Managing Software Design Erosion with Design Conformance Checking

Master's Thesis

N. J. Karsidi

Managing Software Design Erosion with Design Conformance Checking

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

N. J. Karsidi
born in Heemstede, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
`www.ewi.tudelft.nl`

Cannibal Game Studios
Molengraaffsingel 12-14
Delft, the Netherlands
`www.cannibalgamestudios.com`

© 2012 N. J. Karsidi. All rights reserved.

Managing Software Design Erosion with Design Conformance Checking

Author: N. J. Karsidi
Student id: 1257315
Email: `n.karsidi@cannibalgamestudios.com`

Abstract

Software design erosion is a well known process; however, once it becomes noticeable it may already have progressed so far that repairing it is difficult and costly. Design conformance assessment techniques can help developers to detect – and mitigate – the effects of design erosion, before they cause problems to the long-term maintainability of software systems. Existing techniques have already been proven successful in controlled cases, but are not yet ready for widespread adoption in production environments.

This thesis studies the requirements and effects in the context of a real-world production environment and serves as a step towards making design conformance assessment techniques an economically viable investment for businesses. The contributions of this thesis are: an evaluation of the maturity of existing techniques, an inventarisation of requirements that arise from business environments with respect to design conformance assessment, and the implementation of the *SharpDCA* prototype tool that was evaluated in an ongoing development project.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dr. M. Pinzger, Faculty EEMCS, TU Delft
Company supervisor:	Ir. J. Dobbe, Cannibal Game Studios
Committee Member:	Dr. S. Dulman, Faculty EEMCS, TU Delft

Preface

I consider this thesis to be the culmination of much that I have learned and experienced in the past half-decade. While the goal of this work has been to develop – and prove – my abilities as a scientist, the lessons that I have learned go far beyond only the ones related to my specialization; I have learned at least as much about myself as I have about software engineering.

I could not have accomplished what I have without the people around me. In particular, I want to thank my parents for their unwavering support whenever I have chosen to pursue a goal in my life, Martin for his guidance and for the occasional dose of realism when my expectations were about to take off into orbit, and my colleagues at Cannibal for the great place to work and learn.

Last but not least, I also want to thank my dearest friends (you know who you are) for supporting me whenever I needed it and making me a wiser man in the process.

Contents

Chapter 1

Introduction

The 'erosion' of software design is a well known side-effect of the evolution of software systems, as it has been studied for the past two decades [35]. To be more precise, small inconsistencies between the original design and the implementation are often introduced whenever software is changed. Individual inconsistencies may have no noticeable effects; however, as they accumulate, maintainability will degrade significantly.

As the maintainability of a system decreases, changes become harder to implement. From a business perspective this means that software is *less flexible* with respect to the business processes it is supposed to support, and implementing changes becomes *more expensive*. Moreover, these effects increase exponentially over time, because preexisting design erosion will only make subsequent changes more difficult.

This thesis explores *design conformance assessment* (DCA) techniques as a possible solution to the design erosion problem. By automatically monitoring the consistency between design and implementation, developers can be made aware of inconsistencies much earlier, and the overall software quality can be better guarded.

1.1 Problem Statement

There already exist several approaches to design conformance assessment, e.g., [32, 25, 24]; however, these are not very likely to be applied in real-world environments. The reason for this, is that any DCA technique should be tightly integrated into the development process in order to be effective; however, changing the existing workflow comes at the risk of disturbing the production process. This is a risk that businesses are not eager to take, since such a change may affect the effectiveness (and thus the profitability) of a project or organization [16].

As with any addition to the development process, introducing a DCA workflow will require some extra structural expense, in terms of the time and effort that is required to perform DCA. It is important that this new workflow will not distract from existing activities and that this added cost also translates into added value.

This is the key problem throughout this thesis. The overarching goal is to create a DCA approach that is mature enough to be applied in the real world; this thesis is aimed at formulating and testing a concrete approach that makes a step towards reaching that goal.

1.2 Goals

In order to advance DCA techniques to the more demanding production environment, three goals have been formulated for this thesis. Achieving each of these goals subsequently produces a better insight into the interactions of the design conformance assessment cycle with the existing development process. The concrete goals of this thesis are:

1. To determine the requirements for DCA from the perspective of a production environment.
2. To formulate an approach to integrate DCA techniques into an existing development process, and create a prototype implementation.
3. To evaluate the effectiveness of this approach, with respect to a production environment.

1.3 Research Questions

In order to test the formulated approach a prototype implementation of a DCA tool, called *SharpDCA*, will be created and evaluated in a production environment. This evaluation is aimed at answering the following research questions:

R.Q. 1: Are developers more aware of inconsistencies, when using the SharpDCA tool?

Only if developers are *aware* of inconsistencies can they take action to prevent design erosion. This is the most important requirement for any DCA approach. Hence, this question is intended to obtain validation for the effectiveness our approach, with respect to mitigating the effects of design erosion.

R.Q. 2: Are developers better capable of effectively managing inconsistencies, when using the SharpDCA tool?

In a business environment, however, the effectiveness of the approach is not limited to the ability to detect design inconsistencies. It should result in a net positive added value to the value chain of an organization. This essentially means that the existing development process should not be negatively impacted by the addition of a DCA workflow. Moreover, developers should be able to use the functionality of the DCA tool to resolve design inconsistencies that they would otherwise not be able to.

1.4 Research Approach

In order to answer the aforementioned research questions, a study is performed in the context of a real-world software development project at Cannibal Game Studios. Requirements for automated support by a DCA tool are gathered from the developers working on this project, and a prototype implementation is created. This prototype specifically aims to also fulfill the *process requirements* besides the *technical requirements* of a DCA implementation.

The same real-world environment is then used to evaluate the effectiveness of the prototype, by means of a pre-experimental study that allows developers to use the DCA implementation on a project which is currently being developed.

1.5 Structure

Chapter 2 first discusses the case which motivates this study, and looks at Cannibal Game Studios in detail. Chapter 3 continues with an in-depth look at related work in the area of managing design erosion, as well as the individual advantages and disadvantages of existing design conformance assessment techniques.

The requirements for a concrete DCA tool, *SharpDCA*, will be formulated in Chapter 4, based on the context of the presented case. Based on these requirements, in turn, Chapter 5 presents the design and implementation of the *SharpDCA* tool. Chapters 6 and 7 present the evaluation of this approach and the prototype tool, the results of which are presented in Chapter 8.

To conclude, Chapter 9 reflects back on the original motivation for this study and discusses possible future work.

1.6 Summary

Enforcing consistency between the intended design and the actual implementation is a logical approach towards preventing design erosion, and thus towards managing software evolution. However, applying this approach in the real world is not a trivial exercise. Businesses have strict requirements in terms of *return-of-investment* and the *impact of change*, when changing the existing development process. In order to mature existing techniques to the point at which they become economically viable, we need to understand these real-world requirements.

The next chapter will take the case of Cannibal Game Studios, a company developing middleware and customized products, to understand these requirements better.

Chapter 2

Case: Cannibal Game Studios

There is a clear business case for techniques that can help to manage design erosion, but current techniques do not yet meet the specific requirements that arise from business environments. In order to explain these requirements, this chapter describes a concrete case: a middleware development company called Cannibal Game Studios.

Section 2.1 describes the demand for a way to manage design erosion, from a business perspective, which is the direct motivation for this thesis; Section 2.2 discusses the specific environment of Cannibal Game Studios in more detail; and Section 2.3 continues with the discussion of an initial case study, aimed to identify the specific causes for design erosion at the company.

2.1 Motivation

The necessity of software evolution is a well know fact [35], as is its consequent effect: design erosion. Since the need for change is inescapable in any (successful) software system [35], its evolution should be *managed* in order to mitigate as much of the negative side-effects as possible.

There are two situations where design erosion is of particular concern; when software is changed frequently, or when the expected lifespan of a software system is very long. In both cases there is a clear potential for the accumulation of design violations during the lifespan of the system. Organizations that have to deal with these types of software systems will likely experience the need for ongoing management of design erosion.

2.2 Cannibal Game Studios

Cannibal Game Studios is such a company, which strategically focuses on the reuse of software frameworks for customized products. Where reusable frameworks indicate a long lifespan, product customization indicates the need for change; the combination of which creates a rather volatile balance. As business expands, and the demands for the software products change, the company increasingly experiences reduced maintainability due to the divergence of the original software design and its implementation. Moreover, the original designs also diverge with the actual requirements for the system. Hence, there is a need for more agility in changing designs as well as their implementations during the lifespan of software systems, and in particular reusable software frameworks.

2.2.1 Company and Strategy

In order to reduce the turnaround time of projects while creating complex and highly customized applications, a repository of proprietary modules is maintained. These modules are reused, extended and customized as the need arises, while any newly created code is also being added to this repository for future usage.

This strategic reliance on both long-term reuse and customization gives rises to a myriad of software evolution issues, of which design erosion poses an ongoing concern to the developers at the company. For this reason, mitigating the effects of design erosion directly supports the business strategy of Cannibal Game Studios.

2.2.2 Business Case

The business case of managing design inconsistencies is that it can reduce development time and extend the product lifespan. This, in turn, results in more flexibility when software is required to adapt to a changing market environment and saves cost whenever maintenance is performed.

These results can be achieved by managing the identified causes of these inconsistencies (see Section 2.3.2); i.e., *increasing design understanding among developers*, *enforcing consistent design documentation*, *discouraging hacks to save time* and *improving awareness of problematic designs*.

2.3 Case Study

In order to formulate an approach to 'manage' design erosion, a case study was performed to identify the particular needs of Cannibal Game Studios. The study is based on the observation of development activities, as well as interviews with developers and managers. While there are many ways of managing design erosion, there are also many possible trade-offs in doing so. For example, intentionally not doing anything might very well be a valid strategy to reduce cost and can be successful for smaller software systems. On the other hand, another (extreme) way of managing design erosion is to redesign the entire system before each single change. While this is probably extremely expensive, it can be a valid strategy for certain critical systems. In any case, before formulating any approach it is important to understand the context of the problem.

2.3.1 Brainstorming Session

As a first step in the context gathering process, a structured brainstorming was conducted with 4 experienced developers, a technical manager and a project manager at the company. The aim of the brainstorming session was to allow all responsible team members to share their perception of the development process, and the cause of design erosion. Specifically, by studying the existing design-implementation cycle the impact of changing this process can be better understood.

Session design

The session was structured to ask the following questions:

Q1. How is a software design specified?

- a) What common elements do designs contain?
- b) What format is used (e.g., text, diagrams, etc.)?

Q2. What may cause you do deviate from the intended design?

- a) What are valid reasons to violate a design?

Q3. Who is responsible for...

- a) design communication and understanding?
- b) enforcement of the intended design?
- c) deciding when a design should be changed (i.e., when it is insufficient)?

These questions were presented to the participants, one at a time, and they were asked to write down three or more answers. Participants were asked to provide their answers without consulting their colleagues to make sure that they were reflecting on their personal experience. After every participant produced several answers, a short discussion was used to see if there was a general consensus about the answers.

Results

The result of the brainstorming was a list of possible answers to each questions, where certain similar answers may have been given multiple times by different participants. By categorizing all answers a number of common concerns became apparent, moreover by counting the number of answers in a category these concerns could be prioritized.

Definition of 'software design' (Q1) When developers refer to 'software design', they often do not have a strict definition of the term. However, most often the terminology is used to indicate constructs that are less abstract than architectural patterns and more abstract than algorithmic patterns. This corresponds to the level of abstraction used by the 'Gang of Four' Design Patterns [13], albeit less well defined.

Specifically, a 'software design' was described by many developers as a set of rules describing the 'relations and interactions' between 'entities' (e.g., modules, classes, methods, etc.). The reason that they often refer to this level of abstraction is because changes at this level are frequently required, which makes their activities often center around this particular level. Moreover, the high number of entities and relationships gives rise to a high level of complexity, compared to architectural design for example.

Causes of design violations (Q2) There are several reasons for the introduction of design violations; much cited are inadequate design documentation, inadequate communication in general, or a lack of design understanding.

One issue with the current form of technical designs that was identified, is the informal nature of the documents. With respect to the information contained, there is no enforced consistency between different technical designs in terms of what they describe and how they describe it. This can make technical designs harder to understand. And, with respect to the semantics of a technical design, designs are often ambiguous and the information may be incomplete. This means that automatic verification or even mapping to the source code is often very difficult.

Most design knowledge is stored tacitly in the minds of the developers. Much cited reasons for this are that the overhead of maintaining explicit documentation is perceived to be too expensive, the effort is tedious and it feels redundant if the design is likely to change again soon. Moreover, developers have stated that they often need to start writing code (i.e., create a prototype) in order to gain sufficient knowledge to write a good design. Then, once working code has been written, developers seem averse to 'go back' to writing technical designs.

Responsibility of design conformance (Q3) Developers believe that good software design is a team effort, which places a shared responsibility on each team member. However, they are very aware of the fact that design violations are often intentionally introduced simply to save time or effort. This is quite common in production environments, where there are ever impending deadlines and where time-to-market is a key business driver. Moreover, this an accepted practice, given that such design conformance violation are fixed as soon as possible; however, developers say, in reality such 'TODOs' are often forgotten.

2.3.2 Identified Causes

Using the data gathered during the brainstorming session, a number of specific causes for design erosion have been identified at Cannibal Game Studios, which will be summarized in this section. It was already known that the company strategy inherently increases the likelihood of design erosion occurring, but changing the business strategy is not a realistic way of managing the problem. However, there are also other processes that *can* be managed, which are the main focus of the approach that is presented in Chapter 4.

First of all, a design violation can be accidental or intentional. Accidental design violations are often introduced when developers do not have sufficient understanding of the intended design. Developers feel that this type of design violation is most common, as was indicated by the prioritized results. Much cited causes for insufficient design understanding are:

- Incorrect or outdated documentation.
- Missing design documentation.
- Lack of communication of knowledge.

Intentional violations, on the other hand, can be an indication that the intended design does not sufficiently capture the requirements imposed on the software system. If developers are often forced to circumvent the intended design (a.k.a. '*hacking* a solution'), then this design might require rethinking. Cited causes for such intentional violations are:

- The requirements were not yet fully understood when the original design was made.
- There is not always enough time to implement features according to the design.
- Previous intentional violations that should have been fixed, are forgotten.

These causes are specific to Cannibal Game Studios, and are the precise factors that we want to manage. When formulating an approach (see Chapter 4) these will be guiding for the choices in techniques that are being used to manage design erosion.

2.4 Summary

This chapter discussed the need for 'managing' design erosion in businesses and looked at the situation of Cannibal Game Studios in particular. The initial case study was presented, which was used to study the particular strategy and environment at Cannibal Game Studios. Moreover, a number of concrete causes of design erosion have been identified.

The next chapter will focus on existing techniques for managing design erosion, and particularly discuss design conformance assessment techniques.

Chapter 3

Related Work

This chapter discusses the existing work, related to managing design erosion. Moreover, *design conformance assessment* (DCA) techniques in particular will be treated in detail.

Section 3.1 discusses techniques for managing design erosion, and in particular the techniques that can manage the causes that were identified in the previous chapter. Section 3.2 continues with an in-depth discussion of DCA techniques and classifies the different approaches in two separate taxonomies.

3.1 Overview

Design erosion can occur due to a variety of reasons, such as a lack of design understanding or inadequate design communication. Since there are different causes for design erosion, different techniques for managing the effects have also been developed. This section provides an overview of related research areas.

3.1.1 Design Erosion

Software evolution effects, such as design erosion, are well known. Parnas [35] describes these effects as the 'aging' of software. He remarks that any software system that is successful enough to be used for any extended amount of time, will inevitably have to change to keep up with its environment if it is to stay successful.

Others have created taxonomies of the causes of design erosion, arguing that design erosion is a direct consequence of change to a system and then identifying the possible reasons to change an existing system [8, 39]. The causes cited by these taxonomies are different than the ones that we have identified in Section 2.3.2; while these taxonomies focus on the possible reasons to perform maintenance (i.e, change as the cause of erosion), we have found that design erosion occurs whenever there is *improper maintenance* (e.g, due to lack of design understanding or time constraints).

3.1.2 Program Understanding

Bosch argues that the key to properly maintaining software is understanding the design and specifically the rationale behind previously made design decisions [7]. Communicating the rationale behind a design is not a trivial exercise. For that reason new ‘views’ on software design are being developed that aim to explicate the design rationale, which is usually only implicitly stored in resulting implementation [19].

3.1.3 Formal Design Definition

Formal design specification can help design understanding in any case [14]. However, if the goal is to unambiguously verify the consistency between an implementation and the intended design, a well defined design becomes a strict requirement.

Many existing design specification formats may not be unambiguous or expressive enough to meet the requirements imposed by any automated design conformance checking approach. UML is a prime example, because of its widespread adoption as the *de facto* standard for design specification. Although it is quite well defined, it is actually not formal [37] and it is not expressive enough to specify certain design patterns [12]. While several approaches that are being developed attempt to extend UML into a more formal language [37, 12], others attempt to extract information contained in UML models and translate this into a more formal model [17].

3.1.4 Design Recovery

The automatic detection of certain design constructs in source code is relevant for both verifying design conformance and the retroactive generation, or updating, of documentation. Much of the work in this area seems to focus on the *‘gang-of-four’* design-patterns [13], since this apparently represents a complex level of abstraction.

Some techniques focus just on the recovery of known patterns from source code [27, 3, 15], while later techniques also focus on making these traceable to the intended design [2]. When attempting to generate documentation from source code, much information might be lost. Especially when recovering design-pattern realizations, the rationale behind the design decisions that underlie the choice for specific patterns contains value. There are techniques that attempt to also recover this rationale [20].

Not all design-pattern detection techniques focus on the source code, however. Analyzing intended designs for known constructs can also be done to assist software designers, by detecting their intentions and suggesting better alternatives [6, 42].

Furthermore, other techniques that do not focus on the recovery of known constructs (such as design-patterns) also exist. These techniques are able to give a much more abstract design overview and are often based on clustering of program entities [29, 28].

3.1.5 Design Conformance

Checking consistency between a design model and an implementation is not only useful for the detection of design erosion, but is also a helpful tool for program understanding.

Murphy et al. [32] use their Reflexion Models technique to iteratively test – and refine – their program understanding, based on a model of the *expected design*. Such a technique can be directly extended to check design conformance, by replacing the expected-design model by an intended design model [33].

It is however noted by Balzer that design inconsistency, up to a certain level, is an expected and temporary side-effect of the change process that will resolve itself [5]. If inconsistencies are managed too aggressively, this could prove counter productive.

The main differences between the various approaches for detecting inconsistencies, are the models used to represent the intended design and implementation. First of all, the models implement various levels of abstraction. While some techniques are specifically aimed at the architectural level [36, 10, 41], the design-pattern level [38, 24] or system modularity [18, 40], others are not dependent on a specific abstraction [32, 33, 25].

Most current techniques focus on structural design aspects [9, 23, 24, 21], but a complete design also comprises behavioral aspects of the software. A few techniques already focus on this 'dynamic' design [26, 22], but many of the techniques for structural checking can theoretically be extended with behavioural checking as well.

Two detailed classifications and explanations of the aforementioned approaches, as well as a discussion of their advantages and disadvantages, will be presented in Section 3.2.

3.1.6 Documentation Consistency

Besides checking the consistency between the intended design and its realizations, the general case of consistency between other artifacts is also a concern. One of the most significant artifacts next to the design and source code is the documentation of the system.

Since the documentation of systems is primarily intended for the human developers, the format is natural language and a myriad of diagrams. This means that there is no formal and well-defined format for documentation, which means that automatically testing the correctness of documentation is difficult. The best most approaches can hope to achieve is to monitor the links between documentation and source code, as to make developers aware of documentation that is possibly outdated or affected by a certain change in the code; Antoniol et al. [3, 2] and Marcus [30] try to do this by creating a mapping between the source code and documentation artifacts.

3.2 Design Conformance Assessment

This section discusses the class of *design conformance assessment* (DCA) techniques in detail. These techniques are of particular interest, because they can be used to manage design erosion in multiple ways.

The primary benefit of DCA approaches is that it can provide awareness of the existence of design inconsistencies. Moreover, comparing the expected design with the actual implementation can help to increase design understanding [32]. Hence, a DCA approach is capable of tackling exactly those issues that have been identified in Section 2.3.2.

There are a number of advantages to DCA techniques that make it a particularly interesting approach to manage design erosion; it is *extensible* to many aspects of software design, the implicit requirement of a formal design specification can help to *improve design understanding and communication*, and the techniques can be automated and made part of an existing *continuous integration* cycle.

This makes such approaches particularly well suited for application in a production environment. The flexibility of evaluating only the relevant design aspects can result in lower cost and complexity, improving design understanding and communication deals with some of the primary causes of design erosion and the ability to perform DCA as part of the continuous integration cycle means that it can be performed frequently with minimal impact on the existing development process.

While DCA is a flexible solution to manage design erosion, this flexibility does imply that the particular approach should still be tailored for a specific environment. This chapter explores different approaches and techniques that can be used to implement design conformance assessment.

3.2.1 Taxonomies

Among the existing DCA approaches, there are two ways by which they can be classified; either by the type of design they analyze, or by the type of design model is used. Such taxonomies are useful; since the advantages and disadvantages of each approach may differ in different environments, it helps to understand the differences between the various techniques.

3.2.2 Classification by Design Type

As there are many possible views on software design (i.e., different 'types' of design), there also are corresponding approaches to design conformance assessment. Common views on design are *architectural styles*, *modularity design*, *object-oriented design patterns* and *coding styles*, hence there are design conformance assessment techniques for each one of those.

When implementing DCA in a business environment, it should be clear which aspects of the software design should be tested, and a technique should be chosen accordingly.

Architectural styles

At the most abstract design level architectural designs can be expressed as certain common 'styles'; architectural styles can describe structure and behavior of a system at the highest level. Some examples are: client-server-, distributed-, event-driven- or monolithic- architectures.

Modularity design

Another type of high-level design concerns how the system is modularized. How 'good' modularity is defined may differ. For example, functional cohesion might be desired for reusability, but a program might also be modularized based on information hiding, low coupling, etc.

In [18] it is argued that actually implementing decoupled structures as they are designed is not trivial and an automated conformance checking mechanism is required to feasibly implement modular software designs. *Design Structure Matrices* are chosen as a comparable representation for both program and design.

Tool support for modularity conformance checking is presented in [40]. The presented *Clio* tool does not, however, define a modularity violation as something that can exhibit itself in a single version of the software. Instead, a violation is said to have occurred if modules change in unison over two consecutive releases. This is considered a modularity violation, since modules are supposed to be independent units that evolve separately. It should still be noted that some of these detected 'violations' are in fact semantic dependencies and not structural violations.

Design Patterns

Design patterns can describe both structural and behavioral designs at the object-level. This is an important class of designs, since they are often used to solve common design problems. On top of that, they represent a much lower level of abstraction than software architecture, which results in much more code changes at that level and subsequently require more care to keep design patterns intact.

Keller [20] describes the belief that understanding the rationale behind design decisions is required to properly perform work on existing source code. It is argued that design patterns hold this information, as they are the result of a choice for a particular design solution. In order to support this, their *SPOOL* environment for pattern-based reverse engineering can be used to provide visual views of these patterns.

One particular problem with design patterns is that a single pattern might have many realizations [21]. This makes it difficult to detect the proper implementation of a pattern based on a predefined template. Moreover, design patterns may be cross-cutting and intertwined [19], making it even harder to detect single instances. Kim et al. have implemented a design specification format in an attempt to deal with these problems [21, 23, 22].

Coding styles

On the lowest level are coding styles and guidelines. It is arguable whether these constitute a 'design', but they are certainly related. Moreover, the tools that support checking conformance to a specific coding style are very related to the tools for checking design conformance. Coding style conformance enforcement can be of substantial value in maintaining code understandability and maintainability, however, it still only targets concrete code structures and not the more abstract design structures.

Such 'coding style checkers' are much more mature than their higher-level counterparts, 'design conformance checkers'. Many such style checkers already exist, such as *StyleCop*[1] for .NET languages or *CheckSyle*[2] for Java.

[1] http://stylecop.codeplex.com/

[2] http://checkstyle.sourceforge.net/

3.2.3 Classification by Model Type

Where concrete DCA techniques often differ based on the type(s) of design that they analyze, an (arguably) even more fundamental classification can be made based on how they model the design and implementation before comparing them.

Since the abstractions made by these models have consequences in terms of preserved information and maintainability of the design models, the particular choice can have a significant impact on the effectiveness of a DCA technique. This section will discuss these differences for existing techniques.

Dependency Structure Matrices

A dependency structure matrix (DSM) [36] essentially summarizes the dependency relationships between a number of program elements. These program elements are enumerated in the margins of the matrix, where the matrix cells denote the existence (or possibly even the *count*) of dependencies between each pair of program elements. Moreover, modular structures can be described by sorting the rows and columns of the DSM to represent cohesion of certain program elements into design structures.

One main advantage of DSMs is their simplicity, but this comes at the cost of reduced expressiveness for describing design constructs. Specifically, only the presence or absence of a dependency can be expressed [36]. Nevertheless, the DSM is suitable to represent a hierarchical view of the modules in a software system.

The semantics of the ‘modularity’ that can be modeled by a DSM is defined by the choice of the elements that are used to construct the matrix as well as their order. Huynh et al. [18] choose to model the design-level modularity, based on the *design rules* [4] of a system. In order to accomplish this the rows and columns of a DSM are used to specifically represent so called *design variables*, which are a representation of either an environmental condition or a ‘design decision’. A set of design decisions, in turn, can describe a program element (e.g., an object method, which in [18] is illustrated as being described by a decision for a method signature, a data structure and an implementation). This way a DSM can express how design decisions are dependent on each other. When such a DSM is then clustered, a program structure of so called ‘dependency clusters’ arises which modularizes the program based on the related design decisions.

However, it is noted that specifying a high-level model in the form of a DSM is not straightforward [17], since engineers often find it difficult to explicitly enumerate all design decisions. This results in manually specified DSMs often containing errors and ambiguities. Instead, Huynh, Cai and Shen propose to let engineers use a well known modeling language, such as UML, and automatically derive a DSM model from the user-specified model [17].

Source Code Query Languages

Design conformance assessment essentially comes down to asserting that the source code meets certain expected conditions. While other approaches might encode these assertions implicitly in some reference model, source code query languages (SCQL) take a more direct approach by explicitly formulating queries over the source code that test for the conformance to these expected conditions.

One of the earlier design conformance checking approaches by Sefika et al. [38] is based on directly defining constraints of the source code using such queries. They propose a tool-aided approach, *Pattern-Lint*. SCQL offer an efficient method to gather information on static data that is inferred from the source code. However, Sefika argues that *both* static and dynamic checking are required to fully assess design conformance.

The actual conformance checking step is based on the rules and constraints for the design that are formulated as logical queries. The source code model is then tested against this design model in order to obtain positive evidence and violations of design conformance (i.e., converging and diverging of program elements with the design).

One of the trade-offs that SCQL techniques make, is the sacrifice of abstraction for high expressiveness in return. Each constraint requires that a new query be formulated, which makes it less practical for structural integration in a development process. That said, it is an effective method of targeted information gathering from source code [36], e.g., when gathering source-code metrics.

Structural Constraint Languages

A Structural Constraint Language (SCL) models programs and design constraints using a first-order logic language. General purpose logic languages (e.g., *Prolog*, or similar languages) allow design conformance assessment by modeling the design constraints as logical queries. The fact that a structural design generally can be expressed as a set of entities which are related according to certain rules, makes such a language well suited to describe the structural design of a program.

SCL are related to SCQL, as both operate by specifying queries (representing the design) over a model of the implementation. The main difference is the type of model that is used for the design representation; SCL model the design as a logical proposition, while SCQL use a relational database containing facts about the design.

While SCL offer highly expressive semantics and an efficient representation for design conformance assessment, the main issues with this approach are the complexity of the syntax and the lack of industrial-strength tools [36].

One approach based on a SCL is implemented by Eichberg et al. [11]. In the approach dependencies are defined between so called *ensembles* which are groups of program elements (classes, methods, etc.); which kind of program elements may be contained in such an ensemble depends on the level of abstraction of the dependency. However, the specification itself is done using a visual language only after which it is transformed to *Datalog* (which is a subset of *Prolog*), thereby mitigating some of the usability concerns.

While such techniques are generally indeed aimed at structural conformance checking, the same technique can also be used for checking behavioral features, as Eichberg et al. mention [11].

Domain Specific Modeling Language

Domain Specific Modeling Languages(DSML) allow the explicit separation of the design and the functional code. The approach depends on a *domain-specific language* (DSL) to define a design model which can be used to generate the code structures that essentially are 'the design', which in turn can be extended with any functional code. The expressiveness of the DSL determines what kind of designs can be modeled. A notable subclass of DSML are *Architectural Description Languages* (ADL), which are constrained to describing only architectural designs.

The obvious benefit of using a DSL-based approach is that design conformance is ensured implicitly because the related code is entirely generated. However, the DSL-based approach also brings along its main disadvantages, which are the current lack of support by matured tools and the inability to apply the method to existing projects, because the approach works at the language level.

Zheng proposes a method for both structural and behavioral architecture conformance preservation, *1.x-Way architecture-implementation mapping* [41]. One key observation made is that changes that cause architectural drift can either be made to the code or the architectural itself. This gives rise to a two-way model of architectural erosion.

The proposed technique is based on the so called *deep separation* of architectural structures and any functional code. Specifically, the architectural elements are generated automatically and maintained as completely independent from the functional, user-specified, code. Only later are the two merged by a program composition mechanism. This would indeed mean that design conformance *checking* would become unnecessary, since it would be guaranteed by the generation of the design structures and the automatic composition with functional code defined by the user. However, this new approach poses substantial new challenges for properly propagating changes to either architecture or user code to the actual program as it hinges on how the two are composed.

Structural Pattern Matching

Murphy has originally introduced the concept of *software reflexion models* [32] to improve program understanding; by iteratively comparing what an engineer believed to be a correct high-level view of a program to an automatically extracted view, the understanding of the program could be refined. The approach works by generating two models, one of the source code and another one of the user-specified high-level model which can then be compared. The exact information contained in the models is dependent on the features under investigation; as long as both models contain comparable data, reflexion models can be used to summarize the convergences and divergences between the two. Besides aiding in program understanding, the general concept of reflexion models also can naturally be used for checking design conformance between an initial design and an actual implementation [33], depending on how the resulting reflexion models are interpreted.

Because the reflexion models approach is not limited to the comparison of a specific type of artifact, it can be applied on virtually all types of high-level models, which is reflected by the myriad of later techniques that are based on this approach. Other benefits are the high-level of abstraction at which engineers are allowed to specify their designs and the relative maturity of the tool support. On the other hand, some disadvantages are a limited expressiveness of the designs that can be modeled and the fact that one has to define a mapping between the models manually, which might be tedious. [36]

One shortcoming in the expressiveness of the original approach of Murphy was the inability to express hierarchical models, because the rules used for mapping high-level model to the source model yielded ambiguities for hierarchical structures. However, Kroschke and Simon in [25] refined the technique to allow the hierarchical decomposition models by formalizing the semantics of the mapping relationship to solve these ambiguities.

Another limitation to the expressiveness of reflexion models is the inability to model generic design patterns, but only specific instances [23]. This essentially means that the only way to capture possible variations in the implementation of generic patterns is to explicitly model each variation. *Structural Pattern Matching* (SPM) approaches allow designs to be characterized at a high level by groupings of structural entities and their relations. The program itself can also be viewed as a large number of entities with relationships among them and by matching the program structures to the predefined design structures design conformance can be verified.

One caveat is that combinatorial explosion lurks when programs grow large, so an efficient search algorithm is a must-have for this approach. However, the advantage of this specification method, over defining explicit constraints, is that it allows more generic design definition. This is especially true for design patterns. Although design patterns offer an intentionally more limited solution space to design problems, Kim et al. [23] note that checking conformance to common design patterns is still an issue.

Design pattern matching This approach is particularly used to check design pattern conformance. Two related problems are identified specifically related to design patterns. First, that performing conformance checks manually requires intimate knowledge of the particular design pattern, which is not always widespread among developers. This calls for tool support which can assist engineers in performing the conformance check. Secondly, even with tool support it is hard to automate validating design pattern implementations, since there often is a range of possible realizations to a single pattern. This fact makes techniques which simply compare a model of the source code to a model of a single design inherently unsuitable for checking conformance to design patterns [23].

Antoniol et al. [3] suggest an early SPM-based approach of specifically detecting design patterns in existing code. The approach defines a design pattern as a tuple of classes and relations among them. One of the main challenges of this approach is the possibility of combinatorial explosion when searching in code with many classes and relations. This is partly mitigated by looking at metrics and structural properties to filter for potential candidates for a particular pattern match.

Among the listed future work is the detection of ‘other pattern families’. However, since the pattern definition is based on a tuple of classes and relations, the expressiveness of the detectable design patterns is limited by this which might return in either false positives or false negatives. Experimentation has, however, revealed that the number of false positive matches is relatively small, so it might be feasible for a human to filter these manually.

Heuzeroth [15] discusses a method of detecting design patterns using a combination of static as well as dynamic analysis. A design pattern is also defined as a tuple of program entities and interrelationships, but additionally dynamic analysis is used to find run-time relationships.

Kim et al. [23] present a more accurate approach by replacing the simple tuples by more strictly defined clusters of structural elements. These clusters define the so called *pattern roles* that make up a design pattern and which may be defined in an extension of UML called the *Role Based Modeling Language* (RBML). The technique, in its current state, only checks conformance for structural properties; however, future work includes extending it to work for behavioral properties as well.

In addition to their earlier work, Kim and Shen [24] continue work on semantic conformance checking and go into more detail about the limitations of design conformance checking using RBML. They note that their technique is not suitable for checking conformance to simpler patterns (i.e. with less structural features). Furthermore, they propose to use model-level constraints in their language to specify run-time invariants and pre- and post conditions. Additionally they are currently working on other (non-static) pattern specifications; the *Interaction Pattern Specification* and the *StateMachine Pattern Specification* which are intended specifically to allow the specification of behavioral rules.

Reflexion Models

The Reflexion Models approach is different from the other techniques, in the sense that it is independent of the exact type of model that is used.

Murphy has originally introduced the concept of *software reflexion models* [32] to improve program understanding; by iteratively comparing what an engineer believed to be a correct high-level view of a program to an automatically extracted view, the understanding of the program could be refined. The approach works by generating two models, one of the source code and another one of the user-specified high-level model which can then be compared. The exact information contained in the models is dependent on the features under investigation; as long as both models contain comparable data, reflexion models can be used to summarize the convergences and divergences between the two. Besides aiding in program understanding, the general concept of reflexion models also can naturally be used for checking design conformance between an initial design and an actual implementation [33], depending on how the resulting reflexion models are interpreted.

Because the reflexion models approach is not limited to the comparison of a specific type of artifact, it can be applied on virtually all types of high-level models, which is reflected by the myriad of later techniques that are based on this approach. Other benefits are the high-level of abstraction at which engineers are allowed to specify their designs and the relative maturity of the tool support. On the other hand, some disadvantages are a limited expressiveness of the designs that can be modeled and the fact that one has to define a mapping between the models manually, which might be tedious [36].

One shortcoming in the expressiveness of the original approach of Murphy was the inability to express hierarchical models, because the rules used for mapping high-level model to the source model yielded ambiguities for hierarchical structures. However, Kroschke and Simon in [25] refined the technique to allow the hierarchical decomposition models by formalizing the semantics of the mapping relationship to solve these ambiguities.

Another limitation to the expressiveness of reflexion models is the inability to model generic design patterns, but only specific instances [23]. This essentially means that the only way to capture possible variations in the implementation of generic patterns is to explicitly model each variation.

3.3 Summary

There is a myriad of techniques that can help in mitigating the effects of design erosion that have been identified at Cannibal Game Studios. An overview of these techniques was presented, which has been the starting point for formulating the approach that is discussed in the following chapter.

Design conformance assessment can be a particularly helpful tool in a real-world production environment, however, there are many aspects that should be taken in consideration. Each DCA approach has its specific advantages and disadvantages based on the environment that it is used in. When considering to implement DCA, it is important to consider which design aspects should be checked and what kind of model is most appropriate to represent the design. Making the wrong choice can result in a significant added complexity without adding any business value.

The next chapter describes how a concrete DCA approach is formulated, specifically in the context of the company.

Chapter 4

Tool Requirements

The previous sections have discussed why design erosion is a significant problem, particularly for organizations that want to combine reusable software frameworks with highly customized products. An approach is formulated, based on the use of design conformance assessment (DCA) techniques to detect design violations as soon as they are introduced.

This chapter describes that approach in more detail, in particular the requirements and scope for a prototype implementation are formulated. Section 4.1 starts with an overview of the goals of such a prototype. Section 4.2 discusses the requirements for the prototype and the process of gathering those, so that the scope of the implementation can be discussed in Section 4.3.

4.1 Goals

The overarching goal is to reduce design erosion, by detecting the introduction of design violations as soon as they occur. Specifically, such violations are detected as close to the source as possible in terms of time and location, which makes fixing a violation much easier than when significant erosion has already occurred.

This requires a DCA technique that can test for design violations as often as possible, which in turn suggests an automated approach. A prototype tool will be implemented to test this approach of ‘continuous’ design conformance assessment.

Besides providing quick feedback, an automated approach has other benefits as well. For example, an automated approach implicitly enforces formal design specifications, which can help in better design communication and understanding. Moreover, if design violations can be detected automatically and traced to a specific location in the source code, it becomes possible to see when developers violate a specific part of the design frequently, indicating a possible problem with the design.

However, besides performing automated design conformance checks, such a tool must be able to integrate seamlessly with existing development practices if it is to be used by software developers. This gives rise to the additional goal of tuning the tool to a production environment.

4.2 Requirements Analysis

In order to integrate the prototype tool with existing practices, we need clear insight into the development process and the reasons why software design is violated in the first place. To this end, several aspects of the development process at Cannibal Game Studios have been studied. As described in Section 2.3.1, a brainstorming session was performed to learn from first-hand what the added value of a DCA tool would be (and what not). Additionally, an analysis of existing projects was performed to serve as sample cases on which the tool should be able to work. This combined knowledge is used to formulate concrete requirements for a DCA solution.

4.2.1 Design Analysis

While the brainstorming mainly focused on how a DCA tool should interact with the developers, an analysis of existing technical designs was performed to learn how the tool should interact with the code. Specifically, the design-analysis is aimed at identifying the concrete design constructs that the prototype should be able to test for design conformance.

This analysis was done using two complementary methods; by extracting the structural and behavioural descriptions from existing design documents, and by manually inspecting source code. The resulting data served as the input for creating a prioritized taxonomy of design constructs. The prioritisation determines the most common design constructs, which was used for selecting a limited subset of constructs to be supported by the prototype.

The analysis confirmed the lacking formality of designs that was found to be an important cause of design erosion during the brainstorming. A significant portion of technical designs consists of natural language descriptions which make automated verification difficult. Moreover, many design documents mix the description of different design types and levels of abstraction, which makes it more difficult to fully comprehend its larger system context.

Taxonomy of designs

A classification of the different types of information contained in the technical design was made. This section gives an overview of the common constructs that were classified. First of all, technical designs generally contain 3 classes of design information:

- Structural features of the code.
- Behavioral features of the program.
- Rationale behind design solutions.

Furthermore, this information describes either the structure or behaviour on one of three levels of abstraction; the *architectural*, *design-pattern* and *algorithmic* levels.

At the architectural level, entities are generally systems or software modules. Relations at this level are either the hierarchical containment of modules or interactions between entities in the form of method invocations or messages.

At the design-pattern level, entities are the typical object-oriented programming constructs, such as namespaces, classes, interfaces, structs, enumerations, methods and fields. The most common relationships between these entities are inheritance, structural containment or method calls.

Finally, at the algorithmic level there are mostly dynamic constructs, where the entities are variables and expressions, which are connected by assignments and logic branches.

4.2.2 DCA Technique Analysis

Besides identifying the needs from the perspective of the development process, we want to identify which existing DCA techniques might be suitable for implementation in a business environment. A survey of existing techniques has already been performed in Chapter 3, but given the list of common design constructs that are used at Cannibal Game Studios a more concrete comparison can be performed.

From the brainstorming and the analysis of designs, we have gathered that the most complex designs occur at the design-pattern level, so ideally the DCA technique would support the definition of relations between methods, classes, etc. Moreover, at least call, containment and inheritance relationships should be expressible. Furthermore, since design-patterns are often composed of multiple structural constructs, we are interested in how these rules can be grouped, which types of constraints can be expressed on individual rules and whether it is possible to define multiple types of relations in a single model.

The different approaches that were discussed in Chapter 3 have been compared based on these criteria. Table 4.1 shows the types of static constructs that can be tested for by several existing DCA techniques.

As can been seen, some techniques have a higher level of expressiveness than others. However, as noted in Section 3.2, this expressiveness is a trade-off with the complexity of the design specification, which in turn translates to a significant impact on the design specification process. Moreover, there is a significant variation in the level of maturity of the different techniques.

Source Code Query Languages (SCQL) [36, 38] are theoretically very expressive and have already been used in commercial applications; e.g., [34]. However, the design specification is very verbose and much better suited for gathering low-level code metrics. Moreover, it is not possible to model dynamic designs in such languages. For the same reasons Structural Constraint Languages (SCL) [36, 11] are not practical for this purpose.

Dependency Structure Matrices (DSM) [36, 18, 17] can express the relationships that make up the structural design of a program at a much higher level, which is preferable. However, the approach is quite rigid in the sense that it is not easily extensible and does also not allow the definition of dynamic constructs.

The Role Based Modeling Language (RBML) [36, 21, 23] approach seems promising for matching designs specifically at the design-pattern level. Also, it can theoretically be extended to model dynamic designs. However, its maturity is relatively low compared to the other methods and design specification is limited to specific design patterns.

The Reflexion Model (RM) [32, 33] approach and its extension, Hierarchical Reflexion Model (HRM) [25], appear the most mature and flexible of all. While not allowing the

		DSM	SCQL	RM	HRM	RBML	SCL
Entity support	Field	o	o	o	o	o	o
	Method	o	o	o	o	o	o
	Property (Getter/Setter)	*	o	o	o	o	o
	Type	o	o	o	o	o	o
	Namespace	o	o	o	o	-	o
	Assembly	-	o	o	o	-	o
Grouping	Annotations	-	-	*	*	*	-
	Naming	*	*	*	*	*	-
	Manual mapping	*	o	o	o	o	o
Relations	Implementation	o	o	o	o	o	*
	Extension	o	o	o	o	o	*
	Call	o	o	o	o	o	*
	Containment	o	o	-	o	-	-
	Multiple types per model	-	o	-	-	o	-
Constraints	Must always occur	o	o	o	o	o	o
	May occur	-	-	*	*	o	-
	Negation operator	-	o	*	*	*	o
Meta-constraints	Multiplicity	*	-	-	-	o	-

Table 4.1: Comparison of expressiveness for existing DCA approaches (supported: o, supported with extension of existing techniques: *, not supported: -).

expressiveness of SCQL or SCL, it is very well suited to express the static constructs that have been identified. Moreover, the technique is inherently capable of comparing any type of model, including behavioural designs.

4.3 Prototype Scope

After analyzing the needs and possibilities for DCA implementation in a production environment, a concrete set of bounds for the prototype were formulated.

4.3.1 Design Types

The results of the brainstorming have shown that developers would most value a tool that can check design conformance at the ‘design pattern level’, because this level contains complex relationships and is changed frequently. Based on this insight, it seems most effective to limit the initial scope of the design conformance assessment prototype to this level.

At this level of abstraction, there is still the distinction between static (structural) and dynamic (behavioural) design. The scope for the prototype tool is limited to checking static design constructs, because a static or dynamic analysis tool would require fundamentally different analysis methods. Implementing both is beyond the scope of this thesis, instead it is part of the future work. Static analysis was chosen over dynamic analysis because of several reasons:

- Developers have indicated that problems with static constructs are most common.
- DCA techniques for analysis of static constructs are more mature.
- Developers generally have more experience with documenting structural constructs, which makes evaluation easier.

4.3.2 Language Expressiveness

Given the limitation to static design-pattern level constructs, the required expressiveness of the language is also bound. The entities that should be expressed are *namespaces*, *classes*, *interfaces*, *structs*, *enumerations*, *methods* and *fields*. The relationships that can be expressed between these entities are limited to *interface implementation*, *class extension*, *method calls* and *entity containment*. For a more detailed description of the language design, see Section 5.2

4.3.3 User Interaction

An obvious way to integrate the prototype neatly into the existing tool suite, is by implementing the functionality as part of the IDE. This means that the developer should be able to specify an abstract model of the intended design, initiate the conformance assessment and interpret feedback all from within the IDE.

The design specification will be limited to a low-level format, since the focus of the prototype is not on design specification, but on the effects of the design conformance assessment itself. Furthermore, the execution of the process will be done every time the project is compiled. The assumption is that this is frequent enough to be effective, but the main motivation for this choice is technical in nature (also see Chapter 5).

4.4 Summary

This chapter described the analysis steps that have been performed to gather the requirements for a prototype implementation of a DCA tool in a production environment. A brainstorming session was performed to elicitate requirements from developers and better understand the existing practices, an analysis of existing technical design documentation and source code was performed to categorize common design constructs and a comparison of design conformance assessment techniques was made with an eye on the requirements.

The Hierarchical Reflexion Model technique has been found the most appropriate for implementation in this particular environment. Moreover, the scope of the prototype has been limited to checking static design constructs at the design-pattern level.

Chapter 5

SharpDCA

This section describes the design and considerations that lie at the base of the prototype implementation of our design conformance assessment (DCA) tool, *SharpDCA*. Furthermore, some concrete examples of the usage of the tool are provided.

Section 5.1 provides an overview of the tool and its functional components and dependencies. Section 5.2 through Section 5.6 continue with a discussion of the design decisions that have been made, as well as their rationales.

5.1 Overview

The very first step in the design of the tool, was to determine its functional components. Since the tool is designed to be based on an arbitrary design conformance assessment technique, it is built from components that are exchangeable. Figure 5.1 depicts an overview of the various components, which are individually described in the following sections.

5.1.1 Specification Language

The specification language defines which software design constructs can be modeled as the 'intended design'. Depending on the level of expressiveness that is required, a specification language should be chosen or developed.

The required expressiveness for the prototype tool was determined by studying the software projects at Cannibal Game Studios (see Chapter 4). A taxonomy of design constructs found in these projects was created, from which the most common constructs were selected (see Section 4.2.1). The design specification language for the prototype has been designed to be able to express these particular constructs (see Section 5.2).

The language is implemented in the form of a parser that loads the intended design model into a meta-model.

5.1.2 Design Meta-Model

The design specification parser will read the intended design model from disk and store this in memory as the design meta-model. The design meta-model is dependent on the design

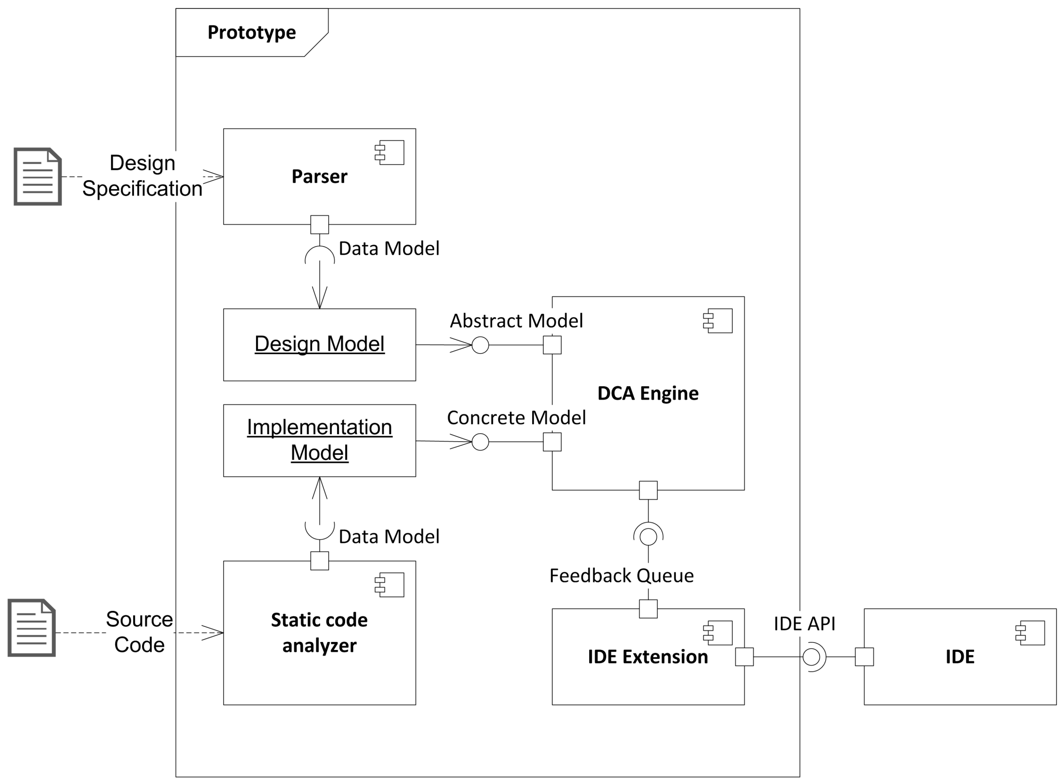


Figure 5.1: Overview of the components.

specification language, since it must be at least as expressive.

The design meta-model in the prototype tool hierarchically defines the *design*, *pattern*, *rule*, *entity* and *mapping* elements. A design can contain patterns which are collections of rules. A rule, in turn, represents a relationship between two entities, which are mapped to a part of the implementation. Each rule also defines a *semantic* that defines the type of the relationship, (e.g., containment, extension, implementation, calls, etc.).

5.1.3 Implementation Model

The actual implementation will be represented by its own model. This model is dependent on the programming language, as it describes all language-specific entities (e.g., namespaces, classes, methods, etc.) and their relations. Together with the design meta-model, the implementation model will serve as input to the DCA engine.

The prototype will use a code model that represents C# programs and should be capable of expressing programs written in most other object-oriented languages as well.

5.1.4 DCA Engine

The ‘design conformance assessment engine’ is the component responsible for comparing the design meta-model to the implementation model. The prototype implements this as a Reflexion Model comparison [32].

Since this component is functionally dependent on the underlying models, it might be necessary to adapt these models when implementing another approach to perform the model comparison.

5.1.5 IDE Extension

One of the requirements for proper process-integration of the tool is that it is integrated into the IDE. This means that all input and output must be routed through the extensibility API of the IDE.

For input, the tool will address the API to automatically search through the project files for any stored design-models. The execution of the DCA process will hook into the build-process of the IDE, so the analysis will occur every time the code is compiled. The feedback will be displayed in a tool window that is dockable and controllable with the IDE, furthermore, the results will be hyperlinked to project files to allow for easy navigation.

5.1.6 User Interface

The user interface is a crucial aspect of the prototype, since it is responsible for giving developers a succinct yet informative overview of the state of their project. The DCA process can potentially produce a lot of feedback. If this feedback is not presented in a manner that the developer can effectively consume, there will be no effect on design awareness as well.

5.2 Language Design

One of the most fundamental components is the design specification language. Besides a syntax and semantics, this also includes a parser.

5.2.1 Expressiveness

Much work has been done on design specification languages, also in the context of design conformance assessment specifically [22, 14, 42]. However, many languages intended for formal software design specification are either very specific or very complex. For that reason, the prototype will use its own specific design specification language that implements the level of expressiveness that is required.

The required level of expressiveness is defined by the programs that will be modeled. More specifically, the programs that will be modeled implement a set of structural and dynamic design constructs. Since this prototype is intended to check static design conformance, we want to know which static design constructs will need to be modeled. To that end, three representative development projects at Cannibal Game Studios have been selected. Based on these projects, a taxonomy of design constructs was created. This taxonomy classifies the occurring design constructs by structure and frequency. Finally, the most frequently occurring static design constructs have been taken as the minimally expressible constructs for the language.

When generalized, the most occurring constructs were found to be compositions of the following types of relationships *method calls*, *entity containment*, *class extension* and *interface implementation*. Moreover, all of the these basic relations exist between exactly two entities, while more complex constructs can be composed from the basic relations.

5.2.2 Syntax

The syntax definitions has two aspects; an abstract part and a concrete part.

Abstract syntax

The abstract syntax is the conceptual language definition. This is defined as follows:

$$\begin{aligned}
design &\rightarrow patterns\\
patterns &\rightarrow patterns|pattern\\
pattern &\rightarrow rules\\
rules &\rightarrow rules|rule\\
rule &\rightarrow (source, target, semantic, constraint, direction)\\
source &\rightarrow entities\\
target &\rightarrow entities\\
entities &\rightarrow entities|entity\\
entity &\rightarrow (type, < mapping >)\\
type &\rightarrow \text{“}Namespace\text{”}\\
&\quad |\text{“}Class\text{”}\\
&\quad |\text{“}Struct\text{”}\\
&\quad |\text{“}Interface\text{”}\\
&\quad |\text{“}Field\text{”}\\
&\quad |\text{“}Getter\text{”}\\
&\quad |\text{“}Setter\text{”}\\
&\quad |\text{“}Method\text{”}\\
semantic &\rightarrow \text{“}Call\text{”}\\
&\quad |\text{“}Containment\text{”}\\
&\quad |\text{“}Extension\text{”}\\
&\quad |\text{“}Implementation\text{”}
\end{aligned}$$

$$\begin{aligned} constraint \rightarrow & \text{“}Always\text{”} \\ & |\text{“}Never\text{”} \\ & |\text{“}AtLeastOnce\text{”} \\ direction \rightarrow & \text{“}OneWay\text{”} \\ & |\text{“}TwoWay\text{”} \end{aligned}$$

Note that the $< mapping >$ element represents the fully qualified name of the entity in the source code that is being mapped. The above definition can be summarized as follows:

- A *design* contains 1 or more patterns.
- A *pattern* is a composition of 1 or more rules.
- A *rule* represents a basic design construct, which defines a relation between two (groups of) entities. An entity represents an object-oriented structure; e.g., a *namespace*, *class*, *method*, etc.
- A rule also defines a *semantic*, a *constraint* and a *direction*, which describe the relationship.

The *rule* is the most basic design construct that can be defined, which in turn can be aggregated into *patterns* and eventually a *design*. A rule represents a relationship between two sets of static entities; while the nature of this relationship is defined by the *semantic*, *constraint* and *direction* of the rule.

The semantic defines what *kind* of relationship is being described; for example a *method call*, *entity containment*, *class extension* or *interface implementation*. These semantics essentially describe the basic design constructs that were identified in Section 4.2.1.

Note that not each semantic makes sense given each two entities, for example; a ‘field’ entity cannot be related to another ‘field’ entity with the ‘containment’ semantic. Although such relations can validly be expressed, they have no meaning and should be ignored by the DCA engine.

Since the programming language defines the structural entities as strictly hierarchical, it is valid to define a containment relation between entities that cannot be contained directly; for example, indirect containment of a method in a namespace.

Furthermore, the *constraint* on a rule describes when a rule is expected to be satisfied. Possible constraints are: *always*, *never* or *at least once*. These constraints can, for example, be used to model negated rules.

Finally, the direction of a rule can be either *one-way* or *two-way*, where a two-way rule is equivalent to a pair one-way rules with inversed source and target entities.

Concrete syntax

The experimental nature of the prototype has led to the choice of using the extensible markup language (XML) syntax to define designs, patterns, rules and entity mappings. The concrete syntax is structured as shown in the following example:

```
<Design>
  <Pattern>
    <Rule Constraint="Never" Semantic="Extend" Direction="OneWay">
      <Source>
        <Entity Type="Class" Mapping="*"/>
        <Exceptions>
          <Entity Type="Class" Mapping="App.UI.*"/>
        </Exceptions>
      </Source>
      <Target>
        <Entity Type="Class" Mapping="App.UI.Control"/>
      </Target>
    </Rule>
  </Pattern>
</Design>
```

The user can specify constraints, semantics and entity mappings to model the design. The example above models the case where no class may ever extend the `App.UI.Control` class, *except* for classes in the `App.UI` namespace.

Note that the 'source' and 'target' are abstract entities which can be defined as one or more concrete entity mappings, which are based on the fully qualified name of program entities. The entity mapping can either refer to a single concrete entity in the source code, or use wildcards (using an asterisk) to define a generic range of entities. When a wildcard is used, it is possible to define exceptions to the generic mapping.

The following example shows several ways to refer to a class-entity; all of the entity mappings refer to `Program.Objects.ClassA`, although the generic mappings may also contain other entities.

```
<Entity Type="Class" Mapping="Program.Objects.ClassA"/>
<Entity Type="Class" Mapping="Program.Objects.*"/>
<Entity Type="Class" Mapping="*.Class*"/>
<Entity Type="Class" Mapping="*"/>
```

This generic naming scheme makes the mapping of design entities to the source code more flexible, since it is up to the designer to perform this mapping based on either a naming convention or containment in a namespace, or to simply map individual entities by their full name.

In some situations it might be more convenient to exclude a small number of entities from a generic mapping, than it is to define a large number of concrete mappings. A set of exceptions can be defined over either the source or target set; the resulting set of entities is the difference between the two defined sets. The following example shows the definition of such an exception:

```
<Source>
  <Entity Type="Class" Mapping="*"/>
  <Exceptions>
    <Entity Type="Class" Mapping="App.UI.*"/>
  </Exceptions>
</Source>
```

Also in the more general case the XML format has advantages. Particularly, its easy extensibility allows the abstract design model to be modified without making extensive changes to the parser, since this can simply use ‘off-the-shelf’ XML libraries.

5.2.3 Model Definition

Model definition is an important part of the DCA workflow, however, it is out of scope for this thesis. For the current purposes it will suffice to allow the model definition to be done by directly editing the XML definitions. However, the editing of designs might be done more efficiently when done using a specialized front-end application.

5.2.4 Comparison with the RM Approach

Although the *SharpDCA* prototype will be based on the Reflexion Models by Murphy et al. [32] and the hierarchical extension by Koschke [25], our approach will differ in the way that the intended design model can be specified. Where the RM/HRM approaches are only able to express the existence of one specific type of relationship per model, we explicitly allow the *absence* of a relationship to be modeled as well. Moreover, the models in our approach are not limited to a single type of relationships (e.g., *only* calls or *only* inheritance). This means that designers can specify much more comprehensive views on the design.

5.3 Model Design

The two internal design models form the input for the design conformance assessment algorithm. Good design of the model can make the comparison much easier. A single model is used for both the design meta-model and the implementation model (Figure 5.2). This model implements a comparison method in the base class which is the main hook used by the DCA engine.

Inheriting from the model base class, are two subclasses that are used to represent concrete entities and abstract entities respectively. In the intended design meta-model, entity mappings are simply stored as the fully qualified name of the entity. In the implementation model, however, each entity is represented by a specific class containing the properties of that entity-type.

In practice, this model also contains external components, as is the case for the prototype. For example, the prototype uses the implementation of the concrete model provided by the component performing static analysis. This does require an extra abstraction layer which queries the external component, before the this data is translated to the internal model.

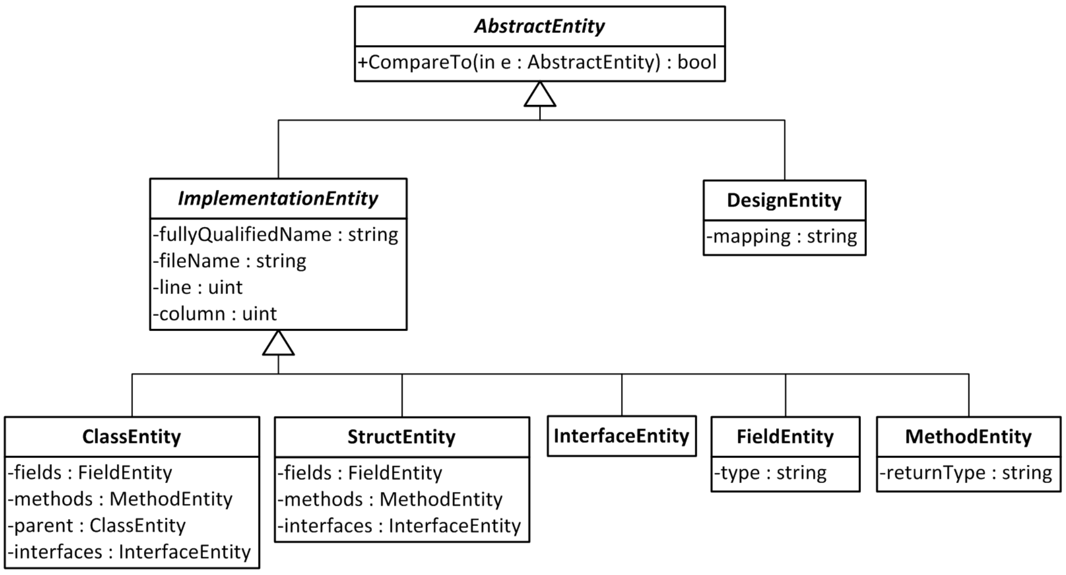


Figure 5.2: Data model used for storing both the intended design and implementation model.

5.4 Reflexion Engine

The 'Reflexion engine' is the component responsible for testing whether a particular semantic is realized given a source and a target entity, it is the realization of the DCA engine (as described in Section 5.1.4) using the Reflexion Model technique of Murphy et al. [32]. As can be seen in Figure 5.1, this component takes two models as input. The reflexion engine performs a comparison of the two models by verifying whether all entities and relations that are defined in one model, are also defined in the other. If entities or relations cannot be found in both models (i.e., design inconsistencies exist), the engine can raise a consistency violation which is displayed by the user interface.

The reflexion engine, in the general case, maintains 3 sets of relations (i.e., the *rules* defined in the intended design-model) that characterize the conformance between the two models, according to the original approach by Murphy et al. [32]; *convergences*, *divergences* and *absences*.

However, in this specific case, divergences are ignored. The reason for this is that divergences only provide useful information if the intended design is modeled for the *entire* system. If this is not done, all unmodeled parts will be detected as a divergences. In a real-world scenario, design patterns might be intertwined, legacy systems might be undocumented or systems might be too complex to fully model. This would result in a large amount of false-positive violation reports. Instead, we allow designers to model when an entity or relationship is *expected to be absent* (using a negation on the rule definition); i.e., the cases in which divergence is explicitly not permitted.

The computation of the inconsistencies is based on the original approach of Murphy et al., with the extension for hierarchical models by Koschke [25]. Algorithm 1 shows this computation.

Algorithm 1 Hierarchical Reflexion Models

Let $ref(a,b)$ denote the existence of a relationship between entities a and b in the implementation model ($M_{implementation}$) and $ref(A,B)$ the existence of a relationship between entities A and B in the intended design model (M_{design}). Furthermore, let $mapped(A,a)$ denote the existence of a mapping between entity $A \in M_{design}$ to entity $a \in M_{implementation}$. Then the the sets of entities resulting from the comparison are defined as:

$$\begin{aligned}
ref'(A,B) &\iff \exists(a,b \in M_{implementation}) : (ref(a,b) \\
&\quad \wedge mapped(A' \in A, a) \\
&\quad \wedge mapped(B' \in B, b)) \\
convergence(A,B) &\iff ref(A,B) \wedge ref'(A,B) \\
absence(A,B) &\iff ref(A,B) \wedge \neg ref'(A,B)
\end{aligned}$$

However, in the original form, this approach is not able to handle the model definitions that are expressible using the language presented in Section 5.2. Since a model may contain relations with different semantics, and constraints on the satisfaction of rules, the DCA engine should take this into account. The algorithm used by *SharpDCA* is shown in Algorithm 2.

In practice, the two inputs for the DCA engine are not directly comparable, since the ‘intended design’ model is provided in the form of the design rules and the ‘actual design’ model is a static model of the implementation. This means that extra logic has to be performed by the reflexion engine. The reflexion engine takes the rule definition as input and, for each rule, searches the implementation model for realizations of the entities defined in that rule. If both the source and target entities are found in the model of the implementation, it checks whether the intended relationship between the two is realized. If this is not the case, or either an entity is missing, an inconsistency is detected.

In the case of a detected inconsistency, the location of the entities is stored as well as the nature of the inconsistency. This information is then later used by the UI component to format a human-readable feedback message.

5.5 Tool Implementation

The prototype is built for a very specific environment, which is reflected in its implementation. Specifically, the tool is built to be used on C# projects, in conjunction with the Microsoft Visual Studio 2010 environment.

This section discusses the particular implementation choice that have been made, and their motivations.

Algorithm 2 SharpDCA: DCA Engine

Let S denote the set of semantics for a relationship between two program entities, as described in Section 5.2.2, let $s \in S$ be the intended semantic and c the constraint of a relationship as specified in (M_{design}). Furthermore, let $semanticOf(ref(a,b))$ denote the semantic of the relationship between the two entities $a, b \in M_{implementation})$.

$$\begin{aligned}
ref_{Always}(A,B,s) &\iff \forall(a,b \in M_{implementation}) : (ref(a,b)\\
&\quad \wedge mapped(A' \in A, a)\\
&\quad \wedge mapped(B' \in B, b)\\
&\quad \wedge semanticOf(ref(a,b)) = s)\\
ref_{Never}(A,B,s) &\iff \nexists(a,b \in M_{implementation}) : (ref(a,b)\\
&\quad \wedge mapped(A' \in A, a)\\
&\quad \wedge mapped(B' \in B, b)\\
&\quad \wedge semanticOf(ref(a,b)) = s)\\
ref_{AtLeastOnce}(A,B,s) &\iff \exists(a,b \in M_{implementation}) : (ref(a,b)\\
&\quad \wedge mapped(A' \in A, a)\\
&\quad \wedge mapped(B' \in B, b)\\
&\quad \wedge semanticOf(ref(a,b)) = s)
\end{aligned}$$

$$\begin{aligned}
convergence(A,B,s,c) &\iff ref(A,B,s) \wedge ref_c(A,B,s)\\
absence(A,B,s,c) &\iff ref(A,B,s) \wedge \neg ref_c(A,B,s)
\end{aligned}$$

5.5.1 External Components

There are two major external dependencies; an API for the static analysis of the C# code and an API for the extension of the IDE.

FxCop API

Microsoft FxCop [31] is a tool for the static analysis of .NET languages, including C#. It provides an API which provides a static model of the program.

The FxCop API does not directly use the C# program as input, instead it operates on the object code (Common Intermediate Language, or CIL) produced by the .NET compiler. The advantage of this is that the static analysis is actually compatible with other .NET languages such as C++/CLI, F#, Visual Basic .NET and even some more exotic ones [1]. The main disadvantage of this, is that some language specific implementation features are lost, most notably the traceability of constructs to specific lines and columns is not remembered.

More specifically, *dynamic* constructs actually *are* traceable through debugging metadata files generated by the compiler. The problem is that all traceability to the *static* constructs is discarded. A workaround is implemented, which queries the IDE by the unique identifier of program constructs, which in turn can return the exact file and line number of the definition.

This issue was only encountered during an advanced stage of the implementation of the prototype; so in retrospect, the preferable choice would be to construct a static model internally solely using the Visual Studio Extensibility API. The advantage of this is that it operates directly on the source files, so traceability is not lost and it would no longer be required to rebuild the assemblies before performing the design conformance assessment.

Visual Studio Extensibility API

The Visual Studio Extensibility API is used for extending the Microsoft Visual Studio IDE. The API offers hooks into IDE services such as the compiler, extension of the graphical user interface and access to the loaded project files – including a language specific static code model.

The DCA functionality is invoked through this API, by hooking into the build-event. Feedback is then passed back to the API and shown in an IDE window. To provide traceability of the feedback messages, these are linked to the code model and hyperlinked to the corresponding files.

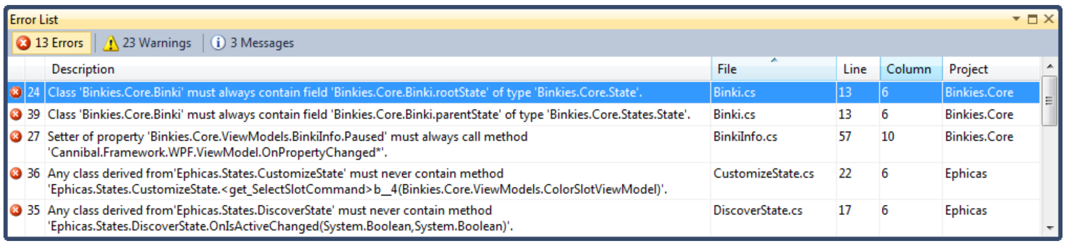
As noted above, the Visual Studio code model is queried for static entity locations. This is one of the more expensive operations that have to be performed, given the potentially huge amount of program entities. In order to minimize this overhead, a table is constructed that maps each program entity identifier to a location in the source code. This reduces the search to a simple lookup.

5.6 DCA Feedback Dialogs

Since good integration with the existing tool suite is important, the interaction through the IDE should be as seamless as possible. For this reason, a first version of the prototype integrated any notifications of design inconsistencies in the IDE error-dialog (see Figure 5.3), together with compilation errors and warnings. Just as with compilation errors, the inconsistency is displayed as human-readable presentation of the rule, and information on the file and line where it has occurred. User testing showed that, even though developers liked that this gave the feedback a sense of 'severity', this made it hard for users to maintain an overview of the rules that have been defined.

After the first round of user testing, a second 'overview dialog' was added (see Figure 5.4), which only displayed the rules and whether they are satisfied or not. The concrete inconsistencies were still shown in the error dialog. Furthermore, the source, target, constraint and semantic are displayed in separate columns to give developers a better overview of the rules that are being checked. A second round of user testing confirmed that developers found it much easier to maintain an overview of the errors and their origin like this.

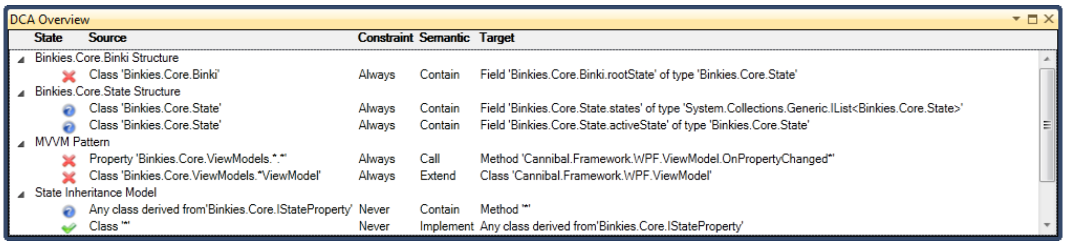
As an example, the topmost entry in Figure 5.4 shows a single rule that was defined. In this case, the source entity *must always* contain the target entity, as shown in the constraint and semantic columns. The source is the class `Binkies.Core.Binki` and the target is


Error List

13 Errors | 23 Warnings | 3 Messages

		Description	File	Line	Column	Project
✖	24	Class 'Binkies.Core.Binki' must always contain field 'Binkies.Core.Binki.rootState' of type 'Binkies.Core.State'.	Binki.cs	13	6	Binkies.Core
✖	39	Class 'Binkies.Core.Binki' must always contain field 'Binkies.Core.Binki.parentState' of type 'Binkies.Core.States.State'.	Binki.cs	13	6	Binkies.Core
✖	27	Setter of property 'Binkies.Core.ViewModels.BinkiInfo.Paused' must always call method 'Cannibal.Framework.WPF.ViewModel.OnPropertyChanged*'.	BinkiInfo.cs	57	10	Binkies.Core
✖	36	Any class derived from'Ephicas.States.CustomizeState' must never contain method 'Ephicas.States.CustomizeState.<get_SelectSlotCommand>b__4(Binkies.Core.ViewModels.ColorSlotViewModel)'.	CustomizeState.cs	22	6	Ephicas
✖	35	Any class derived from'Ephicas.States.DiscoverState' must never contain method 'Ephicas.States.DiscoverState.OnIsActiveChanged(System.Boolean,System.Boolean)'.	DiscoverState.cs	17	6	Ephicas

Figure 5.3: Error dialog.


DCA Overview

State	Source	Constraint	Semantic	Target
Binkies.Core.Binki Structure				
✖	Class 'Binkies.Core.Binki'	Always	Contain	Field 'Binkies.Core.Binki.rootState' of type 'Binkies.Core.State'
Binkies.Core.State Structure				
?	Class 'Binkies.Core.State'	Always	Contain	Field 'Binkies.Core.State.states' of type 'System.Collections.Generic.IList<Binkies.Core.State>'
?	Class 'Binkies.Core.State'	Always	Contain	Field 'Binkies.Core.State.activeState' of type 'Binkies.Core.State'
MVVM Pattern				
✖	Property 'Binkies.Core.ViewModels.*.*'	Always	Call	Method 'Cannibal.Framework.WPF.ViewModel.OnPropertyChanged*'
✖	Class 'Binkies.Core.ViewModels.*ViewModel'	Always	Extend	Class 'Cannibal.Framework.WPF.ViewModel'
State Inheritance Model				
?	Any class derived from'Binkies.Core.IStateProperty'	Never	Contain	Method '*'
✓	Class '*'	Never	Implement	Any class derived from'Binkies.Core.IStateProperty'

Figure 5.4: Overview dialog.

shown to be a field called `rootState` of type `Binkies.Core.State`. The red cross icon indicates that there is at least one instance in the implementation where the rule is violated.

Since the rule does not contain wildcards, there is only one realization of the rule, which is shown as topmost entry in Figure 5.3. The entry shows (from left to right) the sequence number of the error (generated by the IDE), the description of the violated rule (the source class was expected to always contain a field of the target type), the file in which the violation occurred together with the line number and column, and finally the name of the project containing the file.

5.7 Summary

The tool prototype is designed in a modular way, such that individual components are exchangeable for other implementation. This is particularly useful if the tool is later extended to use other DCA techniques or design specification languages. However, certain trade-offs were made particular for prototype design. While this is beneficial for the evaluation in the context of this thesis, they might not be optimal in all cases.

Chapter 6

Evaluation

The initial research question asked how the integration of design conformance assessment (DCA) techniques, into existing development practices, could be done viably in a business environment. The main topic of discussion up to this point has been the design and implementation of a tool that can automatically perform the assessment and communicate feedback through the familiar interface of the IDE.

The following chapters will describe the evaluation study of the implemented tool. However, before discussing the study itself, this chapter discusses which particular aspects of the tool should be evaluated, and why.

The following sections outline what effects *can* be measured (6.1), what effects are relevant in the current context (6.2) and the methodology that has been used to evaluate these effects (6.3).

6.1 Measurable Effects

In order to formulate clear goals for the evaluation, it is important to understand the possible effects that design conformance assessment can have on the development process. Particularly, one can see that the tool affects the development process from two distinct viewpoints.

From the viewpoint of the *software designer* it can assist communication by expressing design structures more explicitly and provide increased awareness of design problems by creating a shorter feedback loop between designers and programmers. From the *programmer* point-of-view, it can facilitate an increased awareness of the consistency between code and design. Also, it can provide a shorter refactoring cycle for resolving design violations. It is assumed that an increased awareness of design violations directly reduces design erosion and, over time, reduces the impact of architectural drift.

In this thesis, we focus on the programmer point-of-view. Furthermore, it can be noted that the mode of design communication used by the tool may certainly affect *design understanding*. While design understanding is clearly related to design erosion and design conformance, it is a very broad subject by itself. So even though the effects on design understanding are relevant, a dedicated study might be needed to fully explore these effects.

6.2 Evaluation Goals

There are three aspects of the tool that need to be evaluated. First of all, the design specification language should be expressive enough to define the required[1] design constructs. Secondly, the actual design conformance assessment output should be tested for correctness. And finally, it should be evaluated whether the feedback from the tool does actually help developers in managing design violations, without hampering their process. This latter aspect of the evaluation is used as measure of how well the presented approach would assist the adoption of DCA in real-world projects (see Chapter 8).

6.3 Methodology

Each of the three aspects discussed in the previous section have been verified using different methodologies, which are discussed in this section.

Some of the underlying techniques of the tool are actually interchangeable with other existing techniques, e.g., Reflexion Models for comparing an implementation to an intended design model. This evaluation does not focus on these underlying techniques, since that has already been done in their respective works [32, 33, 25]. The focus of the current evaluation is on the correctness of the implementation as well as the interaction of the tool with the development process.

6.3.1 Design Specification Expressiveness

The required expressiveness is completely dependent on the design that should be represented and, hence, may be different per project. However, it can reasonably be assumed that there is a common subset of design elements that can be used to express a significant part of object-oriented structural designs. By sampling the applied design constructs from real-world projects, a representative subset can be obtained (given that the samples are acquired from a population that is diverse enough).

In this case, the samples have been obtained from a range of projects at Cannibal Game Studios and the design specification language is constructed to be able to express these common constructs. So when using the tool with these projects at Cannibal Game Studios, by construction, the design specification language is expressive enough. This does imply that the expressible constructs may be biased towards those that are common at Cannibal Game Studios.

When the goal is to develop a DCA tool that can be used for an arbitrary development project, the expressiveness becomes a key concern. This would mean treading the domain of formal design specification. Much work on this topic has already been done, such as [14], [12] and [11]. However, for the current evaluation we only need to express a limited set of design constructs, so the expressiveness is acceptable even if the design specification language is not complete in the general case.

[1]Depending on the specific project; for the purpose of this evaluation the requirements of an actual development project are used as an example.

6.3.2 Correctness of the Results

The correctness of the results can be determined by creating a number of unit-test cases to detect any false-positives. In total 25 test cases were defined to test rules containing all possible types of relations and entities. These unit-tests are defined to cover each of the scenarios that are part of the effectiveness study (see Section 7.2.3).

A false-positive is defined as a hit returned by the tool, while that hit was not expected to occur based on the semantics of the design specification language. This implicitly assumes correctness of the design specification language. Although the language is correct by construction in this specific case, in the more general case this would be a point of attention.

Note that there might be positive hits that do not actually represent violations; other types of false-positive hits may be due to intentional program constructs or can be caused by incorrect modeling of the design. These are not counted as false-positives produced by the tool, since they are the expected result of user-action.

Since the scope and semantics of possible inputs are clearly defined, no false-positives caused by implementation errors are expected. Any occurrence of such a false-positive should be repaired before user testing. Once all unit-tests pass, the correctness of the results (within the scope of the original requirements) is deemed acceptable to continue.

False-negatives are also an obvious concern, but unfortunately the only way to confidently exclude the possibility of their existence would be through the gathering of long-term statistical data. Although false-negatives are not of any concern for immediate purposes, their existence may pose a significant impact (i.e., creating a false sense of consistency). The effectiveness study (Section 6.3.3) attempts to gain qualitative insights into the effects of false-negatives.

6.3.3 Effectiveness in a Production Environment

To evaluate the effectiveness of the tool, a pre-experimental study has been conducted. The goal of the study is to gain better insight into the ways that a developer would use the tool in a real-world environment. It specifically aims to qualify an increase in awareness of- and ability to- resolve inconsistencies between the intended design and its implementation. Besides evaluating any positive effects, the experiment is also used to evaluate the possible dangers, such as the earlier mentioned false sense of security caused by over reliance on the tool.

The format used for the study is a ‘one-group pretest posttest’ approach, followed by a semi-structured interview to gather in-depth feedback from the participants. Specifically, the *treatment* is a session of guided user-testing. Two questionnaires are provided to participants, respectively one before and one after the treatment to obtain a baseline and preliminarily quantify their change in attitude after using the tool. Details on the design and execution of the study can be found in Chapter 7.

6.4 Summary

In the general case, three aspects of the tool should be tested; the expressiveness of the design specification language, the correctness of the conformance checks and whether the tool is able to perform its function as part of the actual development process. Since this study primarily focuses on the design conformance assessment *process*, the evaluation focuses on the interactions with the development process. The next chapter continues with a description of how the evaluation study is set up.

Chapter 7

Study Design

In order to evaluate whether the presented approach of process-integrated design conformance assessment is effective, there are two questions that are of particular interest;

- *R.Q. 1: Are developers more aware of inconsistencies, when using the SharpDCA tool?*
- *R.Q. 2: Are developers better capable of effectively managing inconsistencies, when using the SharpDCA tool?*

As such, answering these questions is the main goal of the evaluation.

The answer to these questions also helps to understand the interactions within the larger problem-context. Specifically, since it is assumed that an increase of design conformance awareness will directly lead to reduced design erosion, the results should indicate whether the DCA tool can help to prevent design erosion. Moreover, the study also attempts to provide insights into the exact mechanisms that may help to increase design awareness.

A pre-experimental study of the tool, seems appropriate given the exploratory nature of the research. Such a study is able to provide some preliminary answers to the aforementioned questions. The study uses a pretest questionnaire to obtain a baseline measurement, which is compared to a corresponding posttest questionnaire as to measure the effect of the tool. Moreover, an important source of data is the semi-structured interview, which provides a more in-depth context that is used to interpret the results.

7.1 Goals

The evaluation goals, as discussed in the previous chapter, have been the main focus when designing the study. There are myriad effects of design conformance assessment; but this study is specifically concerned with the design-consistency awareness among programmers and the ability of programmers to act on a violation once it is detected.

Effects that concern design specification, as well as long-term effects, are explicitly not part of this evaluation.

7.2 Study Overview

The main part of the study consists of 'guided user-testing', which in essence is a trial of the tool in a real-world setting. While the user is working, any actions are observed so the underlying motivations and considerations could be discussed in an interview directly following the user-testing. The evaluation sessions are structured as follows:

1. Pretest questionnaire. (10 min.)
2. Demo and explanation of the tool (10 min.)
3. Exercises for the user. (30 min.)
4. Posttest questionnaire. (10 min.)
5. Semi-structured interview. (30 min.)

7.2.1 Pretest Questionnaire

The pretest questionnaire serves to provide a baseline measurement of the extent to which the participants feel able to detect and resolve design inconsistencies. This data is based on the tacit knowledge and experience of the user, and the questionnaire is intended to explicate this data in a measurable way.

Additionally, each participant is presented several questions intended to indicate how motivated they are to use design conformance assessment techniques in the first place. This data can be used to validate that a programmer indeed does have the required level of experience for the evaluation.

The full pretest questionnaire can be found in Appendix B.

7.2.2 Tool Explanation and Demonstration

The programmer is given a short introduction on the purpose of the tool and all its functions are be demonstrated, as to provide some familiarity with all the available actions and dialog windows. No information is given about any intended workflows for the tool, to prevent biasing the user to a specific way of performing the exercises.

7.2.3 User Exercises

After the participant has been briefed on the usage of the tool, they are asked to perform a couple of programming tasks. Specifically, they are presented with a number of design inconsistencies and are asked to resolve them as they would normally. They are told that they can use any feature of the tool or their IDE that they would like, but should work in a way that they feel comfortable with. Furthermore, participants are made clear that they do not have to concern themselves with modifying the design model; if they feel that a task cannot be completed without changing the design or consulting a designer they can state so an continue with the next exercise.

During the session, it is observed how they handle a number of situations; specifically, the following cases are be presented to the user:

- Trivial inconsistencies (e.g., wrong field name, wrong field type, missing call).
- False positives due to incorrect modeling of the intended design (e.g., expecting a call that is not actually needed).
- Artificially introduced false negatives (e.g., a design pattern which is obviously incomplete, while the conformance checker does not detect this).
- Unclear design rationale (e.g., a rule which does not seem to contribute to any design pattern).
- Large amounts of feedback data (e.g., a single generic rule that produces 50+ concrete inconsistencies).

These cases are realized using one or multiple design violations that are detected by the tool in a test project.

The observation during the evaluation of the prototype was done by the author, while sitting next to the participant and recording their comments and actions. This approach was chosen over recording the actions, so the actions of the participant could be discussed immediately in the following interview.

Test Project

For this thesis, two separate test-projects have been used for two different groups of participants. This is because external participants are not allowed to work on proprietary code due to non-disclosure requirements, so a separate test project was created. The reason that testing was still performed on the proprietary project as well, is that internal participants have a very high level of familiarity with the code that is hard to create in any artificial scenario. The difference between the two groups, in turn, indicates the effects of pre existing design knowledge.

Although different projects have been used, they have been modified as to produce a similar set of inconsistencies, thus keeping the evaluation consistent for both groups of participants.

7.2.4 Posttest Questionnaire

Directly after performing the programming tasks, the participant are asked to fill out a posttest questionnaire. The questions map almost directly to the ones in the pretest questionnaire, but are formulated in a slightly different way that puts the focus on the experience with the tool instead of general experience with software development.

7.2.5 Interview

The interview allows the participants to give some in-depth feedback on how the they perceive the usefulness of the tool. Besides focusing on how the tool influences their ability to detect and resolve design inconsistencies, one part is specifically dedicated to their personal motivation for design conformance assessment. This allows the other answers to be interpreted in the context of their personal experience.

The interview structure can be found in appendix A.

7.3 Participant Selection

In total, 8 participants were selected for the evaluation. There were 4 participants in the group of internal testers, that have a high level of familiarity with the code and design. The other 4 participants have been assigned to another group that performed the same exercises with significantly less prior knowledge about the design.

The participants are be required to have a basic understanding of the Microsoft Visual Studio 2010 IDE and the C# programming language. Furthermore, all participants should have a basic grasp of the concepts *design erosion* and *architectural drift*. Ideally the participants have experience developing software in a production environment, although this is not a strict requirement.

7.4 Pilot Study

The performed study was based on the results of an internal pilot study, where two programmers have been given the tool for user testing. The pilot study was intended to test the stability of the tool and fine-tune the structure of the user-exercises. During the pilot sessions, users were asked to perform exercises in a similar way as described in the previous section. However, the objective in this case was to evaluate the usability of the tool and whether users understood what was expected of them. There were two iterations of internal pilot sessions and consequent changes to the tool. In both rounds of testing, feedback concerned changes to the user interface to make the tool easier to use. In earlier tests, users found it hard to keep an overview given the dense feedback information, so the presentation was changed to improve this.

7.5 Summary

The study is designed to observe what the effects of the DCA tool are, when used by programmers in a real-world environment. The main focus is on finding out if the tool increases awareness of design inconsistencies and makes resolving them easier, these are the primary effects that have been evaluated.

Some secondary effects of the tool have been evaluated as well. The two groups of participants were used to test how prior design knowledge influences how the tool is used.

Furthermore, the study attempts to find out if the tool might induce a false sense of security in programmers, if there are false negatives.

Chapter 8

Results

The study has yielded interesting results, in particular about the means by which the tool improves design awareness. While it should be noted that the study results are still pre-experimental in nature, they have provided positive evidence that the tested approach of a process-integrated tool is indeed an effective way to introduce design conformance assessment (DCA) techniques into real-world software development projects.

8.1 Overview

The study has focused on answering the main research questions; *are developers more aware of inconsistencies, when using the SharpDCA tool* (R.Q. 1) and *are developers better capable of effectively managing inconsistencies, when using the SharpDCA tool* (R.Q. 2)? This section will give an overview of the findings.

8.1.1 Initial Expectations versus Results

The initial expectation was that the tool would directly influence awareness of design inconsistencies by explicitly pointing out the existence and location of such inconsistencies. Furthermore the belief was that the tight IDE integration would help developers in efficiently managing known inconsistencies by providing hints and tools, which in turn help to reduce the time needed to resolve individual inconsistencies.

The results of the study have shown positive evidence for both expectations. The exact mechanism by which these expectations were realized, however, is much more subtle than was assumed. Specifically, not only the awareness of design inconsistencies but also the awareness of the overall design increased. The tool did not accomplish this directly, as was expected, but indirectly by influencing the dynamics of the development process. And in the case of managing inconsistencies; participants exhibited vastly different behaviour when trying to resolve known inconsistencies, depending on their level of familiarity with the project. The group with a high level of familiarity ignored most hints as how to resolve an inconsistency, whereas the participants with only little knowledge of the code and intended design were very reliant on the feedback and actually tried to infer their design understanding from the tool.

8.1.2 Effects on Awareness

Participants of the study have unanimously indicated that tool-based design conformance assessment provided an increased level of awareness of design inconsistencies. After performing the exercises, 75% of participants from the group with high design-familiarity indicated that they have encountered design violations that they were not previously aware of. The programmers generally reckon that it is inevitable to overlook inconsistencies without the support of an automated DCA tool. This is primarily due to the fact that projects often are too large to viably perform continuous integration testing of design consistency by means of human code review.

The fact that the tool was integrated into the IDE, and checks are automatically executed at build-time, requires developers to make an explicit decision in the case of a detected inconsistency. The participants found this to have a positive effect on their workflow, since they are able to produce better code faster. Interestingly, none of them said to feel constrained by a tool that requires them to think about design decisions much more frequently.

This small change to the development process appears to be the main mode by which the tool helps to prevent design erosion; design conformance feedback is provided explicitly and in-line with the existing development process, which continuously forces the developer to think and communicate about the desired design. Contrary to the initial expectation, the study has indicated that the main mechanism of increased awareness is not the mere feedback from the tool itself, but the fact that it stimulates (possibly even forces) developers to think and communicate about the design more frequently. The effect of this is even stronger than expected, since this does not only increase awareness of localized design-inconsistencies, but increases awareness of the overall design as well.

Moreover, 63% of interviewees expressed a positive attitude to a tool that does not allow the execution of a program that contains unresolved design inconsistencies. Still, they do note that in specific cases the interference of a design conformance checker would likely be too constraining, e.g., when writing prototype code. The participants with more software design experience were inclined to favor the mandatory resolving of all inconsistencies, whereas others favored a more optional approach.

In general, developers liked the tool to be enabled by default and execute as often as possible, i.e., update at build time or even update while writing code, but with an option to temporarily disable this functionality that is ‘not too easily accessible’. The latter is because developers do require the flexibility of writing ‘ugly’ code at times, but want to avoid the temptation when writing production code.

8.1.3 Effects on Resolving Inconsistencies

How participants chose to handle inconsistent design once they are aware of it, and how they are influenced by the tool, greatly depended on their prior knowledge of the design. In the case where developers were already familiar with the design of a project, the tool did not noticeably affect their way of resolving inconsistencies.

In the other case however, where the developer did only have a general understanding of the design, they tended to interpret the feedback from the tool in a much more rigid manner. In this latter case, participants were less focused on making a motivated design choice as much as trying to figure out *why* the inconsistency exists and how to 'make it go away'. Given little pre-existing knowledge of the rationale behind design rules, it was to be expected that a developer may not have clear direction in mind when resolving design inconsistencies. Interestingly, however, this group of participants often went looking if the tool could provide more hints as to what this rationale was – even as far as searching for comments in the XML files that specified the design rules, even though it has been made clear that the design definitions were not the focus of the exercise.

This might hint at a potential role such a tool might have in the communication of design rationale. Indeed, in the individual interviews participants of both groups have indicated that they would like to see the tool provide insights into the rationale behind the rules.

8.2 User Perceptions

Besides simply asking participants to rate the value of the tool on a subjective scale, an additional questionnaire was given to provide a baseline measurement for their attitude towards design conformance assessment. A second questionnaire was then provided afterwards to measure any change of attitude towards the tool and design conformance assessment in general.

This section discusses the main results of the questionnaire. The complete results can be found in Appendix C.

Motivation to use design conformance assessment

The questions asking how 'compelled' the participant felt to applying DCA techniques were primarily aimed to validate that the person possessed an appropriate amount of experience with design conformance assessment. In particular, it was tested whether the participant understood the importance of maintaining design consistency. At the start of the study all

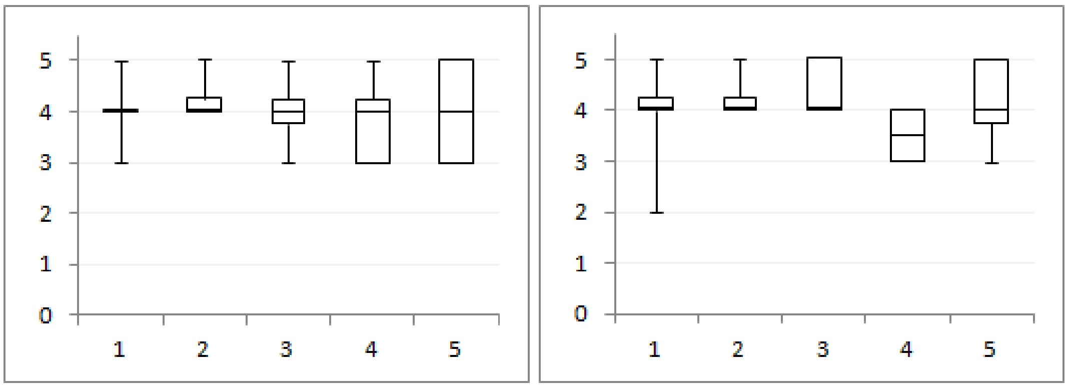


Figure 8.1: Questionnaire results 1-5 (left: pretest, right: posttest)

participants did indeed indicate to understand the importance of maintaining design-code consistency.

However, as can be seen in Figure 8.1, there still were some notable changes in attitude. Question 1, for example, asked how important the participant would rate *absolute* consistency between intended design and implementation. It appears that the participants were more doubtful about this after using the tool. It is possible that this is related to the realization that *perfect* consistency is not always a realistic expectation (see Section 8.2.1). The results of Question 3 indicate a significant increase in the likelihood of adopting a DCA tool, indicating that the participants agree with the usefulness of the prototype tool. Question 4 and 5 indicate that programmers were more inclined to take responsibility for design conformance than before. Similarly they were less inclined to 'stick it' to the designer.

8.2.1 Awareness of Design Inconsistencies

Participants generally indicated that finding inconsistencies becomes easier with the tool, as shown by the change in the results of Question 6 and 7 (see Figure 8.2).

Interestingly, the participants seem to have become slightly *less* confident of their ability to produce code that is free of design violations (Question 8). This is noteworthy because the expectation was that the use of a an automated tool might provide a false sense of security due to the possibility of false negatives. However, it might be possible that the increased awareness of specific design violations makes developers more conscious about the possibility of other problems as well.

Question 9 asked how the participants valued a DCA tool specifically for creating more *awareness* of inconsistencies. After the testing session, participants unanimously gave the highest possible rating to this question. Moreover, in one particular case the participant noted that the scale could not sufficiently express the value of such a tool, in his opinion.

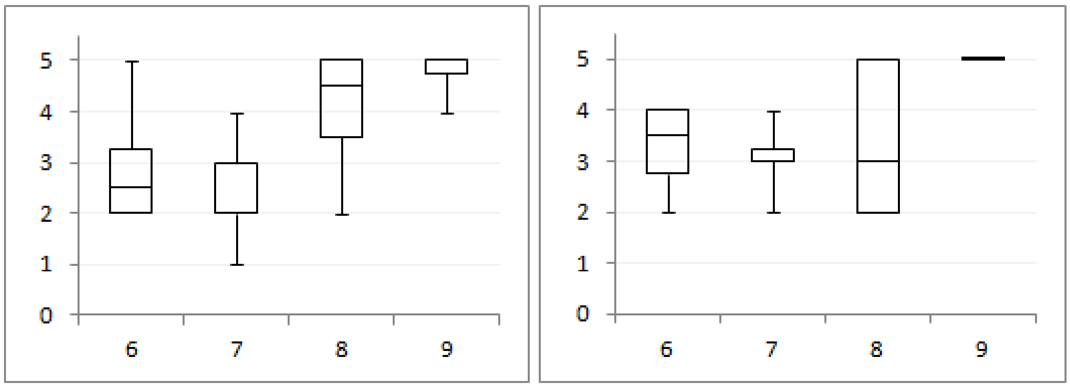


Figure 8.2: Questionnaire results 6-9 (left: pretest, right: posttest)

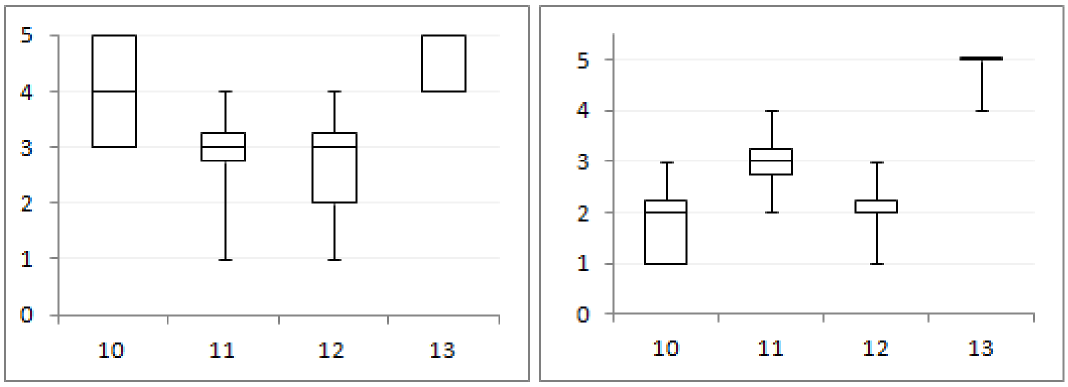


Figure 8.3: Questionnaire results 10-13 (left: pretest, right: posttest)

8.2.2 Resolving Inconsistencies

In line with the conducted interviews, the tool did affect the ease with which violations can be handled, as can been seen in Figure 8.3. The answers to Questions 10 through 12 indicate that the tool makes it easier to locate, understand and resolve inconsistencies.

However, it should be noted that this effect is mainly ‘mechanical’ in nature and is due to the ability to quickly navigate to relevant documents. This is opposed to helping to resolve inconsistencies by increasing the understanding of violation (also see Section 8.4.2).

Finally, as the results to Question 13 show, the participants value a tool that helps to trace design inconsistencies slightly more after using the prototype.

8.3 Interview

The interview was performed right after the posttest questionnaire. The questions of the interview overlap with the questions found in the questionnaire, as to allow participants to elaborate on their answers.

8.3.1 Motivation and Experience

All interviewees believed that design conformance assessment, be it automatic or manually, is an important activity in any ongoing development project. However, even though they agree that DCA should be performed *in theory*, not one of the participants indicated to actually do this in practice. One person noted ‘At one time we *did* change the process to include periodical code reviews to check design conformance, but it still was not done in practice’. Another person commented on this ‘Human code review did not work, because it is just too time consuming, tedious and, frankly, it is boring’.

The lack of design conformance assessment in practice implies that the initial design documentation is never updated as well. All interviewees confirm this, noting that the only time a design document can ever be considered up-to-date is at the start of implementation. One programmer added to this ‘We feel a lot of pressure to get the product functionally ready, so I would rather fix a problem in the code directly than changing the design first. Although, I do think that this perceived time-pressure among developers often is just an illusion'.

8.3.2 Awareness of Inconsistencies

75% participants from the internal group, with a high level of project-familiarity, indicated that the tool presented design inconsistencies that they were not yet aware of. Moreover, they agreed that these were actually violations that should be resolved. Even though a violation might be obvious, interviewees acknowledge that the large amount of code in a project means that they are unlikely to detect it manually.

8.3.3 Resolving Inconsistencies

The general consensus among interviewees was that resolving any inconsistency ultimately requires the judgement of a human. However, during the interviews a number of ways for the tool to assist were mentioned; *hyperlinking to design and code*, *explaining the design rationale of rules* and *communicating possible ways to resolve an inconsistency*.

8.3.4 Interaction with the Existing Process

When asked whether the tool constrains programmers *too* heavily, the response was negative in all cases. Interviewees feel that the tool is a valuable addition to their workflow, however, they differ in their opinion when asked how aggressively design consistency should be enforced. Three interviewees explicitly disliked the idea of a tool that would not allow program execution until all inconsistencies were resolved. The others were proponents of such an approach, to enforce consistency on production code, but while still allowing to disable the tool when working on prototype code. One person noted '...of course people should be able to disable the tool, but that option should not be *too* easily accessible, so that they will not use it all the time', while another said 'I think it is a good idea to enable DCA by default in the IDE and provide the option to temporarily disable it during the current session. If a programmer restarts the IDE, it should be enabled again'.

8.3.5 General Attitude Towards the Tool

Throughout the interview, participants were encouraged to elaborate on how they feel that the prototype could help them in real-world scenarios. They mentioned *increased software quality*, *lower maintenance effort* and *making it easier to communicate about the design* as the added values provided by the tool. The means by which these effects are achieved, according to interviewees, are that the tool *stimulates delivering neater work* and *forces developers to communicate more about the design*. One interviewee summarized this as '...the tool would cause a proliferation of design understanding, which means that everybody has the same knowledge and everybody is talking about the same thing'.

8.4 Observations

Besides the main focus points of the study, a number of other observations were made. Although these were not the intended focus of the study, they can be considered valuable nevertheless.

8.4.1 Design Specification

Although it was specifically mentioned that the user was not expected to analyze of modify the design specification files, many users did chose to do so nevertheless. Apparently the need of programmers to gain a higher level overview of the design has been underestimated. Indeed, maybe the distinction between programmers and software designers that has been assumed is too strict.

8.4.2 Communicating Design Rationale

When a user came across a part of the design that they were not familiar with, their tendency was to try understand it by themselves. Even though they were told that it was allowed to ask questions if they got stuck and that it was a valid option to indicate that they, in real-life, would take no immediate action before having consulted the responsible designer.

Users have noted, on several occasions, that they would have liked the tool to provide some form of rationale for rules that were non-trivial and were hard to understand on a first glance.

As noted in Section 8.2.2, providing the rationale behind a design rule (thus increasing understanding of the design), can help the user to better resolve inconsistencies.

8.4.3 Feedback Mode

Although the generated feedback messages contained quite a lot of information detailing the nature of the problem, most users completely ignored this information. Moreover, the information actually appeared *too* dense, since even when users attempted to analyze the messages in detail (even to the point where they were reading them out loud) they sometimes still failed to take note of important information.

In order to make sure the user can process the right information at the right time, it seems necessary to provide the feedback in a more context-sensitive and filtered format so no unnecessary information is shown and feedback is easily understandable at a glance.

8.4.4 Navigation

Unconsciously users take one-click navigation for granted, if it is possible in a particular context. In pilot tests the lack of integrated navigation to specific files by clicking on error messages was instantly noticed and deemed as a must-have by the users. User are so used to the hyperlinked nature of the IDE user interface, that generally did not accredit the tool for helping them work faster even when the functionality was added.

Moreover, given the tendency to also invoke the design specification files in the exercise, users also indicated that they would like to be given hyperlinks to these files. In the interviews, participants confirmed that the main mode by which a tool can help them resolve inconsistencies more efficiently are these type of 'shortcut' functionalities.

8.5 Research Questions

Given the results from the study that were presented in the previous sections, we are able to reflect back on the original research questions and formulate concrete answers.

R.Q. 1: Are developers more aware of inconsistencies, when using the SharpDCA tool?

Yes; developers who have used the prototype tool have indicated that they have detected more design inconsistencies than they would have normally. The automated approach was able to cover much more code than developers would have been able to using manual approaches. Moreover, developers agree that the tool detects valid problems that they were not previously aware of.

This verifies that our approach can effectively perform design conformance assessment in the specific environment of Cannibal Game Studios. Furthermore, it shows that the Reflexion Models [32] technique can prove effective in this specific business environment.

R.Q. 2: Are developers better capable of effectively managing inconsistencies, when using the SharpDCA tool?

Developers participating in the study felt better able to understand the design and were stimulated to explicitly formulate their design decisions, as well as the underlying motivations. In that sense, their ability to manage design inconsistencies has improved by using the tool.

How a developer manages an inconsistency, however, was observed to be very dependent on the amount of program understanding that is available. In the case of extensive preexisting program understanding, developers mostly depended on their own knowledge when deciding the best course of action. The tool was purely used for providing the initial awareness and consequently proved to be a stimulant for communication among developers. Whereas developers with little program understanding used the feedback and design rules provided by the tool to infer this knowledge.

8.5.1 Discussion

In the first place, we have achieved the successful integration and application of our approach within a *business environment*. This proves that DCA techniques successfully be used by companies, such as Cannibal Game Studios. While not all existing techniques are equally mature, there exists a range of different approaches which are able to cater to the different needs of individual organizations, which is an important step towards the adoption of DCA techniques by businesses.

What makes DCA successful in a business environment?

Inherently, there is no single concrete answer to this question, since it should always be put in the context of the specific requirements that arise from an environment. Specifically, different businesses can have different products or strategies which influence how DCA can be effective.

The key to success is to understand the advantages and disadvantages of the possible approaches, and base any solution on the approach that fits best with the environment in which it will be used. Moreover, it is important to minimize the risk of disturbing existing business processes, which can be done by carefully considering how developers have to depend on and interact with new tools and workflows.

In the process of creating an approach that is specifically suited to the needs of Cannibal Game Studios, these advantages and disadvantages (see Section 3.1.5) have been identified for the existing techniques, such as levels of expressiveness of different model types, the impact on maintainability of designs and the usefulness of checking specific design types. While this is useful to adapt the approach to a specific environment, it also presents an opening for new approaches that combine the advantages of different existing techniques.

Reflexion Models in practice

This thesis has particularly focused on (Hierarchical) Reflexion Models [32, 25], which already have been proven useful on real-world projects [32], be it in a controlled environment. We have taken this a step further and performed the evaluation of *SharpDCA* during an active software project, with the actual developers of that project.

Effects on the flow of information

The study has shown that design conformance assessment techniques can also be used to increase design understanding and communication among developers. By confronting developers with design inconsistencies, they are forced to think about the design decisions that have been made and to gather this information from their colleagues. This means that the presented approach might also have other useful applications in the areas of design communication and managing the flow of information within a development team.

8.6 Threats to Validity

This section will discuss, respectively, the internal and external validity of the results.

8.6.1 Internal Validity

Participant bias

Throughout the study, participants were aware of the fact that the objective of the study was to evaluate the tool created in the context of this thesis. This makes it possible that the participants answers during the interview and questionnaires were biased due their perception of what was expected of them. This was mitigated as much as possible by asking the same questions multiple times, reformulated and from different angles.

Observational effects

The fact that people were observed by a person sitting directly next to them might have changed their behaviour with respect to what they would have normally done in a real-world scenario. However, it is assumed that the impact of this is minimal, since it is not uncommon in a production environment to practice pair programming or to have the work of programmers monitored by during code review or by technical managers.

Level of design-familiarity

Among the participants there is clear distinction between their level of familiarity with the software project that was used for their trial session. This was expected, and has been proven, to significantly influence the behaviour of the participant. For that reason, the results obtained from two groups have been explicitly treated separately to study this influence.

Differences between projects

Due to the wish to have at least half of the participants perform the exercises on a project that they are intimately familiar with on one hand and the need to nondisclosure of proprietary code on the other, two separate projects have been used for the user testing sessions. The project used for external testing was adapted as to generate similar design inconsistencies, in order to keep the exercise as consistent as possible.

8.6.2 External Validity

Population size

Given the pre-experimental nature of the study and the relatively small population of participants, one should in any case be careful to draw any generalized conclusion. Although promising, the next step towards generating more definitive conclusions would require a larger experiment validate the current findings.

Expressiveness

The design specification language used by the tool has been constructed based on the expressiveness required by a specific set of reference cases. Although these cases are believed to be representative for the majority of other real-world projects, this is still an assumption. In any case, there will certainly be scenarios in which the expressiveness of the design specification language is not sufficient in its present form. So it is important to consider what kind of design constructs should be expressible beforehand.

False negatives

False negatives can occur due to a variety of reasons. In the case of a incompletely modeled design, it will always be possible that there exist design violations that go undetected. Furthermore, implementation errors might exist in any tool. The possibility of false negatives can never be excluded with absolute certainty, and can only be excluded with high certainty after long-term study of the outcomes.

False security

Related to the aspect of false negatives, there is a possibility that if users depend on the tool too heavily, they might falsely assume that there are no inconsistencies when false negatives occur. This could possibly cause design erosion directly, instead of preventing it.

Still, the results of the study do show that the awareness of potentially existing false negatives is slightly higher after using the tool. However, this may possibly due to the setting in which the experiment was performed or the novelty of the tool. It is possible that this effect subsides when programmers get used to the presence of the tool.

8.7 Summary

The findings of the study confirm that the approach of using a process-integrated tool for automated design conformance checking indeed increases design awareness, and consequently would be able to reduce design erosion significantly. The primary mechanism by which the approach can assist the development process, interestingly enough, is not by providing a new tool but by stimulating different team members to communicate more frequently and direct about specific design aspects. In essence, the tool works not by changing the process, but by focusing it more appropriately as soon as design violations occur.

Although these first results show a promising potential, it should be noted that these are still pre-experimental results. A larger scale experiment seems justified, and indeed would be required to validate the results, so this would be a logical next step.

Chapter 9

Conclusions and Future Work

In conclusion of the work, this chapter reflects back on the original goals and motivation for the performed study, and discusses what steps lie ahead.

9.1 Conclusions

The motivation of this thesis project was to improve the long-term maintainability of software by trying to prevent design erosion as close to the source as possible. Study of a real-world software production environment at Cannibal Game Studios has indicated that much design erosion is introduced because developers lack awareness of both the intended design and the current state of the implementation. This lead to the belief that a tool which can automatically monitor design conformance and communicate the results to developers, would increase the awareness of inconsistent design and allow developers to prevent design erosion from occurring.

Much work has already been done towards maintaining conformance between design and implementation, as described in Chapter 3; some approaches have even successfully been evaluated in case studies. However, our initial study has shown that such existing techniques are unlikely to be adopted in a production environment, simply because their impact on the development process would be too costly.

Hence, an approach (*SharpDCA*) to lower the cost of implementing such design conformance assessment (DCA) techniques was developed and evaluated. The idea behind this approach was to implement DCA techniques in such a way that they integrate better with existing development processes. Specifically, any new tools should integrate seamlessly with the existing tool suite of developers and feedback should be generated as close to the detected inconsistencies as possible.

In order to evaluate the effectiveness of this approach, the *SharpDCA* tool was tested in a real-world environment, and the results are promising. The approach did indeed have the intended effect of increasing design awareness. Moreover, it was observed to stimulate communication among developers, which in turn increased overall design understanding.

9.2 Contributions

The following concrete contributions have been made:

- An approach was formulated to effectively implement design conformance assessment techniques in a production environment.
- A prototype tool was implemented, and successfully tested in a real development project.
- The Reflexion Models DCA approach was implemented and confirmed to be sufficiently mature to be used in a production environment.
- An initial study has shown that the proposed approach can fundamentally enhance communication in development projects, and in turn prevent design erosion and increase design understanding.

By doing so, the viability of the approach and the *SharpDCA* prototype have been demonstrated; the effectiveness of the Reflexion Models technique for design conformance assessment has further been validated; and the effects of design conformance assessment in the real world have been explored.

9.3 Future work

9.3.1 Experimental validation

The pre-experimental study has shown very interesting results, some of which were unexpected. The logical next step is to search validation of these results in a larger experimental study. By evaluating the effects of the tool over a longer period of time, with more users and a control group, more definitive conclusions can be drawn.

9.3.2 Effects on design understanding

Although the prototype was considered to be quite non-invasive to the development process, the effects on communication and design understanding were observed to be significant. These effects deserve more detailed study in any follow-up experiments.

9.3.3 Extending the DCA tool

The current version of the tool is only a prototype, so it is logical to extend it with more advanced functionality to be tested further.

Communication of design rationale

User testing has shown that the tool creates a suitable platform for communicating design knowledge. By allowing the rationale to be communicated to programmers along with the design rules, design understanding can be communicated in a very context sensitive manner.

Moreover, when designs are formalized in the form of design models, they could even be versioned and their changes be traced over time. This could potentially add a new dimension to design understanding.

Live feedback

Developers have indicated to appreciate DCA feedback to be provided as often as possible and as close to the source as possible. Instead of checking design conformance at each build, it could theoretically be done while the developer is still writing code. This could provide even better traceability of inconsistencies.

Statistics gathering

When making automated DCA a part of the development process, it becomes possible to gather statistics on the detected violations and how they are resolved. This can in turn be used to gain new insights into the quality of the code and the design. For example, if developers often resolve an inconsistency by intentionally ignoring it, this would likely indicate a shortcoming in the design. If such statistics become visible, the responsible architect could decide to change the design.

9.3.4 Design specification expressiveness

For this study, a design specification language with a limited expressiveness has been used. More study is needed to determine what level of expressiveness is needed to describe a ‘software design’ in the general case.

9.3.5 Design specification tool

A part of the DCA workflow that has not been explored in this thesis is design specification. More study is needed to determine a good way of modeling the intended design, while still making the effort viable in practice. A graphical front-end, or a whole new design specification language all together might be implemented to allow designers to model the intended design easily.

9.3.6 The next frontier: Generative programming?

Generative programming is a completely different approach to maintaining design conformance; since the implementation is generated from a model, conformance to that model is implicitly guaranteed. So this begs the question, if generative programming is the future, will design conformance assessment ultimately become obsolete? Assuming that generative programming approaches such as domain-specific languages (DSLs) are indeed the future, the fundamental problem remains. Just as object-orientation is ultimately and abstraction of assembly code and, in turn, machine code, DSLs are just another layer of abstraction.

This means that even if code is generated directly from a model, that model will still be based on an intended design. So, design conformance assessment will likely stay a relevant challenge, although it might converge with techniques such as DSLs.

Bibliography

[1] 2012 (accessed March 29, 2012). `http://en.wikipedia.org/wiki/List_of_CLI_languages`.

[2] Giuliano Antoniol, Gerardo Canfora, Andrea De Lucia, and Ettore Merlo. Recovering code to documentation links in oo systems. In *WCRE*, pages 136–144, 1999.

[3] Giuliano Antoniol, Roberto Fiutem, and L. Cristoforetti. Design pattern recovery in object-oriented software. In *6th International Workshop on Program Comprehension (IWPC 98), June 24-26, 1998, Ischia, Italy*, page 153. IEEE Computer Society, 1998.

[4] Carliss Y. Baldwin and Kim B. Clark. *Design Rules, Vol. 1: The Power of Modularity*. The MIT Press, first edition edition, March 2000.

[5] Robert Balzer. Tolerating inconsistency. In *Proceedings of the 13th international conference on Software engineering*, ICSE '91, pages 158–165, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[6] Federico Bergenti and Agostino Poggi. Improving uml designs using automatic design pattern detection. In *In Proc. 12th. International Conference on Software Engineering and Knowledge Engineering (SEKE 2000*, pages 336–343, 2000.

[7] Jan Bosch. Software architecture: The next step. In Flvio Oquendo, Brian Warboys, and Ronald Morrison, editors, *Software Architecture, First European Workshop, EWSA 2004, St Andrews, UK, May 21-22, 2004, Proceedings*, volume 3047 of *Lecture Notes in Computer Science*, pages 194–199. Springer, 2004.

[8] Ned Chapin, Joanne E. Hale, Khaled M. Khan, Juan F. Ramil, and Wui-Gee Tan. Types of software evolution and software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 13(1):3–30, January 2001.

[9] Selim Ciraci, Pim van den Broek, and Mehmet Aksit. Graph-based verification of static program constraints. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 2265–2272, New York, NY, USA, 2010. ACM.

[10] Florian Deissenboeck, Lars Heinemann, Benjamin Hummel, and Elmar Juergens. Flexible architecture conformance assessment with conqat. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 247–250, New York, NY, USA, 2010. ACM.

[11] Michael Eichberg, Sven Kloppenburg, Karl Klose, and Mira Mezini. Defining and continuous checking of structural program dependencies. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 391–400, New York, NY, USA, 2008. ACM.

[12] Robert B. France, Dae-Kyoo Kim, Sudipto Ghosh, and Eunjee Song. A uml-based pattern specification technique. *IEEE Trans. Softw. Eng.*, 30:193–206, March 2004.

[13] Erich Gamma. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley, 1 edition, November 1995.

[14] John Guttag and J. J. Horning. Formal specification as a design tool. In *Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '80, pages 251–261, New York, NY, USA, 1980. ACM.

[15] Dirk Heuzeroth, Thomas Holl, Gustav Hgstrm, and Welf Lwe. Automatic design pattern detection. In *11th International Workshop on Program Comprehension (IWPC 2003), May 10-11, 2003, Portland, Oregon, USA*, page 94. IEEE Computer Society, 2003.

[16] Andrzej A. Huczynski and David A. Buchanan. *Organizational Behaviour*. Prentice Hall, sixth edition edition, 2007.

[17] Sunny Huynh, Yuanfang Cai, and Wuwei Shen. Automatic Transformation of UML Models into Analytical Decision Models. Technical report, Drexel University, 01 2008.

[18] Sunny Huynh, Yuanfang Cai, Yuanyuan Song, and Kevin Sullivan. Automatic modularity conformance checking. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 411–420, New York, NY, USA, 2008. ACM.

[19] Anton Jansen and Jan Bosch. Software architecture as a set of architectural design decisions. In *Fifth Working IEEE / IFIP Conference on Software Architecture (WICSA 2005), 6-10 November 2005, Pittsburgh, Pennsylvania, USA*, pages 109–120. IEEE Computer Society, 2005.

[20] Rudolf K. Keller, Reinhard Schauer, Sbastien Robitaille, and Patrick Pag. Pattern-based reverse-engineering of design components. In *ICSE*, pages 226–235, 1999.

[21] Dae-Kyoo Kim. Evaluating conformance of uml models to design patterns. In *10th International Conference on Engineering of Complex Computer Systems (ICECCS 2005), 16-20 June 2005, Shanghai, China*, pages 30–31. IEEE Computer Society, 2005.

[22] Dae-Kyoo Kim and Charbel El Khawand. An approach to precisely specifying the problem domain of design patterns. *J. Vis. Lang. Comput.*, 18:560–591, December 2007.

[23] Dae-Kyoo Kim and Wuwei Shen. An approach to evaluating structural pattern conformance of uml models. In Yookun Cho, Roger L. Wainwright, Hisham Haddad, Sung Y. Shin, and Yong Wan Koo, editors, *Proceedings of the 2007 ACM Symposium on Applied Computing (SAC), Seoul, Korea, March 11-15, 2007*, pages 1404–1408. ACM, 2007.

[24] Dae-Kyoo Kim and Wuwei Shen. Evaluating pattern conformance of uml models: a divide-and-conquer approach and case studies. *Software Quality Control*, 16:329–359, September 2008.

[25] Rainer Koschke and Daniel Simon. Hierarchical reflexion models. In *Proceedings of the 10th Working Conference on Reverse Engineering*, WCRE '03, pages 36–, Washington, DC, USA, 2003. IEEE Computer Society.

[26] Johannes Koskinen, Markus Kettunen, and Tarja Systä. Behavioral profiles-a way to model and validate program behavior. *Softw. Pract. Exper.*, 40:701–733, July 2010.

[27] Christian Kramer and Lutz Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In *Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE '96)*, pages 208–, Washington, DC, USA, 1996. IEEE Computer Society.

[28] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *Proceedings of the IEEE International Conference on Software Maintenance*, ICSM '99, pages 50–, Washington, DC, USA, 1999. IEEE Computer Society.

[29] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, and E. R. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *Proceedings of the 6th International Workshop on Program Comprehension*, IWPC '98, pages 45–, Washington, DC, USA, 1998. IEEE Computer Society.

[30] Andrian Marcus and Jonathan I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 125–135, Washington, DC, USA, 2003. IEEE Computer Society.

[31] Microsoft. 2012 (accessed March 29, 2012). `http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=8279`.

[32] Gail C. Murphy, David Notkin, and Kevin J. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *SIGSOFT FSE*, pages 18–28, 1995.

[33] Gail C. Murphy, David Notkin, and Kevin J. Sullivan. Software reflexion models: Bridging the gap between design and implementation. *IEEE Trans. Softw. Eng.*, 27:364–380, April 2001.

[34] NDepend. *Static code analyzer for .Net*, 2012 (accessed March 29, 2012). `http://www.ndepend.com/`.

[35] David Lorge Parnas. Software aging. In *Proceedings of the 16th international conference on Software engineering*, ICSE '94, pages 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[36] Leonardo Passos, Ricardo Terra, Marco Tulio Valente, Renato Diniz, and Nabor das Chagas Mendonca. Static architecture-conformance checking: An illustrative overview. *IEEE Softw.*, 27:82–89, September 2010.

[37] Gianna Reggio, Maura Cerioli, and Egidio Astesiano. Towards a rigorous semantics of uml supporting its multiview approach. In *Proceedings of the 4th International Conference on Fundamental Approaches to Software Engineering*, pages 171–186, London, UK, 2001. Springer-Verlag.

[38] Mohlalefi Sefika, Aamod Sane, and Roy H. Campbell. Monitoring compliance of a software system with its high-level design models. In *ICSE*, pages 387–396, 1996.

[39] Jilles van Gurp and Jan Bosch. Design erosion: problems and causes. *J. Syst. Softw.*, 61:105–119, March 2002.

[40] Sunny Wong, Yuanfang Cai, Miryung Kim, and Michael Dalton. Detecting software modularity violations. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 411–420, New York, NY, USA, 2011. ACM.

[41] Yongjie Zheng. 1.x-way architecture-implementation mapping. In *Proceeding of the 33rd international conference on Software engineering*, ICSE '11, pages 1118–1121, New York, NY, USA, 2011. ACM.

[42] Hong Zhu, Ian Bayley, Lijun Shan, and Richard Amphlett. Tool support for design pattern recognition at model level. In *Proceedings of the 2009 33rd Annual IEEE International Computer Software and Applications Conference - Volume 01*, pages 228–233, Washington, DC, USA, 2009. IEEE Computer Society.

Appendix A

Interview

This appendix describes the structure of the interviews that were performed. The actual interview was performed using a semi-structured format, where the described order and formulation of questions was used as a guideline, but not stricly enforced. Particularly, interviewees were allowed to give an in-depth motivation of their answers.

A.1 Structure

1. *How experienced and motivated are developers with design conformance assessment?*
    a) What roles in the development process do you have experience with?
    b) What is in your opinion the main benefit of design-code consistency?
    c) Do you actively practice design conformance checking?
        i. If yes, by what means? Do you use tool support?
        ii. If no, why not? Was this an explicit choice?
2. *Are developers more aware of inconsistencies?*
    a) Have you found design inconsistencies with the tool than you would not have otherwise?
    b) Would you have ignored inconsistencies that you otherwise would not have?
    c) Do you understand why the hits are being detected as being inconsistent?
    d) Do you agree that these are indeed violations?
3. *Are developers better capable of handling inconsistencies?*
    a) Is it easier to identify the location of an inconsistency?
    b) Is it easier to resolve an inconsistency?
    c) Do you feel constrained by a tool?

4. *How does the tool influence the development process?*
    a) What would you do differently without versus with the tool?
    b) Describe how you would like to go about solving design violations.
    c) Do you value good design just as much a working code? Why?

Appendix B

Questionnaires

The following sections describe the exact questions that were presented to the study participants. The participants were asked to answer the questions on a 5-point Likert scale.

Questions 1-5 test the willingness to apply design conformance checking, questions 6-9 test the awareness of design inconsistencies and questions 10-13 test the ability to resolve known inconsistencies.

B.1 Pretest questionnaire

1. I think it is important to maintain absolute consistency between the intended software design and the actual implementation.
2. I think the the time and effort of checking design conformance is worth the investment.
3. I am likely to adopt a tool to help me perform design conformance assessment.
4. Design/code consistency is the responsibility of the software designer.
5. Design/code consistency is the responsibility of the programmer
6. While programming, I generally find a lot of new inconsistencies.
7. It is easy to identify inconsistencies when checking code manually.
8. I generally feel confident that there are no inconsistencies, between design and implementation, that I am not aware of.
9. I would value a tool to assist in detecting violations between design and implementation.
10. It is generally difficult to determine the location of inconsistencies between design and implementation.
11. It is generally difficult to determine why an inconsistency exists.

12. It is generally difficult to fix an inconsistency.
13. I would value a tool that would guide me to where inconsistencies exist.

B.2 Posttest questionnaire

1. I think it is important to maintain absolute consistency between the intended software design and the actual implementation.
2. I think the the time and effort of checking design conformance is worth the investment.
3. I am likely to adopt a tool to help me perform design conformance assessment.
4. Design/code consistency is the responsibility of the software designer.
5. Design/code consistency is the responsibility of the programmer
6. With the tool, I have identified many inconsistencies that I was not yet aware of.
7. It is easy to identify inconsistencies when checking code using the tool.
8. With tool support, I feel confident that there are no inconsistencies, between design and implementation, that I am not aware of.
9. I would value a tool to assist in detecting violations between design and implementation.
10. With tool support, it is generally difficult to determine the location of inconsistencies between design and implementation.
11. With tool support, it is generally difficult to determine why an inconsistency exists.
12. With tool support, it is generally difficult to fix an inconsistency.
13. I would value a tool that would guide me to where inconsistencies exist.

Appendix C

Questionnaire Results

The table below shows the results of the questionnaires that have been performed as part of the evaluation study. The columns represent the answers of the individual participants and the rows represents the answers per question.

	Pretest								Posttest							
	Internal				External				Internal				External			
Question 1	4	4	4	4	4	5	3	4	4	5	4	2	4	5	4	4
Question 2	4	5	4	4	4	4	4	5	4	4	5	4	5	4	4	4
Question 3	4	4	3	5	3	4	5	4	4	4	4	5	4	5	4	5
Question 4	3	5	4	3	3	4	4	5	3	4	4	3	3	4	3	4
Question 5	3	5	5	3	5	4	4	3	3	5	5	3	5	4	4	4
Question 6	2	2	5	4	2	3	2	3	2	2	4	2	4	3	4	3
Question 7	3	1	3	4	3	3	2	2	3	3	3	3	4	3	4	3
Question 8	5	4	5	2	2	4	5	5	2	2	2	4	2	3	5	3
Question 9	5	4	5	5	4	5	5	5	5	5	5	5	5	5	5	5
Question 10	5	5	3	3	4	4	5	3	1	1	2	2	3	1	3	1
Question 11	1	3	4	3	3	4	2	3	2	2	3	2	3	4	3	3
Question 12	1	4	2	3	3	3	4	2	1	1	2	2	2	3	3	2
Question 13	5	4	5	5	4	5	5	4	5	5	5	5	4	5	5	5