Master's Thesis

Automated verification of design adherence in software implementation

Rune Flobakk

Trondheim, June 2007

Supervisors:
Carl-Fredrik Sørensen, IDI
Ole-Martin Mørk, BEKK

Norwegian University of Technology and Science
Department of Computer and Information Science
Faculty of Information Technology,
Mathematics and Electrical Engineering

Abstract

Software design and architecture specify how a system should be implemented to achieve the required quality attributes. Being able to automatically verify the design adherence during implementation will continuously assure that the system realizes the quality attributes, as well as over time does not drift away from them. This thesis investigates how a software design can be used to automatically verify and enforce rules for implementation. The current tool support for automatic design enforcement is assessed and reviewed. In addition, a prototype contribution to this practice, a plug-in for the Maven software project management system, is presented.

Preface

This thesis is the final dissertation of computer science master studies at the Department of Computer and Information Science, NTNU. The thesis topic was provided by BEKK Consulting AS, which was interested in how they may further introduce additional automated processes to their system development methodologies.

Although this project was initiated by a company external to the University, any results gathered are public and viewable for anyone.

I would like to thank my supervisor Carl-Fredrik Sørensen (Calle) for great help and guidance during my thesis work. We have had several enlightening discussions (and an occasional digression from the main topic as well) which for me has been both engaging and motivating. He also granted full access to his office library of technical and scientific literature.

I would also like to thank my external supervisor Ole-Martin Mørk at BEKK Consulting AS. He has provided great advice on how my thesis topic can be transfered to real-world value. In addition, he uses free beer as a motivational factor which is an indicator that we speak the same language.

Further, I have received great feedback from Alef Arendsen at Interface21, the company behind the Spring Framework. Arendsen is a consultant at the Technology Department of Interface21 and is one of the initial inspirational sources for my thesis topic.

As this thesis marks the end of several years of study for me, I would like to take the opportunity to thank my mum for letting me raid the refrigerator after each of my visits, and my step-father for bravely ignoring his back pain when helping me moving to new flats.

Trondheim, Sunday 24th June, 2007

Rune Flobakk

Contents

List of Figures

Listings

List of Tables

I claim that nobody has eaten a fruit. Lots of people have eaten apples, bananas, and oranges, but nobody has eaten a .3-pound red fruit.

Arthur J. Riel
explaining the concept of abstract classes.

Part I

Context

Chapter 1

Introduction

1.1 Project description

1.1.1 Problem context

Software engineering is a complex area which often requires both coordination and integration of different software components from different technologies, platforms, and languages. It is common to speak of *component-orientation*, where the interdependencies between the components are rigorously defined by the design.

In addition, development is often also performed across different geographical locations. Due to Internet's enabling of real-time correspondence and various forms of groupware, the actual production does not really need to be centralized to a single location, i.e, a software system may be implemented and maintained from different corners of the world.

This clearly opens up new possibilities, but also enables more error-prone solutions. The role of the software architecture is important to propose a design which makes use of relevant patterns and best practices. This is both to support the distributed development process and to ensure an implementation which adheres to the required quality attributes.

1.1.2 Motivation

A system development project may consist of a vast amount of architecture and design documentation with nicely plotted diagrams giving various views of the system together with a fully working implementation. However, this does not actually imply that the implementation adheres to the original design. As a worst case scenario it is quite possible for the programmers to implement a working system without any regard to the original design at all ("The software is the design" [Martin, 2002]). This extreme case is hardly the case in practice, but

it demonstrates the point that an implementation may be realized completely independent from its proposed design.

According to Brown et al., 84 % of all software projects are unsuccessful, and the reason is the lack of pattern use in the software design. The book AntiPatterns also makes an important distinction on patterns from anti-patterns, where the latter is a "commonly occurring solution to a problem that generates decidedly negative consequences". It may also be the application of a pattern in a context where it does not belong [Brown et al., 1998].

Is it then possible to impose a kind of formal dependency from the code to the architectural design in such way that the code is *automatically* rendered not valid if it does not adhere to the design? This concept is referred to as *design level assertions* (DLA) throughout this thesis, and is explained in more detail in section 3.4.

1.1.3 Project initiator: BEKK

The thesis topic was originally proposed by BEKK Consulting AS (Oslo, Norway) as they are interested in more ways to automate the development process to assure quality. BEKK especially mentions low coupling and high cohesion as goals for this automated design verification. BEKK's interests in this topic should be compatible with the university's public research interest, as BEKK does not wish to keep any results from the thesis work exclusively in-house. The thesis report is publicly available, and any relevant software artifacts produced during the thesis work are considered to be released as open-source software (OSS).

1.2 Goals

The goal of this thesis is to review and contribute to the current state of automated verification of design adherence.

1.2.1 Practical goals

This thesis addresses the DLA concept from a developer's point-of-view. In particular the goals of this thesis are the following:

G1 Investigate and review the current practices and tools for automated assertion of design adherence.

G2 Evaluate the strengths and weaknesses of these practices and tools.

G3 By the results of G1 and G2, develop a DLA tool prototype to contribute to this discipline.

1.2.2 Research questions

Some specific research questions that are discussed and also partly answered are:

RQ1 *How may the use of DLAs improve project efficiency?* Automated operations, as opposed to manually executed tasks, will in general contribute to better project efficiency [Clark, 2004]. This is especially crucial to repetitive quality assurance (QA) operations, which are conducted in iterative development, for instance on each nightly build by a continuous integration (CI) system. So to answer this question, it is necessary to address a few preconditional aspects of DLA automation itself:

RQ1.1 What elements of a software design can be codified into assertible rules enabling automated testing of these rules against the program code?

RQ1.2 In what ways must a DLA tool be able to support interests of the various project stakeholders?

RQ2 Given the current state of tool-support for DLA, *how may next-generation tools address the problem in a better way?* There exist several tools for measuring software design and quality, and a representative selection of such tools must be assessed and evaluated to identify any shortcomings and propose any potential improvements.

1.3 Reader's guide

1.3.1 Parts and chapters

The thesis is structured using 4 main parts.

Part I - Context

Part I presents the topic and problem area, and explains the motivations why research on this topic is beneficial for an increased software quality. The goals of the work are stated, and the methods to pursue these goals are presented.

Chapter 3 contains a discussion on design/software quality, relevant metrics, and a review of the current tool support is presented. As software quality is not only given by a system's adherence to its design, but can also be measured by extracting various structural aspects of the source code, both tools and practices for measuring design quality and automatic enforcement of design rules are assessed.

Part II - Contribution

Part II presents my own contribution to the problem area addressed in this thesis; a plug-in developed for the Maven 2 project management tool.

Part III - Evaluation

The evaluation part discusses the effort for a positive contribution to the problem area. This leads to a conclusion and a proposal for further work and studies.

Part IV - Appendix

The Appendix part contains full representations of details not suitable for the body text in the report. This includes relevant source code examples, more detailed diagrams, etc.

Glossary and References

At the very end, a glossary of relevant terms used in the thesis, and the references list are presented.

1.3.2 Typography and conventions

Some typographic styles and conventions are used throughout this thesis to distinguish certain elements, especially to when used in-line the body text. The conventions are shown in Table 1.1.

Typ. convention	Explanation
`monospaced`	To present code snippets, input commands, console output and similar computer related text, monospaced typeface is used. References to Web locations, URLs, are also presented in monospaced font.
italics	Mathematical expressions and symbols use italic typeface. In addition, captions for figures, tables are presented in italics to distinguish them from the remaining body text.
[Author (et al.), year]	References use square brackets containing the surname of the author (in case of three or more authors, "et al." is used) and the year of the referred to work.
term (abbr.)	Any term with a following abbreviation in parenthesis notes that this is a term explained in the glossary at the end of the thesis. The terms in the glossary are however not limited to abbreviations used. For well-known terms, only the abbreviation may be used for the remainder of the thesis.

Table 1.1: *Typographical conventions*

Chapter 2

Research

This chapter presents the research approach used throughout the thesis work.

2.1 Limitation of scope

As with any research project it is important to set a clear boundaries for the scope of what is considered relevant to include. It may also be useful to, if possible and applicable, to state any obviously related subtopics which are not included within the thesis scope. This will diminish the risk of the project loosing focus and required depth because of a too loosely defined topic, as well as the chance of breaking the time-frame. The mentioning of related subtopics also serves as suggestions for relevant further studies.

2.1.1 DLA from a developer's point of view

The project has chosen to focus on how the use of DLA fit into developer processes, and in particular the implementing process with the use of automated tests at build-time. Thus, the actual creation of a software design/architecture is beyond the scope of this thesis, but what aspects of a design that are relevant for codification into rules for an automated DLA tool should be addressed.

Besides architects and programmers, other relevant stakeholders for DLA are project leaders and customers. Project leaders could be interested to see that an implementation adheres to the given design, and depending of the technical knowledge of the customer, the customer appreciates reports proving the agreed quality attributes of the software which the architects have realized in the software design.

While the use of DLA is worthless without feedback from the tool being used, the focus for this thesis is kept on the technical output for the developers to use while implementing the software. The creation of summarizing reports for customers and/or project leaders are outside the scope of this thesis.

2.2 Constructive research

To create an artifact contributing to a discipline is a common task of research in information systems. The software engineering discipline can in some respects still be considered as immature due to:

- software systems are delivered with a considerable amount of errors (bugs),
- projects are commonly not delivered on-time,
- implementations commonly do not satisfactory meet the clients' requirements.

In most industries, these kinds of failures would not be tolerable at all, but "...software technology is in the Stone Age. Application developers, managers, and end users are paying the price" [Brown et al., 1998]. Software engineering needs extensive constructive research to positively support the development of the discipline itself. Frameworks, automation of routine work, and formalized processes are examples of such contributing 'constructs'.

2.3 Software engineering research

This thesis is written in consequence of an assessment of available tools and practices or lack thereof. In addition, a prototype implementation is conducted to propose a contribution to the discipline of automated DLA.

Using Glass et al.'s article on research in computer disciplines, the majority of research on software engineering treats systems/software concepts (54.8%), and this thesis falls into the following topics [Glass et al., 2004]:

1. software life cycle/engineering (incl. requirements, design, coding, testing, maintenance)
2. tools (incl. compilers, debuggers)
3. product quality (incl. performance, fault tolerance)

The use of DLA involves design codification and in particular *testing* of how an implementation adheres to this design (1). The testing is done in an automated fashion, and thus *tools* are an important part of the DLA discipline (2). Without tool-support, an automated process is not possible. Hypothetically, DLA also positively affects the *quality* of the software (3), not per se, but by continuously ensuring the quality implicated by the software design.

2.4 Approach and method

The project aims mainly at a qualitative approach. These kinds of research methodologies have during the later years been approved as equal in value to the more traditional quantitative approaches, whereby action research, which combines theory with practice, has often been mentioned from the end of the 1990s as a relevant qualitative method for research in information technology. [Baskerville, 1999, Avison et al., 1999]

This project has chosen to work with *descriptive* and *formulative* approaches by conducting a review of the tools and practices currently available and evaluate strengths and weaknesses. From the results of the review, an implementation of a DLA tool has been planned, also with a subsequent evaluation of this prototype.

Although a full-scale action research project is too comprehensive for a master thesis project, the action research process described by Baskerville still suits the planned procedure of this project. Figure 2.1 shows the common activities of an action research project. These activities will also be conducted for this project, although using a one-pass linear model:

Diagnosis. This is the state-of-the-art review, a "diagnosis" of the current state.

Planning. After the diagnosis follows a small planning phase of how to realize a contribution to the DLA discipline.

Implementation. This is the actual implementation of a DLA tool prototype.

Evaluation. The evaluation comprises the implementation itself, how the prototype may serve as a platform for further development, and any obstacles encountered because of design decisions. In addition, the implementation must be evaluated with respect to the *diagnosis*, how the implementation may contribute to the state-of-the-art.

Learning. This is the conclusion of the work conducted, and consequently the conclusion of this thesis; the concrete additions to the domain's knowledge base.

While using the action research activity sequence as inspiration, this thesis project is conducted in particular using *conceptual analysis*, presented in Chapter 3, and *concept implementation* (proof-of-concept), presented in Part II, as the specific methods of research.

2.4.1 Conceptual analysis

The conceptual analysis is presented in Chapter 3 and involves an assessment and discussion on design quality, and and a presentation of two concepts to ensure quality of software design; design patterns and design metrics.

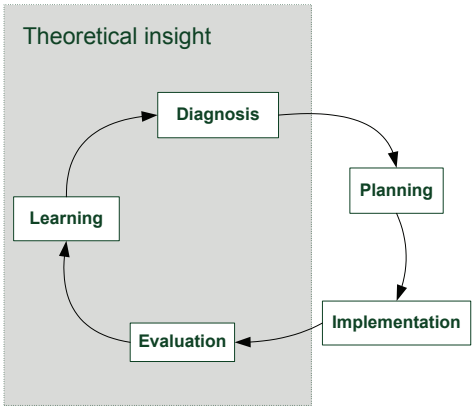


Figure 2.1: *The activities of action research conducted in a cyclic iterative manner. The grey bounding box shows which activities utilizing and/or contributing to theoretic insight of the domain.*

2.4.2 Concept implementation

The concept implementation is presented in Part II, and describes a tool prototype which will verify that an implementation adheres to a given design. In addition, the target platform, Maven 2 is described, as well as the rationale for choosing this platform.

Chapter 3

State-of-the-Art

In this chapter, the conceptual analysis (recall Section 2.4) is presented. An assessment and discussion on design quality, and two concepts to ensure quality of software design; patterns and metrics, are presented. Following, the concept of *design level assertions* is discussed in more depth. In addition, a selection of tools for assisting in ensuring architectural quality attributes when implementing software systems.

3.1 What is "good design"?

There are good designs and obviously there are bad designs. We want good designed software, and we want to employ good design when developing to achieve a certain required quality of the final product. The International Organization for Standardization (ISO) defines *software quality* using a model of the following six characteristics [ISO9126-1]:

- functionality
- reliability
- usability
- efficiency
- maintainability
- portability

In software system modeling, such qualities are often referred to as non-functional requirements (NFR), and are handled by certain modeling languages, e.g. goal-oriented requirements languages (GRL), as *softgoals*. The software architecture research refers to these as *quality attributes* and the means to achieve them as *quality tactics* [Bass et al., 2003].

However, such qualities are not easy to unambiguously prove as they are a bit vague. It is often argued for and against desired qualities, often by using several sub-characteristics, and if a quality receives a satisfactory amount of positive arguments, it is said to be fulfilled.

Failing to implement such required global qualities of a system constitutes a great factor in unsuccessful projects, and still they are most commonly expressed in an informal matter [Mylopoulos et al., 1999].

Quality attribute will be the term used for the remainder of this thesis.

3.1.1 Evaluation of design

Clearly, it can be difficult for instance to evaluate if a system indeed is easy to *modify*, consists of a large amount of *reusable* components, or *performs* satisfactory in a vast amount of different environments. Evaluating quality attributes is a lot more challenging than the binary behaviour of functional requirements (FR); either the system is able to fulfill an FR (or parts of an FR), or it is not.

If an FR is not fulfilled, it is a matter of implementing the missing functionality. In the case of quality attributes, there is, in addition to the difficulties in measuring and proving them, commonly a matter of competing attributes. The classic scenario here is the balance between *security* and *usability*; security measures often hurt how a system's ease of use is experienced[1].

Thus, the quality of a software system cannot necessarily be viewed simply as the *sum* of the involved quality attributes. Instead, the attributes must be balanced on their importance, and when each quality attribute exceeds a defined threshold, the overall software quality requirement is met.

3.1.2 The role of patterns

When speaking of good design, it is common to think of the use of patterns to solve implementation problems. Patterns provide general solutions, which are often thought of as undisputable positively affecting the resulting system, to commonly reoccurring problems. One of Gamma et al.'s characteristics of patterns also indicates why patterns must be considered when dealing with design adherence and analysis above the class level: "Patterns identify and specify abstractions that are above the level of single classes and instances, or of components" [Gamma et al., 1995].

[1]Microsoft has over the years earned itself a reputation of producing insecure operating systems, but there is little doubt about their contribution to bring computer software into the everyday life, i.e. their software generally have high usability. This unevenness is however less valid today for Microsoft's case, but nevertheless, it has business-wise proven to have been a successful quality tradeoff decision

Larman defines patterns this way, not including the emotional aspect often related to patterns of being some magic path to good design:

> [...] a pattern is a named problem/solution pair that can be applied in new contexts, with advice on how to apply it in novel situations. [Larman, 1998]

This definition, together with similar definitions in various other literature sources, simply implies that a pattern is an often used solution to a common problem. It does not say that a pattern must be proved to have a positive effect to deserve to be acknowledged as a 'pattern' per se.

Buschmann et al.'s definition is more biased toward the common positive perception of patterns.

> [A pattern is] a particular recurring design problem that arises in specific design contexts, and presents a *well-proven* generic scheme for its solution. The solution scheme is specified by describing its constituent components, their responsibilities and relationships, and the ways in which they collaborate. [Buschmann et al., 1996]

As patterns form the building blocks of a design on an abstraction level above the mere components and their interdependencies and communication, but also constitute an important communication vocabulary, this level of abstraction should optimally also may be utilized when working with a software design tool.

3.1.3 AntiPatterns

Because of the evolved consideration of patterns, and especially design patterns, to be equal to 'best practice', there has grown some research on negative consequences of the use of patterns, termed as *AntiPatterns*. Patterns were shortly viewed as the "Holy Grail" (and still are) of best practices in software engineering when it became mainstream as late as 1994, but two years later Michael Akroyd presented a formal model of AntiPatterns which "focused on recognizing harmful software constructs, which were *reoccurring across several software projects*" [Brown et al., 1998].

So to avoid AntiPatterns, is it simply to ensure to apply correct patterns to the intended context? If it only were that easy. While after the definition of AntiPatterns as a pattern which generates more negative than positive consequences, there is still the discussion of what is considered positive and negative as well as the challenge of actually capturing every consequence of a pattern.

3.1.4 Example pattern: The MVC.

The Model-View-Controller (MVC) pattern is often regarded as *The* Software Architecture Pattern, promoted by numerous frameworks such as Spring or Ruby on Rails [Spring, Hansson, 2007]. It is based on the classic separation of the user interface (view) from the underlying independent data (model) with an added intermediary in between (controller). MVC has become a buzzword which is often mentioned in promotional descriptions.

3.1.4.1 MVC *may* lead to an AntiPattern

Riel's book, Object-Oriented Design Heuristics, presents a critical view on the use of controller classes as a possible "violation of a fundamental principle of the object-oriented paradigm[2], namely, data and behavior are bidirectionally related as a conceptual block", and thus should be modeled into the same entity (e.g. a class, package, component, etc). Riel draws a relation from the now abandoned action-oriented (structural) programming to the use of separate controller components to do operations on the data model, and that any class with an unreasonable amount of `getXYZ()` methods should bring up a warning alarm. To support the principle that related data and behaviour should be kept in one place, the data model may implement operations as necessary as there are no rule saying that a class using another must exploit its entire public interface. [Riel, 1996]

In fact, Riel says that any class name which is derived from a verb should raise a warning sign. These kinds of classes are very common for instance in the standard class library of the Java platform, which enables pluggable behaviour to generic data structures. The most simple example of this is perhaps the use of Comparator classes (from the verb "to compare") to separate the actual sorting behaviour from sorted data structures of the Java Collection Framework [Bloch, 1996–2006].

The book "Design Patterns: Elements of Reusable Object-Oriented Software" [Gamma et al., 1995] is considered essential reading on the topic of design patterns, and the four authors are often referred to as the Gang of Four (GoF). Already in the introduction chapter, the MVC pattern is briefly described as means for achieving decoupling of the data from its presentation.

An important principle of software engineering is humorously named KISS, an acronym for Keep It Simple, Stupid[3]. A funny detail is that GoF say "We've left out the controllers for simplicity". Not claiming that this is contradicting the motivations of employing the MVC, it still makes an interesting observation together with Riel's view on controllers.

[2]This was before the common advice to avoid the use of "paradigm" in texts. Since then it has lost some of its original meaning from overuse, and is considered a rather vague term today.

[3]A maybe more formally appropriate principle than KISS may be Einstein's quote "Make everything as simple as possible, but not simpler".

From this, it can be concluded that pattern employment is not simply an issue of inquiring a map of situations and their correct pattern(s). What is considered a good pattern applied to the right problem is not an absolute science.

3.2 Patterns

This chapter will present a selection of patterns and a discussion of how they may be identified in program code.

3.2.1 Pattern types

Patterns are commonly grouped into what abstraction level of a system they address and/or what problem area they provide solutions for.

3.2.1.1 Abstraction level

Architectural patterns are described to present the overall structure(s) a system is based on. Examples of such patterns are, e.g, client-server architecture and the Model-View-Controller. On a lower level, design patterns provide solutions for problems inside the larger architectural entities. For instance, both the client and server component of an architecture may employ the same design pattern to solve an implementation problem present in both components.

Whereas architectural and design patterns are generic and applicable for any object-oriented language, idioms are even lower level patterns which are specific to a programming language. Idioms provide implementation-specific solutions in a particular programming language. An idiom may also be programming guidelines, naming conventions, source code formatting, etc. Idioms are not portable to the same extent as architectural and design patterns, as they depend on features of the language and its optimal use Gamma et al. [1995]. For instance in C++, multiple class inheritance is supported by the language itself, but in Java, which is a single inheritance object-oriented language, an idiom must be used to achieve multiple inheritance [Thirunarayan et al., 1999].

3.2.1.2 Problem-specific

Patterns may also be specified by what problem areas they are related to. Examples of such pattern groups are (but not limited to):

- structural
- behavioral
- conceptual

- analysis
- organizational

Bass et al.'s definition of software architecture is:

> The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them [Bass et al., 2003].

By this definition, it is evident to group architectural patterns as also structural patterns. Several design patterns are specifying optimal structures for how to solve problems using several classes with discrete responsibilities, but design patterns also cover a wide range of other problems which makes it reasonable to divide them into problem-specific groups.

3.2.2 GRASP patterns

General Responsibility Assignment Software Patterns (GRASP) is a set of fundamental patterns for object-oriented design. It is important to *grasp* the principles of these patterns in order to successfully design object-oriented software, hence the acronym GRASP.

Two of the patterns of GRASP directly address the problems of keeping low coupling and high cohesion (see Sections 3.2.2.3 and 3.2.2.4) in an object-oriented design, and thus should be highly relevant for BEKK's original thesis proposal. The discussion on GRASP patterns are to a large degree taken from the book *Applying UML and Patterns – An introduction to object-oriented analysis and design* [Larman, 1998] to a large degree.

3.2.2.1 Expert pattern

Expert, or Information Expert, is one of the most fundamental patterns of object-oriented design. It addresses the problem of responsibility delegation, and states that the class which encapsulates the least amount of information needed for an operation should also be responsible to implement this operation. This may seem obvious, but it is no less important to ensure that operations are implemented by the correct class to support maintainability, extendability, and reusability.

3.2.2.2 Creator pattern

The Creator pattern lists some possible conditions where a class `B` should be responsible for instantiating class `A`. `B` thus becomes a Creator of `A` objects. The conditions for assigning class `B` to create instances of class `A` can be one or several of the following:

- `B` *aggregates* or *contains* `A` objects.
- `B` *records* instances of `A` objects.
- `B` *closely uses* `A` objects.
- `B` *has the initializing data* that will be passed to `A` when it is created (`B` is actually an Expert with respect to creating `A`).

Assigning the Creator role to the right class of an object-oriented design supports low coupling.

3.2.2.3 Low Coupling pattern

The Low Coupling pattern is an evaluative pattern and a goal which a designer must apply while creating the system design. While the Creator pattern specifically deals with the responsibility of class instantiation, this pattern is a generic goal of responsibility assignment to support low coupling between classes.

Coupling is a measure of how a class is dependent of another class. Low coupling is desirable to prevent ripple effects [Bass et al., 2003], make classes easier to understand in isolation, and support reusability. Summarized, the Low Coupling pattern "[...] encourages assigning a responsibility so that its placement does not increase the coupling to such a level that it leads to the negative results that high coupling can produce." [Larman, 1998]

Low coupling is not an absolute measure. It is context dependent and often acts as a guidance of how to apply other patterns. Coupling should be considered as in how increasing it will generate problems. Generally, it should be focused on keeping classes which are generic of nature low coupled if cost/benefit is an issue to consider.

3.2.2.4 High Cohesion pattern

The High Cohesion pattern encourages the design of focused classes which do a minimum of related work.

A class has low cohesion if it does a plethora of unrelated work, i.e, the system intelligence is poorly distributed. An example of this is the "God Class", a class which performs most of the work of a system and uses a lot of other classes to abstract minor details. This is often a problem for people moving from structural programming to object-oriented programming [Riel, 1996]. The classes are solely used to implement sub-routines which are called from the centralized main execution path, i.e, the God Class. It is, thus, indeed possible to create non-object-oriented design in an object-oriented language.

3.2.2.5 Controller pattern

The Controller pattern is the delegation of high-level tasks to a special controller class. Controller classes often act as an interface for Use Case operations, and are responsible for carrying out these tasks by delegating the relevant subtasks to business objects. Controller classes represents the highest level of operations carried out by the system.

Even though a controller class is responsible for handling user input, it is important that this is not given to GUI classes. A GUI class is responsible for accepting user interactions, but the actual execution of the use cases should be delegated to a controller class. Controllers should be named and modularized by the actual real-life entities they act on behalf of. For instance, an operation `makePayment` would be natural to delegate to for instance a `CashRegister` controller.

Recall Section 3.1.4.1 for Riel's critique on the use of controller classes. Poor controller design may result in *bloated controllers*; a controller with too many areas of responsibility and low cohesion. The controller should delegate work, not perform everything itself. A system should also have several controllers, not one which is responsible for all use cases.

3.2.3 Behavioural patterns

Behavioural patterns generally describe how to optimally implement a certain behaviour using collaborating objects. Examples are how to iterate through data structures (Iterator pattern), and how changes to data should automatically trigger any relevant views to update their presentations (Observer pattern, explained further in Section 3.2.3.1).

3.2.3.1 Observer pattern

The Observer pattern is an important pattern in GUI-based applications where data is often presented in several GUI components. It must be guarantied that when a user interacting with one component resulting in change in the data, the change is immediately reflected throughout the entire GUI.

Figure 3.1 shows a typical implementation of the Observer pattern. `TableView` and `GraphView` show two different representations of the `ConcreteSubject` data, they are *observing* `ConcreteSubject` which will notify all its objects registered as Observers. This makes the data decoupled from its presentation, and plugging in new presentation classes, e.g, a `HistogramView`, is a simple matter of implementing the `update` method to query the required data.

There is however a problem with the Observer pattern in single-inheritance languages which is discussed in Section 3.2.4.

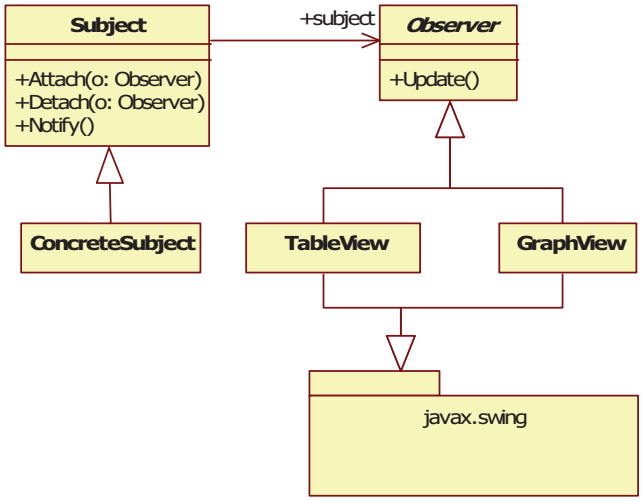


Figure 3.1: *Typical implementation of the Observer pattern. The concrete GUI classes,* `TableView` *and* `GraphView`*, extends classes from the* `javax.swing` *package, and also implements the* `Observer` *interface and must consequently implement the* `update` *method.*

3.2.4 Object-oriented patterns does not solve everything

The literature on patterns in general focuses on object technology [Larman, 1998, Riel, 1996], among them the GoF book [Gamma et al., 1995] which uses C++ and Smalltalk for implementation examples. There are also numerous other books on design patterns, e.g. for Java [Grand, 1998, Cooper, 1998] and Eiffel [Jézéquel et al., 2000]. While object-oriented programming works great for clean separation of responsibilities into discrete components, there are certain concerns that are often unanimous for several components. As design patterns traditionally provide generic solutions for object-oriented design, they may not satisfactory solve such problems. [Laddad, 2003]

The Observer pattern already discussed in Section 3.2.3.1, solves the "one-to-many dependency between objects so that when one object changes its state, all its dependents are notified and updated automatically" [Gamma et al., 1995]. This pattern does however touch an issue which object-orientation cannot cope with; *crosscutting concerns*.

3.2.4.1 Crosscutting concerns in the Observer pattern

An Observer implemented in a single-inheritance language, such as Java, will typically subclass/extend a generic GUI class, and therefore cannot inherit any common logic for how an Observer updates its presentation. As shown in Figure 3.1, both `TableView` and `GraphView` must implement the crosscutting concern of how to update their views, i.e, query data and present it.

Equally, every Subject must implement the responsibility to notify Observers when their data change, which is perhaps a responsibility even farther away from

its main purpose of "being data" than the view-classes' responsibility of getting updated data for their views. For each operation of a Subject which results in its data being updated, the programmer must remember to invoke the `notify` method. This hurts cohesion, as triggering notification of external components is in principle not part of the data model's (i.e, Subject) responsibility.

3.2.4.2 AOP in brief

There are certain concerns that are often unanimous for several components. For instance, logging is a common problem often solved using AOP and what most beginner tutorials on AOP uses as a practical implementation example. Logging is described using AOP-terminology as a crosscutting concern; a system-level requirement which needs to be included in several components. Component- and object-oriented programming and its design patterns are not able to satisfactory address such requirements.

Other examples of crosscutting concerns:

- monitoring
- data persistence
- security

In object-oriented programming, such requirements are bound to result in redundant code as identical code fragments have to be present in every class which is affected by the system-level requirements. Such tangled code complicates reasoning about the system and hurts modifiability as one modification needs to be consistently changed in all the involved code [Clement and Kersten, 2004]. In addition, crosscutting concerns means added responsibility to classes which results in decreased cohesion.

AOP gives the ability to define behaviour which involves several components of a system in one centralized idiom called an *aspect*. By defining aspects in AOP one can specify common behaviour shared by several components of a system, and thus these responsibilities does not need to be repeatedly implemented in every involved class.

3.2.4.3 AOP-implemented patterns

In a paper by Hannemann and Kiczales, it is suggested that patterns may be implemented in a more modular and reusable way using aspect-oriented programming. They implemented the 23 GoF patterns in Java as well as developed corresponding AspectJ implementations to these patterns for a comparative analysis. Using AspectJ, they were able to implement for instance the Observer pattern and achieve decoupling of the participant classes from the behavioural pattern.

The pattern code was localized, and thus potential changes to each instance of the pattern only require changes in one place.

The paper specifically points at improvements of behavioural pattern implementations where "roles are *superimposed* (...) to classes which originally have functionality and responsibility outside the pattern" [Hannemann and Kiczales, 2002]. This is the case with the Observer pattern where, as already showed in Section 3.2.3.1 and Figure 3.1, GUI components are in addition assigned the role (and thus the responsibility) as Observers.

3.3 Metrics

The section presents some metrics to evaluate code quality. It should noted that the same metric abbreviations may occur in several of the Sections 3.3.x, but they do not necessarily mean the same. This is to stay coherent with any references and/or related tool implementations discussed later in Section 3.5.

3.3.1 Counting

To calculate various metrics, the counting of some relevant source code aspects is required. These numbers merely say something about the size of the system in question, not anything about complexity. However, it is of course reasonable to assume that a system with 10,000 classes is more complex than a system with 100 classes.

NOP, number of packages Most modern languages supports a packaging mechanism to help modularizing a system beyond the class concept (namespaces in C++, packages in Java, modules in Python, etc).

NOC, number of classes The number of classes of an object-oriented system.

NOM, number of methods/operations The number of user defined operations, methods, global functions etc. This excludes methods inherited from "stable" packages as, for instance, standard classes provided by the language, or released 3rd party dependencies. The system should specify the version of the dependency through the documentation, bundling, or dependency handling of the build system.

LOC, lines of code The amount of code lines, not counting comments and whitespace. Depending of the size of the system it is common to postfix the metric as for instance kLOC – thousand lines of code.

To which extent it would be useful at all, the sums obtained from the above metrics should be viewed with consideration to language abstraction level (e.g. Java is more high-level than C), code reuse, and other influencing factors. The

role of these numbers are first and foremost to take part in the calculation of other more quality characterizing metrics.

3.3.2 McCabe metric - Cyclomatic Complexity

In 1976, McCabe proposed a metric to measure complexity in terms of the amount of discrete possible operation paths of execution units of a system. This "unit" can for instance be a function/procedure/method or some other abstraction which has exactly one entry and exit point. Although McCabe at the time was probably addressing structural programming, this is still a valid quality metric in object-oriented programming, albeit on a micro level. Especially with a focus on automated unit testing and the code coverage of such tests, it may be interesting to be able to describe the *testability* of methods (i.e, units) using this metric. [McCabe, 1976]

3.3.2.1 Usage in brief

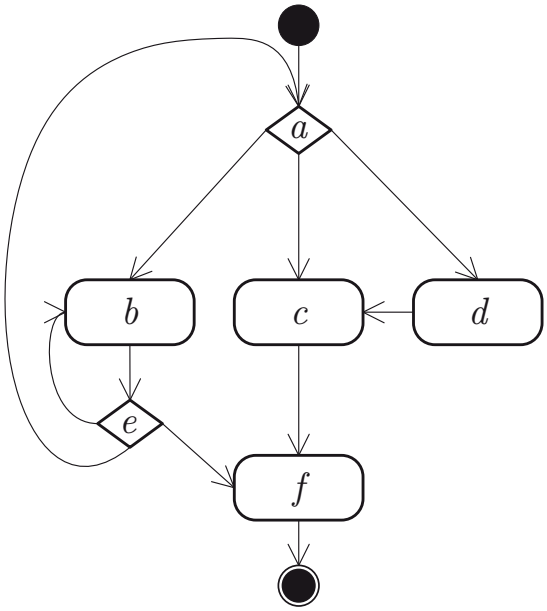


Figure 3.2: *UML Activity diagram of an arbitrary method.*

McCabe explains the *cyclomatic complexity number* using a standard graph containing nodes and edges (arcs). However, a UML Activity Diagram would be an applicable substitution by modern modeling standards. Figure 3.2 shows a simple example program flow of an arbitrary operation (same example as used in McCabes article [McCabe, 1976]). It consists of

- states, S (b, c, d, and f)
- decisions, D (a and e)

- control flows, F (not counting the flows connected to the start and finish points)

Using McCabe's definition, the cyclomatic complexity is calculated like this:

$$CC = F - (S + D) + 2p \tag{3.1}$$

In Equation (3.1), p is the number of connected components which is included in the calculation. If for instance the method in question also uses other methods, which internal operation is also included in the calculation, accordingly $p > 1$.

The Cyclomatic Complexity for the method shown in Figure 3.2 is calculated using Equation 3.1 like this:

$$CC = 9 - (4 + 2) + 2 = 5 \tag{3.2}$$

In Appendix A.1, the example operation used throughout this section is realized as a Java method. The Cyclomatic Complexity of this method has been verified to be 5 by using the Cobertura test coverage tool [Doliner et al., 2007].

3.3.3 Interdependency analysis metrics

An article published by Object Mentor describes metrics "to measure the quality of an object-oriented design in terms of the interdependence between the subsystems of that design." [Martin, 1994]

It is important to be aware of that the article by Martin does not cite any references. This is somewhat peculiar since the article refers to numerous already defined practices to ensure low coupling between components in its discussion toward the proposed "[...] design pattern in which all the dependencies are of the desirable form [... and the] set of metrics that measure the conformance of a design to the desirable pattern."

The metrics described in the article are discussed in the following sections and summarized in Table 3.1. JDepend, which is discussed later in Section 3.5.1, is an implementation of an analysis and assertion tool for these metrics.

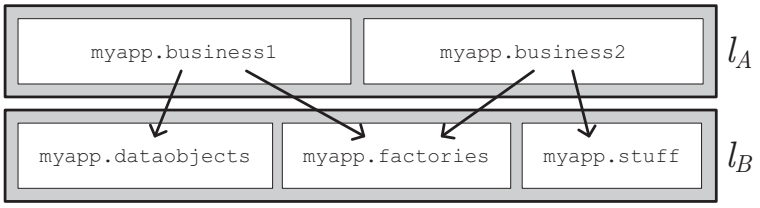
3.3.3.1 Required sums

As with the Cyclomatic Complexity metric discussed in Section 3.3.2, the metrics characterizing interdependencies also need some sums to use for further calculation.

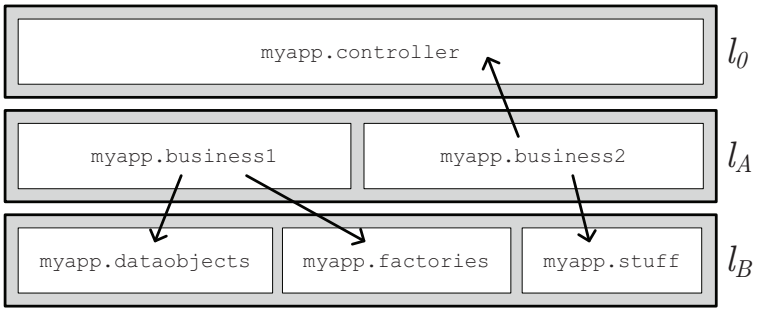
In addition to the total amount of classes of a system (TC, same as NOC in Section 3.3.1), this sum is grouped into concrete classes (CC), and abstract classes including interfaces (AC).

Metric	Abbr.	Description
Number of classes	TC	The total number of classes in a package.
Concrete classes	CC	The number of concrete classes.
Abstract classes	AC	The Number of abstract classes and interfaces.
Afferent couplings	Ca	The number of classes from other packages depending on classes in this package. This describes the package's responsibility.
Efferent couplings	Ce	The number of packages the classes of this package depend upon. This describes the package's independence.
Abstractness	A	The ratio of the number of abstract classes to total number of classes ($\frac{AC}{TC}$) to indicate the abstractness of a package.
Instability	I	The ratio of efferent couplings to total amount of couplings ($\frac{Ce}{Ce+Ca}$).
Distance	D	Martin describes the ideal package as when abstractness and instability sums up to 1 ($A + I = 1$). This metric is the distance from this ideal.
Package dependency cycles		Any dependency cycles are reported.

Table 3.1: *Metrics to describe interdependencies between components of a software system.*



(a) Properly implemented



(b) Implementation not adhering to the design

Figure 3.3: *The figure shows two different implementations of a layered architecture where both has the expected amount of efferent couplings in layer l_A ($Ce = 4$) though only (a) is correctly implemented.*

3.3.3.2 Afferent and efferent couplings

Afferent (Ca) and efferent couplings (Ce) describe dependencies between groups of classes, either by using the language's built-in package mechanism, or by grouping several packages into larger components.

Ca is the number of other packages depending on one package. This describes the responsibility of each package. A high number here may be an indication of bad design, or that the package is used for crosscutting concerns such as logging (see Section 3.5.3 for further explanation). In an MVC architecture, the View component should be depended on only by one component, the Controller, thus afferent couplings for the View should not exceed the number of packages which comprises the Controller component of the design.

Ce is the opposite of afferent couplings; it is the amount of other packages a package depends on. This describes the independence of the package. This can be used to point out non-adherence to the design if certain packages have a unreasonable high number of efferent couplings.

Still, the Ce-number is not enough to identify dependency flaws of an implementation not adhering to for instance a n-tiered top-down layered architecture where each layer is only allowed to access the immediate lower layer external to itself. [Bass et al., 2003, pages 62, 341–345]. If layer l_A (defined as x number of packages where $x \geq 1$) is designed to access layer l_B (defined as y number of packages where $y > 1$), it is highly possible to break a design while still keeping the number of allowed efferent couplings. Figure 3.3 shows two different implementations of a layered architecture where the efferent couplings of layer l_A are the same for both.

Looking at Figure 3.3 (b), it seems that two things may have happened:

1. The package `myapp.business2` accesses the higher level layer l_0 and thus creates an efferent coupling caught by JDepend.
2. The package `myapp.business2` looks like it may be instantiating classes from `myapp.stuff` directly instead of using the provided factories in `myapp.factories`. Thus it does not create the maximum amount of efferent couplings to l_B.

However, this is not information retrieved from the metrics, but for instance an old-fashioned architectural analysis, which has identified a design breach not detected by the use of metrics. This effectively renders the report on efferent couplings for layer l_A not usable.

These metrics are difficult to use as possible warnings unless they are extensively deviating from what is normal, and thus 'normal' also must be extracted from the defined architecture design. In a large-scale application, this low-level view of the amount of interdependent relations between packages may not be feasible to use as an accurate measure of design adherence. As new packages may often be added to the system, to what at the design level are considered components, any added package must be correctly defined as part of a component recognized by any tool calculating couplings of the system.

3.3.3.3 Abstractness and instability

The abstractness (A) is the ratio of the number of abstract classes to the total amount of classes ($\frac{AC}{TC}$) of an analyzed package. When $A = 1$ indicates a package containing only interfaces and/or abstract classes, and $A = 0$ means the package consists only of concrete classes. This number is a component in the calculation of the metric described in Section 3.3.3.4.

The instability metric (I) is an indicator of the package's resilience to change. It is the ratio of efferent couplings to the total amount of couplings ($\frac{Ce}{Ce+Ca}$). If a package is fully independent, i.e, it does not have any efferent couplings, the fraction yields ∞ but is usually considered 1 (for reasons explained in the Section 3.3.3.4), which means that external changes does not affect this package.

Both these metrics will usually be somewhere in between 0 and 1, and the next section explains the proposed optimal combination of these two metrics.

3.3.3.4 Distance from Main Sequence

Using the calculated values *stability* (I) and *abstractness* (A), it is possible to define an ideal relationship between those to numbers. This is called the "Main Sequence", and is per definition the ideal relationship when I pluss A equals 1. The distance from the ideal Main Sequence can thus be calculated like this:

$$D = \left| \frac{1 - (A + I)}{2} \right| \tag{3.3}$$

A component which has a distance from the Main Sequence far from zero should be refactored to meet the optimal balance between stability and abstractness.

3.4 Design Level Assertions

This section will explain the term design level assertion (DLA), its motivations, and how it relates to this thesis.

3.4.1 What are DLAs

When designing and implementing a software system, the developer(s) uses architectural patterns and rules to solve various design problems and to ensure that the design is adhering to certain quality attributes.

3.4.1.1 Verification of design adherence

A design can be verified using various visualizations, most commonly expressed as UML diagrams made during the planning and designing phase. However, it is not given that the diagrams reflect the actual implementation. Another way is of course to do an architectural analysis of the existing code. This should generate an accurate architectural view of the system enabling the identification of the original design (or lack of). This is of course a lengthy and costly process, as well as it involves a great risk of having to do extensive refactoring of the implementation each time the analysis is done. In fact, this concept of an implementation over time slowly moving away from its original design is so common that it has been properly named as *architectural drift* [Bass et al., 2003].

3.4.1.2 Verification – Automated

Design Level Assertions (DLA) is a concept of automated verification of design adherence. The term seems to originate from the people behind the Spring Framework [Interface21, 2000 – 2007, Spring], and is used to ensure that systems which are continuously pieced together using code from various corners of the world, still adheres to the original design [Arendsen, 2006].

In the same way as unit testing does assertions on isolated unit output, a DLA verifies a design aspect of a system. As already discussed in Sections 3.1 and 3.2, it makes sense to design systems using relevant patterns. Thus, a DLA tool should be able to verify the adherence to patterns which comprise the design of a system.

3.4.2 Motivations

Asserting certain output from functions and methods is a relatively easy binary metric of true or false, and thus makes unit testing an easy concept to grasp. The *design* of a system is however at a more abstract level and is thus not as easily codified into quantifiable tests. Another challenge is to be able to actually extract this kind of information from a codebase, ideally using the least possible amount of meta data.

Software quality, as discussed in Section 3.1, is achieved with a well-designed architecture, which "embodies decisions about quality priorities and tradeoffs [...]" [Clements et al., 1995]. Since system quality is evaluated on its architectural design, it is critical that the implementation actually adheres to its intended design. The problems involving design adherence and architectural drift are still mostly solved in a manual fashion. Although there are a few tools to assist in this work, they are still not used in common practice.

3.5 Tools and Practices

This chapter will describe some of the available tools and practices to help enforce design rules and constraints to support quality assurance (QA) in software projects.

3.5.1 JDepend

JDepend is a tool to perform measuring of the metrics described in Section 3.3.2. These metrics describe "the quality of a design in terms of its extensibility, reusability, and maintainability to manage package dependencies effectively" [Clark, 1999–2006].

3.5.1.1 "Include and forget"

When using JDepend, either by running it manually or including it in an automated build process, there is little control on what the tool should do. It is specialized on one thing, which is to calculate the metrics described in Table 3.1. It is possible to adapt it for a target system, although this is not required. This should make it a great tool to quickly get some hints on an implemented design with little extra learning overhead besides how to interpret the output metrics.

3.5.1.2 Configuration options

It is possible to adapt a JDepend report to a specific system. This includes

- defining several packages as a *component*,

- excluding packages or classes for being included in the report.

3.5.1.3 Relevance for DLA

As JDepend is merely a metric tool, it obviously cannot be used to enforce adherence to a specific design, but to identify flaws in the general object-oriented design. If JDepend outputs irregular metric results, there is something wrong with the way the classes, packages, or components of the system are structured. Importantly, this is given that the developers and architects acknowledge the validity of the McCabe metric for the system in question.

See the Evaluation part, Part III, for a discussion on how metric tools as JDepend indeed can be relevant for DLA when used in combination with a DLA tool.

3.5.2 Macker

Macker [Cantrell, 2003] is a tool to do specific assertions on how the various classes and components of a Java software system accesses each other, and thus how they depend on the presence of other classes.

Macker is usually incorporated into an automated build-process. It comes with a preset task specification to be included in an Ant build [Ant], and it is also possible to configure a Maven project [Maven] to run Macker[4].

3.5.2.1 Rulesets

The access rules are defined using XML, and Macker uses compiled classes to check if these rules are adhered to by the code. The rules are specified on class level with the possible use of wildcards to specify several classes within a certain package, or by following a naming convention.

The `macker.xml` file contains rules for a project. Although it is possible to explicitly refer to classes (with or without the use of wildcards), it should be considered good practice to define *patterns* which are used to create reusable entities to be used in access rules. Listing 3.1 shows how to define the set of classes in the view sub-package of a codebase to be later referred to as "`view`".

```
7       <pattern name="view">
8          <include class="**.view.**"/>
9       </pattern>
```

Listing 3.1: *A Macker pattern defining which classes belongs in the "view" component.*

[4] A proposed configuration has been contributed to the Maven community by this project, and is available here: `http://docs.codehaus.org/display/MAVENUSER/Running+Macker+with+Maven+2`.

Further, an access rule to help enforce adherence to the Model-View-Controller pattern can be defined as showed in Listing 3.2.

```
23        <access-rule>
24           <message severity="error">View breaks the MVC pattern!</message>
25           <deny>
26              <from pattern="view"/>
27              <to pattern="all-but-view"/>
28              <allow>
29                 <to pattern="logging"/>
30              </allow>
31           </deny>
32        </access-rule>
```

Listing 3.2: *A Macker access rule which effectively codifies the MVC constraints of the View component: the View is independent and is not allowed to access business logic of the system.*

The examples in Listings 3.1 and 3.2 are taken from the full Macker configuration example found in Appendix A.2.

3.5.2.2 Reporting

In addition to render a build invalid, and optionally in consequence halting it, Macker has the ability to create a report on the adherence to a rule-set. The report can be output to XML or plain HTML, which is viewable using any web browser. The report shows custom error messages on any violations on the specified design rules, together with the involved classes causing the violation.

3.5.2.3 Weaknesses

It has not been released any updates for Macker since 2003. There seems not not be much activity with the development, and although the source code management log shows a submit in May 2007, this is has been the only update in a year. The project

Macker is still in early development, though it is very usable for the functionality it currently implements. The rules are specified on the class-level, with no template support for making reusable patterns on a higher abstraction level. The expressiveness of the rules are thus relatively fine-grained, and this may force a lengthy process of codifying a design into Macker rules.

3.5.3 AspectJ

AspectJ is an implementation of an aspect-oriented programming (AOP) extension to Java. A description of AOP is beyond the scope of this thesis, but a brief overview has already been given in Section 3.2.4.2.

3.5.3.1 Enforcement of architectural rules

Clement and Kersten give an example of utilizing AspectJ for enforcing architectural rules of an implementation. Arendsen[5] terms this practice in general as Design Level Assertions, and also uses AspectJ examples for this [Arendsen, 2006].

Since AOP is about system-level requirements, it enables the developer to clearly define system-level entities of a codebase, i.e, entity abstractions which may comprise several classes and/or packages. Using AOP, it is possible to formally define components, layers, or other architectural or *design level* entities of a system. AspectJ also gives explicit language support for declaring warning and error conditions which are caught by the compiler, much in the same way as pure syntax errors or any other checks which is statically built into the compiler (e.g, checking for unreachable code, operations on possibly uninitialized variables, etc). With AspectJ, it is possible to define custom checks and policies which are done at compile time, and these are specified in the source code itself, giving the benefits of a typed language.

3.5.3.2 Immediate feedback

Given that a proper design definition is created in AspectJ, using integrated development environments (IDEs) such as Eclipse, which supports incremental compilation, a developer will get immediate feedback the instant a statement which breaks the intended design is written. This is shown in Figure 3.4.

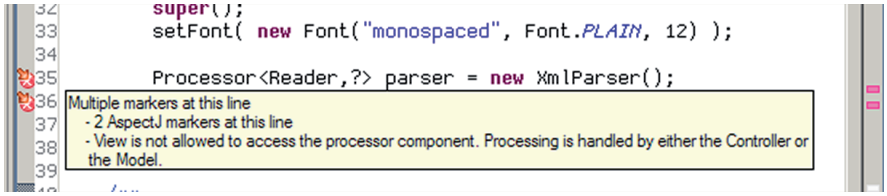


Figure 3.4: *The screen-shot shows the Eclipse IDE giving an immediate error message when trying to instantiate a class which should not be used from the GUI component.*

3.5.3.3 Possible incompatibility issues

The fact that AspectJ offers an extension to the standard Java syntax, but still produces Java-compliant bytecode which is executed by the standard JVM, it will break the usual relation between source code and compiled classes. This may create a potential incompatibility issue with other tools which expects a $1:1$ map between source classes and compiled classes.

[5] In addition to his blog entry [Arendsen, 2006], the project has also had email correspondence with Arendsen to get views and discussion on DLA with AspectJ.

Tools which do some kind of reverse-engineering may for instance look for compiled class-files as well as the source code for any complementary data. Since the compiled class files are not the result from a corresponding number of source classes, but includes aspect sources which are not compatible with Java syntax, such tools can run into problems. One possibility to resolve this issue would be to define exclusions of the specific classes from AspectJ compilation, provided the tool in question offer this kind of functionality.

3.5.4 Other tools

While researching, the project have come across several other interesting tools as well, but due to time-frame limitations was not able to assess them (besides a minor verification example using Cobertura in Section 3.3.2). They are mentioned here as suggestions for others to investigate their appliance for DLA.

Cobertura is a test coverage tool, measuring how much of a software system's code is accessed by unit tests. In addition, the tool shows the average complexity of classes and packages using McCabe's Cyclomatic Complexity metric, already discussed in Section 3.3.2 [Doliner et al., 2007].

SonarJ is an architecture enforcement tool, which seems to be very comprehensive. It is able to check if an implementation adheres to a logical architecture specified in XML, and exists as a standalone program, Eclipse plug-in, and recently also a Maven 2 plug-in [SonarJ].

QALab works as a collator for the output of several metric tools, and is able to show how these metric results change over time. This enables identification of how various quality aspects of the system will progress, and making it possible to take preventive actions on any negative trends [QALab].

Part II

Contribution

Chapter 4

Task

This chapter discusses possible contributions that may be a reasonable 'next step' to conduct in consequence of the review presented in chapter 3.

4.1 Build-time DLA tool

The project has chosen to make a contribution to the problem area by making DLA tool which should be invoked by a build system. The tool is based on

XML for the DLA rules language [W3C XML],

AspectJ language and compiler for the DLA engine,

XSLT for the transformation from XML into AspectJ code [W3C XSLT].

This strategy has been to employ the power of the enforcing capabilities of AspectJ together with the possibility to use custom constructs in XML to be able to define rules at an abstraction level above what is possible in AspectJ. Since there is a risk of loosing expressiveness when moving to a higher level of abstraction, there is also the possibility to define DLA directly using the AspectJ language.

The design of the DLA tool is discussed in more depth in Chapter 6.

4.2 Another considered option

Another evident possibility has been to create a GUI-based tool, for instance an Eclipse plug-in to assist in defining DLA directly in the IDE, and providing feedback while the developers are doing the implementation. Still, the project has considered that a build tool will provide greater flexibility, especially since

the AspectJ source code for the tool can at the same time be used by Eclipse's existing functionality to give immediate feedback as already shown in Figure 3.4.

Following Arendsen's advice, for projects reaching a certain size, there may be a performance issue with having Eclipse constantly executing DLA on the project code, and providing feedback to the user.

Chapter 5

Target platform

The DLA tool should be developed for an already commonly used development tool for easy integration with projects. The chosen platform is Maven 2 because of its automatic treatment of external dependencies and its reporting framework. Maven 2 is also used by BEKK to handle many of their projects.

Maven 2 will further on be referred to as only "Maven" without any version indicator. At the time of writing, the latest stable version of Maven is version 2.0.6, which is the version used throughout the thesis work. [Maven]

5.1 Overview

Maven is presented as a "software project management and comprehension tool" on its website [Maven].

It is a *project management tool* in that it handles compiling, testing, resources, deployment, and other aspects of building a software system from sources. This functionality is analogous to other build systems, such as Ant [Ant], or the much older Make [Make].

Maven is a *comprehension tool* in that it adds the concepts of patterns and best practices to the build phase of software development to place emphasis on uniform project structures and built-in conventions. In addition, it contains extensive automated reporting functionality to further support the understanding of a software project. The reporting is discussed in more detail in Section 5.2.4.

5.2 Notable features

This section elaborates on the features which believed justified Maven as the platform of choice for this contribution.

5.2.1 Plug-in execution framework

Maven can in some respect be viewed as a plug-in execution framework [Sonatype, 2007], as all functionality of Maven is provided by plug-ins, and the Maven core itself is mostly responsible for providing information from the Project Object Model (POM, see Section 5.2.2) needed by the plug-ins' execution.

This architecture, together with Maven's automatic handling of dependencies (Section 5.2.3), enables highly portable projects as Maven will download whatever functionality needed if not already present on the target machine.

5.2.1.1 Build lifecycle

Maven uses the concept of a well-defined *lifecycle* for its build operation, consisting of several *phases* which are executed in sequence. The lifecycle of a specific build task can be described as the sequence of phases which comprises the build.

For instance, two common lifecycle phases, which should be self-explanatory, are *compile* and *test*. The compile phase should be used for compiling the source code into binary artifacts (e.g, classes or native executables), whereas the test phase is used for running for instance automated unit tests. The build lifecycle specifies that the compile phase precedes the test phase, since most tests (for instance unit tests) are performed on compiled classes.

Maven is mostly run by instructing it to execute a specific phase. To execute a phase requires successful execution of all its preceding phases. This ensures that when for instance the developer wants to run unit tests, it is sufficient to indicate the test phase using the command `mvn test`, and all of the preceding phases of the build lifecycle will also be executed, among them the compile phase which enforces that all classes are up to date before any tests are conducted.

For a complete reference of the various phases of the Maven build lifecycle, see the documentation on the Maven website [Maven].

5.2.2 The project descriptor

Maven uses another approach to automated building than Ant, Make, or similar tools. Where as the latter uses procedural descriptions (scripting) on how the build tasks of a project should be performed, Maven has a declarative approach to describe a project using a project object model (POM). The POM states various aspects about a project, and the build tasks are carried out by plug-ins which are of a generic nature and should be applicable to any projects needing that certain plug-in functionality.

The separation of the declarative POM and the actual task execution in plug-ins, enables clean build definitions (project descriptions are actually a more correct term, as a POM does not really say anything about how the various build tasks are executed) for common project types, as well as it encourages the community to

develop reusable task components (plug-ins) that others may use in their projects as well.

Since Maven emphasizes a uniform project structure, it has certain built-in conventions. If a project follows these conventions, it is not required to specify at all, or a minimum of information is required to be specified, by the POM. As an example of this, see Listing 5.1. Given that a project adheres to Maven's expected directory structure, these 14 lines will suffice to enable source compilation, automated unit testing, packaging to JAR-file and even the creation of a website containing basic project reports. In addition, it will also automatically download the dependent JUnit library required to perform the unit tests.

```
1  <project>
2    <modelVersion>4.0.0</modelVersion>
3    <groupId>com.mycompany.app</groupId>
4    <artifactId>my-app</artifactId>
5    <version>1.0-SNAPSHOT</version>
6    <dependencies>
7      <dependency>
8        <groupId>junit</groupId>
9        <artifactId>junit</artifactId>
10       <version>3.8.1</version>
11       <scope>test</scope>
12     </dependency>
13   </dependencies>
14 </project>
```

Listing 5.1: *A small POM which enables Maven to perform for instance compilation, unit testing with JUnit, packaging to JAR-file, and creation of a website with basic project reports.*

5.2.3 Dependency handling

Lines 6–13 in Listing 5.1 shows another one of Maven's strength. A project's POM may define any dependencies on other libraries, and these will be automatically downloaded if not already present on the system.

In addition, Maven also supports *transitive dependencies* which enables the developer to mind only the libraries which a project directly depends on. Any dependencies of the dependent library are automatically downloaded and handled transparently by Maven.

5.2.4 Reporting

Automated generation of reports and documentation is an important part of Maven's functionality, and one of the most important reasons for why many developers choose Maven as their project management tool[1].

[1] In several forums and other Internet channels, "Maven evangelism" is a recurring term.

Generating a full-fledged website containing documentation on a project is a simple matter of running the command `mvn site`. The contents of the resulting website depends on what reporting plug-ins is specified in the POM.

Chapter 6

The DLA Maven Plug-in

This chapter presents the contribution artifact of this project, including its design, how it is used, test cases, and encountered challenges.

6.1 Design

This section explains how the developed DLA Maven Plugin is designed. Some design decisions comes as a consequence of the chosen target platform, already discussed in Chapter 5, and the platform's development framework.

6.1.1 Overview

The realization of the DLA Maven Plugin is grounded in an idea of combining the familiar syntax of XML with the power of the AspectJ language and compiler to achieve an automated DLA tool which can both invalidate builds based on assertion results, as well as present a report on the implementation's adherence to its intended design. The hypothesis is that by creating a "map" from a custom developed XML language to a relevant subset of the AspectJ language, which in particular comprise declarations of warning and error conditions, will simplify the introduction of DLA to projects. Proposed arguments for this hypothesis are:

- XML has become a very common, if not the most common language for various descriptive purposes in combination with modern programming languages, frameworks, distributed architectures, etc, and is therefore already a well-known concept for most programmers [Daconta et al., 2003, Chapter 3].
- The creation of a specialized (domain-specific) language will eliminate the presence of redundant language constructs if instead there was a general purpose language as AspectJ to use for DLA.

- If building a language on top of another, it should be easy to create abstractions of commonly used constructs for the domain.

Of course, there are also possible problems with this approach:

- To develop a new language may be costly.
- XML has received some criticism for being a too verbose for manual editing[1].

To accommodate with the possible problems mentioned above, the focus has been to create a solution which offers both AspectJ and a custom XML language to perform DLA on a project. The XML language used by the DLA Maven Plugin is currently at an early prototype level. The focus has been to "XML-enable" the plug-in for simple test-purposes in relation to this thesis, and arrange for easy further development of the language.

6.1.2 Configurability

A tool which should be applicable to arbitrary projects must offer a satisfactory amount of configuration options to make it possible to adapt its operation. Maven handles configuration of plug-ins automatically using a configuration section for each plug-in in the POM. Each element corresponds to a field in a MOJO class. See Section 6.1.4 for an explanation on how the fields are populated with objects using either configured or default values.

There are several options available:

`failOnWarning` If the plugin should fail the build if a *warning* message occurs. Defaults to `false`. If this is set to `true`, then `failOnError` is not regarded as it is of higher severity and will implicitly always be `true`.

`failOnError` If the plugin should fail the build if an *error* message occurs. Defaults to `true`.

`dlaFile` The filename to use for the DLA rules.

`sourceDirectory` Directory where the DLA rules file is located.

`enableAspectJ` To include the generated AspectJ sources with other aspects of a project (for instance, the project uses AspectJ for general AOP), set this option to `true`. This is also applicable in situations when needing to define DLA directly using AspectJ instead of XML in the `dlaFile`, or even a combination of the two languages. The generated AspectJ sources, if any, will be placed in the source directory instead of the build directory.

[1] http://www.xmlhack.com/read.php?item=1213

aspectDirectory Specifies a directory to put generated AspectJ sources if the need to use them external to the plug-in is present. This option is only regarded if `enableAspectJ` is set to `true`, though it should not be required, since it defaults to the directory expected by the AspectJ Maven Plugin[2].

javaSourceDirectory The directory where the project's Java sources are located.

compileTarget The directory where compiled classes from the AspectJ compiler will be placed.

A full configuration showing all the available options with their default values is included in Appendix A.4. Several of the options should not be necessary to change from their defaults at all. By following the project conventions expected by Maven, the most important options is to configure how Maven should react to the various severities of design violations; `failOnWarning` and `failOnError`. One highly likely setting would be to specify `failOnError` as `false` if it is not desirable to ever fail the build, only to report on the DLA outcome. Also, to specify a strict DLA policy, set `failOnWarning` to `true`.

It is possible to use the plug-in without making any configuration as well. Even though there is possible to do extensive adoptions of the plug-in's operation, each setting not explicitly specified is set to fall back to defaults which is coherent with the recommended Maven project conventions[3].

6.1.3 MOJOs

A Maven plug-in consists of one or several so-called MOJOs[4]. A MOJO is the entry point of execution for one specific task of a plug-in. A plug-in may indeed consist of one single MOJO regardless of the complexity of the plug-in's functionality, though Maven's build lifecycle encourages to break the functionality into a logical sequence of discrete subtasks. The way these subtasks (i.e. MOJOs) are sequenced is simplified by using phases of the build lifecycle.

The DLA Maven Plug-in implementation enables its behaviour to be adapted in such a way that the user may for instance choose to execute only the `DesignVerifierMojo` without having the plug-in to generate AspectJ source code, which is intended to happen in the `AspectGeneratorMojo`. This may indeed be appropriate if the user is already familiar with using AspectJ for design level assertions and only wishes to use the plug-in to automatically carry out this verification during build-time.

Another possibility is to actually combine the two ways to define DLA. It is indeed possible to use both XML and AspectJ to define the assertions. In this case, it is

[2]http://mojo.codehaus.org/aspectj-maven-plugin/

[3]http://maven.apache.org/maven-conventions.html

[4]Maven plain Old Java Object. Technically, a MOJO is a Java class which implements the `Mojo` interface, most commonly by extending the `AbstractMojo` class.

important to not manually maintain source aspects using the same file name(s) as output by the plug-in as it will overwrite without asking any existing AspectJ source files having the same file name as it tries to generate.

6.1.4 Dependency Injection

Maven is built on a lightweight framework utilizing the inversion of control (IoC) pattern[5]. There is really nothing special about this pattern, and it was already discussed by Johnson and Brian in 1988 as means to create reusable classes [Johnson and Brian, 1988]. Newer frameworks are commonly characterized using the IoC term almost as an embellishment[6], and it simply says that instead of that the developer controls when operations should happen, it is the framework which controls this. The developer extends framework classes, implementing *how* operations are conducted, and the framework is responsible for invoking the operations at appropriate system states. This is one evident characteristic which separates a framework from a library. [Fowler, 2004]

The MOJOs of a Maven plug-in are invoked using the `public void execute()` method which any MOJO class must implement, and is trivial enough. In addition, a MOJO must obviously have access to information of the current project the MOJO is executed on. This is also handled by the framework using a specialized form of the IoC pattern called *dependency injection*. The IoC framework is responsible for injecting the MOJO class with required objects, i.e, the MOJO can expect to have these objects available, but neither the MOJO or any other of the developed plug-in classes need to do anything themselves to obtain these objects.

```
21  /**
22   * The current project.
23   *
24   * @parameter expression="${project}"
25   * @required
26   * @readonly
27   */
28  protected MavenProject project;
29
30
31  /**
32   * The directory where the compiled classes and aspects are put.
33   *
34   * @parameter expression="${project.build.directory}/rune"
35   * @required
36   */
37  protected File targetDirectory;
```

Listing 6.1: *Two examples of dependency injection to fields of a MOJO class. The first field, `project`, is populated by an object model of the POM. The second, `targetDirectory`, is constructed using the build output directory specified in the POM with an appended subdirectory, and injected as a File object.*

[5] http://plexus.codehaus.org

[6] Again, buzzwords are common in information technology.

The injection is configured using plain javadoc annotations for fields in each MOJO class. Listing 6.1 shows the power of Maven's dependency injection mechanism in that injected objects are specified using expressions indicating values from the POM, and especially the ability to construct objects by combining dynamic project values with static plug-in values. The `targetDirectory` specifies an own subdirectory for artifacts created by the MOJO located wherever the current project has configured its output directory. This is a very elegant way to ensure that plug-ins are made generic and usable for any project. In addition, `targetDirectory` is configurable in the POM should the default value not be appropriate for a particular project. The `project` object is however annotated with `@readonly`, making it impossible to configure it.

The snippet in Listing 6.1 is an excerpt from a complete MOJO class definition. The full source code can be found in the appendix section A.3.

6.1.5 Part of build lifecycle

Figure 6.1 shows a UML sequence diagram of how Maven's lifecycle phases interact with the plug-in's MOJO classes. "Typically, an interaction diagram captures the behavior of a single use case" [Fowler and Scott, 2000]; the use case in question here being *verify design adherence*.

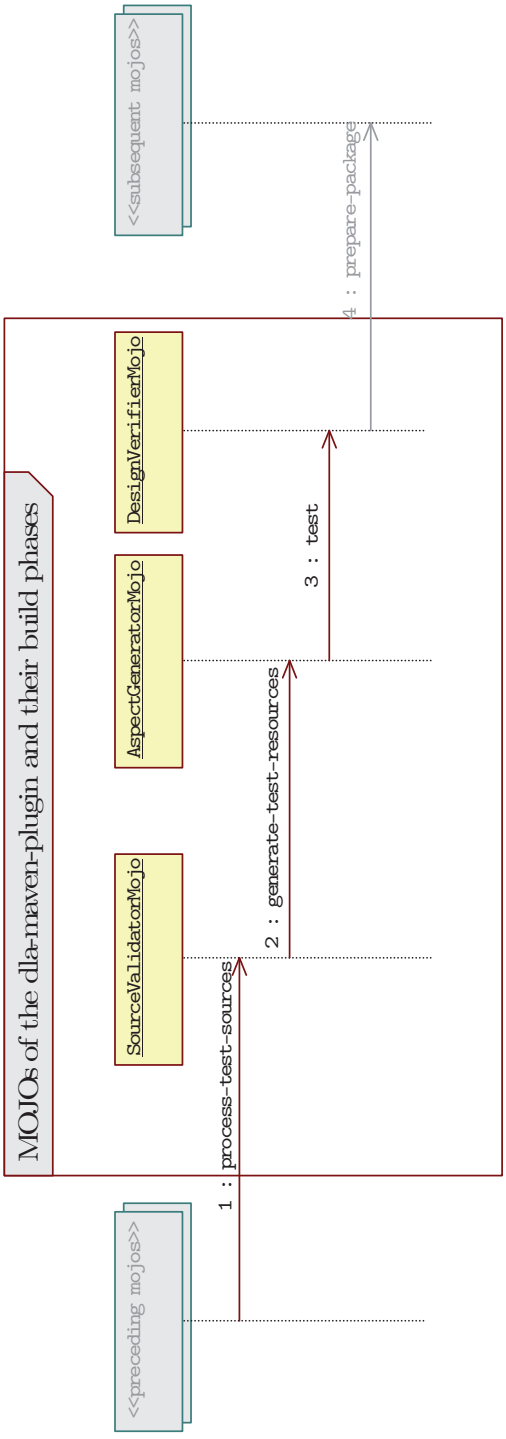


Figure 6.1: *UML sequence diagram showing which phases of the Maven build lifecycle the different MOJOs are bound to. MOJOs are indicated by objects, and the messages between objects are phases triggering the execution.*

As explained in subsection 6.1.3, MOJOs are discrete subtasks which together comprise the intended purpose of a Maven plug-in. The MOJOs are in principle independent of each other, which also makes this plug-in architecture appropriate for prototype implementations such as for this thesis. The MOJO classes does not need to be production-ready code, but it is simple to get the MOJOs up and running by offering a convenient modularization of the overall task. This simplifies further development of the prototype.

6.2 Test Cases

The conducted test cases emphasize the various scenarios that is possible to configure the DLA Maven Plugin for. The plug-in is actually tested on itself, i.e. it is used to enforce design adherence on the DLA Maven Plugin project.

The test scenario tables in the following sections states

- Test result: either PASS or FAIL
- Description: a textual description of the scenario(s)
- Output #: Console output from the DLA Maven Plugin
- Build execution: either SUCCESS or FAIL (Note: this does not indicate if the test scenario was successful, but specifies Maven's handling of the situation. A fail may be expected depending on the DLA specification and how the code adheres to it)

The first scenario (Default configuration, Section 6.2.1) contains a textual description of how the test was conducted, while the remaining tests only presents the results. Refer to Appendix A.4 for how to configure the DLA Maven Plugin for each described test.

6.2.1 Default configuration

The default configuration is the way the plug-in will execute when defined to be included in the build process without any explicit configuration. Listing 6.2 shows how the plug-in is configured to be executed as part of the default build lifecycle in Maven.

```
1  <build>
2    <plugins>
3      <plugin>
4        <artifactId>dla-maven-plugin</artifactId>
5        <groupId>no.rune.mavenplugins</groupId>
6        <version>0.1-SNAPSHOT</version>
7        <executions>
8          <execution>
9            <goals>
10             <goal>generate-aspects</goal>
```

```
11              <goal>verify</goal>
12            </goals>
13          </execution>
14        </executions>
15      </plugin>
16    </plugins>
17  </build>
```

Listing 6.2: *DLA Maven Plugin default configuration*

In addition, the DLA specification in Listing 6.3 was placed in the file `src/test/dla/dla.xml`, relative to the project's base directory, which is its default location expected by the DLA Maven Plugin.

```
1  <?xml version="1.0" ?>
2
3  <dla>
4     <component name="mojos">no.rune.mavenplugins.dla.*Mojo</component>
5
6     <rules>
7        <independent>mojos</independent>
8     </rules>
9  </dla>
```

Listing 6.3: *Sample `dla.xml` file containing DLA specifications.*

The DLA specification defines a component called `mojos` (line 4), and this component is declared as *independent* in the `rules` section (lines 6-8), meaning that the component is not allowed to be accessed by other components.

Summary

Test result:	**PASSED**
Description:	DLA was executed on both design adherent code, and code with intentionally non-adherent statements, shown as respectively Output 1 and Output 2.
Output 1	
`[INFO] Testing design adherence`	
Build execution:	SUCCESS
Output 2	

`[INFO] Testing design adherence`
`[WARNING] Violation of mojos component's independence!`
`[WARNING] constructor-call(void no.rune.mavenplugins.dla.AspectGeneratorMojo.<init>())`
`[WARNING] Violation of mojos component's independence!`
`[WARNING] method-call(org.apache.maven.plugin.logging.Log`
`no.rune.mavenplugins.dla.AspectGeneratorMojo.getLog())`
`[WARNING] There were design violations!`	

Build execution:	FAIL

Table 6.1: *DLA Maven Plugin test scenario: Default configuration*

6.2.2 Manual aspect editing

Test result:	**PASSED**
Description:	To be able to supply the plug-in with manually coded aspect, it is necessary to set `enableAspectJ` to true. This will make the plug-in pick up aspect code from a source directory instead of just temporary storing the aspects generated from a `dla.xml` file. The AspectJ code in Appendix A.5 is used as DLA specification.
Output	
`[INFO] Testing design adherence`	
Build execution:	SUCCESS

Table 6.2: *DLA Maven Plugin test scenario: Manual aspect editing*

6.2.3 Using both XML and AspectJ

Test result:	**PASSED**
Description:	The previous section used only AspectJ source code for DLA. It should be possible to also use DLA definitions in the `dla.xml` file without further configuration. The result should be that both DLA are executed seamlessly.
Output	
`[INFO] Testing design adherence`	
Build execution:	SUCCESS

Table 6.3: *DLA Maven Plugin test scenario: Using both XML and AspectJ*

6.2.4 Using the plug-in with an AspectJ project

The scenario in Table 6.4 is specified as only PARTLY PASSED, because the handling of AspectJ source files are not optimal in this case. The design enforcement is seemlingly executed correctly, but the additional aspects are also included in the DLA Maven Plugin's execution, though they do not generate any output. This generates an potentially substantial extra overhead for larger AspectJ projects, and should be possible to avoid by configuring DLA Maven Plugin to only include generated and/or explicitly specified aspects.

Test result:	**PARTLY PASSED**
Description:	This is an interesting case because projects utilizing AspectJ for general AOP and also using the DLA Maven Plugin for design enforcement, this will result in the AspectJ compiler being used twice during the build. First of all for weaving the software system aspects with the regular Java classes, and secondly to perform DLA. It is important that these two uses of AspectJ does not interfere with each other, especially since the DLA Maven Plugin's default aspect directory is set to the same as the aspect directory for AspectJ AOP projects with Maven.
Output	
`[INFO] Testing design adherence`	
Build execution:	SUCCESS

Table 6.4: *DLA Maven Plugin test scenario: Using the plug-in with an AspectJ project*

6.3 Challenges

Although Maven 2 is presented using promises on how it will impose best practices on the build process, the project have encountered some serious challenges which probably will face other developers wishing to extend Maven's functionality as well.

6.3.1 Poor documentation

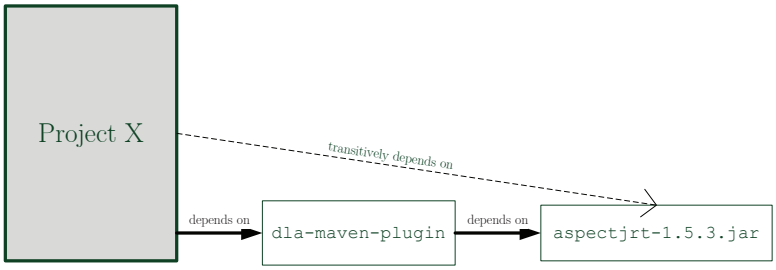
There exist a few tutorials on how to write plug-ins for Maven. They tend to focus on the requirements as to how to make them recognizable for Maven as plug-ins, as well as elaborate on the concept of the build lifecycle (see Section 5.2.1.1). These issues are essential to comprehend for the plug-in development. In addition to a few books [Massol et al., 2006, Sonatype, 2007], there exist some smaller guides on the Maven website [Maven].

What these texts generally leave out is to elaborate on the large object model which is needed to access information on the executing environment and current project. The javadoc reference for this API does not contain much additional documentation besides the extracted class names, method signatures, fields, etc.

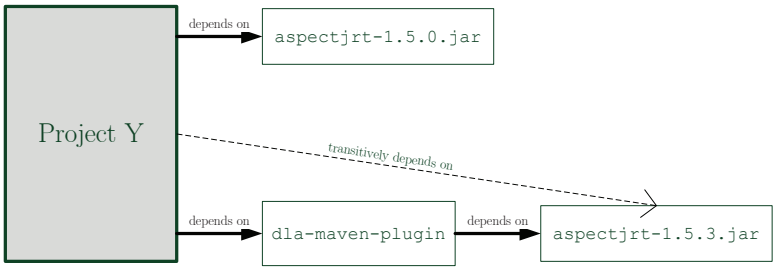
6.3.2 Case: Retrieving the path for a dependency artifact

For this section it is assumed a familiarity with basic Java development and terminology.

One particular challenge of the development has been that the AspectJ compiler requires a certain library (`aspectjrt`) to be specified on the `-classpath`



(a) Project X uses `dla-maven-plugin` which in turn depends on the `aspectjrt` library, making it a transitive dependency of Project X.



(b) Project Y uses `dla-maven-plugin` resulting in the same dependencies for Project Y as Project X. In addition, Project Y also directly depends on `aspectjrt`, but another version, resulting in dependencies on versions 1.5.0 *and* 1.5.3.l Y

Figure 6.2: *The figure shows two different dependency scenarios for a project using the* `dla-maven-plugin`.

argument passed to the compiler. Now, this may not immediately seem as a problem. Because as previously stated, Maven features an automatic handling of dependencies which ensures that all required libraries are present locally, both direct and transitive dependencies. This means it is a simple matter of defining the required library as a dependency in the project descriptor (POM) of the plug-in, which in turn will make it a transitive dependency of any Maven project which uses the plug-in, and Maven will make sure it is downloaded to the local system if not already present and properly added to the classpath when executing the plug-in. This scenario is showed in Figure 6.2 (a).

The problem with the AspectJ compiler is that even though it is written entirely in Java, and it is thus possible to interface to it as with any other Java API, it still acts as a standalone program which requires its own specified classpath to use for the compilation task. Consequently, it is necessary to get the absolute path of the plug-in's dependency on `aspectjrt`, which at runtime will be a *transitive* dependency of the current project.

A Maven plug-in has access to all information of the current project, which also means it can resolve the absolute path to both direct and transitive dependencies. It is desirable to resolve this path using the name of the library, namely "aspectjrt", and accept whatever version which is defined as dependency of the plug-in.

It is possible to iterate all dependencies of a project, and in the case of Project

X in the scenario showed in Figure 6.2 (a), this is guaranteed to find the particular `aspectjrt` version which the plug-in is depending on. However, in Figure 6.2 (b), Project Y directly depends on what happens to be another version `aspectjrt` than `dla-maven-plugin` depends on, and thus making both libraries dependencies of Project Y. When iterating Project Y's dependencies, it is not guarantied which version the first occurrence of `aspectjrt` is retrieved. This may not affect the execution of the AspectJ compiler at all, but it is still not optimal.

The book *Better Builds with Maven* mentions how to inject a `PluginDescriptor` object for the plug-in to access meta data on itself, for instance its dependencies [Massol et al., 2006, Appendix A.2.2]. However, this class is at the time of writing marked as *deprecated*, meaning that it is considered obsolete and should not be used. In addition, if ignoring the deprecated annotation and trying to utilize it for the `dla-maven-plugin` it yields an error message.

After extensive use of a common web search engine, a solution was finally encountered in a Frequently Asked Questions (FAQ) document [Zyl et al., 2007] (not even referred to from the official Maven website). The document gives an example on how to inject information on the plug-in's dependencies, and the IoC configuration is showed in Listing 6.4.

```
103     /**
104      * The plugin's dependencies.
105      *
106      * @parameter expression="${plugin.artifacts}"
107      * @required
108      * @readonly
109      */
110     protected List<Artifact> pluginDependencyArtifacts;
```

Listing 6.4: *Dependency injection of a Maven plug-in's dependent artifacts.*

As shown in Listing 6.4, `pluginDependencyArtifacts` will be populated with the dependencies of the running plug-in. It is now trivial to resolve a particular plug-in dependency based on its name/`artifactId`, which is showed in Listing 6.5.

The listings 6.4 and 6.5 are excerpts taken from the `DesignVerifierMojo` class of the DLA Maven Plugin implementation.

This is a remarkable trivial solution which concludes hours of searching for answers, trial and error with traversing the mostly undocumented Maven API. To add to the challenges regarding the documentation discussed in the past sections, the existing documentation also suffers from heavily decentralization, as well as literature as new as from 2006 does already contain obsolete information. However, on a positive note, this does indicate a very active development of the Maven tool.

```
268     /**
269      * Get the file in the local repository for a dependency of
270      * this plugin.
271      *
272      * @param artifactId The artifact to look up
273      * @return The file located in the local repository, or <code>null</
             code>
274      * if the dependency was not found.
275      */
276     private File getPluginDependencyFile( String artifactId )
277     {
278         for( Artifact artifact : pluginDependencyArtifacts )
279             if( artifact.getArtifactId().equals( artifactId ) )
280                 return artifact.getFile();
281 
282         return null;
283     }
```

Listing 6.5: *Retrieving a particular dependency artifact from a plug-in's set of dependencies*

Part III

Evaluation

Chapter 7

Discussion

7.1 The role of metrics

Metrics may be good indicators to discover weak object-oriented code, or code which does not testify to optimal object-oriented design. However, as demonstrated in Section 3.3.3.2 and Figure 3.3, they are not necessarily sufficient to discover implementation details which are major design violations if they are not having negative effect on the generic metric.

Used in combination with DLA, metrics are relevant to spot suboptimal solutions of a design. If a DLA tool has proved that an implementation adheres to the intended design, suboptimal results from a metric may indicate flaws in the design. This may be summarized using the following premises:

Premise 1 *Optimal DLA results means that the implementation adheres to its intended design.*

Premise 2 *Optimal metric results means that the implementation follows general good object-oriented principles.*

Then, following Premises 1 and 2, if an implementation gets optimal metric results *and* optimal DLA results, then one can conclude that the intended *design* follows good object-oriented principles.

7.2 Limitations of AspectJ

At the time of writing, there are some pattern enforcements which seems difficult to achieve using AspectJ, in particular if one needs to distinguish interfaces from concrete classes in general.

One of the failed attempts to codify DLA into AspectJ code includes the design pattern, or rather a design principle which is presented in a paper by Henkel. It

discusses a subset of the Java language called Constrained Java "which does not allow public classes to be used as static types in any piece of code that belongs to a set of packages, defined as the *Constrained Java Package Space*. Instead of classes, only interfaces are are used. The enforcement of this rule leads to a very clear distinction between the type hierarchy and the implementation hierarchy." [Henkel, 2000]

Constrained Java could indeed be a relevant rule to implement for the DLA definition language on top of AspectJ, but it would be difficult to translate to AspectJ code without imposing some conventions on the programmers to differentiate interfaces from classes. AspectJ in its current version cannot differentiate concrete classes from interfaces; it handles both as a *type*. This is verified by Ramnivas Laddad, the author of AspectJ in Action (see Appendix C.2).

7.3 Research evaluation

This section will address the research questions stated in the Introduction chapter, Section 1.2.2.

7.3.1 RQ1 – How may the use of DLA improve project efficiency?

The project has not directly been involved in a development project using DLA, but some evaluation can be done in consequence of the thesis dissertation.

Depending on the technology used for DLA, there are both possibilities and limitations for enforcing an implementation to adhere to a design. The use of AspectJ which has been the focus throughout this thesis offers quite a powerful control on how classes are allowed to communicate and depend on each other, but as summarized in Section 7.2,

Practitioners of agile methodologies, and extreme programming in particular, have continuous refactoring as a goal by itself for the entire lifecycle of development projects [Wells, 1999]; there should be small iterations producing executable code, with appurtenant refactoring to continuously improve the code. Thus, using DLA to enforce design from the very beginning of a software development project may interfere with this principle. In such projects, a design may instead emerge while implementing as opposed to be rigorously defined beforehand. Again, "The software is the design" [Martin, 2002]. As such, in order to effectively be able to employ DLA, the enforcement tools must support easy editing of the design specification as it dynamically changes during the development lifecycle. And once design aspects gets codified during the development, there is no need to worry about writing code that violates the agreed design, both for the current development, and future maintenance activity.

Once developers get familiar with a language such as AspectJ or a custom DLA language, dynamic adaption of design enforcement code should be applicable in the same manner as writing unit tests. However, it is not transferable to fully test-driven development (TDD) in the same way as writing unit tests before writing methods, which would be analogous to writing the whole DLA specification before writing the entire software.

Reporting

Another important feature of a DLA is to be able to produce reports proving the implementation adhering to its design. Auto-generation of such reports will save a considerate amount of time and would be of interest for various stakeholders, e.g. project leaders and customers.

Currently, the DLA Maven Plugin does not produce a report based on design enforcement tasks. However, it has been arranged for easy extending of the tool to implement such a feature:

- The necessary configuration parameters are already accounted for by the plug-in
- Meanwhile, the plug-in is able to produce a placeholder for a report. The necessary classes for utilizing Maven's reporting framework are implemented, and is able to execute as an actual report generation plug-in. It only needs to have implemented how to collect the DLA result from the enforcement task and output readable results to HTML.

7.3.2 RQ2 – How may next-generation tools address DLA in a better way than today?

The tools and practices which have been assessed in Section 3.5 have one thing in common: they all require design and architecture rules to be defined using programming language concepts, e.g. classes, method calls, packages, etc. As seen in the Patterns chapter, Chapter 3.2, design and architecture patterns provide a common vocabulary and building blocks for designing a complete software system. For a comprehensive and large-scale system, classes and details on their external interfaces are too fine-grained aspects of an architectural design to be considered during the design phase of systems development. This is especially true for modern development methodologies using a more agile approach for the implementation than the waterfall approach, which by many is considered obsolete today.

The obvious problem with the design definition strategies employed by the current tools is that for new development, it is difficult to create complete designs because of the fine-grained expressivity of the design languages. This may make DLA a

concept challenging to introduce to test-driven development (TDD) until a higher abstraction approach for codification of the design is available.

The approach of DLA Maven Plugin to offer an appropriate abstraction from the code is to offer templates inspired by common design patterns. The templates to enforce are specified in XML, which are transformed into AspectJ code using XSLT. This provides a way to define further templates which are completely decoupled from the plug-in code, as developers may define their templates using custom XSLT files. This functionality is however currently not provided by the DLA Maven Plugin, but in principle, it is a simple matter of exposing a few more configuration options which are accounted for by the plug-in.

As discussed in Section 3.2.4.3, there is possible to decouple pattern code from the classes which participate in patterns using AOP, for instance with AspectJ. This may also be interesting from a DLA perspective if it possible to superimpose patterns to classes from DLA definitions. This is proposed as further study in Section 9.2.

Chapter 8

Conclusion

Design level assertions provide the means to automatically verify that an implementation adheres to it's intended design. The goals of this project has been to assess this practice, and evaluate where it currently stands and may be heading.

When assessing the available tools for DLA support, the focus was quickly drawn to aspect-oriented programming and AspectJ. Being a young concept, AOP is still not widely embraced, but are slowly being adapted into software engineering discipline through frameworks as Spring [Spring], and the AspectJ AOP language extension to Java. The latter has been assessed in this thesis for its features which enables design adherence enforcement, and are used as basis for the DLA tool which is the resulting artifact of this project.

It is not sufficient to prove design adherence to claim that a software implementation possesses good object-oriented design. Metrics should be used in combination with DLA to prove that the implementation, and consequently also the design, follows object-oriented principles.

The project chose Maven 2 as the target platform for the developed DLA tool, realized as a Maven plug-in. While beforehand having mostly good experiences with the *use* of Maven, it appeared to be quite challenging to *develop* new functionality for Maven. This has mainly been due to the current state of developer documentation which was experienced to be inadequate, faulty, and highly decentralized. It may be a paradox that a tool which is claimed to impose good practices to software projects, its own API is mostly undocumented.

The DLA Maven Plugin is currently at a prototype stage, but its design of combining both AspectJ and the intended XML language for DLA specification, makes it capable of performing every design enforcement already possible with AspectJ. Because the plug-in produces AspectJ code, it is also possible to combine with an IDE with AspectJ support as Eclipse, to enable continuous DLA while implementing as already shown in Figure 3.4. This is believed to be a very flexible strategy for a DLA tool.

Chapter 9

Further work

This chapter outlines some proposals for further research on the topic of design level assertions.

9.1 Domain-specific DLA languages

This thesis uses XML for its prototype language to define DLA, but this is necessarily not the best choice. While XML has become an industry standard and has tremendous tool support for editing, validating and presentation, it is commonly criticized for being verbose and not being human legible *enough*, even though this is one of the original design goals of the language [W3C XML].

Are there better ways to express a software design than using XML that can also serve as rules for an automatic DLA tool? For instance, YAML, a recursive acronym[1] for YAML Ain't Markup Language, is a language to describe entities and their structure, but uses a cleaner syntax than XML to aim for a human readable syntax. [Ingerson et al., 2001–2006]

A project proposal may be to assess and evaluate other possible ways to code DLA rules. As a concrete artifact output of the research one can for instance extend the DLA Maven Plugin to transform these rules into AspectJ code in the same manner as currently being possible with XML.

9.2 Superimpose design patterns to system code

As discussed in Section 3.2.4, it can be argued that design patterns may actually hurt cohesion if they impose roles to business objects which are not a natural to them. Hannemann and Kiczales proposes a way to separate pattern code from

[1] A recursive acronym is referring to itself in its definition and is a humorous way to construct names often used in computer technology. In the case of YAML, the 'Y' stands for the acronym itself, whereas 'A', 'M' and 'L' stands for the real words Ain't Markup Language.

the participating classes using AOP [Hannemann and Kiczales, 2002], and using this article as inspiration, a suggestion for further research is to investigate if it would be feasible to transform DLA into AspectJ code which superimposes design patterns onto the system code. Hannemann and Kiczales have managed to separate all the original GoF patterns from classes which participate in the design patterns, and this should be a good starting point for such a project.

9.3 Nomothetic and idiographic research

The problem area of this thesis invites to investigations on how the DLA discipline affects for instance:

overall project complexity; How does the use of DLA affect the complexity of projects? It does indeed impose another element of concern, but to what extent is the team required to be trained as a consequence of this? Given that a software design may be fully formalized to a set of DLA rules, is it feasible that only the system architects need a thorough understanding of the concepts, or is this required by all development staff?

business concerns; To what degree does utilizing of DLA positively affect business concerns as time-to-market, quality of project output, required learning-curve, or other cost/benefit issues.

These points may invite to conduct extensive case studies or action research in conjunction with formal-mathematical analysis [Cornford and Smithson, 2006] to observe and measure various implications of the introduction of DLAs to new or existing projects.

Design level assertions are already being used in the industry. BEKK Consulting, one of the initiative parts of this thesis, are interested in incorporating such an automated process to their software development projects, and Alef Arendsen and several of his colleagues in Interface21[2] are already using it to enforce architectural rules on clients' systems. However, according to Arendsen, this is currently being done in an ad-hoc fashion, not part of any formal development process.

It would be interesting to cooperate with a firm or other industry actors and introduce DLA to a real-world project, and observe, measure, and evaluate how it is embraced by various stakeholders, affects project efficiency, post-mortem analyses, etc. The concrete project output can be a formal process description proposal on how DLA should be incorporated into the entire project life cycle. However, framework(s) and/or better tool-support for utilizing DLAs should maybe be developed before doing research on the topic in the enterprise world.

[2] `http://interface21.com`

Part IV

Appendix

Appendix A

Code

This appendix contains various code example listings.

A.1 High Cyclomatic Complexity

This listing shows a Java implementation of the example for calculating Cyclomatic Complexity used in Section 3.3.2.

```
1  package no.rune.mccabe;
2
3
4  public class McCabeProgramFlow
5  {
6
7      public void runOperation( int input )
8      {
9          do
10         {
11             System.out.println( "Decision: a" );
12             if( input >= 10 )
13             {
14                 if( input > 10 )
15                     System.out.println( "State: d" );
16                 System.out.println( "State: c" );
17                 break;
18             }
19
20             do
21             {
22                 System.out.println( "State: b" );
23                 input -= 2;
24
25                 System.out.println( "Decision: e" );
26             }
27             while( input > -3 );
28         }
29         while( Math.abs( input ) % 3 == 0 );
30
31         System.out.println( "State: f" );
32     }
33
34 }
```

Listing A.1: *Method with Cyclomatic Complexity, $CC = 5$*

A.2 Example Macker ruleset

This listing shows a Macker ruleset defining access rules for the "View" of a MVC architecture pattern. The View are denied all access to all classes not in the same package, thus enforcing the component to maintain complete independence.

```
<?xml version="1.0"?>
<!DOCTYPE macker PUBLIC "-//innig//DTD Macker 0.2//EN"
   "http://innig.net/macker/dtd/macker-0.4.dtd">

<macker>
   <ruleset name="Architectural rules for StoryType">
      <pattern name="view">
         <include class="**.view.**"/>
      </pattern>

      <pattern name="all-but-view">
         <include class="**.storytype.**"/>
         <exclude class="**.storytype.view.**"/>
      </pattern>

      <pattern name="logging">
         <include class="**.storytype.logging.**"/>
      </pattern>

      <!--
      Model-View-Controller
      -->
      <access-rule>
         <message severity="error">View breaks the MVC pattern!</message>
         <deny>
            <from pattern="view"/>
            <to pattern="all-but-view"/>
            <allow>
               <to pattern="logging"/>
            </allow>
         </deny>
      </access-rule>

   </ruleset>
</macker>
```

Listing A.2: *Macker example ruleset*

A.3 A simple MOJO

The listing shows a very basic example of the MOJO concept as explained in subsection 6.1.3.

```
package no.rune.mavenplugins.examples;

import java.io.File;
import org.apache.maven.plugin.MojoExecutionException;
import org.apache.maven.plugin.MojoFailureException;
import org.apache.maven.plugin.AbstractMojo;
import org.apache.maven.plugin.logging.Log;
import org.apache.maven.project.MavenProject;


/**
 * This will print the project's name.
 *
 * @author Rune Flobakk
```

```
 * @phase initialize
 * @goal echo-name
 */
public class ProjectNameMojo extends AbstractMojo
{

  /**
   * The current project.
   *
   * @parameter expression="${project}"
   * @required
   * @readonly
   */
  protected MavenProject project;


  /**
   * The directory where the compiled classes and aspects are put.
   *
   * @parameter expression="${project.build.directory}/rune"
   * @required
   */
  protected File targetDirectory;


  /**
   * @see org.apache.maven.plugin.Mojo#execute()
   */
  public void execute() throws MojoExecutionException, MojoFailureException
  {
    log = getLog();
    log.info( "This is the " + project.getName() + " project." );
  }

}
```

Listing A.3: *Basic MOJO class example*

A.4 Full DLA Maven Plugin configuration

This listing shows a reference configuration for the DLA Maven Plugin where all options are given their default values.

```
<plugin>
   <artifactId>dla-maven-plugin</artifactId>
   <groupId>no.rune.mavenplugins</groupId>
   <version>0.1-SNAPSHOT</version>
   <executions>
      <execution>
         <goals>
            <goal>validate</goal>
            <goal>generate-aspects</goal>
            <goal>verify</goal>
         </goals>
      </execution>
      <configuration>
         <failOnError>true</failOnError>
         <failOnWarning>false</failOnWarning>
         <dlaFile>dla.xml</dlaFile>
         <enableAspectJ>false</enableAspectJ>
         <aspectDirectory>src/main/aspect</aspectDirectory>
         <sourceDirectory>src/test/dla</sourceDirectory>
         <javaSourceDirectory>src/main/java</javaSourceDirectory>
         <compileTarget>
            ${project.build.directory}/generated-classes/dla
```

```
          </compileTarget>
        </configuration>
    </executions>
</plugin>
```

Listing A.4: *DLA Maven Plugin reference configuration*

A.5 DLA with AspectJ

```
package no.rune.mavenplugins.dla;

public aspect DesignLevelAssertions
{

  public pointcut mojosSource():
      within( no.rune.mavenplugins.dla.*Mojo );

  public pointcut mojosTarget():
      call(* no.rune.mavenplugins.dla.*Mojo.*(..) ) ||
      call( no.rune.mavenplugins.dla.*Mojo.new(..) );


  declare warning: mojosTarget() && ! mojosSource():
      "Violation of mojos component's independence!";

}
```

Listing A.5: *DLA example implemented in AspectJ.*

Appendix B

Requirements specification

This is the formal specification of a prototype DLA tool.

Realization	Java implementation
Platform	Maven 2
Tool invocation	Through MOJO interface (i.e. as a Maven 2 plugin). Standalone application if time permits.
Overall functionality	Fail builds or issue warnings on given credentials.
Reporting	Simple console output to STDOUT. This may also be dumped into a simple HTML document for inclusion with Maven's website generation output.
Configuration	Maven POM, and design-specification given by a tool-specific XML document.

Table B.1: *Overall functionality.*

Table B.2 describes in detail the functionality of the tool, specifically what kinds of assertions that must be possible to perform, and how to react to them. Possible reactions are

- create a report entry (R)
- fail and terminate the build process (F)
- issue a warning during the build process (W)

Design issue	Description	Reaction
Components	Possibility to define components beyond the package mechanism of Java	*(NA)*
Dependencies	Any components' dependencies not coherent with the design	RFW

Table B.2: *Design-level issues detectable by the implemented tool.*

Appendix C

Conversation transcripts

This appendix contains transcripts of relevant conversations with persons external to this project.

C.1 Correspondence with Alef Arendsen

This is an extract of the mail correspondence the project has had with Alef Arendsen of Interface21 regarding DLA. Arendsen has approved the correspondence to be reproduced in this thesis.

Rune Flobakk, March 26th 2007, 16:10.

[. . .]

I am writing my master's thesis on automatically enforcing design rules when implementing software, or Design Level Assertions as you elegantly term it in your blog.

[. . .]

I thought I'd ask you if you have any kind of documentation on how you at Interface21 use this technique, and if you were willing to share this? Maybe you have something on conventions to follow, best practices, or other experiences you have gained? You blog entry about DLA stands very much in solitude in cyber space ;)

I have also checked out Macker (http://innig.net/macker/) which offers a bit of the same functionality by defining architecture rules in XML, though I think you get more "fine grained" control using AspectJ. Do you have any thoughts on Macker vs. DLA in AspectJ? Any other tools or practices you would think are worth checking out?

[. . .]

Alef Arendsen, March 26th 2007, 19:22.

[...]

There are some colleagues (one is me) using DLAs at clients to do architectural enforcement, but so far it's been pretty ad-hoc. We've been looking into potentially creating a product that helps you do this kind of stuff, but so far we haven't had the time or resources to further pursue this. Internally we have a small AspectJ message listener that picks up on DLA warnings and publishes them in HTML format, but this is all pretty crude right now.

We usually use tools such as Structure101 or SonarJ (commercial tools, of which I personally like Structure101 very much) to analyze dependencies and then use AspectJ to enforce them. The thing I like about AspectJ is that it's compile time and can potentially be integrated in Eclipse (although for large projects, this is sometimes too heavy a burden for Eclipse). This allows you to embed your policies and architectural guidelines in your source code, as opposed to tools like PMD, JDepend, Checkstyle, but from the looks of it, also Macker where you have to run an Ant build or similar tool. Plus you get the beauty of a typed language of course (AspectJ).

Of course, you can't do all enforcements with AspectJ though.

[...]

C.2 Interface matching in AspectJ

The following is a transcript of a thread initiated by this project in the Manning Publication Co.'s *Author Online* forum [Manning], where it is possible to ask questions directly to the authors of Manning books. This particular question was addressed to Ramnivas Laddad, the author of AspectJ in Action [Laddad, 2003].

Rune Flobakk, June 19th 2007, 21:47.

Hello, and complements on a great book! Love the in Action series in general really.

I am using AspectJ for design enforcement here, and declaring errors and warnings for the ajc to throw at compile-time. Among the things I would like to detect is when any static reference is done to a class instead of an interface, and declare a warning/error for that. In particular, I am aiming to enforce Constrained Java as explained in this paper: `http://www-plan.cs.colorado.edu/henkel/stuff/ConstrainedJava.pdf`

Is this possible at all? I have tried things like:

public pointcut violateCJPS():

withincode(* no.rune.processor..*.*(..)) &&
if(! thisJoinPoint.getSignature().getDeclaringType().isInterface());

declare warning: violateCJPS(): “Violating Constrained Java Package Space.”;

But I think I am not at the right track because I get an compiler error saying that the if() pointcut designator cannot be used in declare statement.

So to sum it up, I want to make a pointcut which matches interfaces only. Possible? Thank you in advance!

Ramnivas Laddad, June 19th 2007, 22:34.

Thanks.

The if() pointcut is a runtime-determinable pointcut. You can use only the statically determinable pointcut for static crosscutting. Unfortunately, you seem to have a need to determine if a certain type is an interface and that is not possible with current AspectJ.

-Ramnivas

“agou”, June 19th 2007, 22:35.

p.97 “Note that since these declarations affect compile-time behavior, you must use only statically determinable pointcuts in the declaration.”

So in other words you want to do something dynamically using a static crosscutting. I had a similar problem where I wanted to capture join points that were only inside classes and not aspects (`http://www.manning-sandbox.com/thread.jspa?threadID=19243\&tstart=0`)

But I am only a beginner. Maybe Ramnivas has some ideas!

Rune Flobakk, June 19th 2007, 21:47.

Thanks for quick replies! It was as I suspected then. Crossing my fingers for future AspectJ updates.

Glossary

AntiPattern A pattern used in a context where it causes more negative than positive consequences. A recurring solution to a common problem which does not positively affect the system.

architectural drift The phenomena of a software system over time evolving away from its original intended (architectural) design.

architectural pattern A pattern to solve an architectural problem. See pattern.

aspect-oriented programming (AOP) As object-oriented programming deals with assigning responsibilities to discrete classes, aspect-oriented programming enables assignement of crosscutting concerns, responsibilities applicable for several possibly non-related classes. (See also subsubsection 3.2.4.2.).

best practice What is considered to be the best way to solve a certain problem in a given context.

continuous integration (CI) The process of frequent building, testing, and integration of approved code to the final system. To support iterative development, this is usually performed automatically by a CI system.

crosscutting concern A term commonly used in relation to aspect-oriented programming and describes a responsibility which is applicable to several, if not all, classes of a system. Typical crosscutting concerns are logging, persistence, or access control (security).

dependency injection A specialization of the inversion of control pattern which hands the responsibility of wiring objects together into the hands of a framework instead of the developer.

design level assertion (DLA) The concept of automated verification of adherence to a (software system) design.

design pattern A pattern to solve a design problem. See pattern.

Gang of Four (GoF) A common reference to the authors of the book *Design Patterns. Elements of Reusable Object-Oriented Software* [Gamma et al., 1995] which is regarded as essential reading on design patterns.

General Responsibility Assignment Software Patterns (GRASP) Group of design patterns to properly assign discrete responsibilities to classes. These patterns are fundamental for good object-oriented design.

goal-oriented requirements language (GRL) Modeling language for defining goals of software systems.

groupware Software to support work collaboration.

inversion of control (IoC) A pattern where the assembly/wiring of components and operation invoking are handled by a framework instead of custom developed classes of a software system.

javadoc A syntax for writing code documentation in Java source code. The documentation can be extracted by an appropriate tool to autogenerate an API reference.

Model-View-Controller (MVC) A common architectural pattern for separation of data, presentation and the interaction in between.

non-functional requirement (NFR) A requirement comprising the system as a whole. Also known as quality attribute.

open-source software (OSS) Software which source code is publicly available for anyone to view and modify.

pattern A common and well-known way to solve a specific problem.

project object model (POM) A project description for a Maven project. The POM describes various aspects of a software project, and Maven uses this information to correctly perform various build tasks.

quality assurance (QA) A generic term for any efforts done to ensure that a product (e.g. software) has a certain degree of measureable and/or perceived quality.

quality attribute See non-functional requirement.

quality tactic A strategy supporting the achievement of a quality attribute.

softgoal A GRL concept for modeling non-functional requirements and its success premises. Also see goal-oriented requirements language.

software architecture "... the structure or structures of a [program or computing] system, which comprise software elements, the externally visible properties of those elements, and the relationships among them." [Bass et al., 2003].

test-driven development (TDD) A development methodology concept common in agile development where programatic tests for the system are written before the actual system code. When a test passes, the component it tests is in principle finished.

transitive dependency A module of a software system may depend on other modules, but these modules may also in turn depend on other modules. These will then become transitive dependencies of the system. A transitive dependency is not directly required by a module, but since it is required by a dependent module it still must be present for proper operation of the system.

References

Ant. Apache Ant Project, 2007. URL http://ant.apache.org. Last accessed: April 30th, 2007.

Alef Arendsen. On the ServiceLocatorFactoryBean, DLAs and the sustainability of code and design, October 2006. URL http://blog.arendsen.net/index.php/2006/10/05/on-the-servicelocatorfactorybean-dlas-and-the-sustainability\-of-code-and-design/. Last accessed: January 18, 2007.

David Avison, Francis Lau, Michael Myers, and Peter Axel Nielsen. Action research. *Communications of the ACM*, 42(1):94–97, January 1999.

Richard L. Baskerville. Investigating information systems with action research. *Communications of the Association for Information Systems*, 2(19), October 1999.

Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, 2nd edition, 2003.

Josh Bloch. Collections (The Java^{TM} Tutorials), 1996–2006. URL http://java.sun.com/docs/books/tutorial/collections/. Last accessed: March 5th, 2007.

William J. Brown, Raphael C. Malveau, Hays W McCormick III, and Thomas J Mowbray. *AntiPatterns. Refactoring Software, Architecture and Projects in Crisis*. John Wiley & Sons, Inc., 1998.

Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley, 1996.

Paul Cantrell. Macker, 2003. URL http://innig.net/macker/. Last accessed: February 19th, 2007.

Mike Clark. *Pragmatic Project Automation: How to Build, Deploy, and Monitor Java Apps*. The Pragmatic Programmers, http://pragmaticprogrammer.com/, 2004.

Mike Clark. JDepend, 1999–2006. URL http://clarkware.com/software/JDepend.html. Last accessed: January 28, 2007.

Andy Clement and Mik Kersten. AJDT: getting started with aspect-oriented programming in Eclipse. Presentation at the technical track at EclipseCon 2004., 2004. URL http://www.eclipsecon.org/2004/EclipseCon_2004_TechnicalTrackPresentations/29_Clement-Kersten.pdf. Last accessed: April 12th, 2007.

Paul Clements, Len Bass, Rick Kazman, and Gregory Abowd. Predicting software quality by architecture-level evaluation. In *Proceedings of the Fifth International Conference on Software Quality*, pages 485–497, Austin, TX, October 1995.

James W. Cooper. *The Design Patterns Java Companion*, pages 177–185. Addison-Wesley, 1998.

Tony Cornford and Steve Smithson. *Project Research in Information Systems*. Palgrave MacMillian, 2nd edition, 2006.

Michael C. Daconta, Leo J. Obrst, and Kevin T. Smith. *The Semantic Web: A Guide to the Future of XML, Web Services, and Knowledge Management*. John Wiley & Sons, 2003.

Mark Doliner, Grzegorz Lukasik, and Jeremy Thomerson. Cobertura version 1.9. Sourceforge project, 2007. URL http://cobertura.sourceforge.net.

Martin Fowler. Inversion of control containers and the dependency injection pattern, January 2004. URL http://www.martinfowler.com/articles/injection.html. Last accessed: June 8th, 2007.

Martin Fowler and Kendall Scott. *UML Distilled*. Addison Wesley, 2nd edition, 2000.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, January 1995.

Robert L. Glass, V. Ramesh, and Iris Vessey. An analysis of research in computer disciplines. *Communications of the ACM*, 47(6):89–94, June 2004.

Mark Grand. *Patterns in Java. A Catalog of Reusable Design Patterns Illustrated with UML*, volume 1. Wiley, 1998.

Jan Hannemann and Gregor Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings of the 17th Annual ACM conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 161–173, November 2002.

David Heinemeier Hansson. Ruby on rails, 2007. URL http://rubyonrails.com. Last accessed: February 28, 2007.

Johannes Henkel. Constrained java and its application for refactoring large java programs. URL http://www-plan.cs.colorado.edu/henkel/stuff/ConstrainedJava.pdf. Last accessed: June 17th, 2007, November 2000.

Brian Ingerson, Clark Evans, and Oren Ben-Kiki. YAML Aint'n Markup Language, 2001–2006. URL http://yaml.org. Last accessed: November 15, 2006.

Interface21. Interface21 - the company behind the Spring Framework, 2000 – 2007. URL http://interface21.com. Last accessed: January 29, 2007.

ISO9126-1. *ISO/IEC 9126-1:2001 Software engineering – Product quality – Part 1: Quality model*. International Organization for Standardization, 2001.

Ralph E. Johnson and Foote Brian. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June 1988.

Jean-Marc Jézéquel, Michel Train, and Mingins Christine. *Design Patterns and Contracts*. Addison-Wesley, 2000.

Ramnivas Laddad. *AspectJ in Action*. Manning Publications Co., 2003.

Craig Larman. *Applying UML and Patterns – An introduction to object-oriented analysis and design*. Prentice Hall, 1998.

Make. Gnu make, 1997–2006. URL http://www.gnu.org/software/make/. Last accessed: April 30th, 2007.

Manning. Matching interfaces. Author Online Web forum thread, June 2007. URL http://www.manning-sandbox.com/thread.jspa?forumID=6\&threadID=19476.

Robert C. Martin. OO design quality metrics. An analysis of dependencies, 1994. URL http://objectmentor.com/resources/articles/oodmetrc.pdf. Last accessed: March 12th, 2007.

Robert C. Martin. *Agile Software Development. Principles, Patterns, and Practices*. Prentice Hall, 2002.

Vincent Massol, Jason van Zyl, John Casey, Brett Porter, and Carlos Sanchez. *Better Builds with Maven. The How-to Guide for Maven 2.0*. Mergere Library Press, Marina del Rey, California, October 2006. URL http://library.mergere.com.

Maven. Apache Maven Project, 2002–2007. URL http://maven.apache.org. Last accessed: April 30th, 2007.

Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4), December 1976.

John Mylopoulos, Lawrence Chung, and Eric Yu. From object-oriented to goal-oriented requirements analysis. *Communications of the ACM*, 42(1), January 1999.

QALab. QALab. Statistics for your build. Sourceforge project, 2006. URL `http://qalab.sourceforge.net`.

Arthur J. Riel. *Object-oriented Design Heuristics*. Addison-Wesley, 1996.

SonarJ. Sonarj version 3.2. Website, 2007. URL `http://www.hello2morrow.com/en/sonarj/sonarj.php`.

Sonatype. Maven: The definitive guide (version 1.0 alpha 1), 2007. URL `http://www.sonatype.com/book/`. Last accessed: April 30th, 2007.

Spring. Spring framework, 2006. URL `http://springframework.org`. Last accessed: January 29, 2007.

Krishnaprasad Thirunarayan, Günter Kniesel, and Hariprivan Hampapuram. Simulating multiple inheritance and generics in java. *Computer Languages, Systems and Structures*, 25(4), 1999. URL `http://www.elsevier.com/locate/complang/`.

James Donovan Wells. Refactor mercilessly. Website, 1999. URL `http://www.extremeprogramming.org/rules/refactor.html`.

W3C XML. *Extensible Markup Language (XML) 1.0*. World Wide Web Consortium, 4th edition, August 2006. URL `http://www.w3.org/TR/2006/REC-xml-20060816/`.

W3C XSLT. *XSL Transformations (XSLT) Version 1.0*. World Wide Web Consortium, Novembebr 1999. URL `http://www.w3.org/TR/1999/REC-xslt-19991116`.

Jason van Zyl et al. Maven user FAQs-1, 2007. URL `http://docs.codehaus.org/display/MAVENUSER/FAQs-1`. Last accessed: June 1st, 2007.