing the following research questions:

RQ 1. How effective is the method compared to manual inspections?

RQ 2. What can we expect from a software that conforms to the architecture compared to one that does not?

    RQ 2.1. What realistic refactorings are needed to comply to the architecture?

    RQ 2.2. How will refactorings affect source code metrics for coupling, and size?

    RQ 2.3. What modifications to the method are needed to minimize false reports of violations?

    RQ 2.4. How many and what types of violations are not found?

RQ 3. Can the method be automated in a tool and will using such a tool affect the number of violations in software projects?

1.6 Thesis Organization

We use the empirical software engineering method [Bas93; WRH+12] and: 1. Define and verify a problem in reality. 2. Propose a solution to the problem based in engineering and science. 3. Implement the solution. 4. Evaluate the solution and report our experiences (RQ 1 and RQ 2). 5. Demonstrate the usefulness of the solution in a practical environment (RQ 3).

In Chapter 2 we define and verify a problem (Step 1) by presenting a study of erosion in a line of computer games with architectures based on the same pattern. In Chapter 3 we present relevant theory and current techniques for SACC and use this to propose a solution (Step 2). In Chapter 4 we implement (Step 3) and evaluate the solution (Step 4) and in Chapter 5 we create a tool based on the solution and evaluate it in a field experiment (Step 5). In Chapter 6 we present related work. We then provide a discussion of our findings in Chapter 7, threats to validity in Chapter 8, conclusions and contributions in Chapter 9 and end with future works in Chapter 10.

This thesis is primarily based on two published papers, “Evaluation of a Static Architectural Conformance Checking Method in a Line of Computer
“Games” [OTWE14], and “Evolution and evaluation of the model-view-controller architecture in games” [OTWE15]. The contents of Chapters 2 and 4 are mostly from these papers, as well as discussion and validity threats for these parts. A paper, titled Evaluation of an Architectural Conformance Checking Software Service, that describes the field experiment in Chapter 5 has been accepted to the 3rd Workshop on Software Architecture Erosion and Architectural Consistency. It will be published in the proceedings of the 10th European Conference on Software Architecture.
Chapter 2

Erosion in the MVC Pattern

First, we investigate whether software architecture erosion is a problem in the real world. We are particularly interested if using an architectural pattern in itself is enough to control increased complexity in evolving software.

I have previously worked in the game development industry. During this time the focus on architecture was prevalent, and we (the development team) quickly turned to patterns to find solutions to our problems. The problems we wanted to address were to handle large amount of code in user interface, constant changes in user interface technology (real time rendering), ever evolving game rules (fun is a very elusive target), and editing large amounts of game data (editors). We also wanted organizational benefits, like rotating developers between projects and utilizing experts. The solution we all saw was to use the Model-View-Controller architectural pattern. We have a series of games that all use this pattern as the architecture and are developed by the same development team. This provides an opportunity to study erosion in the use of a pattern in a retrospective longitudinal case study [RHRR12] and find if the use of an architectural pattern is enough to control erosion or if there are still problems.

2.1 Background

Model-View-Controller (MVC) is an architectural pattern used in interactive software systems [Ree79; BHS07], systems that need some way of presenting information to a user and to collect the users decisions. The presentation is often visual, ranging from simple text prompts to advanced real-time graphics and interaction is commonly in the form of keyboard and mouse events, but more novel forms of input such as movement gestures and voice recognition can be used. This shows that there is a broad range of technologies for the input and output in interactive systems and that technology is evolving. This is the very problem the MVC pattern address. By separating the system into three
distinct sub-systems and defining their relations, a separation of concerns in the areas of system functionality (Model), handling output (View), and handling input (Controller) can be achieved. The View and Controller together form the user interface. The basic pattern tells us that:

- The model is the element that encapsulates the system functionality. This is where the business rules are implemented and needed data stored.
- The View handles output, the visualization of the data in the Model.
- The Controller handles input and use the Model and View to perform user scenarios.
- The Model may not depend on neither the View nor the Controller.
- If needed, a mechanism to send events from the Model to the View or Controller can be implemented, the Observer pattern [GHJV95] is often used for this.

The advantage of separating Model and user interface is that the Model can be reused with several user interfaces (even in the same system different users have different needs). User interface technology can evolve and changes but should not affect the Model. The distinction between View and Controller, and the exact relations to the Model are not clearly defined in the original pattern. This and advances in user interface technology have led to a number of variants. The MVC-pattern is often referred to as the MV* pattern, where the “*” denotes the different styles of the pattern. We present an overview of three variants of the MVC-pattern, for a more detailed overview of variants see [SW14].

In the Monolithic User Interface architecture variant, there is no real separation of View and Controller. Interaction and visualization is often done through well defined, standardized, user interface components, so called widgets that are defined by an API. The low level input (mouse clicks and key presses) is handled by these widgets and higher level events are handled in an application specific way. The user interface (layout of widgets) can be declared (for example using XML) and customized (color, border styles, etc.). This architecture is good for applications where the user interface should be standardized (use standard widgets to look and work like other applications), and where the interaction is direct—there is no need to move the user through complicated user scenarios. This type of application is a typical single user desktop application, for example a word processor. An example of this type of architecture is the Document-View architecture that Microsoft recommends for applications developed with their Microsoft Foundation Classes (MFC) framework.

In Supervising Controller [Fow06b], the Controller is the director orchestrating the flow of events from the View to executing functionality in the Model. Relations between View and Controller are defined as: 1. The View may not depend on the controller; and 2. The View can only read data from the Model, only the Controller can execute functions that change the data. In Passive
View [Fow06a] variant, the View entities have no relations to the Model but rather work on data structures defined by itself or primitive data types. This variant provides the opportunity to also reuse the view (as it is independent of controller and model).

The Supervising Controller and Passive View are suitable when a more custom user interface is needed (for example a specific visualization of the data in the model) and/or there are more complex user scenarios.

2.2 Method

We analyze how the same team of developers has implemented the MVC pattern in a sequence of projects. Differences in implementations can show architectural erosion is caused by a lack of understanding, mistakes, and convenience. This would confirm the problem of erosion when using an architectural pattern and provide information on how to construct a compliance checking method.

To determine the implemented architecture of each project, we perform a systematic analysis of the C++ source code. The analysis is focused on the user interface, and how interaction and visualization are handled by and divided among the classes. We do not assess the design of the actual game rules (in the Model) since this would require genre-specific analyses. The analysis is grounded in the high-level components of the MVC architectural pattern and how these are separated in terms of relations and responsibility. The projects use a game engine that provides rendering services. It is especially interesting to assess dependencies to these services because they are considered a spot of evolution (rendering engine changes). The View can be considered a layer of indirection between the Controller and the rendering services. The analysis should determine if such a layer exists and whether it is bypassed. We also want to determine if the View has multiple specialized access points or not. The analysis is performed as follows:

1. Create a UML class diagram that show key classes, interfaces and dependencies.
2. Describe as implemented responsibilities and key characteristics of each class and interface by inspecting operation signatures and implementation.
3. Create tags for responsibilities and characteristics and annotate each class and interface with corresponding tags.
4. Synthesize classes and interfaces based on tags and dependencies to produce final architectural variations.
5. Compare architectural variations and note differences.

2.3 Results

We performed an architectural analysis of five game projects developed by the same development team in the same company, all using the MVC pattern as
Table 2.1: Games under analysis

<table>
<thead>
<tr>
<th>Name</th>
<th>Year</th>
<th>Genre</th>
<th>Status</th>
<th>Rendering Engine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time Breaker</td>
<td>2005</td>
<td>Action Arcade</td>
<td>Published</td>
<td>OpenGL</td>
</tr>
<tr>
<td>Frontline</td>
<td>2006</td>
<td>First Person Shooter</td>
<td>Prototype</td>
<td>OpenGL</td>
</tr>
<tr>
<td>TWTPB</td>
<td>2008</td>
<td>Shoot 'Em Up</td>
<td>Published</td>
<td>DirectX</td>
</tr>
<tr>
<td>Hero</td>
<td>2009</td>
<td>Isometric RPG</td>
<td>Prototype</td>
<td>DirectX</td>
</tr>
<tr>
<td>Gears of Love</td>
<td>2010</td>
<td>3D platform</td>
<td>Prototype</td>
<td>DirectX</td>
</tr>
</tbody>
</table>

In the analysis we found three distinct evolutions of the use of the MVC-pattern. We present the synthesized architecture variants and the game projects it was found in below.

2.3.1 Time Breaker and Frontline

Both Time Breaker and Frontline have a clear separation of Model and user interface, cf. Figure 2.1. There is an Observer-like design with an interface (EventTarget) that encapsulates communication between the two major elements. The Game class and classes in the user interface implement this interface. By using the interface, events from the Model can be sent to the user interface and input can be sent into the Model from the user interface. In
Figure 2.2: Architecture for the TWTPB project

Frontline, a game bot class (AIPlayer) implements the same interface, which allows the Model to also send events to bots so they can react like human players. The EventTarget interface essentially describes the game and the game rules at a high level. Classes in the Model handle generic game domain objects (GameToken), e.g., an enemy, as well as game levels and game rules. The design is similar in both games; there are no dependencies from the Model element to the user interface. The user interfaces in both games consist of a few, large classes that implement both rendering and interaction for the major usage scenarios, e.g., play game or edit level. These classes are coordinated by a top level class (MasterView) that is also responsible for the final rendering. Most token classes in the Model have a rendering counterpart in the user interface and most classes in the user interface are connected to the rendering service. In Frontline, Model token-objects are passed as arguments to a rendering function in each type. In TimeBreaker, these are handled via attributes and C++ templates.

2.3.2 TWTPB

TWTPB also has a clear separation between Model and user interface, cf. Figure 2.2. It uses an interface that both the Model and user interface implements. The same Observer-like construct from both Time Breaker and Frontline is used with the difference that a top-level coordinator (MasterController) realize the interface and routes events to the active use-case controller. Most code related to rendering of the user interface is located in a few elements, while other elements are responsible for the execution of usage scenarios, and more complex View-logic and state. We find an example of a sub controller that handles the game menu. Views and Controllers are mixed in both the Time Breaker and Frontline projects, but in TWTPB there is a more clear separation of concerns. We, for example, find an element (ViewCore) responsible for low-level render-
2.3.3 Hero and Gears of Love

In the architectures of Hero and Gears of Love, there is a strict separation between the Model and the user interface, and between the View and the Controller. There are controllers responsible for executing and maintaining state for usage scenarios, and there are View elements responsible for handling rendering that shields the Controller from the rendering parts of the game engine. The Controller elements manage the state for visualization and interaction, collect input at a high-level and refer to the View to control rendering. The top level MasterController receives events from the Model and passes these to the currently active use-case Controller. There are no intended direct dependencies from the Controller elements to the rendering components in the game engine. The View forms a strict layer between the controllers and game engine rendering code, and view elements have read-only dependencies to the Model and no dependencies to the Controller. It also manages low-level input. Complex and performance demanding calculations in the View are executed in a separate thread. The main difference between the games is that the user interface of Hero implements much more functionality, which is typical for role-playing games. This complexity has been handled with more fine-grained Controllers that often use a specialized fine-grained View class with functionality that target a small subset of Controllers.
2.4 Summary

We find three different architectural variants with different degrees of separation between View and Controller elements. In the first variation, there is no separation and the user interface is monolithic. View and Controllers become more and more strictly separated in the second and third variants. In the final variant, the use of rendering services are handled by View elements to a high degree and View elements are more fine grained and specialized for a particular Controller. We consider the implementation in the first variant to be mostly caused by a lack of understanding the pattern. In the second variant, we find widespread signs of mixed responsibilities caused by many small mistakes that have spread. In the newest games there are fewer and less severe mistakes that are probably caused by convenience rather than a lack of knowledge. This suggests that the pattern in itself did not sufficiently control erosion throughout these five games.
Chapter 3

SACC and the MVC Pattern

An architectural pattern is not sufficient to avoid architectural erosion. We focus our attention on Static Architectural Compliance Checking (SACC), a method to find the parts of an implementation of a system that deviates from the intended architecture [DSB12; PTV+10; KLMN06]. First, we provide a background to the area and define some important terms. We then motivate the use of SACC in our particular case and specify requirements on a method for SACC based on this case. We evaluate state-of-the-art techniques and proceed to construct a high-level method for SACC based on the MVC architectural pattern that fulfills our requirements. Finally, we illustrate and evaluate our method with a series of hypothetical scenarios.

3.1 Background

SACC is a process-oriented approach to control architectural erosion [DSB12]. The idea of SACC is to analyze the source code (manually or via tool support) and determine whether dependencies between source code elements conform to the specified dependencies between architectural elements. In the basic case, an analyzed dependency is conformant (allowed), divergent (not allowed), or absent (a dependency that is specified in the intended architecture but not implemented) [PTV+10; KP07]. SACC is introduced to the development process as a verification step. Information on whether dependencies in the implementation conform to the intended architecture or not help developers mitigate problems in the source code and/or architects restructure the intended architecture. The source code and intended architecture can thus be kept consistent. Software inspections can be used to check compliance, however this is labor intensive and can typically not be performed continuously. To reduce the required effort, some approaches to help automate SACC have emerged: Reflexion models, Dependency rules, and Dependency structure matrices. These techniques rely on program dependence graphs.
The following definitions are used throughout the remainder of the thesis. Figure 3.1 shows how these definitions relate to an architectural pattern and its implementation.

**Source Code Element.** A named block in the source code that can contain dependencies to other constructs. It is defined in one or several source code files and can contain other source code elements. A source code element belongs to one and only one architectural element per architectural model. Typical examples include namespaces, packages, classes, structs, and procedures.

**Source Code Dependency.** The implementation of a use between two source code elements, e.g., calling a method from another method, inheriting from one class to another, or using data in one class from within a method in another class.

**Architectural Element.** A high-level construct with defined responsibilities and architectural dependencies. Architectural elements are the building blocks of the architecture’s structural view. Source code elements can be mapped to this level via hierarchical containment (file system placement, package hierarchy, etc.), the naming of source code elements, or manually.

**Architectural Dependency.** The intended use between two Architectural Elements. Source code elements mapped to the Architectural Elements should conform to this intended use. The structural software architecture typically shows architectural elements with their intended architectural dependencies.
3.1.1 Dependency Graphs

Static analysis of source code can be used to create a program dependence graph [FOW87], where nodes represent source code elements and edges represent source code dependencies. In the analysis of an object-oriented system, the syntactical dependencies are typically a combination of inheritance, use of operations, and attributes, expressed in the source code syntax [GDDC97; MMKM94]. If a class, Foo, inherits from, uses, or contains an attribute of a class, Bar, Foo is said to be dependent on Bar and there is a directed edge from Foo to Bar in the graph. From a pragmatic standpoint, Foo cannot function (or compile) without Bar, but Bar can function (or compile) without Foo. Logical or hidden dependencies [CMRH09; GHJ98] are, on the other hand, not explicitly expressed in the syntax of the source code. These are often encoded as identifiers or strings, e.g., a certain name of a table, a column in a database, a class, or an ID of an HTML element. While these dependencies are interesting, they are not within the scope of this thesis.

The dependency graph forms a meta model for the source code that can be traversed to calculate metrics, e.g., coupling metrics, or to find different subsets of edges or nodes [OO84], e.g., all nodes that represent a class that have a direct edge to another class. A general framework to manage and query software dependencies was proposed by Wilde, Huitt, and Huitt [WHH89].

Nodes and edges can be visualized in a graph (which tend to be quite large even in small systems) or in matrix form. Edges can encode a containment hierarchy from operations and attributes, via a class, up to packages and projects. Visualizations or queries can be hierarchical and iteratively refined. While manual inspection of a dependency graph is possible, it becomes hard in practice as the number of classes and dependencies is typically large. Queries can be used to find sub-sets of classes that fulfill some desired criteria [WHH89].

3.1.2 Software Inspections

IEEE defines a standard for systematic software reviews and defines five types of reviews [Ieea]. The software inspection standard is especially interesting as it states that software inspections “Verifies that the software product satisfies its specifications. Verifies that the software product exhibits specified quality attributes. Verifies that the software product conforms to applicable regulations, standards, guidelines, plans, specifications, and procedures.” Quality attributes of software are often a specific focus in architectural design [Som04]. To verify that software fulfills requirements such as maintainability and evolvability, inspections are often the only practical approach available [CLN15]. In the case of SACC, the main input to this process is the intended software architecture and the object of inspection is the system source code. The output is a ranked and classified list of anomalies (architectural violations).

Inspections or reviews of source code are widely adopted by industry [Res10]. These have positive benefits, such as to reduce bugs early in development, to share best practices among developers, and to encourage refactoring to keep the code base as simple as possible. As such, a thorough inspection that involves architects and developers should provide a solid foundation for SACC. Given
enough time and resources, inspections should be able to uncover all violations to the intended architecture. However, as the source code grows, manual inspections do not scale well [DSB12; CLN15]; practitioners report that lack of time is the most problematic part of code reviews [Res10]. Despite this, the use of tools (including tools for SACC), is not widespread [Res10; CLN14; GGIW14].

3.1.3 Reflexion Models

Reflexion Models [MNS95] originally focused on helping an engineer understand a part of a software to perform a task, but have since been modified and improved to support SACC [RLGB+11; Kos13; FKBA07; KS03; BAE+15]. A Reflexion Modeling tool is provided with the intended architecture and then analyzes the source code. Each source code element and its dependencies are lifted to the level of an implemented architectural element. Essentially, this step is a reverse engineering technique that creates the implemented architecture based on the source code. The implemented architecture is then compared to the intended architecture as specified by the architect. High-level architectural elements can be missing (not present in the implementation) or illegal (not present in the intended architecture). Knodel [Kno11] call this part of the compliance checking method the Internal Structure. The aim of this step is to find source code elements that do not follow conventions and will be misplaced when lifted to an architectural element. This is not described as a fully automated process, but rather “The architect has to decide whether or not architectural elements are well composed. Hence, subjective expert ratings are required to identify flaws in the source code model containment.” [Kno11].

If corresponding elements in both the implemented and specified architectures comply, the implemented dependencies of an architectural element can be compared to the specified dependencies of the corresponding intended architectural element. The dependencies can then be classified into three defined types of classifications [MNS95]:

1. **Convergent dependency**: The implementation and specification comply for the dependency.
2. **Divergent dependency**: The implementation contains a dependency that is not present in the specification.
3. **Absent dependency**: The implementation does not contain a dependency that is present in the specification.

Knodel [Kno11] defines a compliance metric by using the harmonic mean F-value of the internal composition and external dependencies for every architectural element in the implementation and specification. A compliance value can thus be determined on the scale 0–100%. However, it is not clear how or if the internal composition metric is automatically determined.

RMs can provide multiple architectural perspectives by allowing several models. You could, for example, have one Reflexion Model for the overall architecture and additional models for each implemented feature. This allows
an architect explore dependencies between architectural elements from different points of view [HB15].

Knodel et al. [Kno11; KMR08] implement a tool based on Reflexion Modeling, SAVELife, that provides detailed and live feedback to developers by analyzing delta changes in the source code. JITACC [BMRA13] is another tool that provides direct feedback for the developer in an IDE, however source code elements are manually mapped in this tool. As far as we are aware, these are the most highly integrated SACC tools. Both require the use of special development environments. Recent tools and methods have focused on using the Reflexion Models technique, possibly because it offers visual modeling and feedback, and multiple perspectives.

3.1.4 Dependency Rules

Dependency Rules or Relation Rules work directly on a dependency graph created from the source code. They exercise rules that govern dependencies between source code elements [PTV+10; KP07]. If source code elements can be mapped to architectural elements and the source code dependencies can be iterated, rules can be defined and checked for each such dependency on an architectural level. This approach has been used to check conformance using, e.g., a generic source code query language [DMSV+08] or more domain specific languages [TV09]. The procedure is straightforward and does not require a high-level specification of the intended architecture, which Reflexion Models does. Rules are instead defined based on dependencies in the intended architecture. The rules that are created use a name-matching pattern to map code elements to architectural elements. The relations between the elements can then be checked for conformance. For example, to define the MVC architecture depicted by Figure 3.1, the following rules need to be implemented:

- *Model* should not depend on *View*.
- *Model* should not depend on *Controller*.
- *Controller* must depend on *Model*.
- *Controller* must depend on *View*.
- *View* should not depend on *Controller*.
- *View* should depend on *Model*.

If we iterate over all source code elements, and use them as source elements and match them to the respective architectural element, we can find outgoing dependencies and target architectural elements. We can then check the corresponding rule that match both source and target to find convergent, divergent, and absent dependencies between architectural elements.

In theory, there is a need to define \( N \times (N - 1) \) rules, where \( N \) is the number of architectural elements, to fully cover all cases. So, the number of rules will grow quickly. The exact implementation of the rules depends on the language used. Query languages are generic languages that focus on performing queries. These queries can be lengthy and complicated, and offer the possibility of greater detail and a scope beyond dependencies. For a brief overview, see Pruijt, Koppe, and Brinkkemper [PKB13]. Domain specific languages are often
declarative and thus easier to create and maintain at the price of being more restricted. Pruijt, Koppe, and Brinkkemper [PKB13] recommends SACC tool developers to “minimize the difference between logical rules, as perceived by the architect, and the technical implementation in the tool.” This indicates that current rule based approaches are too complicated for practical use. However, with a high level of programmability, there is a high level of power. Rules have successfully been used to find violations in medium and large software systems [BK07; Pos03].

3.1.5 Dependency Structure Matrix

A Dependency Structure Matrix (DSM) [SGCH01] is a square matrix with the same source code elements as rows and columns. Each cell denotes a dependency between the elements of the corresponding rows and columns. DSMs can be hierarchical and provide a compact visualization of numerous elements. Sangal, Jordan, Sinha, and Jackson [SJSJ05] describe an approach to manage software architectures using DSMs. The DSM approach offers a compact visualization of the implemented architecture, where architects and developers can discover patterns and possible violations to an intended architecture. Rules can be specified and visualized in a DSM [HCSS08; SJSJ05], much like Dependency Rules. One disadvantage of DSMs compared to Reflexion Models is that DSMs focus on a strictly hierarchical division of source code elements. This works well if a strict hierarchical division is the focus of the architecture. In Reflexion Models, architects have more freedom to explore multiple perspectives. The DSM approach works using the implemented architecture only, so there is no specification of the intended architecture. It can thus be hard to use the DSM approach to assess architectural changes or in the beginning of a project where there might not be as much implementation. The DSM approach is used in the tool suite Lattix Architect.

3.2 Method

First, we define our requirements and investigate the state-of-the-art methods, Reflexion Models and Dependency Rules. We opt to not investigate the DSM technique as it, in our case, can be considered a special form of Dependency Rules. We determine which technique is most suitable to our needs and what challenges remain. We then propose a method that better address our needs. Finally, we illustrate and evaluate the method in a series of hypothetical scenarios.

SACC is a process-oriented method to find which parts of an implementation of a system that deviate from the intended architecture [DSB12; PTV+10; KLMN06]. The architecture we would like to apply SACC to is depicted by Figure 3.2. It is an instance of the MVC-pattern with the addition of the User Interface API (UIAPI) that is outside of the application scope. During the architectural recovery of the five cases presented in Chapter 2 we made the following observations:
• The UIAPI was more strictly used by source code elements in the View in later iterations of the architecture.

• Source code elements in the Controller that directly used the UIAPI tended to be incohesive and have mixed responsibilities. As a result, a larger part of the source code depended on the UIAPI.

• Naming and placement of source code elements in the user interface was not always consistent with their de facto responsibility.

• The later iterations of the implemented architecture show sign of finer grained source code elements in the View where each Controller gets a specific view. This could be a sign of architectural evolution where a new layer in the view is forming.

Based on these observations, we conclude that the use of a pattern did not fully control erosion. While the separation of the Model and the user interface seems to be obvious to developers, they made errors when source code elements should be structured between the View and Controller. These errors were possibly due to confusion or misinterpretation of the pattern, or that it was simply a faster solution. If we can prevent such errors, we can keep the lowest possible amount of source code directly dependent on the UIAPI. This would prevent the formation of a monolithic user interface where changes to the UIAPI directly affects the entire user interface. In our case, this is particularly important as the UIAPI itself is evolving. In the game industry, it is common to develop a reusable game engine in parallel with one or several games. There are other, similar examples from other domains, so this problem is not exclusive to our case. There is also the case where the game engine does not provide all the required functionality, so developers start implementing it themselves using some lower level API (e.g., DirectX) directly. It would be interesting to be able to find such source code elements as they could be reuse candidates for future inclusion in the UIAPI.

The projects studied would benefit from a method that can control dependencies in the user interface to the UIAPI and enforce a separation between the View and Controller architectural elements. The method should not rely on developers experience or discipline. It should also require minimal effort to apply and maintain. It is unlikely that the evolution of the implementation of the architecture will stand still so it is important to support easy modification of the SACC technique in terms of adding architectural elements. This motivates our use of automatic SACC at a pattern level of abstraction. To be effective, the method should be available on a continuous basis for the whole development team. This motivates our requirement to implement automatic SACC as a software service.

The list below summarizes our important requirements of a SACC method. The method should:

1. Be based on the MVC pattern and able to find common mistakes in the implementation of this pattern.
2. Support a heterogeneous and modifiable definition of the UIAPI, i.e., source code elements outside the application scope that are vital for the architecture.

3. Find source code elements in the View that are UIAPI reuse candidates.

4. Be easy to change when architecture evolves.

5. Be adaptable to a service based implementation.

3.3 Comparison of Reflexion Models and Dependency Rules

Reflexion Models is a generic technique for SACC, as such it is not specific for the MVC pattern. However, visual modeling is an important part of the Reflexion Model technique so it is straightforward [PTV+10] to create a structure such as the one depicted by Figure 3.2. Visual modeling can be a disadvantage when defining the UIAPI; if it is not well structured there will be a lot of visual clutter. Typically, Reflexion Models are less expressive than Dependency Rules [PTV+10], which could turn out to be problematic if the UIAPI is not heterogeneous. The Reflexion Model technique is prone to false negatives if only a few source code elements implement a mandatory dependency, e.g., it would essentially only require a single source code element that is lifted to the View architectural element for this dependency to be conformant. This would reduce the ability to find reuse candidates in the View. This problem is also reported by [RLGB+11] in a commercial case study. Visual modeling should provide a powerful abstraction for architectural evolution, however, it is not easy to implement. It would require a substantial graphical user interface implementation in a service.

Dependency Rules are also a generic technique for SACC and thus not specific to the MVC pattern. To our knowledge, there is no specific high-level rule based implementation of the MVC pattern for SACC. The tool
Table 3.1: Comparison of requirements support, Reflexion Models (RM) vs. Dependency Rules (DR)

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Tech</th>
<th>Motivation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Support for MVC pattern</td>
<td>RM</td>
<td>Visual modeling is a high level of abstraction.</td>
</tr>
<tr>
<td>2. Support evolving UIAPI</td>
<td>DR</td>
<td>Rules are more expressive.</td>
</tr>
<tr>
<td>3. Reuse in View</td>
<td>DR</td>
<td>RM tend to generate false negatives in these cases.</td>
</tr>
<tr>
<td>5. Adaption to Service</td>
<td>DR</td>
<td>Rules are easier to implement.</td>
</tr>
</tbody>
</table>

HUSACCT [PKWB14] offer a “layer” module type, which is the highest level of abstraction we have found in a rule based tool. Typically, rules are implemented in a generic query language and depending on the concrete syntax the effort to create the pattern structure [PTV+10] will vary. A generic query language offers the ability to handle a heterogeneous UIAPI that is evolving with language constructs.

Rules are executed on each source code element and should be able to detect any reuse candidates in the View. Rules are essentially source code statements. To fully define an architecture with $N$ architectural elements, $N \times (N - 1)$ rules are needed, which can become hard to maintain and evolve. Rules should not require a complicated user interface implementation, in the simplest case a set of rules could be sent to a service along with the code to analyze.

We summarize our findings from the comparison of Reflexion Models and Dependency Rules in Table 3.1. Both are generic techniques for SACC and none directly support the MVC pattern. The visual modeling used in Reflexion Models provides a powerful abstraction, expressing a pattern visually is more intuitive compared to expressing the pattern using rules. Dependency Rules are more expressive than Reflexion Models and would be more appropriate to use when quick and easy modification of elements outside the application is needed. Dependency Rules execute on the level of source code elements and should not fail to find any mandatory dependencies. If the architecture itself evolves, it can be time consuming and error prone to change Dependency Rules definitions compared to visual modeling techniques. However, Dependency Rules are basically text strings so it should be relatively easy to build a service that let the user define rules or parameters to rules compared to building a visual modeling interface.

The major downside to Reflexion Models is the issue with false negatives that prevents the discovery of reuse candidates in the View. The main downside to Dependency Rules is the low level of abstraction. However, building a method on a rule based foundation and adding a higher level of abstraction should be a feasible approach.
3.4 A Method for Pattern-based SACC

Given that neither Reflexion Models nor Dependency Rules fully support our requirements based on the study in Chapter 2, we create a method for pattern-based architectural conformance checking using a rule-based approach. The goal of the method is to support the requirements and also to be as simple as possible. We want simplicity to minimize maintenance overhead and make it easier for architects to adapt the method to their specific architectures, as well as to facilitate future implementation of the method as a software service.

Algorithm 1 shows the pseudo-code for our method. There are two classification functions that both take a source code element as input and return the corresponding architectural element:

1. **DevC** classifies a source code element according to the developers point of view. Typically, this would use the name and/or containment of the source code element.

2. **ArchC** uses the dependency information of a source code element and compare it to architectural dependencies to make a classification.

Both these classifications essentially form a heuristic representation of how a developer and an architect views a source code element, respectively. If the two classifications do not agree (developer and architect think the source code element belongs in different architectural elements), a third function **DivAbs** determines if there is an absent or divergent dependency that needs to be further inspected based on the classification results from DevC and ArchC.

```
1 forall codeElem ∈ Project do
   /* Classify the source code element according to the
      developers point of view          */
   dc ← DevC(codeElem);
   /* Classify the source code element according to source
      code dependencies                */
   ac ← ArchC(codeElem);
   if dc ≠ ac then
      /* Find the divergent or absent dependencies   */
      DivAbs(dc, ac, codeElem);
   end
```

**Algorithm 1**: Method Overview

The MVC pattern is defined by three architectural elements: Model, View, and Controller (cf. Figure 3.2). The functions DevC and ArchC should thus return one of these elements based on the source code element under analysis, and the function DivAbs takes a pair of these elements as its input. Outside the application scope, there is a fourth architectural element, the UIAPI, which essentially is the platform used by the View elements to provide physical input.
Table 3.2: Table for finding Divergent and Absent dependencies by classification to Model (M), View (V) or Controller (C) by DevC ($dc$) and ArchC ($ac$)

<table>
<thead>
<tr>
<th>$ac = M$</th>
<th>$dc = M$</th>
<th>Absent dependency to the UIAPI</th>
<th>Absent dependency to a View</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ac = V$</td>
<td>Divergent dependency to the UIAPI</td>
<td>Divergent dependency to a View</td>
<td></td>
</tr>
<tr>
<td>$ac = C$</td>
<td>Divergent dependency to a View</td>
<td>Absent dependency to the UIAPI</td>
<td></td>
</tr>
</tbody>
</table>

and output using system devices such as monitor and keyboard. The inclusion of the UIAPI is essential as this provides the ground truth for the ArchC.

We can construct the function ArchC by investigating a source code element’s dependencies to the UIAPI. A source code elements that:

- Directly use the UIAPI can be classified as an architectural View element.
- Indirectly use the UIAPI can be classified as an architectural Controller element.
- Has no dependency on the UIAPI can be classified as an architectural Model element.

This rule can be implemented by calculating the distance from the source code element to the UIAPI in a dependency graph. The function DivAbs can be constructed as a table, cf. Table 3.2.

In Reflexion Models, the architect constructs an architectural model (the intended architecture) and a source code element mapping to this model. The mapping is used to lift the source code elements and source code dependencies to the architectural model level (the implemented architecture). Dependencies at the architectural level can then be compared between the implemented and intended architecture, and violations detected. Our method performs two lifts to the architectural level, one from the developer’s point of view (this is essentially the same as the source code element mapping in Reflexion Models) and another from the architect’s point of view using the source code dependencies. We then compare the two classifications.

Dependency rules are evaluated at the source code element level and each dependency of the element under analysis is examined. First, the destination source code element of the dependency is found. The rules are then scanned for a matching source and destination element pair. If a match is found, the rule is evaluated. This requires that rules for all possible combinations of architectural elements are defined. This can quickly become a large set of rules to maintain, and rules can also become contradictory.
In our example there would be a need to implement six different rules:

1. Model does not depend on Controller.
2. Model does not depend on View.
3. Model does not depend on UIAPI.
4. Controller depends on View.
5. Controller does not depend on UIAPI.
6. View depends on UIAPI.

By classifying based on dependencies implemented as the distance from the UIAPI, there is a benefit of maintenance and possibly performance. However, a drawback is that Dependency Rules can be more detailed and expressive. The ArchC classification can be seen as a heuristic based on an extension of Dependency Rules, e.g., our method could be implemented in a query language with features to compute the distance to source code elements in a dependency graph.

### 3.4.1 Hypothetical Scenarios

In this section, we present a simple example application and some scenarios that show how possible architectural violations are detected. The scenarios are selected to evaluate requirements 1, 3 and 4. For the remaining requirements we rely on the expressiveness of the underlying dependency graph and query language, and the simplicity of the method to be suitable for a service based implementation. These two requirements will be discussed later.

The application is a dice game where two dice are rolled, their results are displayed, and if the sum is seven or greater the player won the game. The UIAPI is defined as console input and output commands and the source code element package name is used in the DevC function.

### Conformant Version

Figure 3.3 depicts a UML class diagram of the conformant version of the game. This diagram compares to a dependency graph of the source code entities (classes and interfaces) and their dependencies (association, dependency, inheritance, realization). The Model contains three source code elements that aggregates the two dice, evaluates the game rules, and sends events to its listeners when the respective die is rolled (Observer pattern). The View contains one abstract class, ConsoleView, with two concrete implementations, English and Swedish. They all use the System.Console package for console input (read key) and output (write line). Finally, the Controller contains one class, Player that is parametrized with a concrete View class and the DiceGame that implements the DiceGameObserver interface and forwards events to its view when a die is rolled. The Controller loops the game for as long as the player wants to play the game. All types are classified as the same architectural element and thus no violations are detected, cf. Table 3.3.
Table 3.3: Conformant Classification Results

<table>
<thead>
<tr>
<th>Source code element</th>
<th>Distance</th>
<th>ArcC</th>
<th>DevC</th>
</tr>
</thead>
<tbody>
<tr>
<td>DiceGame</td>
<td>—</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>Dice</td>
<td>—</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>DiceGameObserver</td>
<td>—</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>Player</td>
<td>2</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>ConsoleView</td>
<td>1</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>English</td>
<td>1</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>Swedish</td>
<td>1</td>
<td>V</td>
<td>V</td>
</tr>
</tbody>
</table>

Table 3.4: Erroneous Observer Implementation Results

<table>
<thead>
<tr>
<th>Source code element</th>
<th>Distance</th>
<th>ArcC</th>
<th>DevC</th>
</tr>
</thead>
<tbody>
<tr>
<td>DiceGame</td>
<td>3</td>
<td>C</td>
<td>M</td>
</tr>
<tr>
<td>Dice</td>
<td>—</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>DiceGameObserver</td>
<td>—</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>Player</td>
<td>2</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>ConsoleView</td>
<td>1</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>English</td>
<td>1</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>Swedish</td>
<td>1</td>
<td>V</td>
<td>V</td>
</tr>
</tbody>
</table>
Figure 3.4: Erroneous Observer Implementation UML Class Diagram

Table 3.5: Erroneous Observer Interface Placement Results

<table>
<thead>
<tr>
<th>Source code element</th>
<th>Distance</th>
<th>ArcC</th>
<th>DevC</th>
</tr>
</thead>
<tbody>
<tr>
<td>DiceGame</td>
<td>—</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>Dice</td>
<td>—</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>DiceGameObserver</td>
<td>—</td>
<td>M</td>
<td>C</td>
</tr>
<tr>
<td>Player</td>
<td>2</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>ConsoleView</td>
<td>1</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>English</td>
<td>1</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>Swedish</td>
<td>1</td>
<td>V</td>
<td>V</td>
</tr>
</tbody>
</table>

Erroneous Observer Implementation

Assume that the developer makes a quick fix and allows the DiceGame to contain Player controller classes directly instead of creating an Observer interface, cf. Figure 3.4. This creates an indirect divergent dependency from the DiceGame to the UIAPI. As a result, the intended Model class DiceGame will be classified as a Controller, cf. Table 3.4. This is an example on how a circular dependency between architectural elements Model and Controller is avoided.

Erroneous Observer Interface Placement

Assume that the developer places the DiceGameObserver in the Controller package cf. Figure 3.5. The DevC classification of DiceGameObserver will be Controller compared to the ArchC classification of Model, cf. Table 3.5.
View Responsibility in Controller

Assume that the developer makes a quick fix and use console I/O directly in the Player controller class cf. Figure 3.6. This means that the Player will be classified as a View, cf. Table 3.6.

View Responsibility in Model

Assume that the developer makes a quick fix and use console I/O directly in the Dice class, cf. Figure 3.7. This means that the Dice will be classified as a View and as a consequence DiceGame will be classified as a Controller, cf. Table 3.7. This is a problematic situation where a violation is reported in a type that does not contain the root of the problem. However, we handle this as a special case in our actual implementation.
Table 3.7: View Responsibility in Model Results

<table>
<thead>
<tr>
<th>Source code element</th>
<th>Distance</th>
<th>ArcC</th>
<th>DevC</th>
</tr>
</thead>
<tbody>
<tr>
<td>DiceGame</td>
<td>2</td>
<td>C</td>
<td>M</td>
</tr>
<tr>
<td>Dice</td>
<td>1</td>
<td>V</td>
<td>M</td>
</tr>
<tr>
<td>DiceGameObserver</td>
<td>—</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>Player</td>
<td>2</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>ConsoleView</td>
<td>1</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>English</td>
<td>1</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>Swedish</td>
<td>1</td>
<td>V</td>
<td>V</td>
</tr>
</tbody>
</table>
Reusable Class in View

Assume that a new class, CoolConsole, is created in the View that is not dependent on System.Console, but provides generic console functionality, cf. Figure 3.8. This class is thus a candidate for reuse in the System.Console context. This class will be classified as a Model element and View element respectively, cf. Table 3.8, and will be reported as a violation.

New Architectural Element

Assume that the architect decides that a new architectural element should be introduced. This element is represented by the wrapper class ViewWraper,

Table 3.8: Reusable Class in View Results

<table>
<thead>
<tr>
<th>Source code element</th>
<th>Distance</th>
<th>ArcC</th>
<th>DevC</th>
</tr>
</thead>
<tbody>
<tr>
<td>DiceGame</td>
<td>—</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>Dice</td>
<td>—</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>DiceGameObserver</td>
<td>—</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>Player</td>
<td>2</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>ConsoleView</td>
<td>1</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>English</td>
<td>1</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>Swedish</td>
<td>1</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>CoolConsole</td>
<td>—</td>
<td>M</td>
<td>V</td>
</tr>
</tbody>
</table>
Figure 3.8: Reusable Class in View UML Class Diagram

Figure 3.9: New Architectural Element UML Class Diagram
Table 3.9: New Architectural Element Results

<table>
<thead>
<tr>
<th>Source Code Element</th>
<th>Distance</th>
<th>ArcC</th>
<th>DevC</th>
</tr>
</thead>
<tbody>
<tr>
<td>DiceGame</td>
<td>—</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>Dice</td>
<td>—</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>DiceGameObserver</td>
<td>—</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>Player</td>
<td>3</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>ConsoleView</td>
<td>1</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>English</td>
<td>1</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>Swedish</td>
<td>1</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>ViewWrapper</td>
<td>2</td>
<td>VS</td>
<td>VS</td>
</tr>
</tbody>
</table>

responsible for creating concrete ConsoleView objects and providing a uniform interface for the Controller elements to use, cf. Figure 3.9. This requires us to change the distance measurement and let Controllers be defined by distance $> 2$, ViewService elements distance $= 2$, and View elements distance $= 1$, cf. Table 3.9.

3.5 Summary

We have constructed a method to perform SACC based on the MVC pattern. It is based on performing two classifications of each source code element to the elements in the architecture, one from the developer’s point of view and one from the architect’s point of view. The two are then compared. If they are inconsistent, we determine if this is due to an absent or divergent dependency. The architectural classification is based on the distance between a source code element and some source code elements that provide a ground truth (UIAPI). From the examples provided, we see that there is a need to parametrize the method with a definition of the ground truth and the distances used to form different layers in the MVC architecture.
Chapter 4

Evaluation of SACC of the MVC Pattern

Our method is a heuristic approach to SACC. This implies that it will likely not produce perfect results; there will be situations that the method cannot handle, it could miss some architectural erosion, or it could report erosion that does not exist. We want to know how effective the method is (RQ 1) and what refactorings are required for the code base to conform to the results of the method (RQ 2.1) as well as how these refactorings affect size and coupling metrics (RQ 2.2). This, and knowledge about what kind of erosion the method does not detect (RQ 2.4) gives an indication of what to expect from a project that uses the method from start. Finally, we want to know if we can adjust our method to minimize false violation reports (RQ 2.3).

4.1 Method

To measure effectiveness and find refactorings, we compare the results from manual inspection to those from the method on a number of projects. We define:

**Erosion** A source code element that contains one or several signs of architectural breakdown. This does not necessarily mean that the architecture has been obviously affected yet (i.e., an illegal dependency), but it is in the process of becoming so.

**Violation** A source code element that is reported by the method to be affected by erosion.

**True Positive (TP)** an existing erosion that is a violation.

**False Positive (FP)** a non-existing erosion that is a violation.

**False Negative (FN)** an existing erosion that is not a violation.
Table 4.1: Inspection Protocol

<table>
<thead>
<tr>
<th>Erosion</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Illegal dependency</td>
<td>Check if there are illegal dependencies according to the MVC pattern.</td>
</tr>
<tr>
<td>Model Logic in UI</td>
<td>Check if the user interface implements business rules</td>
</tr>
<tr>
<td>UI Specific Model</td>
<td>Check if a Model services, and is tailored for, a specific View</td>
</tr>
<tr>
<td>Incohesive Model</td>
<td>Check if a Model has several unrelated responsibilities.</td>
</tr>
<tr>
<td>Painted Type</td>
<td>Check if a generic type is masking a dependency or a more complex type.</td>
</tr>
</tbody>
</table>

4.1.1 Manual Inspection

To find erosion, we perform a manual inspection that will act as the ground truth. It is based on a protocol aimed at finding erosion, cf. Table 4.1, constructed from known common mistakes in the MVC pattern and our own experience. The protocol is used for each source code element.

4.1.2 Method Execution

To find violations, we execute the method in a semi-automatic way. We use a query language available in a code analysis tool to construct a query to compute the distance from each source code element to the UIAPI, and execute it on each project without modification. This data is used to perform the architect classification, $ArchC$.

To perform the developer classification, $DevC$, we manually inspect naming conventions, namespaces, IDE specific organization, and if the source code element is responsible for maintaining state for user interaction ($Controller$), rendering ($View$) or game rules ($Model$).

4.1.3 Effectiveness

When we know the true positives, false positives, and false Negatives, we can calculate the precision and recall. Precision tells us the ratio of violations that are also erosion in relation to all violations. It is calculated according to Equation 4.1. A low precision indicates that we detect many violations that are not erosion. This risks that developers spend time trying to find erosion that does not exist.

$$P = \frac{TP}{(TP + FP)} \quad (4.1)$$

Recall is the ratio between violations that are erosion and all erosion. It is calculated according to Equation 4.2. A low recall indicates that much of the erosion is not detected.
\[ R = \frac{TP}{(TP + FN)} \]  

(4.2)

4.1.4 Refactoring

We study all existing violations in the source code (true and false positives), and suggest appropriate and realistic refactorings, or method change needed to conform to the architecture. These changes are then implemented, and we re-run the method and again check for conformance. We inspect the False Negatives to learn what kind of erosion our method does not find.

After all violations has been addressed, we have the opportunity to compare a non-compliant version with a compliant version of the same software. This gives us the opportunity to study if (software) metrics results change from an architecturally driven refactoring and if so, how? Bourquin and Keller [BK07] suggest that refactorings should be driven by architectural erosion and evaluated by metrics. A static code analysis tool is used to measure the outgoing dependencies (Fan out/Efferent coupling) and Lines of Code (LoC) of each source code element affected by changes of the different versions of the softwares. Saraiva [Sar13] finds that these are the most widely used metrics for maintainability. Changed source code elements will be determined by using a source code management system to compare the projects before and after refactoring.

4.2 Results

We selected four games that use the MVC pattern with respect to age, genre, and maturity (in terms of readiness for commercial release). All games were developed in object-oriented C++ using the Visual Studio IDE. The games use the same game engine (EngineIII) for rendering (using the DirectX API) and common functionality. Table 4.2 provides basic information about the projects and the game engine. We measure LoC as the number of lines of code in C++ source and header files using CLOC v. 1.60. Type metrics are measured by CPPDepend v. 3.1.0.6445 Professional Edition. CPPDepend is also used for source code analysis, the query, and metrics. Figure 4.1 depicts the games’ intended architecture and the UIAPI (dxrnd) used by the View. Note that some architectural elements in the game engine and other domain specific elements have been left out to reduce the complexity.

4.2.1 Effectiveness

We inspected the implementation of each type and found erosion using the inspection protocol as well as the naming, organization, and responsibility of the types to perform the developer classification (DevC). We found Model elements to be well encapsulated within a namespace in general. In some cases, an in-depth inspection was required to classify Controller and View types. Elements that were not part of the MVC-architecture related to copy
Figure 4.1: The Game Architecture. Note that \textit{dxrnd} is the UIAPI.

<table>
<thead>
<tr>
<th>Project</th>
<th>LoC</th>
<th>#Types</th>
<th>Genre</th>
</tr>
</thead>
<tbody>
<tr>
<td>EngineIII</td>
<td>46,390</td>
<td>225</td>
<td>Game engine</td>
</tr>
<tr>
<td>Gears of Love</td>
<td>5,362</td>
<td>56</td>
<td>3D platform</td>
</tr>
<tr>
<td>Pirate Quest</td>
<td>12,046</td>
<td>68</td>
<td>Puzzle</td>
</tr>
<tr>
<td>Hero</td>
<td>21,639</td>
<td>167</td>
<td>Isometric RGP</td>
</tr>
<tr>
<td>TWTPB</td>
<td>14,212</td>
<td>137</td>
<td>Shoot ’Em Up</td>
</tr>
</tbody>
</table>
from t in Application.Types
where t.ParentProject.Name == "ProjectName"
let d = t.DepthOfIsUsing("dxrnd")
select new { t, d }

Listing 4.1: LINQ implementation of measuring distance from source code element to UIAPI

Table 4.3: Effectiveness of the classification results, in terms of true and false positives, false negatives, precision, and recall.

<table>
<thead>
<tr>
<th>Project</th>
<th>TP</th>
<th>FP</th>
<th>FN</th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gears of Love</td>
<td>10</td>
<td>1</td>
<td>6</td>
<td>0.90</td>
<td>0.63</td>
</tr>
<tr>
<td>Pirate Quest</td>
<td>15</td>
<td>3</td>
<td>2</td>
<td>0.83</td>
<td>0.88</td>
</tr>
<tr>
<td>Hero</td>
<td>28</td>
<td>9</td>
<td>8</td>
<td>0.76</td>
<td>0.79</td>
</tr>
<tr>
<td>TWTPB</td>
<td>23</td>
<td>5</td>
<td>9</td>
<td>0.82</td>
<td>0.72</td>
</tr>
<tr>
<td>Total</td>
<td>76</td>
<td>18</td>
<td>25</td>
<td>0.81</td>
<td>0.75</td>
</tr>
</tbody>
</table>

protection and automatic testing were excluded from the analysis. The manual inspection was performed by two persons and took approximately 180 person hours, divided over six weeks.

We used CPPDepend to construct dependency graphs for each of the games and LINQ to construct a generic query. CPPDepend handles common types of dependencies such as inheritance, declarations, type definitions, and references. We used the same query on all projects without any modification (except the project name). The query string was constructed as in Listing 4.1. The query measures the depth of dependencies from an element to the dxrnd (UIAPI) namespace. We then performed the architectural classification using the depth of each element. An element with a direct dependency (distance = 1) to the UIAPI is classified as a View element, an element with an indirect dependency (distance > 1) is classified as a Controller element, and an element with no dependency (distance = 0) is classified as Model element.

We compared the architectural classification and developer classification. If they did not agree we marked the element as a violation. Violations and erosion were then compared for each element to find True Positive, False Positive and False Negative. Table 4.3 present the results as well as precision and recall.

4.2.2 Refactoring

We selected two projects, Gears of Love and TWTPB, for refactoring. We decided to use these to study a project under development (Gears of Love) and a completed project (TWTPB). Based on the extent of erosion, TWTPB was then selected for implementation of refactorings as this project suffered from the most serious erosion of the two.

Upon inspection of the violations, we did not find any violations in the Model architectural element, only in the View and Controller elements. Ta-
Table 4.4: Found Refactorings in Gears of Love (GoL) and TWTPB

<table>
<thead>
<tr>
<th>Refactoring</th>
<th>GoL</th>
<th>TWTPB</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Split Type</td>
<td>0</td>
<td>3</td>
<td>Split into separate View and Controller</td>
</tr>
<tr>
<td>Move Type</td>
<td>6</td>
<td>6</td>
<td>Move type to correct component, possibly reuse</td>
</tr>
<tr>
<td>Move Function</td>
<td>0</td>
<td>4</td>
<td>Move function to the correct type</td>
</tr>
<tr>
<td>Move Variable</td>
<td>3</td>
<td>0</td>
<td>Move variable into the correct type</td>
</tr>
<tr>
<td>Encapsulate Type</td>
<td>1</td>
<td>10</td>
<td>Move type into parent type</td>
</tr>
<tr>
<td>Refine Method</td>
<td>1</td>
<td>5</td>
<td>Refine the method to avoid false positive</td>
</tr>
</tbody>
</table>

Table 4.4 provides information of suggested refactorings based on the analysis. The table is ordered in descending order based on the scope of the refactoring. The suggested types of refactorings are:

**Split Type** A type has been seriously affected by erosion. It contains multiple dependencies that do not conform to the architecture and functions with mixed responsibilities. This refactoring would require a substantial effort to resolve.

**Move Type** A whole type is misplaced and should be moved to the right architectural element. This could include moving the element outside of the application into a supporting framework. In this reuse case there can be a high level of effort required to remove application specific responsibilities and providing a more generic approach.

**Move Function** A function in a type is affected by erosion and should be moved into a more appropriate type.

**Move Variable** A variable has been misplaced and should be moved into a more appropriate type.

**Encapsulate Type** A type should be encapsulated into a parent type in the same architectural element. For example smaller elements that is used for data passing.

**Refine Method** To avoid False Positives the query should be tweaked.

The source code refactoring of Gears of Love was straightforward and took one working day. It mainly involved one variable responsible for handling mouse-scroll-wheel input that needed to be moved to the view (Move Variable). The refactoring of Gears of Love involved six instances of reuse candidates (Move Type). These were all geometric hedra (e.g., Tetrahedron) that could be moved to the math library of the game engine.
Table 4.5: False Positives in Gears of Love (GoL) and TWTPB

<table>
<thead>
<tr>
<th>False Positive</th>
<th>GoL</th>
<th>TWTPB</th>
<th>Heuristic Refinement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dependency outside UIAPI</td>
<td>0</td>
<td>2</td>
<td>Increase the scope of the UIAPI</td>
</tr>
<tr>
<td>Data Type Class</td>
<td>0</td>
<td>3</td>
<td>Use lines of code threshold</td>
</tr>
<tr>
<td>Dependency Not Detected</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.6: False Negatives in Gears of Love (GoL), TWTPB, Pirate Quest (PQ), and Hero.

<table>
<thead>
<tr>
<th>Erosion</th>
<th>GoL</th>
<th>TWTPB</th>
<th>PQ</th>
<th>Hero</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Illegal Dependency</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>Tool Problem, scope of UIAPI</td>
</tr>
<tr>
<td>Has Model Logic</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>UI Specific Model</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Incohesive Model</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Painted Type</td>
<td>2</td>
<td>7</td>
<td>0</td>
<td>4</td>
<td>Related to Color as DWORD</td>
</tr>
</tbody>
</table>

The source code refactoring of the TWTPB project took nine working days. Our main effort was spent on the Split Type refactorings. The refactoring involved six instances of reuse candidates (Move Type). These all involved input via a game pad and the use of DirectInput. They should be moved to the UIAPI of the game engine, but we opted to encapsulate these in the dxrnd (UIAPI) namespace but keep them in the project due to time limitation. Refactoring them to a generic reusable state would have required additional effort.

False Positives

In TWTPB, the false positives were due to source code elements in the View incorrectly classified as Model elements. This was in the form of use of types outside the defined UIAPI and classes that acted as pure data types for passing data. The query could be refined to increase the scope of the UIAPI and include a LoC threshold to filter data type classes. In Gears of Love the false positive was a tool error. A dependency to the UIAPI in the source code was not detected correctly; a View element was classified as a Model element. We regard this as a minor tool problem since other dependencies of the same type were correctly detected.

False Negatives

Table 4.6 presents types of erosion that was not detected by the method in its original form. Problems related to illegal dependencies and painted type erosion could be detectable with a more flexible definition of the UIAPI. This would remove 18 of 25 false negatives.

Three major problems stand out. Pirate Quest has a sub-game with game rules that is implemented entirely in the user interface. In Hero, audio is
Table 4.7: Couplings in TWTPB before and after Refactoring

<table>
<thead>
<tr>
<th>Class</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>GameView</td>
<td>64</td>
<td>46</td>
</tr>
<tr>
<td>ViewCore</td>
<td>45</td>
<td>53</td>
</tr>
<tr>
<td>Menu</td>
<td>38</td>
<td>30</td>
</tr>
<tr>
<td>SoundBox</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Particle</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>ViewManager</td>
<td>28</td>
<td>16</td>
</tr>
<tr>
<td>SpriteView</td>
<td>19</td>
<td>12</td>
</tr>
<tr>
<td>PlayerViewData</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>ViewAssets</td>
<td>35</td>
<td>36</td>
</tr>
<tr>
<td>Soundbox.Sound</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>ViewAssets.BGTile</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>XInputDevice</td>
<td>7</td>
<td>—</td>
</tr>
<tr>
<td>DInputDevice</td>
<td>14</td>
<td>—</td>
</tr>
<tr>
<td>InputDevice</td>
<td>6</td>
<td>—</td>
</tr>
<tr>
<td>InputDeviceFactory</td>
<td>20</td>
<td>—</td>
</tr>
<tr>
<td>GameController</td>
<td>—</td>
<td>37</td>
</tr>
<tr>
<td>GameView.EnergyAnimationState</td>
<td>—</td>
<td>1</td>
</tr>
<tr>
<td>GameView.HealthAnimationState</td>
<td>—</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count</td>
<td>15</td>
<td>14</td>
</tr>
<tr>
<td>Sum</td>
<td>298</td>
<td>254</td>
</tr>
<tr>
<td>Average</td>
<td>19.87</td>
<td>18.14</td>
</tr>
<tr>
<td>SD</td>
<td>18.52</td>
<td>18.38</td>
</tr>
<tr>
<td>Median</td>
<td>14.00</td>
<td>9.50</td>
</tr>
</tbody>
</table>

sent directly from the Model to elements in the user interface in the form of integer identifiers. For Gears of Love, the element representing the game world implemented functionality for both playing and editing the game. These problems would be hard to detect.

4.2.3 Metrics

In the process of refactoring, some source code elements changed, moved, or were created. Table 4.7 shows the Coupling of each changed element in the TWTPB project. We opted to not study the Gears of Love project as source code changes were minimal.

Note that the number of types has decreased since the input related classes were recognized as reusable game engine classes and were moved to the UIAPI. Three new classes were added due to the split of the eroded GameView class into a View and the new GameController class. The menu was refactored into a sub-Controller of the GameController and the view responsibilities were moved to the ViewCore. This resulted in an increased coupling in the ViewCore. The ViewManager was heavily eroded, with direct use of the UIAPI when it
Figure 4.2: Coupling changes in TWTPB before and after refactoring

Table 4.8: LoC per Architectural Element before and after refactoring

<table>
<thead>
<tr>
<th>Element</th>
<th>Pre</th>
<th>Post</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>2,456</td>
<td>2,456</td>
</tr>
<tr>
<td>View</td>
<td>2,411</td>
<td>2,048</td>
</tr>
<tr>
<td>Controller</td>
<td>287</td>
<td>664</td>
</tr>
<tr>
<td>Total</td>
<td>5,154</td>
<td>5,168</td>
</tr>
</tbody>
</table>

actually mainly had Controller responsibility. View responsibilities were once again moved to the ViewCore. In retrospect, we should have created specialized View classes for this.

A boxplot reveals that Coupling is less extreme and, in general, more evenly distributed in the post version. A density plot confirms this finding, though the effect is small, cf. Figure 4.2. The most important result is possibly that the most coupled class, the GameView, reduced its Coupling from 64 to 46.

Table 4.8 shows the LoC in the two versions. Refactorings did only affect the user interface, so only View and Controller elements are interesting. The table shows that the total LoC in the architectural elements is almost identical before and after refactoring. However, the LoC in the user interface is evenly distributed between the View and Controller elements after the refactoring. Note that in this data, LoC is measured by CPPDepend (not CLOC) so only executable statements are reported, not declarations. This is a very conservative approach to measuring the LoC.

4.3 Summary

The effectiveness of our method, in terms or precision and recall is 0.81 and 0.75, respectively, for the total of our five cases (cf. Table 4.3). We found that
refactorings are needed on different levels of effort (cf. Table 4.4). Metrics relevant to maintainability, i.e., source LoC and Coupling are more evenly distributed in a conformant version of the same software (cf. tables 4.7 and 4.8, and Figure 4.2) compared to the non-conformant version.

To avoid false positives the scope of the UIAPI definition needs to be changed and a LoC threshold should be applied to remove small data type classes. 18 of 25 false negatives are related to issues that could be managed by a flexible UIAPI definition (cf. Table 4.6).
Chapter 5

Automatic SACC of the MVC Pattern

In this chapter we are interested in creating a usable tool based on our method and investigate RQ 3. We also want to know if we can generalize our method to a new context and application domain.

5.1 Background

A source code management system is important in software development. It provides a controlled repository for the source code of a system. Examples of such systems are CVS, Subversion and Git. In recent years, such tools are offered as software services, often with additional features such as issue tracking and Wiki pages. To fully benefit from this development, a tool for SACC should be implemented as a software service that can integrate with a source code management system. This makes SACC available to the whole development team on a continuous basis, which is one of our main requirements.

There are several tools that offer SACC [Kno11; Pru15; PKB13]. Most are based on Reflexion Models or Dependency rules. Despite the availability of tools, industry adoption is not widespread [DSB12]. Table 5.1 provides an overview of what we consider to be the tools described in literature and used by professionals. The focus of this overview is to find how a tool can be integrated into the development work flow and whether the tool is actively maintained (as indicated by latest release date). The information is taken from the websites of the respective tool.

In our analysis (c.f. table 5.1 most tools used in academia or maintained by academic institutions seem to be discontinued or unavailable. There are a few commercial, large tool suites where SACC is one part. Typically, these seem to be costly and also require an investment in time and training to get started. Currently available solutions that are actively maintained seem to focus on providing integration in development work flow via either a desktop application (that requires manual execution of compliance checking) or via command-line
Table 5.1: SACC Tools

<table>
<thead>
<tr>
<th>Name</th>
<th>Updated</th>
<th>Integration</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConQAT AA</td>
<td>2005</td>
<td>Command Line, GUI for configuration and inspection of results</td>
</tr>
<tr>
<td>dTangler</td>
<td>2009</td>
<td>GUI, some form of build integration support</td>
</tr>
<tr>
<td>Macker</td>
<td>2003</td>
<td>Command Line</td>
</tr>
<tr>
<td>SAVE</td>
<td>2008</td>
<td>Eclipse plugin</td>
</tr>
<tr>
<td>Sonargraph Architect</td>
<td>2015</td>
<td>Plugins for the project build tools ANT and Maven</td>
</tr>
<tr>
<td>Structure 101</td>
<td>2013</td>
<td>Command Line Utilities, Webb app</td>
</tr>
<tr>
<td>Bauhaus</td>
<td>2006 (?)</td>
<td>Desktop Application, (updated is last time of webpage update)</td>
</tr>
<tr>
<td>jDepend</td>
<td>2008</td>
<td>Support for build tool ANT and unit testing with JUnit</td>
</tr>
<tr>
<td>Lattix Architect</td>
<td>2016</td>
<td>Command Line</td>
</tr>
<tr>
<td>Semmle QL</td>
<td>2015</td>
<td>Desktop Application, IDE-plugins</td>
</tr>
<tr>
<td>Sotoarch</td>
<td>2015</td>
<td>Desktop Application</td>
</tr>
<tr>
<td>Husacct</td>
<td>2015</td>
<td>Desktop Application</td>
</tr>
<tr>
<td>JITACC</td>
<td>2014</td>
<td>Eclipse plugin for roundtrip engineering</td>
</tr>
</tbody>
</table>
tools. Command-line tools offer integration, but require configuration and servers to run them. This also requires an investment in both time and money for the developer. Integration into developer IDEs is another approach that can provide developers with immediate feedback on violations.

Continuous and early use of SACC seems to be important for getting developers to actually address problems [BK07; Kno11; KMR08]. This approach has the main drawback of locking the development team into an IDE. The low rate of industry adoption indicates that current tools are not a sufficient return on investment [DSB12]. Development of tools also seem to be abandoned and this pose a risk for developers, as a big investment in a tool suite and training can quickly be lost if the tool is not up to par with the latest language features, IDE versions, etc.

We note that as far as we can find, there is no tool that offer SACC as a software service. A service oriented approach would offer a high level of integration in development work flow and lower the investment and risk involved with big tool suites.

5.2 Method

To test a fully automatic version of the method in a different context, we implemented a tool for automatic detection of problems in MVC for PHP-based projects. This tool was offered as a service in a student project assignment context to investigate the feasibility of fully automating the method, to find points of improvements, and to compare projects that used the service with those who did not.

We want to answer two questions: 1. How should the method be refined to this new context?; and 2. Is there a difference in the number of violations detected in projects that use the tool and projects that do not use the tool? This aims at answering our third research question: Can the method be automated in a tool and will using such a tool affect the number of violations in software projects.

We also investigate what happens to the violations during the entirety of a project, not just the end result. Continuous use of a tool should prompt developers to remove violations earlier, and not let them exist and accumulate for a long period. Our hypothesis is that developers that use the tool will correct violations more promptly, and possibly learn from feedback and not introduce violations. Tool users will thus have a lower number of reported violations at a randomly selected time during the project compared to non-tool users. Our hypotheses are:

\( H_0 \): The number of reported violations is the same or greater in a project that use the tool compared to a project that do not, at a random point in time.

\( H_1 \): The number of reported violations are lower in a project that use the tool compared to a project that do not, at a random point in time.
This study is conducted as a field-experiment. Students that are randomly assigned to use the tool are provided with further instructions and we log data for each project, other students form a control group. After the projects have terminated, we sent out a survey to all students. Lastly, we analyze the experiment projects and compare projects where the tool was used with projects that did not use the tool to find if we can reject the null hypothesis ($H_0$).

5.2.1 Environment

Our tool, Rulzor, aims to offer SACC as a service and aid developers to promptly correct architectural violations. Figure 5.1 provides an overview of Rulzor, its input, output, and external actors. Rulzor is notified when developers contribute source code and changes to the project repository. The architect supplies Rulzor with heuristics for developers and pattern classification and Rulzor performs the analysis. Any violations are raised as issues posted to the project repository that developers (and the architect) can see and react on. There is also a more detailed log that the architect can analyze, for example to check the heuristics. We use GitHub as the source code repository and rely on an API hook to notify Rulzor whenever a developer pushes to the repository’s main branch. Rulzor then pulls the latest revision from GitHub and analyses it. Types are analyzed and classified and if a violation is found an issue is created on the project GitHub repository. If a previously found violation is removed, the issue is automatically closed. If the developer sets the issue status to “won’t fix”, a new issue is not posted for the type, even if the issue persists. This means the developer can decide that it is a false positive, or that the element should not be part of the analysis, e.g., if the element is part of a third party library.

The internals of Rulzor consist of three major parts, cf. Figure 5.2. The static analyzer is responsible for converting the source code to a dependency graph, the classification algorithm uses the heuristics ($\text{DevC}$ and $\text{ArchC}$) to find mismatches, finally the issue management creates, opens, or closes issues based on violations.

The static analysis was implemented using $\text{PHP-Parser-0.9.4}$. PHP is a typeless language, so it can be hard to determine dependencies. Developers were encouraged to use type hinting, but there can be situations where dependencies cannot be found. One problematic situation is when objects are fetched from another container and then used (cf. Listing 5.2).

```php
public function foo(SomeContainer $c) {
    foreach($c as $obj) {
        $obj->bar();
    }
}
```

Listing 5.2: Problematic dependency analysis situation in PHP Code

The $\text{Architect Classification}$ (ArcC) heuristic defines the UIAPI to be any
of the global HTTP data arrays available in PHP (_GET, _POST, _REQUEST, _FILES, and _SERVER), functions related to HTTP (header, get_headers, urldecode, urlencode, rawurldecode, rawurlencode, http_build_query, parse_url, and setcookie), or if it contained a string constant enclosed within < and > (a crude method to find HTML tags). This is an example of a heterogeneous UIAPI that is not well encapsulated. If a source code element uses any of these it was considered to have a direct dependency to the UIAPI and would be classified as a View element. A breadth-first search is used to find a path from the element under analysis to the UIAPI.

The Developer Classification (DevC) heuristic uses naming of elements, files, and namespaces as criteria. Namespaces, files (with paths), or class names that contain the words view, model or control are used to classify source code elements as View, Model, or Controller elements respectively. The namespace is tried first, then the filename (with path), and finally the type name. If no classification can be found, it is posted as an issue to the repository. This allows the developers to change the naming to help classification.

The problem description in the issue is a generic message that suggests there is a problem which includes the classification results, a link to the class that was found to be problematic, and a problem description. The problem description is determined using Table 5.2. Figure 5.3 shows a screen shot of an issue report.

Before the experiment, Rulzor was extensively tested using both fabricated test data (PHP code) and real projects from previous years. For the tests with
real data, we focused our analysis on projects that seemed to stand out in terms of detected mismatches to find false positives. In our analysis of the detailed log files we found three common problems:

1. Unit tests classes are often a source of false positives.

2. Exception classes are often a source of false positives.

3. Some violations propagate and cause false positives (this problem was also noted in Chapter 3).

Based on the common problems, we modified the method to exclude exception and test classes from the analysis. A propagating violation is caused by a violation in another class. We found two instances of propagating violations that we adapted the method to handle. One is when a Model class depends on another Model class that is classified as a View (it depends on the UIAPI). This generates a false positive rule classification as Controller. The other is a Controller class that is classified as a Model when it depends only on View classes that are classified as Models (not dependent on the UIAPI). The general approach is to handle cases when a class depends on classes that in turn are the cause for the problem. We only want to report the root cause of a violation. In such an event it is important to check if the paths to the UIAPI go through a source code element that is already marked as a violation.
Table 5.2: Issue description matrix based on classification to Model ($M$), View ($V$) or Controller ($C$) by $DevC$ ($dc$) and $ArchC$ ($ac$).

<table>
<thead>
<tr>
<th></th>
<th>$dc = M$</th>
<th>$dc = V$</th>
<th>$dc = C$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ac = M$</td>
<td>You probably do not have any direct view responsibility in this class.</td>
<td>You probably do not generate any output using a view in your controller.</td>
<td>You probably do not generate any output using a view in your controller.</td>
</tr>
<tr>
<td>$ac = V$</td>
<td>You probably have view responsibility in your Model class. For example generating HTML or use of some function in PHP that is specific for HTTP.</td>
<td>You probably have View responsibility in your controller class. For example generating HTML or use of some function in PHP that is specific for HTTP.</td>
<td></td>
</tr>
<tr>
<td>$ac = C$</td>
<td>You probably use a class that has direct View responsibility.</td>
<td>You probably do not have any direct view responsibility in this class.</td>
<td>You probably do not have any direct view responsibility in this class.</td>
</tr>
</tbody>
</table>

Figure 5.3: Posted Issue as seen by Developer

5.2.2 Subjects/Students

The project was part of a PHP web development course held autumn 2014 at Linnaeus University\(^1\). The course is a second-year course on the Web developer and Digital services development programmes. There are strict prerequisites for the course, that include courses in Database technology, Introductory programming, and Web technology, plus at least three more Computer science courses. The students have previously used GitHub for source code management.

The project where done individually by the students, lasted 3–4 calendar weeks, and determined approximately 65% (4.5 of 7 ECTS credits) of the final grade (3–5). All students were given lectures and assignments that explained the use of the MVC pattern in the web context before the projects started. It was mandatory to use the MVC pattern and the examination criteria stated

\(^1\)https://coursepress.lnu.se/kurs/webbutveckling-med-php-ht14/ (The course material is unfortunately only available in Swedish)
that serious violations would result in a fail. It was not allowed to use any frameworks (such as Laravel); the PHP code must be the student’s own. Students could however use JavaScript frameworks (e.g., Bootstrap or JQuery).

There were both campus and off-campus students in the course. The project part of the course was presented on the course webpage with detailed criteria in thirteen categories for grading. Ten days before the final deadline, the students were given an opportunity to get a mid-project check by the course staff to see what was lacking in terms of the examination criteria. Students were also offered tutoring during the projects to solve technical problems. The course staff was informed about the experiment and instructed to not help students with experiment specific parts. The course staff also did not know which students participated in the experiment.

Students could start working on their projects at any time during the course. The experiment was introduced roughly at the starting time of implementation and the students already had a small amount of code in their repositories. At the start of implementation the students were asked to participate and Rulzor was presented as a tool to help them in their development, the purpose of the experiment and also that they should not discuss experiment related issues among each other. They were then further informed on how to set up the tool (webhook) and what would happen via email. Students could opt out at any time by simply removing the webhook from their repository. No further tool help, or help with correcting violations was given.

5.2.3 Data Collection and Analysis

Data was collected during the execution of the projects in the log and issue files created by Rulzor. After the projects where finished, we collected all source code commits for each participating student using a GitHub data mining tool\footnote{Doris: https://github.com/gingerswede/doris}. We used the unique commit hash to link the different versions of the source code to the log files created by Rulzor. From this, we could determine the number of types and LoC for each analysis done by Rulzor. We collected the project grades from the student administration system and asked students to fill out a questionnaire at the end of the projects.

We analyzed the source code for each project and sampled LoC, number of types, and number of violations at the end of each day. If there were no events during a day, the data from the previous day is propagated. The data forms a time line with one row for each day and project. We checked for the true start of implementation in each project and removed days at the start that did not contain any activity after an initial small commit. As projects were of different length, we computed a normalized time scale from 0.0 (project start) to 1.0 (project end). Statistical data analysis was done in R.
Table 5.3: Project overview statistics. Groups $T_n$ used the tool and $C_n$ are the control groups.

<table>
<thead>
<tr>
<th>Group</th>
<th>Types</th>
<th>Violations</th>
<th>LoC</th>
<th>Days</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_0$</td>
<td>57</td>
<td>11</td>
<td>4,083</td>
<td>21</td>
<td>5</td>
</tr>
<tr>
<td>$T_1$</td>
<td>47</td>
<td>3</td>
<td>2,785</td>
<td>35</td>
<td>5</td>
</tr>
<tr>
<td>$T_2$</td>
<td>50</td>
<td>5</td>
<td>3,231</td>
<td>21</td>
<td>4</td>
</tr>
<tr>
<td>$T_3$</td>
<td>18</td>
<td>2</td>
<td>1,588</td>
<td>20</td>
<td>3</td>
</tr>
<tr>
<td>$C_0$</td>
<td>50</td>
<td>8</td>
<td>5,254</td>
<td>19</td>
<td>4</td>
</tr>
<tr>
<td>$C_1$</td>
<td>22</td>
<td>4</td>
<td>1,760</td>
<td>21</td>
<td>3</td>
</tr>
<tr>
<td>$C_2$</td>
<td>25</td>
<td>2</td>
<td>3,819</td>
<td>19</td>
<td>4</td>
</tr>
<tr>
<td>$C_3$</td>
<td>71</td>
<td>13</td>
<td>4,278</td>
<td>35</td>
<td>5</td>
</tr>
<tr>
<td>Tool Avg</td>
<td>43</td>
<td>5.25</td>
<td>2,922</td>
<td>24.25</td>
<td>4.25</td>
</tr>
<tr>
<td>Control Avg</td>
<td>42</td>
<td>6.75</td>
<td>3,778</td>
<td>23.50</td>
<td>4.00</td>
</tr>
<tr>
<td>Tool SD</td>
<td>17</td>
<td>4.00</td>
<td>1,040</td>
<td>7.00</td>
<td>0.96</td>
</tr>
<tr>
<td>Control SD</td>
<td>23</td>
<td>5.00</td>
<td>1,472</td>
<td>8.00</td>
<td>0.82</td>
</tr>
</tbody>
</table>

5.3 Results

In total, eleven students volunteered for participation in the experiment. Six students were randomly assigned to the tool-using group and five to the control group. One student from each group did not complete the final survey and one project in the tool-using group was very small (358 LoC). These projects were removed. This left us with four projects in each group. There are 97 days logged in the tool-using group and 94 days in the control group. Table 5.3 presents an overview of the projects and metrics for the final versions. The tool required one minor update during the experiment period. There was a problem handling string arguments used as type name (cf. Listing 5.3). This caused an unhandled exception and termination of the analysis. Only one project was affected for a short period. We do not think that this affected the results of the experiment.

```java
public function foo($objectClass) {
    return = new $objectClass();
}
```

Listing 5.3: String argument as class name

In an effort to test the null hypothesis we analyze the difference in reported violations for each day in each project. A boxplot reveals a difference in violations between the groups (cf. Figure 5.4). Violations in both groups are skewed with a long tail. Histograms also confirm this, and a Q-Q plot reveals (cf. Figure 5.4) that the distribution of violations is positively skewed. We conclude that the distribution is not a normal distribution.

We chose the nonparametric Exact Wilcoxon Mann-Whitney Rank Sum Test to test if the difference in reported violations between the groups is sig-
Figure 5.4: Box- and Q-Q Plots for Violations

lines of code

Figure 5.5: Boxplot for LoC and the number of Types

types
The result is \( p = 0.000225 \) (\( U = 3176.5, Z = -3.6588 \), and \( r = 0.26 \)), which should be counted as a significant difference (\( p < 0.001 \)) and we thus reject \( H_0 \). A randomly selected day in the tool group has fewer mistakes compared to a randomly selected day in the control group in 5941.5 of 97×94 = 9118 cases. The median difference in violations is 2 (= tool=3 and control=5). We calculate the effect size, \( r \), using Equation 5.1 [Ros91] where \( N \) is the total number of samples. A standard interpretation of our result, \( r = 0.26 \), is a small to medium effect size; 0.30 is the threshold for a medium effect size.

\[
r = \left| \frac{Z}{\sqrt{(N)}} \right|
\]  

(5.1)

The experiment was conducted in the field, so there could be other reasons than tool usage to explain the difference in violations, e.g., the complexity of the projects. LoC and number of types are basic measures of complexity, so it is interesting to know if these differ between the groups. Boxplots reveal that both groups are similar with respect to both LoC and number of types (cf. Figure 5.5). Test of these variables also reveal non-significant \( p \)-values (\( p = 0.15 \) and \( p = 0.082 \), respectively).

We also compare final grades for the project part of the course. The course examiner act as an expert opinion and several factors are examined in the projects: system works as stated in requirements, size and complexity of requirements, system is deployed, system is well tested, quality of code, architectural compliance, code comments, installability, usability, designability, security and oral presentation. We conclude that the difference in grades between the two groups is small, 4.25 vs. 4.0. Expert opinion thus judge overall effort, quality and ambition in the two groups as equal.

Students were asked to answer a survey after the project ended. We wanted
to know how confident the students think they were in the MVC architecture before the project (Q1), whether they found it easy to assign types to components in the architecture during development (Q2), and whether they continuously worked with correcting architectural problems during development (Q3). Finally, we asked how confident they felt about their use of the MVC architecture at the examination of the project (Q4). The survey was distributed via email with a link to an online tool used to collect the answers. Statements were put forth and students graded their agreement to these on a scale they were used to from regular course evaluation surveys. Figure 5.6 shows the number of answers for each question and group, (T) for Tool and (C) for Control. Students in both groups had the same level of confidence in the MVC pattern before the project started; Q1. They also found it equally easy to work with the architecture during the projects; Q2. One interesting result is that one project in the tool-using group did not actively correct architectural mistakes during the project; Q3. This is project $T_0$, that also has the highest number of architectural violations in the final versions of the projects (cf. Table 5.3), in the tool group. This is also the student that is slightly less confident in the examination, Q4, of the projects.

5.4 Discussion

Our experiment shows that our heuristic for SACC can be fully automated and that it makes a significant difference in the number of violations during the entirety of a project. We could reject the null hypothesis using quantitative data and statistical methods. In our data analysis, we found no other obvious parameter that separate the groups so we consider them to be equal. It is interesting that we can get a significant difference in violations between the groups, even in quite small single developer projects. We found the practical effect of using the tool to be medium to small in this experiment. However, as projects grow in size, the effect should be greater, especially if more developers are involved in each project.

A better integration in the development work flow, alternatively more training in issue-management is needed for Rulzor to be effective. None of the students used the option to tag issues with “wont-fix” to prevent further conformance checking and one student (cf. $T_0$ in Table 5.3) stated that he did not actively correct the problems that were reported. This is the project with the most detected violations in the tool-using group. If we plot the average number of violations in each group (cf. Figure 5.7) on a normalized time line ($0.0$ is project start, $1.0$ is project end), we can see that the tool-using group drops and stays below the control group after the tool is introduced. If we also remove project $T_0$ the difference is even more obvious (cf. Tool-$T_0$ in Figure 5.7). After initially having more violations than the control group, the tool-using group drops and keeps quite steady. This indicates that the method itself works as intended and that actively using it makes a difference.

We found it fairly easy to fully automate the method and adapt it to the new context of web based systems. Most of the time in development was spent on integrating the method with the GitHub API and to find an acceptable
This period in the study programmes of the students is a stressful time. We think that this shows in the fact that no projects in the experiment managed to entirely avoid architectural violations. When the deadline approaches, the architecture quality seems to be an acceptable sacrifice. This is also something we could see in the form of code comments in some final versions of the systems. For example, in a controller class we find a use of the HTTP protocol with a comment that states that the function should be moved. (cf. Listing 5.4). The global PHP `$_SERVER` array is part of the UIAPI, so this controller class will be classified as a view by the architecture heuristic.

```php
//This function needs to be moved
private function getUserAgent() {
    return isset($_SERVER['HTTP_USER_AGENT']) ? \
        $_SERVER['HTTP_USER_AGENT'] : "";
}
```

Listing 5.4: Architectural Comment

5.5 Summary

We found that the method can be fully automated as a software service. Care needs to be taken to avoid false positives for unit testing and exception source code elements as well as to avoid the problem with cascading violations. Based on statistical analysis of data there was a significant difference in violations between projects that used the tool and projects that did not ($p < 0.001$ and $r = 0.26$), the median difference was 2. This is a small to medium effect size.
Chapter 6

Related Work

6.1 Erosion in Architectural Patterns

Architectural erosion during software evolution is a widespread phenomenon that is described in multiple sources and case studies [Kno11; KLMN06; BK07; RLGBAB08; ARB12; PB14; RLGB+11]. The Layers pattern (or style) has been used in many software systems to provide software qualities such as maintainability, portability and reusability. Clements and Nord [CN00] note that the practical use of the Layers pattern often violates its intention. Architects want their system to be layered even if it is not, and developers misunderstand the design and introduce erosion.

Sarkar, Rama, and Shubha [SRS06] propose a method to find architectural erosion in the Layers pattern and evaluate it using two large open source systems and one proprietary system. Their approach work on module level and is based on static analysis. They find problems related to back calling (a lower layer calling a higher layer), skip calling (a layer is skipped, not enforcing strict layering), and cyclic dependencies. They find erosion in all categories in all systems except one.

In [TV09] the MVC pattern is used in the case study system and erosion is investigated in three versions of the system. They show a dramatic increase from 7 detected erosion to 194 while the system evolves from 18,000 LOC to 240,000.

6.2 Static Architecture Conformance Checking

Maffort et al. present ArchLint [MVA+13; MVT+15], an approach to detect architectural violations based on static analysis of the project source code history and a high-level component specification. The idea is that dependencies that cause architectural erosion are more rare and more frequently removed over the history of source code revisions. If these can be detected by using the project history, there is no need to specify explicit models or rules for detecting them. Four heuristics are presented that can detect divergent and absent dependencies. They report precision values ranging from 0.48 to 0.89 and recall values
from 0.16 to 0.96. ArchLint seems to be prone to producing false positives, which is a drawback. ArchLint also needs the project version history, making it hard to check for conformance in the early stages of a project. However, the approach is novel and could be especially valuable in a maintenance scenario, where there might be no other reliable documentation than the source code. The heuristics could then produce a first draft of violations that then can be formalized with, e.g., Dependency Rules.

Deissenboeck et al. present SACC in the ConQAT quality analysis framework [DHHJ10]. It uses a hierarchical component description of the intended architecture with specification (allow, deny, tolerate) of dependencies between the components. Project artifacts (e.g., source code elements) are then mapped to this specification and with their dependencies extracted (by the ConQAT framework) they form the implemented architecture. The rules of the intended architecture are then evaluated from the lowest level first. This approach is similar to the Reflexion Model approach. One interesting aspect of this work is the ability of ConQAT to include other artifacts than source code in the analysis.

Code smells or code anomalies are places in the code that developers themselves think are badly designed. There are several well recognized code smells and to remedy these there are standard ways to restructure the code to remove the smell without changing the functionality of the system. This procedure is called a refactoring. There are tools available to detect code smells and even provide automatic or semiautomatic refactoring of these. Automatic detection often relies on code metrics such as LoC, Complexity, and Coupling metrics. Detection systems are then tuned and different thresholds are set for combinations of metrics. Source code elements are measured; metrics collected, and smells detected. Smells tend to be related to low level design issues, but these could also be related to architectural issues. Machia et al. [MGP+12; MAG+12] have made two studies on the subject. They find that automatically detected code smells were not related to architectural violations to a high degree (60% were not related) and that refactorings do not tend to have an architectural impact. In a longitudinal case study, Ferreira et al. find that inspections are as effective as metrics based approaches to find architecturally relevant code anomalies. They also find that inspections require less time than automatic detection of code anomalies. While current methods for detecting code smells can be beneficial for maintaining a good low level design, they do not seem to be a promising alternative for finding architectural violations.

6.3 Architectural Languages and Architecture Constraint Languages

An Architectural Language (AL) or Architectural Description Language (ADL) is used to describe an architecture [MT00]. An AL could be as simple as box-and-line notations to formal languages [LMM+15]. If the description used is well defined, this facilitates the use of tools. For example, a tool could be used to visualize and allow for manipulation of different views of an architecture and
make sure that they do not contradict each other. A well-defined AL could also enable generation of programming code and thus ensure conformance [MT00; PTV+10]. An AL could include a Constraint Language (ACL) to allow for consistency checking [LMM+15]. De Silva and Perera [DSP15] propose the use of the GRASP ADL to formally specify the architecture and use this specification in a SACC tool to allow for automatic and continuous SACC.

Architectural decisions often need to be present in multiple models of a system, from architectural models at a high level, then possibly through several levels of refined and more specific design models, and finally in the source code model. An ACL is meant to formalize architectural decision on a high level and enable automatic tools to find violations in subsequently created or updated models, including the source code model [TFS10]. For implementation conformance, round trip engineering of source code to a design model is assumed, or special programming languages or extensions are used, e.g., annotations in Java [TFS10; Tib14]. Dependency rules are also considered a state-of-the-art method to statically check constraints on an implementation [Tib14]. ACL also consider dynamic conformance checking techniques, for example in the case of a self-adaptive system [Tib14].

### 6.4 Empirical Research

As SACC has been an active research area and there are currently tools to support such activities in development, it becomes interesting to know how SACC affects development of software in practice. As a systematic literature review is outside the scope of this thesis, this section is meant as an overview of empirical evidence regarding the need and effect of SACC.

There is a lack of compliance in professionally developed software. Knodel et al. [Kno11; KLMN06] report on 14 case studies of varying sizes and domains from both academia and industry. Based on interviews and compliance data, the conclusion is that compliance is a recurring problem in all cases and no case had full compliance to the intended architecture. Bourquin et al. [BK07] study a medium sized Java enterprise application and find 7,000 violations in 850 classes at the start of their case study. Rosik, Le Gear, Buckley, and Ali Babar [RLGBAB08] and Ali, Rosik, and Buckley [ARB12] report on inconsistencies that developers were not aware of in commercial software systems in industry case studies. During the development of a prototype for HUSSACT (an ACC tool), six different professional systems from four organizations were used and five of the six cases contained violations [PB14]. Finally, Greenfield development is investigated in a longitudinal case study, which finds that the implementation diverges from the intended architecture [RLGB+11] but also that the Reflexion Model based method used concealed some actual violations.

Will evolution in an eroded architecture require more effort than evolution in a system that displays architectural compliance? While it may seem intuitive that this is the case, this does not seem to be the case for code smells [SYA+13]. Knodel et al. [Kno11; KLMN06] indicate that the impact of architectural erosion does indeed cause an overhead in effort when evolving professionally developed software systems. In [Kno11], an experiment is performed
and replicated three times to evaluate the effect of architectural compliance when performing an evolution task. The experiment is performed on the same system in two versions, one with architectural violations and one without. Randomized groups of students and professional developers then perform the same evolution task, and outcome and effort were measured. On average, lack of compliance caused an effort twice as high, and possibly worse, only one person of 16 managed to solve the task with 100% correctness. In the group that used the compliant version, all but one managed to achieve 100% correctness. However, the evolution task selected can be an important factor to consider. Bourquin and Keller [BK07] report improvement in source code metrics and that developers found it easy to add subsequent new features after refactorings guided by architectural violations.

Will the use of SACC methods improve the evolvability of the software? This is a harder question to answer, as evolvability needs to be well defined and the fact that knowledge of erosion does not inherently cause removal of the erosion. Rosik, Le Gear, Buckley, et al. [RLGB+11] report that detected erosion did not prompt removal. Also, Bourquin and Keller [BK07] report that not all erosion was removed, as some problems “seem to be too complicated to be tackled at the moment.” A contrary example is a rise in compliance from 54% to 95% in a small company, reported by Knodel [Kno11], when they established an architecture-centric approach to development that included regular SACC. In a product line that used SACC, most of fifteen products increased the compliance during the time of the trial even though the compliance was already quite high [Kno11]. Knodel, Muthig, and Rost [KMR08] performed an experiment using students in a project setting. Students were assigned different components to develop and some components were assigned to use SACC in their development. An interesting aspect of this experiment is the use of a live feedback approach to SACC. The developers got prompt feedback on architectural violations directly in their integrated development environment (Eclipse). During the project, the group that used SACC had almost 60% less violations compared to the control group.

These varying results from using SACC indicate that there needs to be a focus on architectural compliance in the organization for developers to be able to perform refactorings to comply with the architecture. Another indication is that if the erosion has gone too far and violations are too complicated they will not be refactored. A highly integrated conformance checking process used from the onset of a project could alleviate both these problems by raising focus and awareness of architectural conformance and not let erosions become too bad.

To conclude, there is ample empirical evidence that architectural erosion exists in software systems in general. There are also indications that this erosion causes problems when evolving a system, both in terms of increased effort and lower correctness. There is mixed evidence from the incorporation of SACC in development projects and there is a lack of industry adoption of SACC tools. Knowing about violations does not always prompt removal of erosion if architecture is not a focus in the development process or if the erosion is not detected early.
Chapter 7

Discussion

7.1 Erosion

Our study of the implementation of MVC over time shows that earlier implementa-
tions are more monolithic than the later ones. The use of a pattern has not been sufficient to manage an increase in complexity in these earlier implementa-
tions. This is in line with previous research [CN00; SRS06; TV09]. We can clearly see that the developers have used the principle of separating Model and user interface in all the projects. One, rather large, interface is responsible for sending events from the model to listeners, which is the standard Observer pattern solution. This has been a stable design in all three variants.

The first iteration resembles the document view architecture, where there is no separation of view and controller but rather a monolithic user interface. The result is a collection of large and highly coupled classes in the user interface. Developers seem to have wanted to address this in the TWTPB project, where we find a division of View and Controller elements. However, there seems to have been a struggle in how to realize this, as responsibilities are mixed and the idea of using the View as a layer of indirection between the rendering engine and the Controllers has not been fully accomplished. In the Hero and Gears of Love projects, this idea is more visible, though there are a few exceptions to this principle, as can be expected. This last design also introduces more fine grained View elements that serve a sub set of Controllers. This should help in clustering features and could, for example, be important if parts of the game, such as an editor, should not be ported to a new platform.

From the analysis we find that the use of the MVC pattern has changed between the games as developers learned more and more of how to use the pattern effectively in game development. We see this as a symptom of ero-
sion (probably due to inexperience). Developers agreed on the MVC pattern but when concrete decisions were made, they had a hard time dividing the responsibilities and managing the key dependency to the rendering engine. The concrete problems are mainly due to elements that have mixed or multiple responsibilities. Separating the responsibilities between View and Controller seem to have been problematic as well as controlling the dependencies to the
rendering engine. The final iteration shows that the View should be considered a strict layer that protects the controller from directly accessing the rendering API.

7.2 Design and Implementation of our Method

The use of an architectural pattern did not control erosion very well. This is not a surprising finding, but it enabled us to study what mistakes were done in early implementations of the pattern compared to later implementations. This provided valuable knowledge when we constructed our method. We then showed that a Dependency Rules-based approach would be suitable as the underlying technique because of its high level of expressiveness and that rule-based approaches are code driven which is suitable for a service approach. Reflexion Models are powerful and often used, but visual modeling is a key feature and more suitable when exploring an architecture.

We used the MVC pattern as the pattern to study since this was the pattern used in the projects from our initial analysis and in-depth knowledge about. In essence, our method compares two classifications of each source code element: one classification from the developer’s point of view and one from the architect’s point of view. This idea emerged when we inspected the source code. In eroded source code elements, we often got a first impression when looking at a source code element (i.e., based on naming) but when looking deeper at the actual responsibility of the element, a different picture emerged. There seemed to be a mismatch between the developer’s original intent and the final responsibility of these source code elements. From this analysis, we also found what we regard as a ground truth in the MVC pattern: View responsibility is determined by a coupling to some UIAPI that is external to the application. Based on this, we created a method that can automatically perform the classification from the architect’s point of view. While not perfect and detailed, it did suit our goals of simplicity and abstraction. Our general idea is to base the architectural classification on as few key dependencies as possible. This means that the method will not be as precise as if a full Dependency Rules-based method was used. In this sense the method is a heuristic approach.

We compared the method to manual inspections of source code from real projects, which provided us with valuable information on what the method would and would not find (cf. Tables 4.4 on page 42, 4.5 on page 43, and 4.6 on page 43). Based on our findings, we performed a refactoring of two projects to see if we could get to a state of full conformance. This provided us with valuable information on problematic situations where the method would report a problem that did not exist. Such situations are undesirable, as developers risk spending time trying to find and understand problems that do not exist. We found that we need detailed dependency information to be able to define the ground truth, UIAPI, as accurate as possible.

The definition of the UIAPI is interesting. We analyzed projects where the UIAPI itself was under development and not fully implemented. This meant that we had to be flexible in our definition of the UIAPI itself and also that each project may implement parts that are reusable and could or should be moved to
the UIAPI. The goal was to be able to offer both flexibility in definition and be able to find these reuse candidates. In our first evaluation, we used an off-the-shelf tool with a generic query language that did offer some flexibility, but this came with the price of added complexity in the rule implementation. In our second implementation, we used a language parser to construct the dependency graph. This provided a much more flexible definition of the UIAPI. We could also add a simple form of logical dependency, in the form of detecting strings that contain HTML. An idea is to expand on this and check for more logical dependencies such as files, database connections, SQL, painted types, etc. Painted types that mask dependencies are responsible for many false negatives in our evaluation (cf. Table 4.6 on page 43).

We made a fully automatic version of our method and packaged it as a service. This service was integrated into a development process using the source code repository to get the source code and to post found violations to the project issue management system. We wanted to know if our method was usable, if it would generalize to another domain, and how its use would affect developers. Current studies have mixed results when using SACC methods; some report a decline in violations while others report an unchanged rate of violations. We found a statistically significant lower amount of violations in projects that used our method. This could be because the method was used from the start of the implementation phase, and that it was used continuously. Erosions that are allowed to stay in the source code tend to become larger and at some point a refactoring becomes too complicated. There is also a need to value architectural conformance on a higher level in projects. We found that violations are ignored if the developer do not perceive conformance to be important enough. Projects tended to accumulate some violations as they got nearer the final release. This could have implications in agile projects, where pressure to continuously deliver a working product could drive erosion.

Knodel et al. [Kno11; KMR08] propose turning the analytical process of conformance checking into a quasi-constructive process. The idea is that if developers get more immediate feedback on architectural violations, they can correct these problems with less effort. Effort is saved because developers do not lose the context of their task compared to feedback days, weeks, or months later. There could also be a learning effect, where developers learns and understands how to address their tasks without violating the architecture. Their experimental result suggests that projects that get live architectural feedback had fewer (60%) architectural violations compared to a control group. Their tool (SAVE LiFe) used a special version of the Eclipse IDE to enable live feedback of architectural violations to developers. In our experiment, we compare a less frequent (controlled by the developer) feedback mechanism, and in our case the control group had one opportunity to get feedback on architectural issues in a mid-project release. We think that we complement the experiment by Knodel et al. and strengthen the importance of architectural conformance checking throughout the development process [Kno11; KMR08].
7.3 Evaluation

The results show that a high-level architectural method can be constructed based on the MVC pattern. The method proved reusable across several games, and in a web-development setting without modification and revealed points of architectural erosion. This suggests that patterns can be used as the basis for a method to check conformance and that this method is reusable across domains without modification. It was easy to perform the analysis required for our method using off-the-shelf tools and the implementation as a service was straightforward. We found architectural violations that traced to erosion in the implementation. The precision and recall of our method are acceptable but might improve if the method was customized to each project and used throughout the development cycle.

As far as we know there are only a few studies that report precision and recall for SACC methods in relation to manual inspections of erosion. Maffort et al. [MVA+13; MVT+15] report precision values ranging from 0.48 to 0.89 and recall values from 0.16 to 0.96 with their heuristic approach and state that it is prone to produce false positives. Reflexion Models are a popular method to use for SACC but [RLGB+11] reports that some erosion was concealed compared to their observation and think-aloud protocols. In this case, 41 of 44 elements marked as convergent did actually contain inconsistencies. This would indicate a low recall value for such methods. The study is only one case and no general conclusions can be drawn from it. However [RLGB+11] state that “it is our intuition and contention that consistent edges would often hide inconsistent source code relationships.” However, the majority of studies we have seen [AP11; DHHJ10; HCSS08; BKL04; MNS95; SJSJ05; FKBA07] do not report precision and recall values and often only include the number of detected violations. In this regard it becomes hard to relate our precision and recall values to other methods. We hope that our result will give an indication of the method’s effectiveness and we urge others to report these values as well.

We find three large refactorings in the TWTPB project and overall more refactorings that are needed compared to the Gears of Love project (cf. Table 4.4 on page 42). The probable cause for this is the size difference of the two projects and the fact that Gears of Love is both younger and a later implementation of the architecture. Developers knew more about how to implement the pattern. This is also indicated by the lower recall value in the Gears of Love project (0.72 vs. 0.63). There are more architectural erosion not detected in Gears of Love than in TWTPB. Looking at the results of the individual games we can see that the recall values of the newest projects are lower (cf. Table 4.3 on page 41). To us, this indicates that the value of the method declines as the developers get more familiar in how to implement the intended architecture, other erosion become more prevalent. In our first study these projects were also the latest evolution of the pattern implementation and displayed a much more thorough separation of View and Controller (cf. Figure 2.3 on page 14). This indicates that the value of our method, as is, becomes lower when the development team is more experienced. Possibly heuristics could be tweaked continuously during software evolution to provide continuous benefit
from SACC. Our data (cf. Table 4.4 on page 42) also indicates that erosion start in the small with a variable, then moves to function level, type level and lastly a type needs to be split to adhere to the architecture. This is in line with other findings on architectural erosion [MGP+12; TVCB12; MGP+12]. Using SACC as early as possible will detect these small erosion and they can thus be hindered from evolving into erosion that require a larger amount of effort to remedy.

Metric differences between the pre and post versions of TWTPB indicate that the compliant architecture manages complexity better by distributing couplings more evenly between classes and that there is an increase in the lines of code related to the Controller and a decrease in the lines of code in the View. However, the overall effect is small. This could be caused by the number of violations and thus the number of refactorings was not large enough to yield a substantial effect. It could also be that refactorings in general have a varying effect on metrics as indicated by previous studies [Als09; BK07]. However, we think that a more even distribution of complexity is desirable and something that is an overall aim of software architecture.

Regarding false negatives (cf. Table 4.6 on page 43) these would be hard to detect by any current SACC method. In our specific case the architect needs to take great care when defining the UIAPI, and possibly tweak this definition. In this case the UIAPI was also under development as is quite common in game development. If the UIAPI had included a type for managing color information many false negatives would have been avoided. If the UIAPI is not mature and the definition of the UIAPI method should be adjusted regularly the information in the dependency graph generated is important. More details (i.e., primitive data types and variable names) could enable a more precise definition of the UIAPI (i.e., include a “DWORD color” as a dependency to the UIAPI). An alternative could be to make a textual analysis of source code and merge this information with the dependency graph.

Painted types and UI specific models often relates to hidden or implicit dependencies. Instead of explicitly using a type this is masked in the form of primitive data types. We found examples in the form of texture coordinates (floating point values) and integer-IDs that are used to find some resource for visualization. While this is often convenient during development, these constructs tend to make Model classes more complex than needed. These mixed elements could also have lower performance as they are mixing game rule responsibility with user interface representation responsibility. An example of this is in Gears of Love where the world representation is subdivided based on a more detailed graphical representation than needed for collision detection (a Model responsibility). This extra level of detail is not used for collision detection and subdivisions should not be needed. This consumes both resources and performance during runtime, as well as adds complexity to the source code. There seems to be a trade off between reuse (of a common more complex class), maintainability (one complex class vs. two slightly less complex classes that need to be kept synchronized) and performance (will one representation tax performance more than two optimized representations). While one might take a more orthodox stance and say that it should be divided, in practice things
are more complicated and trade offs are often needed. However, this should be an informed decision and not just something that happens.

### 7.4 Problematic Situations

As the example with View responsibility in the Model showed there is a risk that one violation generates secondary violations (cf. Figure 3.7 on page 33). This could minimize the practical value of the method as developers could spend a lot of time searching for problems that do not exist. In practice we solved this in our field experiment, but a more generic approach should be defined.

It is quite easy to create a source code element that would generate a violation without it actually being one. For example one could just add an element in the View that did not depend on the UIAPI. We would expect such source code elements to be for example interfaces or elements used for data passing in a real code base.

If the architecture does not provide an unambiguous way to classify a source code element based on dependencies, there is of course no way to create the ArchC function. We used the UIAPI to provide unambiguity, but there may be architectures that do not provide this especially a more detailed architecture may be problematic. This is a problematic situation for any SACC that rely on dependencies. In Reflexion Models, an error in the source code mapping would not be discovered (as there are no violating dependencies) and in a Dependency Rule, an error in source code element naming would not be discovered. It may be more of a problem in our approach as for example Dependency Rules can include more details (at the price of complexity). A problem that relates to this is to find a suitable heuristic that can perform the mapping. While we have relied on our experience to create this from the MVC pattern, there can be cases where it is not straight forward. Possibly some machine learning approaches could be used in complicated cases.

### 7.5 Developer Classification

How to find the developer classification has not been discussed. While this can be a complicated issue if performed by manual mapping we think that looking at naming and containment of a source code element is a sufficient approach. This is essentially what is done in both Reflexion Models and Dependency Rules to provide automatic SACC. An interesting note to make is that we think that this mapping is one of the first visible things a developer does in the source code when performing a task. She decides what name a new element is to have, what file name, directory and package it exists in. After this is done, dependencies are added to fulfill the responsibilities of the new element. Sometimes the developer makes a mistake in this first classification and responsibilities (and dependencies needed) are added that are not in line with this first classification. Sometimes responsibilities (and dependencies needed) are added to existing
elements that are in violation of this initial classification out of convenience or misunderstandings. The initial intended architecture thus erodes.

7.6 Summary

We have created a method for SACC based on the MVC pattern that finds common problems in implementation of MVC-based applications. By showing examples, performing an evaluation and executing an experiment we have demonstrated that we can fulfill our requirements (cf. List 3.2 on page 23). For the method to be flexible it is important that the definition of the UIAPI is easy and straightforward to change. To accommodate architectural evolution and possibly additional patterns the definition of the ArchC classification also needs to be changeable.
Chapter 8

Threats to Validity

In this chapter we discuss validity threats to the studies. We cover construct, internal and external validity factors. For construct and internal validity, we study each research question separately. The external validity is discussed in the context of our goal of providing a method for pattern-based SACC, with a focus on overarching problems and problems in the field experiment.

8.1 Construct Validity

Construct validity is the extent to which the study design supports the underlying research questions.

8.1.1 Research Question 1

To answer RQ 1, we compared the results of the method to the results of manual software inspections in four projects. While this is a labor intensive approach, and probably not feasible in a larger code base, it provides detailed knowledge and a solid base of data to evaluate the effectiveness of the method in this specific context. Unfortunately, it is not common in research to report values of precision and recall for SACC methods or tools, which makes it hard to compare our results with others.

8.1.2 Research Question 2

To answer RQ 2, we studied the discovered violations and erosion. Based on these, we modified both the source code and method specifics to find full architectural compliance. We compared size and coupling metrics before and after refactoring. We opted to study LoC and Coupling as they are a basic measure of complexity. Our SACC method is based on dependency graphs, so refactoring of violations should affect Coupling. The study design provided us with realistic data from one combination of refactorings and project. This makes it sensitive to the actual refactorings performed which are highly subjective, and given more or less resources, the metrics would probably be different. This
part of the study can be regarded as exploratory and should probably not be considered general evidence.

8.1.3 Research Question 3

To answer RQ 3, we implemented a tool and performed a field experiment. The successful execution of the field experiment shows that the method can be automated. We wanted to know the impact of using the tool and whether it can be an effective way to combat architectural erosion. We refine this to be able to say that the average number of violations in a project that uses a tool should be less than in a project that does not use such a tool. The projects were in themselves small and only one developer participated in each project. Also, developers were not responsible for further evolution or maintenance of systems beyond the experiment context. This setting makes the need for architectural conformance lower. Comparing tool use with no tool use may be unfair in itself. For students in the tool-using group, just knowing that a tool is monitoring the code may provide extra incentive to care more for architectural violations and adapt their behavior. We tried to combat this by not giving too exact violation descriptions. The students had to inspect the entire source code element to find the violation and actually understand the architecture, not just mechanically remove violations. This means that there is likely extra effort needed that is not just a change of behavior. Survey question three (cf. Q3 in Figure 5.6 on page 57) compared the attitudes of handling architectural violations during the projects. One project in the tool-using group did not agree with this statement. This indicates that the use of a tool did not prompt more investment in handling architectural violations.

The number of participants in the experiment is low so there is a risk that the random assignment of students can skew the result both in terms of having a complex or ambitious project (possibly resulting in more violations) and different levels of competence. We mitigated this risk by examining the final grades of the projects and the first two survey questions. We found no major difference in these variables. We think that the students that volunteered to participate in the experiment were among the more competent and ambitious students, as reflected by their high average grade. A blocked design could be an option, but this can also produce problems in researcher bias and finding good criteria for blocking.

The use of the issue mechanism for feedback could affect students not familiar with this part of GitHub. Issues could simply be ignored which was also the case in one of the tool-using groups. We could also see a decrease in handling architectural violations towards the end of the projects in all projects and also some traces of marking violations in the source code instead of fixing them (c.f. Listing 5.4 on page 59). This indicates that measurements towards the end of the projects are not always representative. Knodel, Muthig, and Rost [KMR08] also noticed the same behavior. However, as we focus our analysis on the entirety of the projects, we are less susceptible to violations due to crunch time stress towards the end of the projects.
8.2 Internal Validity

The internal validity is the analysis of factors outside the ones studied that could influence the results.

8.2.1 Research Question 1

We used manual inspections as the base line of comparison. Manual inspection of source code is hard, time consuming, and prone to errors; there is always the risk that something is overlooked. In this particular case, the number of false negatives (some erosion is overlooked by inspectors) might be too low, which would inflate the recall. We put extra effort in investigating source code elements that did not seem to contain any architectural problems.

The method is only as good as the tools are at accurately detecting dependencies in the source code. Pruijt, Koppe, and Brinkkemper [PKB13] compare several tools to check dependencies in Java source code, and 74% of the tested dependencies were detected on average. A low accuracy in this regard will affect the result in our study by increasing the number of false positives and false negatives, which affect both precision and recall. One of the false negatives we found was indeed caused by undetected dependencies. We also found implicit dependencies in the code, which are difficult to detect with static dependency analysis.

8.2.2 Research Question 2

To answer part of RQ 2, we performed refactorings. The refactorings we did were in some cases hard and took a lot of effort, especially the three cases where Controller and View responsibility were heavily mixed. Finding a division and entangling the dependencies was complicated and required creative problem solving. This means that there is a high chance that other developers would do things differently, depending on their experiences, time constraints, and other factors. We tried to find acceptable refactorings with the time constraints we had, but things could always been done better. For example, the ViewCore got more responsibility and coupling, and a better approach would have been to create specialized View elements. However, the refactorings we did are realistic, given the information we had in the form of architectural violations. In [BK07], a process for managing refactorings suggests to use metrics to evaluate the impact of architecturally guided refactorings. Following this process we would have used our current metrics as input to the next round of refactorings and improvements. It would be interesting to see the effect of this next step and compare to both the original and current versions.

8.2.3 Research Question 3

The internal validity for RQ 3 is the extent to which other factors, besides use of the tool, could influence the number of violations in the projects.

In these projects the architecture was decided based on application domain (web application), not the problem domain of the individual projects. MVC
might not be the best choice for every project and thus introduce higher than needed complexity. However, students were extensively trained in using the MVC pattern and to find individual architectures for each project was not an option due to time constraints. Students were aware of the requirement to use the MVC pattern, so they could select projects that this architecture is suitable for.

We acted as the architects and supplied the heuristics for both the developer and architectural classifications of the system as well as the definition of the UIAPI. There was no tweaking or project specific adaptations made during the experiment. This could affect the results negatively for the projects that used the tool. To combat this, we tested the tool using projects from previous years of the same course and found the heuristics to give good results with some modifications. In the ideal case, the architect continuously monitors the logs and adapts the heuristics on a per project basis.

As this is a field experiment, we had no control over the tools or methods the students used besides the experiment tool and environment. However, we believe that the students selected the tools and methods that were best for them and their project. This should produce more realistic results compared to being forced into a set development environment of which students could have different levels of experience.

8.3 External Validity

External validity factors are issues that influence how well the results can be generalized, applied to a similar but different contexts.

We used games in the study on effectiveness that were developed by us, at least the majority of the code. Students that participated in the experiment were all students we had met during our lectures. There is a chance that this influenced the results. Our method is possibly too specific to our view (and implementation) of the MVC pattern. There is also an issue with using different patterns, as this work focus on the MVC pattern. We argue that it should be applicable to other patterns as well, but this is nothing we have actually tested. These two issues affect the generalizability of our method. There is a need to study more implementations of the MVC pattern and also to broaden the method to different patterns.

The students produced the code for the systems entirely from scratch and no frameworks were used. This is uncommon in the web domain where numerous frameworks exist. We think that if a framework is to be used, the UIAPI needs to include a set of entities from the framework.

In the study on effectiveness, we have seen that developers learn how to implement the architecture correctly as they gain experience over a number of projects (cf. Table 4.3 on page 41). The value of conformance checking becomes lower as more false positives are reported. The field experiment had students and they are not as experienced. We would not recommend these exact heuristics to be used in a project with experienced developers without testing. Architects need to play an active and continuous role in finding good conformance checking heuristics on a per project basis.
The size of the projects studied is relatively small both in terms of team size and number of lines of code. This is problematic as projects with a small scope may face different problems than larger size projects. For example, the general value of automated SACC becomes greater in a large project with many developers and LoC. In a small project with limited scope, a few or a single developer can keep most of the design and domain in mind. The need for SACC in this case is generally more limited. This could be a reason for the low to medium effect size in our experiment. There can also be entirely different problems that exist in a large project [ER03] and there might be tool or method performance issues, which would make the method impractical to use on a continuous basis, possibly loosing much of its value. There might also be special cases that can not be covered by our method as it is an abstraction. In a large project, there could be too many special cases and which might make prevent the method from being applicable to large projects.
Chapter 9

Conclusions

The overall goal of this research has been to provide developers with a method of SACC that is based on an architectural pattern and is easy to use.

First, we established that using an architectural pattern was not sufficient to control architectural erosion. This is not a surprising result since it is easy to violate the constraints of a pattern. We found that as the development team got more experienced, the structure of the pattern became more clearly expressed in the source code. This could indicate that architectural erosion is not a big problem in small, highly experienced teams that use the same architectural pattern in several products. In reality, it is unrealistic to expect that all developers are highly experienced throughout the lifetime of a system. We consider the motivation to use pattern-based SACC to be valid, but also that needs might vary, e.g., based on the experience of the team.

Second, we established that the definition of the UIAPI was important to define the MVC pattern. In our case, we found that the UIAPI was unstable since it was also under development. We designed our method as a heuristic based on Dependency Rules. RMs offer a high level of abstraction as they are built on visual modeling. This can be powerful, but we found three major problems: 1. Reflexion modeling is less expressive and this prohibits a flexible definition of the UIAPI; 2. As RMs compare dependencies at the architectural level, there is likely that individual source code elements do not fulfill mandatory dependencies; and 3. Visual modeling would be more complicated to implement in a Software as a Service based tool. If the UIAPI used is well defined and stable, and false negatives are not considered a large problem, RMs could offer a valid approach. Building a reusable Reflexion Model and using it for several projects could be an alternative. However, looking at a broad set of systems in different domains using different platforms, the UIAPI will be a varying factor. Our idea is to use this as a parameter to our MVC heuristic to enable the method to be used in different domains.

Third, we evaluated the effectiveness of our proposed method compared to manual inspections in a number of computer game projects. This is especially important as our method is a heuristic approach. We found that our method has a precision and recall of 0.81 and 0.75, respectively, by comparing it to
manual inspections (RQ 1). These are acceptable numbers, but it is hard to compare it to other methods as there is a lack of reports of such measures in the research.

We analyzed the existing erosion and reported violations in detail. We found that the method will find source code elements that are severely eroded, but also such that are less severely eroded. The method also found source code elements that are candidates for reuse. False positives were typically caused by small source code elements that are mainly used to pass data and most of these could be corrected by tweaking the rule (RQ 2.3). False negatives were mostly caused by either problems generating the dependency graph or the definition of the UIAPI parameter. These could be corrected by increasing the scope of the UIAPI definition (RQ 2.3). However, there is also a class of problems that are caused by logical dependencies that are not easily detected by static analysis in general (RQ 2.4). We found that if realistic refactorings of detected violations are performed, complexity in the form of dependencies of classes and lines of code tend to be more evenly distributed in the system (RQ 2.2). The evidence for this conclusion is limited due to the limited scope of the refactored system.

It is not surprising that the method gave acceptable results for effectiveness as it was developed based on experiences from analyzing some of the games we studied and the same team was involved in all projects. This could mean that the method is specific to this particular line of games, and would not work on other instances of the MVC pattern. To address this we aimed to test the method in another technical context with other developers.

Fourthly, we automated the method and implemented it as a Software as a Service based tool. This tool was then used in a field experiment to test if developers would benefit from it. This allowed us to evaluate our method in another domain (web applications) with other developers. We found a significant difference in reported violations between the group of developers that used the tool versus the control group (without tool). We also had the chance to verify that the flexible definition of the UIAPI would allow the method to be used in a different technical platform. Results are promising; but there are some overarching validity threats that need to be addressed by future studies.

We found that a tool could be implemented to fully automate the method. We evaluated the tool by integrating it into the development work flow as a software service and performed a field experiment with participating students as developers. The results show that a project using the tool will significantly decrease the number of violations during development compared to a project that did not use the tool (RQ 3).

We have contributed to the knowledge of SACC methods by comparing our method with inspections and provided measures for precision and recall. This is an important piece of information that developers need to know when adopting a SACC method in a development project. Currently, few studies report these values, but there are indications that current methods can hide violations. This could cause distrust of the practical value of SACC methods in general.

We have contributed to the knowledge of how the incorporation of a SACC method affects actual removal of architectural erosion. Previous empirical stud-
ies in this area report both that erosions are removed and that violations are ignored. The indication is that architectural erosion must be detected early in the project and that frequent feedback is needed for erosion to be removed. We support these indications. However, there is still a need for an overall organizational focus on architecture in projects for developers to spend time on refactoring, especially when deadlines are closing in. This could be a major problem in agile projects that mandate frequent release of products, and SACC will be all the more important. There is a need for more research in the area of how to effectively integrate SACC in the development of software.

We have contributed to the knowledge of how refactoring based on detected architectural erosion affects code metrics. We found that dependencies between source code elements became more evenly spread, and that lines of code between the architectural elements View and Controller became more evenly distributed after refactoring.

The research addressed in this research started out in a pragmatic way. We took a practitioners point of view and devised a practical solution to a problem sprung from our experiences and empirical data. This is an inductive approach to research and the methods used were more qualitative in nature with e.g., manual analysis and synthesis of source code. This allowed us to deepen our knowledge. The latter part of the research, mostly concerning the evaluation, gave us the chance to use a more analytical, and more quantitative, approach. This proved valuable to find problematic situations in our method and ways to improve it. The use of the empirical method in software engineering has provided a framework that we found suitable to perform this kind of mixed method research [Cre13; WRH+12].
Chapter 10

Future Work

In this research we show that our method works for MVC on different systems (i.e., games and web applications). A goal is to increase industry adoption of SACC, but in order to transition to industry use and larger systems we face a number of challenges.

While the method could serve as a starting point for MVC-based applications, there is a need to validate it for layered architectures. Layered architectures are common and could make our approach more generalizable. This would open up opportunities to make case studies on larger code bases in a more realistic setting. We would also like to investigate the possibility to define architectures in a hierarchy, for example a layered structure inside an element in the MVC.

From a practical tool point-of-view, the architect stakeholder needs more attention. There is currently no user-friendly way to adapt the heuristics or inspect logs in the tool. We are considering possibilities ranging from code-driven to data-driven to visual or even automated approaches to adapt and create new heuristics. This is a challenge, since finding an unambiguous mapping heuristic using dependencies can be a non-trivial task. So far, we have relied on our own experience with the MVC pattern. A strict layered structure would be simple to implement checks for, since it would only require checking the distance to the bottom layer. However, other types of architectures could be more challenging and possibly require us to check several dependencies hierarchically. This poses questions to software architects: if it is not possible (or easy) to find a mapping, what is the quality of the architecture itself? Should ease of conformance checking also be an architectural goal?

One major challenge of SACC is automatic construction of a unified dependency graph in a heterogeneous language environment. Ultimately, we would like to be able to check for all logical dependencies between artifacts in many languages. For example, consider dependencies from code to a table or column in a database, or dependencies on a field in a JSON string, or between HTML tags and JavaScript.

The concept of conformance checking could also be broadened to cover other aspects of software development, for example conformance between end
user documentation and source code. To facilitate such analyses, we consider developing a unit test framework that focuses on conformance checking where developers write unit tests that check conformance between important project artifacts and source code.
Bibliography


