



An Approach to Discovering Architectural Patterns in Software

by

Joeri Peters

A master thesis
in fulfilment of the thesis requirement
for the degree of
Master of Science
in
Business Informatics

Utrecht University

15th February 2016

Joeri Peters, BSc.
Student number: 3484998
J.G.T.Peters@students.uu.nl
J.G.T.Peters@uu.nl
JoeriPeters@live.com

Supervisor:
dr.ir. Jan Martijn van der Werf
Information & Organisation group

2nd Supervisor:
dr. Jurriaan Hage
Software Technology group

Software Systems Division
Department of Information and Computing Sciences
Faculty of Science
Utrecht University



Images: <http://www.gobex.es/comunicacion/noticia?idPub=7248#.Vq97NtCaTa8> & http://www.evident.nl/portfolio/klanten/~media/Images/Evident/Portfolio/Cases/Klantlogo/news_1_1288775788.ashx.

Abstract

In this master thesis, an approach to discovering architectural patterns within static modular architectures based on software dependencies is developed and investigated. Architectural patterns represent reusable design of software architecture at a high level of abstraction. They can be used to structure new applications and to recover the modular structure of existing systems. Techniques like Architecture Compliance Checking (ACC) focus on testing whether the realised artefacts adhere to the architecture. Typically, these techniques require a complete architecture as input.

In this thesis, we focus on defining architectural patterns in such a way that we can use ACC tools to recognise architectural pattern instances and on algorithms by which to discover them, for the purposes of Software Architecture Reconstruction (SAR). The central problem is that it is difficult to distinguish a *possible* instance of an architectural pattern from a *genuine* instance. Addressing this requires us to explicitly define architectural patterns in terms of allowed and disallowed software dependencies. We base ourselves on Semantically Rich Modular Architectures (SRMAs). Defining architectural patterns this way allows us to reason about them. For example, about how patterns should be interpreted as incomplete architectures and how different interpretations affect the pattern recognition process. Recognising architectural patterns using ACC techniques thus has great potential in architecture design, ACC and SAR.

Our way of finding potential patterns, what we call pattern candidates, is a two step process. The modular decomposition step establishes the modules of which a selection is used in the second step. There, we go through possible mappings of the found modules (referred to as *software units*) to the modules of architectural patterns (called *pattern modules*). The ACC tool we employ to do this is HUSACCT.

We have devised two main approaches to finding pattern candidates. There is the more cumbersome brute force approach and the more elegant genetic approach. Both algorithms require further development before they can be used in practice, but we manage to show that this is a promising line of research which may already result in several publications. Some preliminary evaluations provide a handful of lessons to be applied in any future work, which we envision as GEAR: Guided Evolutionary Architecture-Reconstruction. This vision is also described within the thesis, by means of providing a rudimentary illustration of a practical tool based on our work.

This master thesis is primarily about providing several proofs of concept, it is not intended as a specification for future realisations or as a formal analysis of the algorithms provided within. It also raises certain questions and discussion points, such as the nature of architectural patterns as open architectures within larger systems.

Preface

This master thesis represents the culmination of years of studying at Utrecht University and thus the end of an era. From highly mathematical courses on physics, through unambiguously business-oriented courses on IT management, I came to software architecture. As my background is an odd blend of technical and non-technical areas, so is the field I find myself in today; albeit not always necessary for a single individual to combine the deeply technical aspects of software architecture with the often neglected organisational side. In fact, this is deemed unnecessary far too often in practice.

Just like software architecture is based on a delicate balance, IT-research as an academic field in general tends to exist on this wonderful point in between the academic extremes of an R&D employee and an ivory tower recluse. When I realised this, it rekindled a passion for doing research and developing ideas that I had lost when, several years ago, I came to the conclusion that although astrophysics is an excellent conversation topic over a glass of dark abbey beer, its sort of research was not what I wanted for myself as a working career. IT seemed more exciting, more practical yet still very broad, probably due to its youthfulness. Even if my work thus far is not quite a turning point in human history, it represents a watershed for me.

Although I did learn to program during my years of studying physics, I had to learn a lot during the course of this thesis. First of all, I had never laid my hands on a single line of Java before. Moreover, computational physics and data processing were always restricted to a handful of code classes, whereas in this thesis I had to learn to work with a relatively large open-source application. I had to get used to Eclipse and GitHub, which demanded a few frustrating moments, and much of the literature mentioned concepts that I had not been taught during my formal education. The sort of programming was quite different from the computer models that a physicist is required to produce, so my work in this thesis project has been a lot of trial and error.

Surely, a more experienced developer would have worked faster and would have produced a cleaner result. Although I find code optimisation a very interesting topic, I do not know much about it. Also, someone with a more pronounced background in software architecture would have read through the relevant papers more swiftly. But maybe that is the point of a master thesis. It could scarcely have been more educational for me.

Experimental design research is a marvellous thing, although its freedom and flexibility come with a certain degree of frustration when attempting to plan ahead more than a few weeks. I wish there had been time to do extensive case studies, proving conclusively that my ideas can indeed form another tool on the tool belt of architecture reconstruction. But these things take time and go through several steps of maturation. This thesis is primarily about proofs of concept.

It is often said that a master thesis should be at least somewhat readable to a layperson. Actually, it can be a burden to make sure that even an expert will not lose track of what is meant by a certain term or why a particular step makes sense. I will not pretend that this work is as technically dense as that produced by my fellows at Computing Science, who regretfully receive more opportunities to apply their mathematical skills than I do and breeze through Java coding tasks, but I have attempted to flesh out certain parts of my thesis whenever I deemed it necessary.

I hope this document succeeds in portraying a rather interesting side of both software architecture and IT in general. I also hope it conveys a sense of excitement over the sort of work that remains to be done.

Cheers,
Joeri

Glossary

“Must use” heuristic :

“x must use y” is one of the seven relation rules within HUSACCT’s SRMA language uses. This rule is only violated when module x does not directly depend on module y at all, which then counts as a single violation. One would presume that the appropriate modules of an architectural pattern do depend on one another, so this violation is of vital importance to the discovery of patterns.

ACC (Architecture Compliance Checking):

The act of studying the degree to which a system’s software architecture complies with its documentation. This may be in order to verify that the architecture is correct or in order to check that the documentation is kept up to date.

Aggregation :

The term used for the scenario when multiple software units are mapped to the same pattern module, creating groups of software units and thus complicating the mapping.

Architectural element :

The word used for any module found within software architecture, whether it be from the as-is architecture or not.

Architectural pattern :

A type of generic software architecture design aimed at promoting qualities such as re-usability, maintainability and separation of concerns.

Brute force :

A brute force approach or brute force attack is a way of finding the solution to a problem by attempting all possible solutions.

Chromosome :

A molecule of DNA, which in genetic algorithms represent individual solutions. In our case, each chromosome is a pattern candidate.

Dependency :

A direct software dependency between two modules. Throughout this thesis, we do not distinguish between any dependency types. By “using”, “accessing”, “depending on” and “calling on” we thus mean the same thing.

Design pattern :

A type of software design that exists on a much smaller scale than architectural patterns, though the distinction is not always clear. The most famous design patterns are known as the Gang of Four (GoF) patterns.

Gene :

A section of a chromosome that encodes for something, in our case the mapping of a single software unit to a pattern module. The value of a gene is called the allele.

Genetic algorithm :

An approach from evolutionary computing inspired by neo-Darwinian evolution to find solutions to a particular problem by modelling the solutions as a reproducing population of individuals.

HUSACCT (Hogeschool Utrecht Software Architecture Compliance Checking Tool):

The ACC tool used in this thesis. It is intended for SRMA support and thus distinguishes itself from other such tools through the richness of its semantics. It analyses Java and C# source code and is itself built in Java.

Pattern candidate :

A particular mapping of software units to pattern modules such that the whole is a possible instance of an architectural pattern and can be associated with a certain fitness score.

Pattern instance :

A genuine occurrence of an architectural pattern.

Pattern module :

An architectural element from an architectural pattern, as opposed to software units.

Remainder :

The collective name for software units that are not part of a pattern candidate in that they are not mapped to any pattern module.

SAR (Software Architecture Reconstruction):

An act similar to ACC, although it implies a lack of architecture documentation. The software architecture of an existing system is reconstruction based on a number of many characteristics. A person performing this task may be known as a re-engineer or re-architect.

Software unit :

The name for an architectural element from the analysed (as-is) architecture, thus it is a class or package from the source code.

SRMA (Semantically Rich Modular Architecture):

The term used for architectures with rich semantics, meaning various types of modules, rules, exceptions, violations and software dependencies. HUSACCT supports a set of architectural elements that are commonly found in SRMAs.

Violation :

A software dependency that exists in spite of an architectural rule. The dependency is then said to violate that rule. The opposite of a violation is a dependency in accordance with a rule, an affirmation.

Contents

1	Introduction	1
2	Research Approach	5
2.1	Research Questions	5
2.2	Relevance	7
3	Architecture Reconstruction	9
3.1	SAR & ACC	9
3.2	SRMA & HUSACCT	10
3.3	Running Example	12
4	Architectural Patterns	17
4.1	Design Patterns	19
4.2	Pattern Definition	21
4.3	Pattern Catalogue	27
5	Proposed Approach	39
5.1	Patterns in SAR	39
5.2	Process-Deliverable Diagram	43
5.3	Modular Decomposition	44
5.4	Pattern Matching	46
5.4.1	Fitness Function	47
5.5	Running Example Continued	50
6	Search Algorithms	53
6.1	Number of Candidates	53
6.2	Realisation of the Brute Force Approach	56
6.3	Genetic Algorithms	56
6.4	Genetic Code	60
6.5	Realisation of the Genetic Approach	65
7	Evaluation	69
7.1	Yasper	69
7.2	aTunes	78
7.3	SweetHome3D	83
7.4	HUSACCT	90
7.5	Lessons Learnt	95

8	GEAR	97
8.1	User Interface Design	97
8.2	Pattern Editor	103
8.3	Custom Fitness Function	105
8.4	Additional Constraints	106
8.5	Pattern-based Architectures	107
9	Conclusion and Discussion	109
10	Acknowledgements	113
	References	115
	Appendices	123
A	PDD Tables	123
B	Additional Pattern Definitions	127
C	Class Diagram	141
D	Additional Code Samples	143
E	WICSA/CompArch 2016 paper	175

List of Figures

1.1	The context of software architecture.	2
1.2	The various viewpoints of software architecture.	3
1.3	Conceptual visualisation (Venn diagram).	3
3.1	The meta-model of Semantically Rich Modular Architectures	11
3.2	The meta-model of Architecture Compliance Checking.	12
3.3	The root, Analyse and Define folders in the HUSACCT v1.0 source directory. . .	13
3.4	HUSACCT’s analysis of the two components, with the Analyse component selected.	14
3.5	A 5-Layered pattern in the Analyse component, with violations in red.	15
3.6	A 4-layered pattern in the Define component, with violations in red.	16
4.1	An example of a Layered pattern.	17
4.2	An example of a Model-View-Controller pattern.	19
4.3	A generic depiction of the Adapter pattern.	20
4.4	A generic depiction of the Bridge pattern.	20
4.5	The 3-Layered pattern and the Remainder, defined with “Is not allowed to use” rules and showing allowed dependencies as dashed lines.	24
4.6	The 3-Layered pattern and the Remainder, defined with “Is only allowed to use” rules.	25
4.7	The 3-Layered pattern and the Remainder, defined with “Is the only module allowed to use” rules.	26
4.8	The 3-Layered pattern and the Remainder, defined with two different rule types (see rule set 5).	26
4.9	The Controller Interface interpretation of the classic MVC pattern.	33
4.10	The Requester Interface interpretation of the Broker pattern.	35
4.11	Class diagram of Reconstruct package.	37
5.1	The proposed method.	44
5.2	A PDD of the modular decomposition using namespaces (C#) or package defini- tions (Java).	46
5.3	Four imaginary software units and their mutual dependencies.	48
5.4	Two candidates for the 3-Layered pattern. See the set below for the architectural rules.	49
5.5	A PDD brute force approach to the <i>Evaluate candidates</i> activity.	52
6.1	Mapping resulting from chromosome.	61
6.2	Analysis results of HUSACCT’s own source code (v1.0) within its Graphics sub- system.	62
6.3	The resulting architecture diagram as if validation were manually called within HUSACCT.	63

6.4	The new architecture diagram.	64
6.5	A PDD of the genetic approach to the <i>Evaluate candidates</i> activity.	65
7.1	The package hierarchy of the Jasper system.	70
7.2	Dependencies among the root packages the Jasper system.	71
7.3	Eclipse console output of the non-aggregation run on the Jasper packages.	71
7.4	Eclipse console output of the aggregation run on the Jasper packages.	72
7.5	Eclipse console output of the aggregation + Remainder run on the Jasper packages.	72
7.6	Eclipse console output of a genetic run on the Jasper packages.	73
7.7	Eclipse console output of a genetic run (including Remainder) on the Jasper packages.	74
7.8	Eclipse console output Jasper.	75
7.9	Eclipse console output Jasper.	76
7.10	Eclipse console output Jasper.	77
7.11	The root of the aTunes package hierarchy.	78
7.12	The dependencies in the root of the aTunes package hierarchy.	79
7.13	Eclipse output of aTunes aggregation + Remainder brute force for Complete Freedom MVC.	80
7.14	Eclipse output of aTunes aggregation + Remainder brute force for Model Interface MVC.	81
7.15	Eclipse output of aTunes aggregation + Remainder brute force for Controller Interface MVC.	82
7.16	The SweetHome3D package hierarchy root.	83
7.17	The dependencies between SweetHome3D's packages.	84
7.18	Eclipse output for Complete Freedom 3-Layered with SweetHome3D's root packages.	85
7.19	The layering as seems to be employed by SweetHome3D.	85
7.20	An interpretation of SweetHome3D's Model-Viewcontroller pattern.	86
7.21	A genetic (with Remainder) run on SweetHome3D's packages for the Model-Viewcontroller pattern.	88
7.22	A genetic (with Remainder) run on SweetHome3D's packages for the Centralised Layering pattern.	89
7.23	The intended architecture of HUSACCT's Analyse component.	90
7.24	The results of the brute force (aggregation + Remainder) run on the Analyse component.	93
7.25	The results of the genetic (aggregation + Remainder) run on the Analyse component's sub-units.	94
8.1	Define intended architecture window.	98
8.2	A very basic mock up of a user interface that conveys the idea of what a future implementation could be like.	99
8.3	Selecting a technique of modular decomposition and choosing software units.	100
8.4	Selecting an existing fitness function or editing one (adjusting weights per rule type).	101
8.5	Selecting a pattern matching technique. Obviously, this list and others could be extended by additional techniques.	102
8.6	The best results displayed as an interactive list.	102
8.7	Studying a single result more closely.	103
8.8	A combination of MVC and the 3-Layered patterns.. . . .	104
8.9	Simple example of a possible pattern editor.	105
8.10	A basic illustration to communicate the idea of a future fitness function editor.	106

8.11	A combination of Complete Freedom MVC (style) and Free Remainder 3-Layered (pattern), which fails to isolate the lower Model layers.	108
B.1	The Complete Freedom interpretation of the classic MVC pattern.	134
B.2	The Free Remainder interpretation of the classic MVC pattern.	135
B.3	The Restricted Remainder interpretation of the classic MVC pattern.	136
B.4	The Model Interface interpretation of the classic MVC pattern.	137
B.5	The Requester Interface interpretation of the Broker pattern.	139
B.6	The Requester Interface interpretation of the Broker pattern.	139
B.7	The Requester Interface interpretation of the Broker pattern.	140
C.1	The class diagram of the Reconstruct package.	141

List of Tables

4.1	The 7 relation rule types, split into two sub-categories.	22
4.2	This legend presents the different rule types for the provisional diagram notation used here.	24
6.1	Candidate numbers.	55
6.2	The initial population.	58
6.3	The selected population marked with crossover positions.	58
6.4	The selected population after crossover and marked for mutation.	59
6.5	The resulting new generation.	59
6.6	The third generation.	59
6.7	The fourth generation.	60
6.8	An example of a chromosome for 7 software units.	61
6.9	The legends for this candidate	62
6.10	An example of a chromosome for 4 software units mapped to the MVC pattern. .	62
6.11	The chromosome post-mutation.	63
6.13	A quick overview of the genetic structure.	64
6.12	Legends for a candidate.	64
A.1	A quick overview of the genetic structure.	124
A.2	The concept table for the various Process-Deliverable Diagrams used throughout this thesis.	125

Listings

4.1	The abbreviated source code for the abstract Pattern class.	27
4.2	The abbreviated source code for the abstract class of the N-Layered pattern. . .	28
4.3	The abbreviated source code for one interpretation of the N-Layered pattern, namely the case with isolated internal layers.	29
4.4	The shortened source code for the abstract Model-View-Controller pattern class .	31
4.5	The source code for the Model-View-Controller (Controller Interface) pattern class.	33
4.6	The source code summary for the abstract Broker pattern class.	34
4.7	The source code for the Broker (Requester Interface) pattern class.	35
6.1	The fitness function class for the genetic approach.	66
7.1	A pattern that corresponds to SweetHome3D's layering structure.	86
7.2	A Complete Freedom interpretation of a Model-Viewcontroller pattern.	86
7.3	HUSACCT's Analyse component.	91
B.1	The source code for the abstract Pattern class.	127
B.2	The source code for the abstract class of the N-Layered pattern.	129
B.3	The Isolated Internal Layers variety of the N-Layered pattern.	130
B.4	The Complete Freedom variety of the N-Layered pattern.	130
B.5	The Free Remainder variety of the N-Layered pattern.	131
B.6	The Restricted Remainder variety of the N-Layered pattern.	131
B.7	The Layer Types variety of the N-Layered pattern.	132
B.8	The source code for the abstract Model-View-Controller pattern class.	132
B.9	The Complete Freedom variety of the MVC pattern.	133
B.10	The Free Remainder variety of the MVC pattern.	134
B.11	The Restricted Remainder variety of the MVC pattern.	135
B.12	The Model Interface variety of the MVC pattern.	136
B.13	The source code for the abstract Broker pattern class.	137
B.14	The source code for the Broker (Complete Freedom) pattern class.	138
B.15	The source code for the Broker (Free Remainder) pattern class.	139
B.16	The source code for the Broker (Restricted Remainder) pattern class.	140
D.1	The main reconstruction class. Currently has a very low cohesion and high coup- ling to other classes, in line with the fact that this is an experimental setup and not a finalised product.	143
D.2	The mapping generator for non-aggregation mappings of the brute force approach.	160
D.3	Test code for the mapping generator.	162
D.4	The mapping generator for the aggregation case of the brute force approach. . .	163
D.5	The test code for the aggregate mapping.	167
D.6	The genetic algorithm class.	169
D.7	The JUnit tests for SweetHome3D's package dependencies.	171

Chapter 1

Introduction

A system’s software architecture is similar to the architecture of a building, in that it tells the knowledgeable observer how it was constructed and how its components fit together. “The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both” (Bass, Clements & Kazman, 2012, Ch. 1).

The software architecture is comprised of multiple structures. Some are static, these are the modules of the system and they can exist on many levels of decomposition. These modules form a tree, with the largest structures in the root and the smallest in the leaves. This tree is usually imagined upside down, so the lowest level consists of the smallest modules, which would likely be code classes in the case of a logical decomposition. The highest levels may contain layers or major components, e.g. separating responsibilities like presentation from database logic. The uses of such decomposition include project structuring, object-oriented design and allowing specific modules to be reused elsewhere.

Dynamic structures are related to runtime behaviour, which means that they are concerned with services, interaction and synchronisation among the various structures. These are also known as component-and-connector structures (Bass et al., 2012). This has far more to do with performance and availability than the static side of the architecture.

Finally, there is the allocation of these structures to development teams, file structure or hardware. Assigning a particular module to a particular tester is a typical example of an allocation structure. These structures are often overlooked or at least not considered part of a software architecture, and one can imagine why, but this is an essential part of complete architecture documentation. Particularly project managers would be interested in the work assignment and deployment aspects of an architecture, as opposed to developers who would be more concerned with the other two types of structures.

Figure 1.1 shows the context of software architecture as a diagram. Another way to categorise various aspects of software architecture is to look at views and viewpoints, which are also indicated in this diagram. A view is defined as “a representation of one or more structural aspects of an architecture that illustrates how the architecture addresses one or more concerns held by one or more of its stakeholders” (Rozanski & Woods, 2011, Ch. 3). In other words, a view is a set of figures and descriptions that together depict a particular aspect of the architecture. This is strongly related to ISO Standard 42010¹.

A *viewpoint* can be thought of as the specification of such a view, which may include things like design guidelines. Rozanski and Woods define it as: “a collection of patterns, templates and conventions for constructing one type of view. It defines the stakeholders whose concerns are

¹<http://www.iso-architecture.org/ieee-1471/ads/>

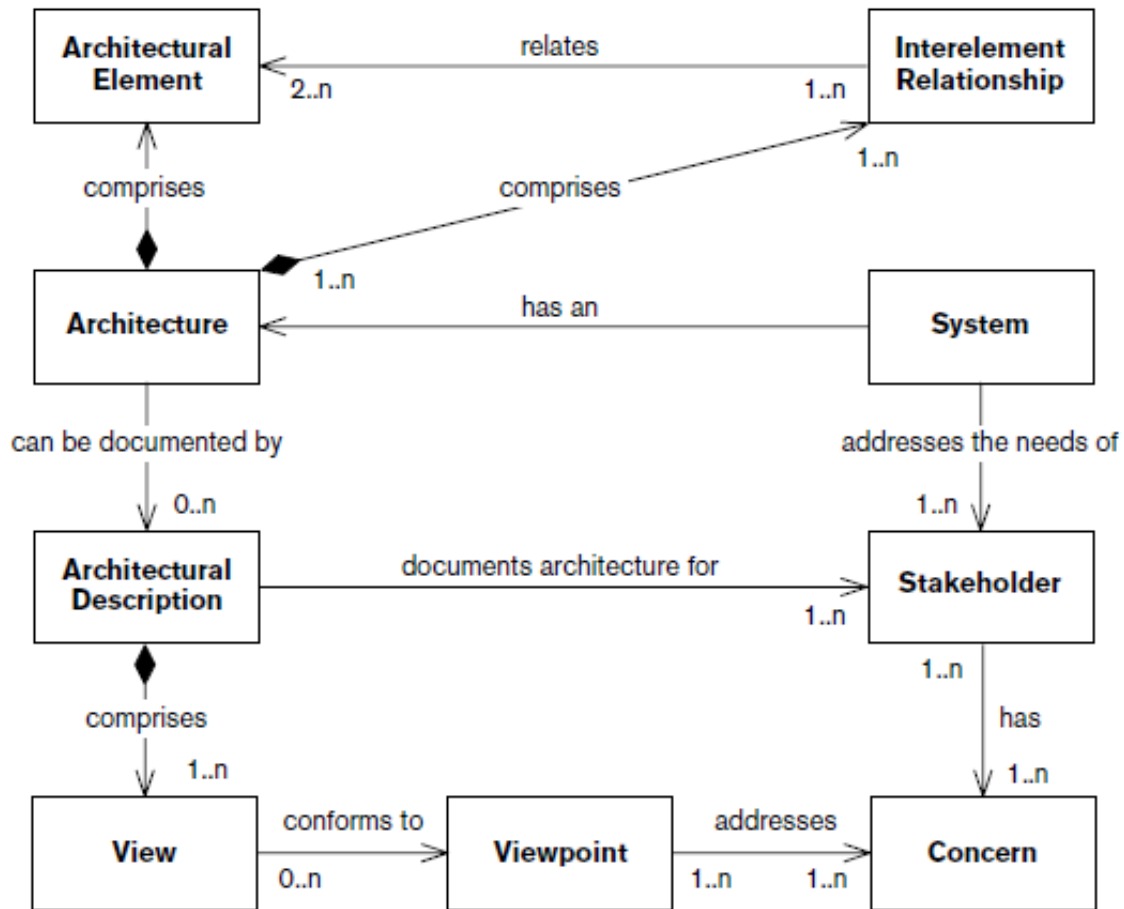


Figure 1.1: The context of software architecture. Source: (Rozanski & Woods, 2011, Ch. 3).

reflected in the viewpoint and the guidelines, principles, and template models for constructing its views” (Rozanski & Woods, 2011, Ch. 3).

There exist seven core view(point)s, each serving a different purpose, which are shown in Figure 1.2. Operational views deal with support strategies, deployment views are about hardware and runtime environments, concurrency views describe threads and interprocess communication, information views are concerned with data flow and storage, the functional views mention functional elements and responsibilities, the development viewpoint is concerned with the system’s construction and the context deals with the system’s environment. For a more in-depth discussion on views and viewpoints, please see the book by Rozanski and Woods (2011).

This thesis is primarily and predominantly concerned with the static, modular and technical structure of software architecture, i.e. the technical aspects of the functional view. To put it differently, this thesis is about the relationship between the object-oriented source code and the technical architecture.

To fully understand what software architecture is, one should also understand how the field of software architecture relates to other academic areas of research. Computer science and information science come together in the domain of software architecture, as it is not all technical and mathematical. Information science itself can be said to be the interface between business

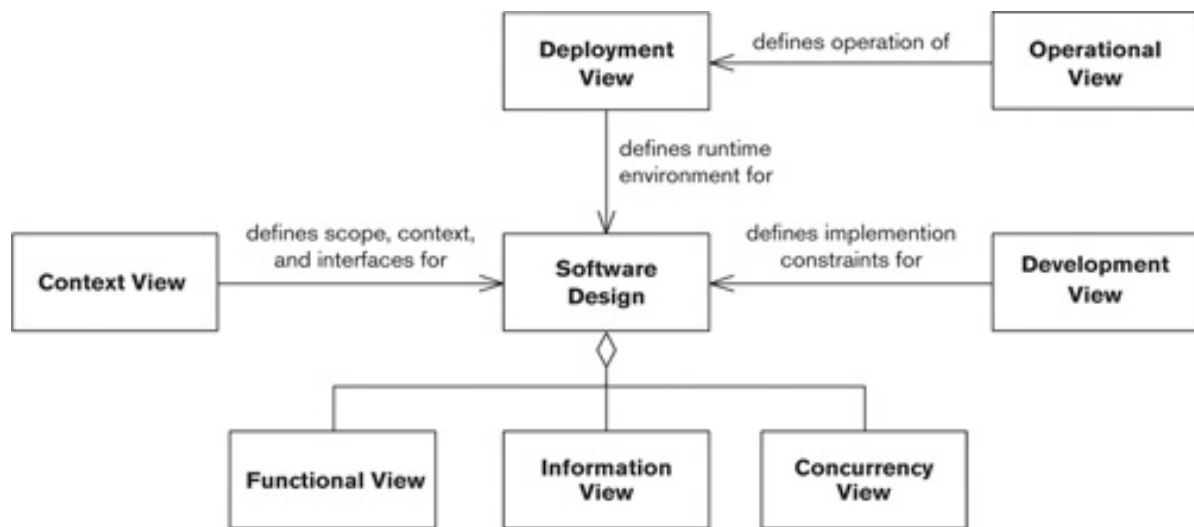


Figure 1.2: *The various viewpoints of software architecture. Source: (Rozanski & Woods, 2011, Ch. 15).*

(science) and IT, so software architecture can be seen as the most technical aspect of information science. These terms are all quite arbitrary and confusing, especially since there are some major differences in the terminology used among researchers and in educational curricula.²

Presumably, many software architects and scientists would completely disagree with this conceptualisation, but the reader is invited to look at the positioning of software architecture in the fashion illustrated by Figure 1.3. Another way to look at it would be a similar interface between Requirements Engineering and Software Engineering, as this too is the contrast between the non-technical side of business clients and their needs with the technical side of software design and development.

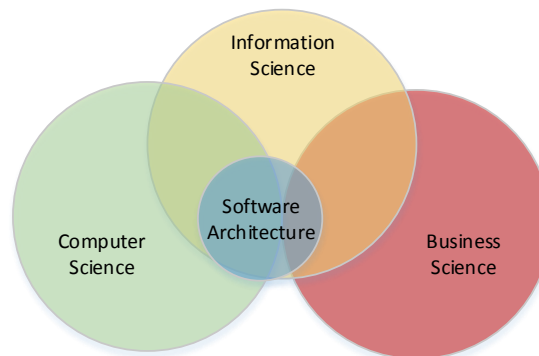


Figure 1.3: *A highly contested conceptual visualisation of what constitutes software architecture.*

The notions explained in the previous paragraphs should assist the reader not familiar with

²There even exists a language barrier. In Dutch, to give one example, computer science is often called “*informatica*” and information science is “*informatiekunde*”, although “*computerwetenschappen*” and “*informatiewetenschappen*” would be the literal translations from these English terms. But how to translate the term *informatics* then becomes slightly problematic and different solutions to this semantic problem may lead to some confusion.

the field of software architecture to grasp these and upcoming basic concepts as they apply to this thesis. Software Architecture is a young and developing field, like many other fields within the broader IT domain filled with unclear terminology and imprecise definitions. Much has been written about the role of the software architect within software producing organisations and even more anecdotes can be found from developers and IT managers about the pains of having an architect be either too technical (more of a lead developer) or not technical enough (more of a project manager). Some companies show little appreciation for the architecture of their software products, while some realise very well how absolutely vital a good software architect and good architecture documentation can be.

Software producing companies that make use of software architecture documentation often do so primarily at the start, before development (i.e. coding) has begun. It is not uncommon for the architecture to get eroded, meaning that the architecture as it was originally intended does not entirely comply with the architecture as it was actually realised by the development team(s). But because the architecture is an abstraction of the system, it is usually not very obvious that the two are not in agreement.

Sometimes, there may not even be any documentation of the intended architecture. Maintaining, updating or reusing parts of the system could then require the reconstruction of the architecture, studying the system until one can decipher what was originally intended architecturally. This is a rather difficult task, since it includes deriving the correct abstractions of the systems logical or functional elements. This is where we hope to make a contribution.

This thesis will emphasise one aspect of software architecture: static modular architectures derived from source code. We are interested in studying the possibility of using architectural patterns, commonly used designs of modular architecture described shortly, to reconstruct architectures. For this, we rely on the software tool HUSACCT, which will be further explained in the subsequent chapters. Suffice it to say, for now, that this tool can be used to design architectures or check whether an existing architecture agrees with its documentation. The main distinguishing properties of HUSACCT are that it is open-source and relies on architectural elements that are commonly found in semantically rich architectures, i.e. it understands a good amount of different architectural elements such as module types.

The research approach will be discussed before delving into the core subject in the next chapter. After that, this thesis will look into software architecture reconstruction (Chapter 3), the role that architectural patterns can play (Chapter 4), and the way a software tool like HUSACCT can be used for these purposes (Chapter 5). The algorithms are presented in Chapter 6. Then, the approach undergoes some preliminary evaluation (Chapter 7) before future work is discussed (Chapter 8). After the Conclusion (Chapter 9) and the Acknowledgements (Chapter 10), the reader can find the bibliography and the appendices filled with tables, images and code samples that were deemed too excessive for the main body of the text.

Chapter 2

Research Approach

As explained in Chapter 1, the aim of this research project is to develop tool support for detecting the use of architectural patterns in architecture reconstruction, using HUSACCT as a framework for development of this tool support by extending it and making use of its rich semantics to empower the method.

“HUSACCT” is an acronym that stands for “Hogeschool Utrecht Software Architecture Compliance Checking Tool.” “*Hogeschool*” is Dutch for *high school*, although this refers to tertiary education and not secondary, as opposed to the American term. Specifically, this is a university of applied sciences, higher vocational university, or college. The University of Applied Sciences Utrecht (the Netherlands) produced an Architecture Compliance Checking (Chapter 3) tool in 2011 as part of an advanced software engineering course. This software tool has been further developed in the subsequent years, with several individuals contributing up to this very day through the open source repository service Github. More will be said about HUSACCT in Section 3.2.

This thesis represents a type of science that falls within the domain of design research. The research process is iterative by nature, meaning that different sub-problems are studied and solved consecutively. The steps of design research are 1) analyse, 2) design, 3) realise, 4) evaluate. Due to the time restrictions of the master thesis project, later iterations of the research are described on a purely theoretical basis, lacking physical realisation (code) and some are reserved for the future work section in their entirety. It is important that a design research approach leaves some room for such considerations, but it is also of great import that concrete goals are set. For this purpose, the next section will elaborate on the main research question and its sub-questions.

2.1 Research Questions

In order to understand our main research question, two things have to be defined. First, a candidate pattern is a particular mapping of several software modules found by modular decomposition to an architectural pattern, with which certain numbers of violating and non-violating dependencies are associated. A pattern instance is an architectural pattern that is genuinely present in the intended architecture. By *genuinely*, we mean the distinction between an architectural pattern that seems to be a good match for the realised system, and the pattern that was actually used by the software architect(s) in the intended architecture. Therefore, the main research question is defined as:

RQ: *How can architectural patterns be identified from source code in such a way that the correct pattern instances, those that were genuinely realised, can be found in a, possibly undocumented, software system?*

Because such an open research question consists of several minor questions, sub-questions should be clearly defined. The ordering of these sub-questions could be taken to imply an ordering by priority. In this case, the ordering refers to the absolute times at which the answers to these questions were anticipated.

Initially, the literature study should answer a question on the research topic's background. Using the definitions of Software Architecture Reconstruction (hereinafter SAR) and Architecture Compliance Checking (hereinafter ACC) as provided in Section 3.1, this sub-question is formulated as:

SQ1: *How are architectural patterns used in Architecture Compliance Checking and Software Architecture Reconstruction?*

After this research based on existing literature, the various methods of decomposition into software modules must be investigated. This is due, in part, to the envisioned reconstruction approach that will be shown in Section 5. Subsequent pattern matching approaches would rely on this decomposition step. It was decided at this point in time that the initial tool support is already provided in this regard, namely decomposition based on Java package definitions or C# namespaces, but a detailed description of this and alternative techniques had to be provided in the final thesis nonetheless.

SQ2: *How can the individual software modules that correspond to an architectural pattern instance be correctly identified in an application's source code?*

Decomposition is one part of the problem, pattern matching is another. Delving slightly deeper into the literature, the question of how exactly a pattern might be detected must be investigated.

SQ3: *How can candidate architectural patterns be identified within a software system?*

Although this question is already partially answered by the literature study, it cannot be fully satisfied until this is realised within HUSACCT and any problems that may occur have been dealt with.

The question remains of how the set of candidates can be whittled down to a few or even a single possible candidate. There has to be some way to compare candidates and find an optimal solution. The first question is how to address the fitness metric. We want to somehow express the degree to which an architecture complies to a particular pattern, the fitness of that pattern candidate. But formulating a function that describes this fitness is not trivial, there may be several formulations that would prove useful.

SQ4: *How can the fitness of a candidate architectural pattern be expressed?*

Based on this, various candidates must somehow be compared.

SQ5: *How can multiple candidate architectural patterns be compared with one another?*

It is not trivial how this comparison leads to a definite conclusion, so this is also a sub-question. The “best” candidate would be the most likely, the most plausible, the most acceptable pattern candidate, because it may just be a genuine pattern instance. Thus:

SQ6: *How can the best candidate architecture patterns be chosen from a list of candidates?*

Once the goals of the research have been established through the listed research question and sub-questions, another issue that remains to be addressed is the rationale of this thesis: why is it relevant to detect the use of architecture patterns with a tool like HUSACCT?

2.2 Relevance

This research project is focussed on the overlapping area of several other topics of scientific rigour: Software Architecture Reconstruction, Architecture Compliance Checking, architectural patterns, design patterns, graph matching, software development, etc. This particular focus, namely the (partial) recognition of architectural patterns in static architecture reconstruction using a modular decomposition based on packages/namespaces and a subdivision of dependencies, modules and rules into various types, appears to be quite unique.

Uniqueness alone does not imply usefulness. As indicated by Pruijt, Köppe and Brinkkemper (2013), however, semantic richness can be a powerful tool in Software Architecture Reconstruction. Since such semantics are currently so underdeveloped and underused elsewhere, this richness allows for much contribution to be made to the scientific knowledge base.

Additionally, as indicated by the collection of papers on the subject, detecting architectural patterns can be a great aid to architecture recovery. The combination of the aforementioned research areas in this thesis can thus be said to provide a scientific contribution to a field in need of maturity.

Research topics as presented here provide purely academic benefits first and further social or market applications later. The results of this project are not immediately user-friendly or even usable beyond an academic context, since it would be of little academic interest and a complete waste of time for a master thesis to overly concern itself with the design of a user interface beyond what is presented in Section 8.1. As such, the results of this thesis are restricted to hard-coded methods. Nevertheless, the fact that HUSACCT is a free and open-source product allows for almost immediate use by third parties. Also, the intention is that this thesis results in more research and development, for which the UI will most definitely be interesting and not at all wasteful in terms of time spent, if only to provide some initial ideas.

Bass et al. (2012, Ch. 2) mention that some 80% of all costs in software development are related to maintenance activities. Architecture reconstruction, and the Architecture Compliance Checking that is automatically enabled by it, is thought to be advantageous in this regard. In conclusion, although it might be relatively haughty to speak of scientific and especially social relevance for an exploratory design research project such as this master thesis, one might reason that it is indeed a relevant topic of research.

Chapter 3

Architecture Reconstruction

In this chapter, the differences between reconstruction and compliance checking are elucidated, as are the software tool HUSACCT and its meta-model for software architectures. These concepts need to be understood before the bulk of the research can be considered, since these are the fundamentals upon which the whole thesis is built. The chapter is concluded with a running example that illustrates how HUSACCT can be used at this time.

3.1 SAR & ACC

In Software Architecture Reconstruction (SAR), the architecture of an already realised system is re-engineered. “Architecture reconstruction is an interpretive, interactive, and iterative process involving many activities; it is not automatic” (Bass et al., 2012, Ch. 20). In this book, two main purposes of reconstruction are listed: documentation and conformance checking. This means that a re-engineer (someone who is re-engineering the system, also called re-architect or some similar term) tries to understand the software architecture of a software system, which is rather difficult to do even if the system is relatively small.

Why might someone want to do such a thing? For a number of possible reasons, in fact, most notably because a particular system might have no proper architecture documentation and someone is in desperate need of some good descriptions. Perhaps a legacy system has to be modified or extended, or several systems have to be combined. It is not uncommon that there is some documentation that is hopelessly outdated or otherwise flawed. Another reason could be that a particular structure or view was not described in the original document and now it has become of the utmost importance to current development or maintenance. Whatever the reason, reconstruction can be an essential albeit challenging task for the re-engineer to perform. Luckily enough, there is tool support.

Bridging the gap between the actual architecture and what was originally intended or, at least, how it is currently documented can be treated as a search for violations, software dependencies that ought not to be there, which can result in updating the documentation or adapting the architecture. This documentation verification process is known as Architecture Compliance Checking.

ACC is subtly different from SAR. It is a process where existing documentation is taken into account. Perhaps the documentation is outdated, but not hopelessly so. Perhaps it is incomplete, but not entirely useless. Or perhaps you just want to see if the development team built as they were bid and did not ignore the designs of the software architect as they realised

(or implemented, as programmers tend to say, perhaps erroneously¹) the software system you have been anticipating.

Two concepts are strongly tied to the rationale behind doing ACC or SAR: architecture erosion and architectural drift, which are real problems in software architecture. The first is defined as “the introduction of architectural design decisions into a system’s descriptive architecture that violate its prescriptive architecture” (Taylor, Medvidović & Dashofy, 2009, Ch. 3). Architectural drift is a highly similar concept, where changes in the descriptive architecture (other terms include implemented, realised and as-built architecture) may not be described in the prescriptive architecture (also known as intended, defined, as-designed architecture and so on) but do not actually violate it. ACC is therefore focussed on erosion but not on drift, as the latter is by definition not detectable through violations. Static ACC tools use the modular decomposition of source code and dependency relations between these modules to check for compliance with user-defined architecture rules (Pruijt, Köppe & Brinkkemper, 2013).

So where does ACC stop and SAR begin? Is the possession of existing architecture documentation the be-all and end-all distinction in these matters? Consider the following scenario: we have no documentation and are trying to reconstruct a technical component of some software system. We look through the source code, draw a number of diagrams on a white board and at some point we think we might start to fathom at least a portion of the structural decomposition. What do we do? We verify our hypothesis, we try to find out whether our idea agrees with reality. We check whether the architecture we just drew on the white board complies with the actual architecture of the software system. So, how is that different from compliance checking? The source of the hypothesis is documentation in the case of ACC or someone’s ideas with SAR, that is essentially the distinction.

Because ACC and SAR blur together in such circumstances software tools intended for ACC support may well be used for SAR purposes. This is a fundamental insight for this thesis (albeit unlikely to be an original one), as it allows for a reconstruction approach with an ACC tool at its very core. However, how does this translate to the chosen ACC tool used in this thesis, HUSACCT?

3.2 SRMA & HUSACCT

As mentioned in Chapter 2, this thesis’ working tool is HUSACCT, developed by Leo Pruijt at the University of Applied Sciences Utrecht as part of his PhD thesis at Utrecht University. HUSACCT is obviously made for the purposes of Architecture Compliance Checking, but it can also be used to support a re-engineer during the process of Software Architecture Reconstruction. The reason why this is possible boils down to the argumentation as to why ACC and SAR overlap, presented in Section 3.1.

HUSACCT’s distinctive characteristics include its support for rich sets of module and rule types (Pruijt, Köppe, Van der Werf & Brinkkemper, 2014). It allows the user to define a software architecture using a series of modules, rules and exceptions with relation to specific kinds of dependencies. It then allows this architecture to be compared to the architecture as derived from source code, in either Java or C#. By mapping the discovered software elements to the defined

¹*Implementation* is used by IT managers, CEOs and information scientists to refer to a system being adopted by and inserted into a real-life organisation, while they call the actual creation *realisation*. Developers and computer scientists tend to think of *realisation* as too broad a term for the *implementation* of software design. The question on the interchangeability of these terms is debatable though ultimately meaningless, in this student’s humble opinion, due to the fluidity of language and semantics in the real world. Since academic writing demands technical precision, this thesis will favour the information scientists’ terminology.

architecture, violations of architecture rules can be identified by the user. Different severity ranks may even be used to highlight certain types of violations over more minor irregularities.

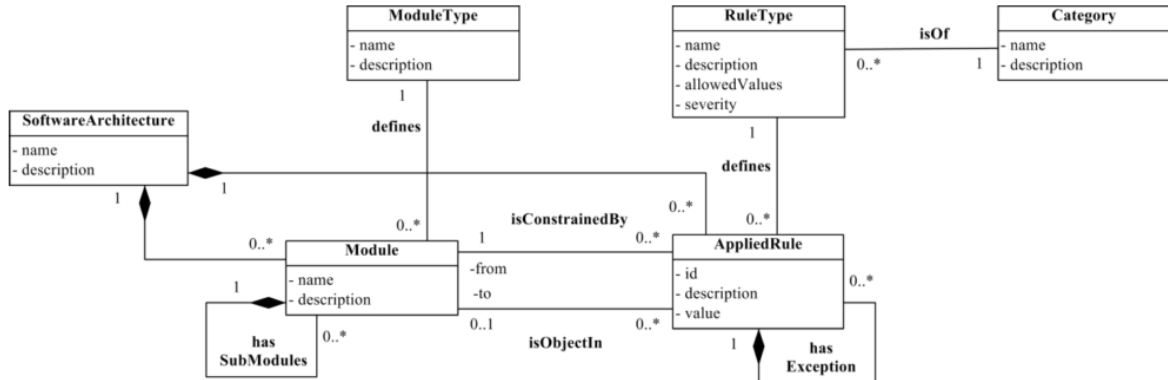


Figure 3.1: The meta-model of Semantically Rich Modular Architectures (SRMAs). Source: (Pruijt & Brinkkemper, 2014).

Seven tools for the purpose of ACC were compared by Pruijt, Köppe and Brinkkemper (2013), when the first version of HUSACCT had already been created, showing that on average 74 % of dependency relations were discovered in a benchmark test. These tests resulted in a series of recommendations, such as enriching violation messages with a severity ranking. It partially underlines the strength of HUSACCT, with its semantic richness described in another paper (Pruijt et al., 2014).

A portion of the meta-model of Semantically Rich Modular Architecture Compliance Checking, as was published in a subsequent paper, is displayed in Figures 3.1 and 3.2, showing the relationships between the different concepts. Figure 3.1 shows, for instance, that both modules and rules consist of multiple types. Multiple rules can apply to the same module and a rule can apply to any number of modules, albeit that exceptions are allowed. It is important to note that a sub-module has only one parent module, although that parent can itself be a sub-module of another individual parent module.

In Figure 3.2, the relationship between dependencies and violations is most relevant. Each dependency is of a specific type and each violation breaks a specific type of rule.

SRMAs combine various types of modules with rule types to express architectural elements and their constraints. This enhances expressiveness and supports architecture reasoning in terms comparable to regular language (Pruijt & Brinkkemper, 2014). Using the module and rule types understood by HUSACCT, we want to express architectural patterns.

HUSACCT supports the following types of software modules: subsystems, layers, components, interfaces and external systems. Subsystems are modules with clear responsibilities. A layer has the additional property of a hierarchical level, which enforces strict layering as HUSACCT automatically adds rules banning skip-calls and back-calls to the relevant levels. Components are modules whose contents are hidden behind and accessed through an interface module. Finally, external systems represent libraries, modules that are not actually part of the system under consideration (Pruijt & Brinkkemper, 2014)(Pruijt et al., 2014).

The rule types can be placed in one of two categories: property rule types and relation rule types (Pruijt & Brinkkemper, 2014). The former consists of conventions such as naming and inheritance. Only the façade convention is used in this thesis. This convention states that interface Module A always has to act as the interface for component Module B. The second category, relation rule types, consists of a further subdivision into: “Is not allowed to use” and

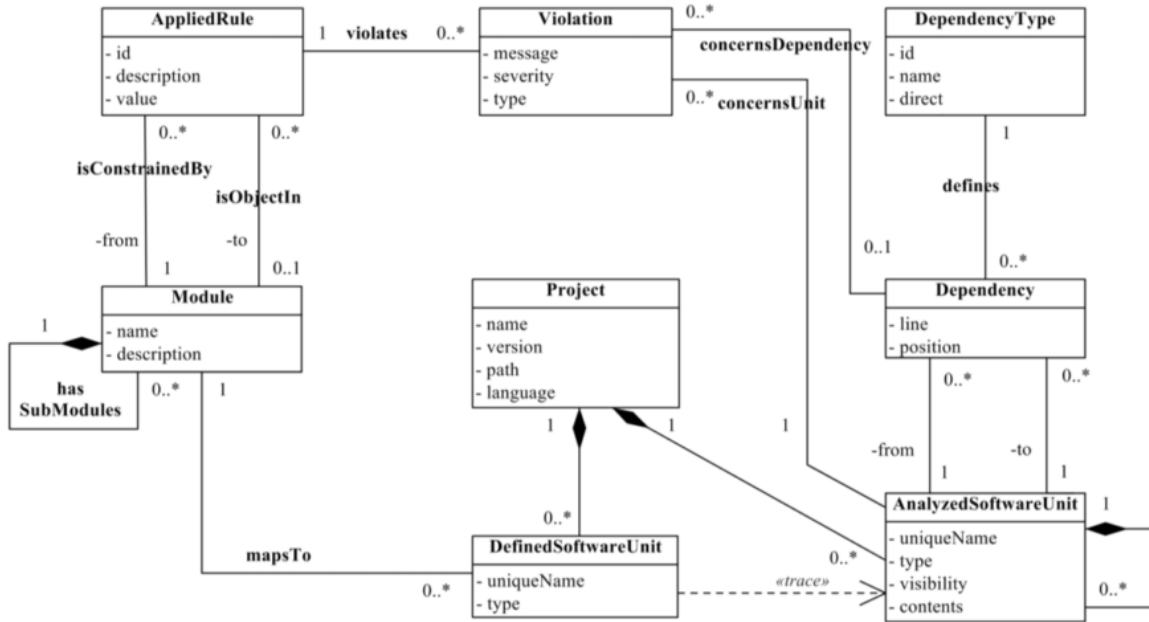


Figure 3.2: The meta-model of Architecture Compliance Checking. Source: (Pruijt & Brinkkemper, 2014).

“Is allowed to use” rules, which are themselves two basic rule types that can be used to express rules like “Module A is not allowed to use Module B”. These rule types are further explained in Section 4.2.

In the next section, we will look at a brief illustration of what HUSACCT can do.

3.3 Running Example

Starting off, “we”, i.e. the architecture re-engineer(s), have the source code of an undocumented system. Its software architecture is currently unknown and too large for a mere mortal to comprehend by simply reading the code. What we do have, however, is the HUSACCT tool. It allows us to study the static dependencies in the code, so that it is perfectly clear to us which classes call on which, what inheritance relations hold, etc. This dependency analysis is all to a certain degree of completeness, though, since no tool guarantees that all relations can be found.

Without any kind of documentation to go on, we would have to make several poorly supported decisions. Depending on our process, we could start with grouping the most interconnected modules, essentially assuming the system was designed with the guidelines of low coupling and high cohesion in mind. We could allow naming conventions and regular expressions to guide our efforts, slowly building up to an architecture that makes sense, i.e. complies with our biases and preferences. Or perhaps we could use a tool/algorithm to identify design patterns, hoping to come to a better understanding of the architecture that way.

Instead of a bottom-up approach, we could proceed in a top-down manner by defining large modules first and determining which sub-modules fit them best. We could, for example, assume the system was created with Module-View-Controller in mind and thus force the implemented modules into this pattern. In order to do this properly, we might look at comments, names for classes and methods, or perhaps external libraries that are accessed.

One would presume that it is in the interest of the re-engineer to have as many options and techniques available as possible, allowing him or her to select those most appropriate in a given situation. Although static architectural patterns have been noted as a possible approach, their use in a more bottom-up fashion seems, as of yet, underused in the scientific literature.

Ideally, we want to be told by a tool that the architecture under analysis contains a specific architectural pattern. Instead of forcing Module-View-Controller upon a system, we want to know how likely it is that this pattern was used and whether other patterns might actually be more suitable.

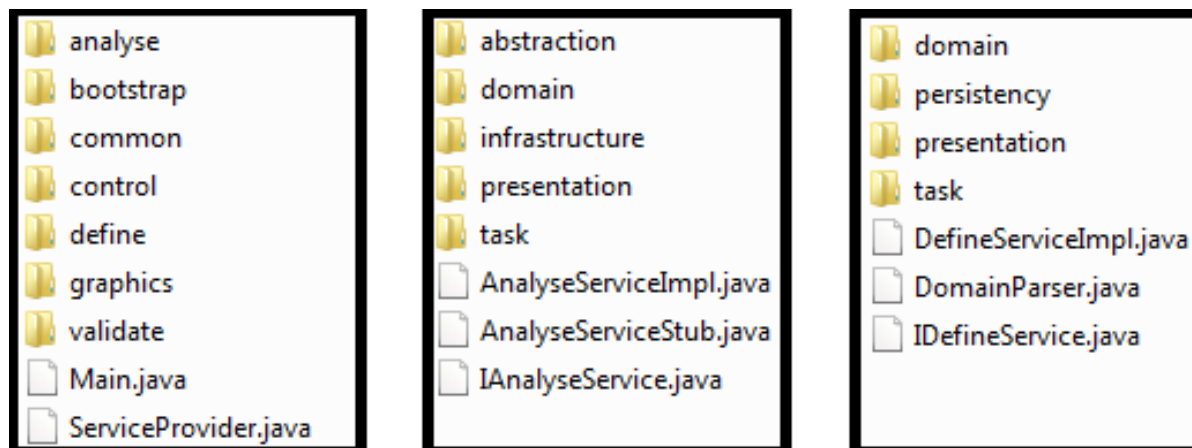


Figure 3.3: The root, Analyse and Define folders in the HUSACCT v1.0 source directory.

If we look at Figure 3.3, we see some code files from HUSACCT 1.0, which was built in 2011. On the left is the root directory, and to the right are two components of the architecture on display, namely Analyse and Define. The former is responsible for analysing an existing system and registering as many of its dependencies as possible, while the latter is concerned with the user defining an architecture to which the actual architecture can then be mapped (either based on documentation in ACC or the user’s ideas in SAR). Just by looking at the source code directory, we cannot be sure what the architecture looks like, as it would be a rather crude guess. And yet, there is too much code within these folders (113,155 LOC) to go through it all.

Using HUSACCT itself (version 4.1) to analyse these files, we get Figure 3.4. This is based on package definitions within the Java code. There is no reason why this has to comply with the source directory, but we can see that there is some similarity. Package definitions also do not have to reflect the intended architecture, but as we will see, they most certainly might. Modules are linked to each other by a variety of dependency types and sub-types. We can tell that some elements are more interconnected than others. Although not yet known to HUSACCT, Analyse and Define are modules of the component-type, which means they both have an interface (IAnalyseService and IDefineService).

In a perfect world, either the directory structure or the packages/namespaces could show exactly how the architecture was designed by corresponding to the appropriate modules. Instead of relying on these, one could rely on a clustering algorithm to create various modules based on coupling. Since the re-engineer would have no knowledge of the number of modules, the clustering approach would have to be one that determines an optimal amount of clusters (e.g. hierarchical clustering). HUSACCT finds a large amount of dependencies compared to many tools, including underused types such as Import statements (Pruijt, Koppe & Brinkkem-

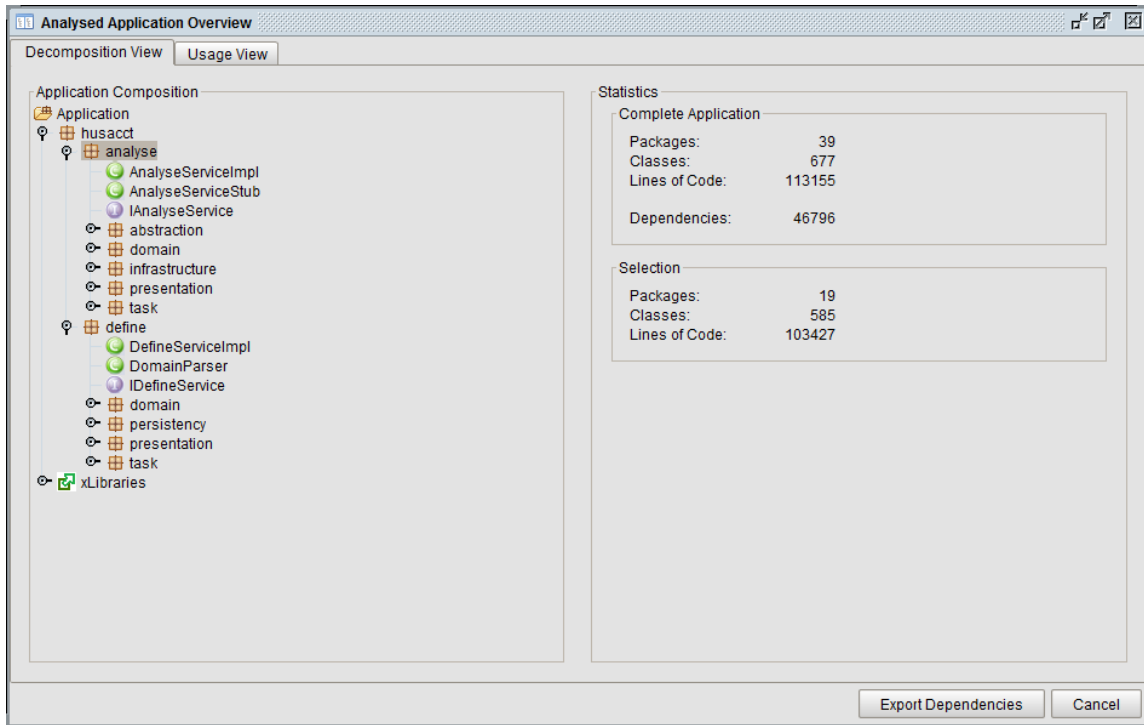


Figure 3.4: *HUSACCT's analysis of the two components, with the Analyse component selected.*

per, 2013)(Pruijt, Köppe & Brinkkemper, 2013), which provides a good basis for dependency analysis and perhaps also clustering/layering.

Based on these modules, we want to make an educated guess as to what the original creator(s) intended for this system. We are going to attempt to do this by looking at architectural patterns. The higher level modules are fed to a pattern matching algorithm that attempts to map known architectural patterns to the dependency graph. Contrary to design patterns, architectural patterns do not specify particular dependency types and imply an even greater implementation variety. As such, pattern violations are to be expected and do little to indicate that an architectural pattern was not intended. The pattern with the smallest number of violations should, however, be the most plausible one.

We use the following definitions, here and throughout the thesis:

Candidate pattern:

A particular mapping of software units to pattern modules in the defined architecture.

Pattern instance:

A pattern candidate that is genuinely part of the actual architecture.

As a result of the matching algorithm, we receive a list of architectural patterns, their mappings to various modules and the number of violations associated with each candidate pattern. By studying the mapping and looking at module names, we can determine the most likely intended pattern instance. We subsequently come to the conclusion that two architectural patterns

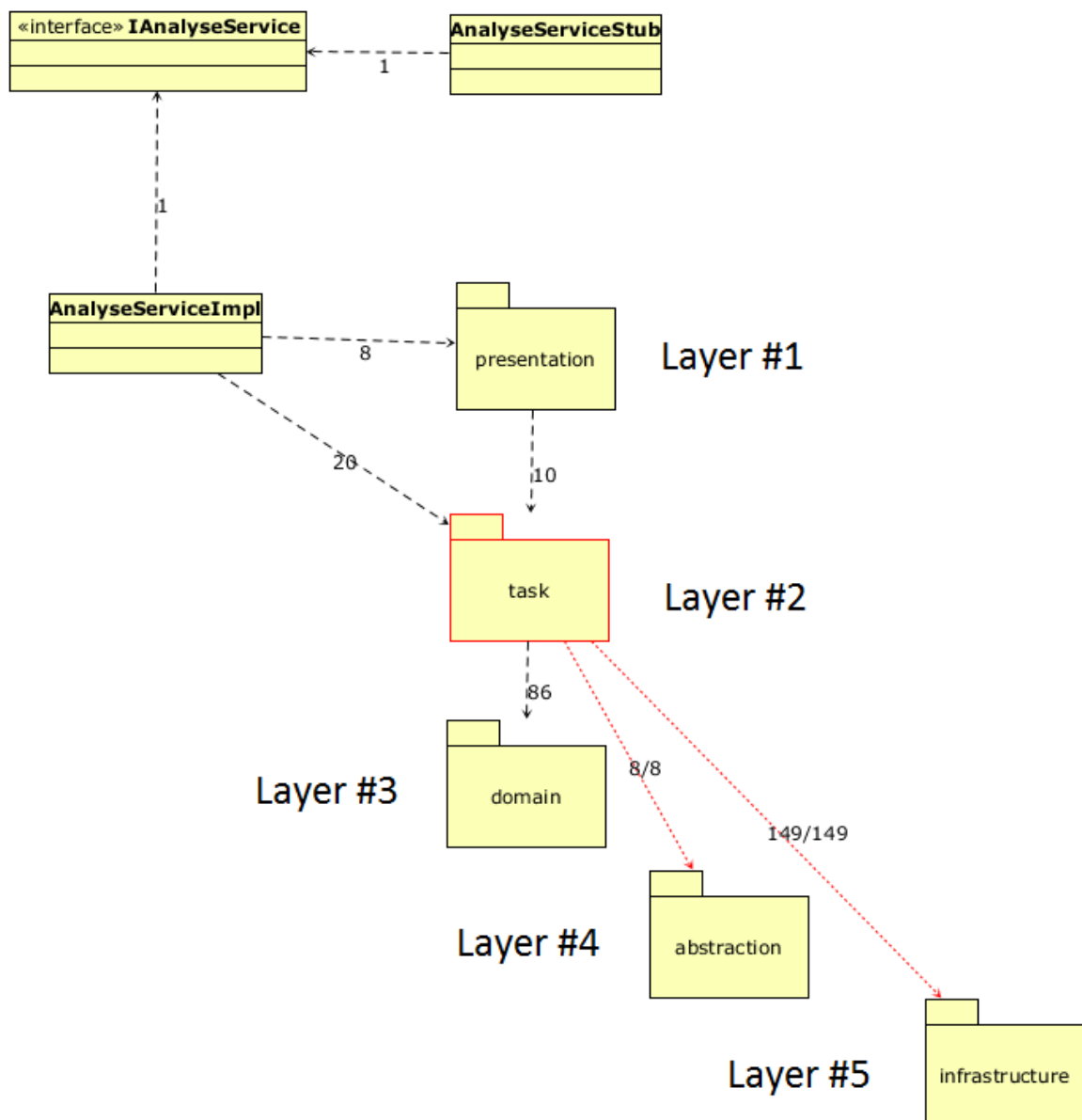


Figure 3.5: A 5-Layered pattern in the Analyse component, with violations in red.

were used here: a slightly violated 5-Layered pattern in Analyse (Figure 3.5) and a more severely violated 4-Layered pattern in Define (Figure 3.6).

Although one would not normally have architecture documentation in a reconstruction process, the documentation of HUSACCT is available and these two pattern instances match the descriptions rather well, including the violations of the pattern rules. Task (as shown in Figure 3.5) is allowed to use Infrastructure for external libraries and Abstraction for export mechanisms.

The documentation on the Define component is incomplete for HUSACCT v1.0, so its pattern would not be verifiable were we to use this example as a test. However, it does seem like a similar layered structure, albeit one layer fewer. The most dubious set of relationships is that going up from Task to Presentation, since here a layer is not circumvented (which is a common exception with layered patterns), but a layer uses the layer above. It was confirmed by

Dr. Pruijt, since he was HUSACCT's primary creator, that these dependencies were unwanted pattern violations of which he was well aware.

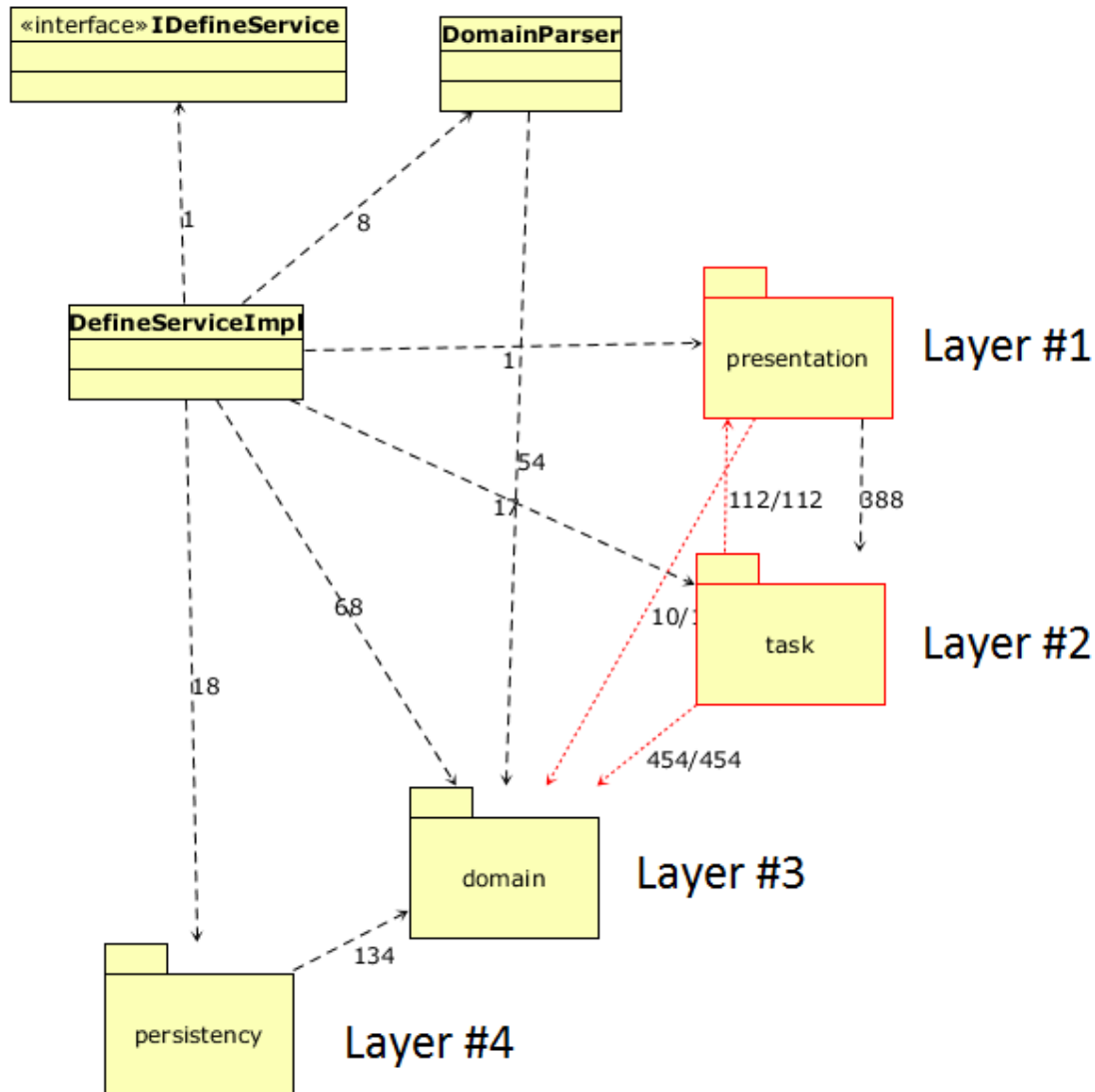


Figure 3.6: A 4-layered pattern in the Define component, with violations in red.

This process can be repeated within different parts of the architecture, which may contain multiple patterns, more complex (user-defined) patterns, or unhelpful names in their package definitions. If successful, one can thus come to a greater understanding of the intended architecture. As such, this thesis' approach stands out from similar approaches due to the fact that it looks at architectural patterns specifically instead of focussing on design patterns and that it relies on HUSACCT, which incorporates many types and sub-types of static dependency relations as well as various types of modular elements. This latter aspect allows for rich definitions of architectural patterns in terms of components, subsystems, layers and external libraries.

In the next chapter, we will look at architectural patterns; at how they relate to design patterns and how they can be defined in terms of HUSACCT's architectural language.

Chapter 4

Architectural Patterns

Architectural patterns are a kind of reusable software design intended to prevent architects and developers from trying to reinvent the wheel. Architectural patterns establish the relationship between a context, a problem and a solution (Bass et al., 2012, Ch. 13). Throughout the scientific publications, the various terms for these patterns are favoured by different authors, such as architectural *styles* instead of *patterns*. Some authors may consider there to be a distinction between styles and patterns, e.g. Taylor et al. (2009), whereas others may simply consider them to be interchangeable concepts. Throughout most of the thesis, *architectural pattern* will be the preferred terminology, as this is more similar to the term *design pattern*. Since these two kinds of pattern may be considered extremes of the same spectrum, with architectural patterns representing system level design and design patterns more detailed design, similar sounding terms seem most appropriate. The two kinds of pattern represent the same idea (proven, maintainable, scalable, reusable design), though on a different level of abstraction whilst lacking complete clarity on what would be the distinguishing level.

We do use a notion of *style*, though. We take style to be the overall pattern, such as layering. This really only becomes relevant if we are concerned with additional patterns as well, such as when this layered style contains more detailed patterns within the layers.

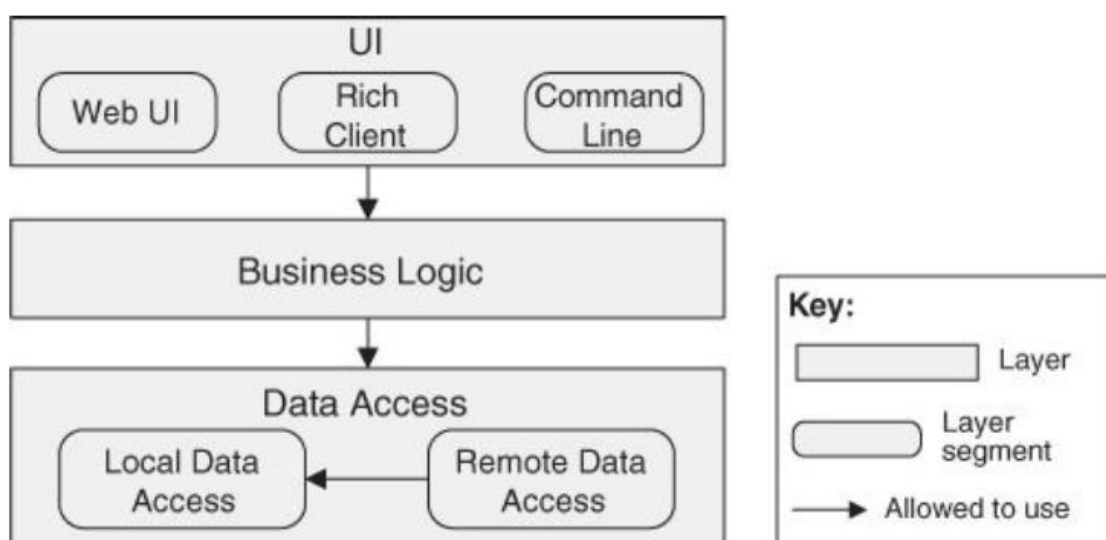


Figure 4.1: An example of a Layered pattern. Source: Bass et al. (2012, Ch. 13).

Architectural patterns exist on a high level of abstraction, which means that they have a less concrete mapping to source code than other kinds of reusable design. To illustrate what such a pattern would look like, let us take a relatively straightforward pattern. Figure 4.1 is an example of an N-Layered pattern. Specifically, this is a 3-Layered solution with additional segmentation within the layers. A pattern is incomplete without architectural rules, which forbid some of the pattern's modules from interacting. Often, architecture rules connected to such a pattern are not apparent in such a diagram. In this particular case, the “Not allowed to use” rules, or however else one might formulate them, are completely absent, but the main point of a layered architecture is that classes within a layer may only call on classes in that same layer or the layer below. They may not call on the layer above (back-calls) and they may not circumvent a layer (skip-calls). UI is not to call on Data Access, Business Logic is not to call on UI and Data Access is not supposed to call on either of the other layers.

If this were the architecture of an existing system and its software producer wished to build another type of system that would work differently underwater but utilise the same user interface, it would be fairly convenient to simply take the UI layer and reuse it. Not only would the layer be a single structure, its connections to the other layers ought to be relatively simple. As long as the pattern was used correctly, the UI layer should only be connected to the Business Logic layer thanks to the skip-call rule, and only in one direction of dependency thanks to the back-call rule. Similarly, the same system can be smoothly shifted from a standalone to a browser-based interface by replacing the UI layer.

This is just an example of reusability being benefited, but similar advantages apply to attributes like maintainability and extensibility, to mention two examples. All in all, this architectural pattern essentially adds the benefits of object-oriented design to a modular software architecture.

Another well-known architectural pattern is presented in Figure 4.2: the Model-View-Controller pattern. There happens to be some similarity between this pattern and the diagram in Figure 4.1, because of the separation of the presentation portion of the system into a module of its own. The same principle of the diagram merely implying architectural rules is at work here. The idea is that the allowed dependencies are, in fact, the only dependencies that are allowed between these three parts of the system.

This pattern is sometimes described as a combination of three design patterns, a class of patterns discussed in the next subsection.

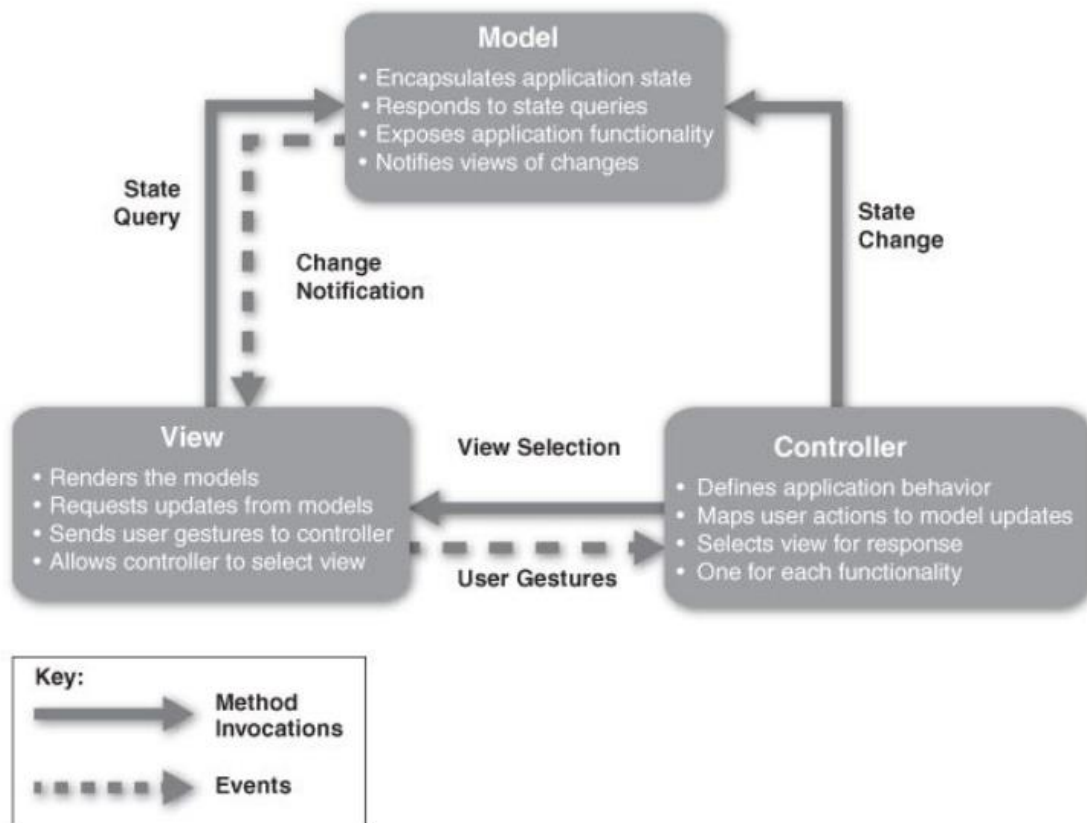


Figure 4.2: An example of a Model-View-Controller pattern. Source: Bass et al. (2012, Ch. 13).

4.1 Design Patterns

Design patterns were first introduced by the so-called *Gang of Four* (GoF), Gamma, Helm, Johnson and Vlissides (1994). Each design pattern describes a template for solving a particular object-oriented software design problem, accompanied by motivations and consequences. They are essentially well-documented solutions to common problems, establishing a terminology and promoting reuse of the designs of others. Examples are the Adapter pattern (Figure 4.3), which includes converting an interface into one or more usable by client classes, and the Bridge pattern (Figure 4.4), which says to decouple an abstraction from its implementation in order to allow both to vary (Shalloway & Trott, 2004). There is no need to go into further detail here, suffice it to say that design patterns are established instruments in software systems design, even if there are some who object against relying too much on them.

Design patterns can be relevant to SAR and ACC since they are so widely used in software design that recognising them can assist the re-engineer in understanding the intended architecture. Design patterns tend to be more familiar to developers than architectural patterns, which are more the domain of software/system architects. However, architecture reconstruction approaches that use patterns in some way, whether as input or as output, tend to use either of the two types of pattern. In order to evaluate existing approaches to pattern-based reconstruction, it is imperative that the distinction between the two kinds is abundantly clear. We identify the following five distinguishing characteristics: scale, specificity, frequency, strictness, and the sign

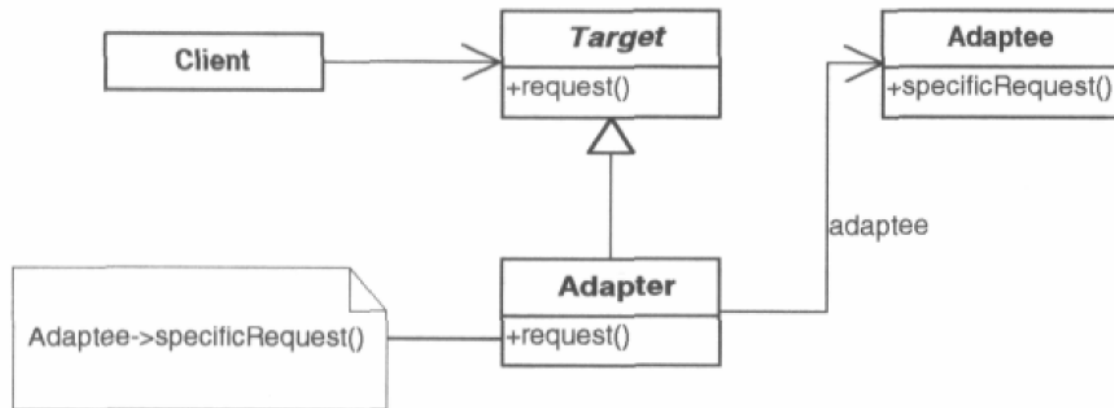


Figure 4.3: A generic depiction of the Adapter pattern. Source: Shalloway and Trott (2004).

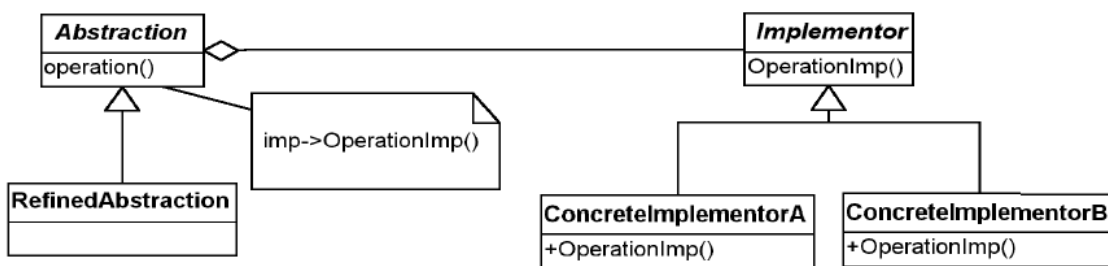


Figure 4.4: A generic depiction of the Bridge pattern. Source: Shalloway and Trott (2004).

of rules. Some of these characteristics might be seen as consequences of the difference in scale, in fact. They represent our response to the questions asked about the distinction between the two kinds of pattern raised in Chapter 13 of the book by Bass et al. (2012).

Scale: Design patterns tend to be about individual classes of code or perhaps small aggregates of such classes, whereas an architectural pattern might split a whole system into a number of divisions. Thinking of the modular hierarchy as a tree, architectural patterns exist near the root, whereas design patterns exist near the leaves. This is by no means a clear division, however, since design patterns can exist on several levels of abstraction and an element in an architectural pattern may well be identified with a large class in the source code. Although in the latter case, this class is likely to be massive in size and might have an extremely low cohesion, i.e. it does too many different things. This is strongly in line with the emphasis on cohesion and coupling in the book by Shalloway and Trott (2004) about design patterns. If this were true in all cases, one could state that when a single class corresponds to a single element of an architectural pattern one-to-one, then that class has been poorly written according to object-oriented design principles. But this is by no means an exhaustive or accepted rule.

Specificity: Design patterns are concerned with properties of rather high granularity, such as class abstraction, method visibility and instantiation (like in Figure 4.3). One class is supposed to be the child of another and it is clear which class instantiates this child. Architectural patterns,

on the other hand, speak of software elements in much broader and vaguer terms (Figure 4.1). There is no mention of instantiation or inheritance here. This implies that the existence of specific properties of small scale has little or no relation to the existence of architectural patterns. Another way of phrasing this might be that architectural patterns are more language agnostic than design patterns.

Frequency: If an application has been made with object-oriented design principles in mind, one would expect to see the same design patterns over and over again. A large system may contain the Bridge pattern from Figure 4.4, for example, hundreds if not thousands of times. See Dong, Sun and Zhao (2008) for examples of phenomenon. Such a system may also contain several instances of the same or similar architectural patterns, but it is unlikely that these will be anywhere near as frequent. Approaches that look for frequent patterns therefore have a significant disadvantage when trying to identify architectural patterns.

Strictness: One might come up with one's own version of a particular design pattern, but this is not highly encouraged among developers. Most deviations from a design pattern are simply taken to be rule violations. Books like those by Gamma et al. (1994) and Shalloway and Trott (2004) do not claim that there are no more design patterns to be formulated, but the specificity of their pattern definitions do imply a kind of strictness. The rules of an architectural pattern, however, are almost literally made to be broken, in the sense that exceptions are to be expected. This implies that the number and nature of violations of a pattern's rules are far more relevant for identifying architectural patterns than for design patterns, the latter of which could often be found by looking for an exact match. A useful design pattern detection algorithm may be unable to identify a previously unknown variation; an architectural pattern detection algorithm that only recognises instances of strict adherence to the patterns' rules is quite useless.

Sign: The specification of design patterns is usually about which classes use which, who instantiates whom and what the method calls ought to be. Compare Figures 4.3 and 4.1, for instance. Architectural patterns' rules contradict this observation, as they tend to be most clearly and concisely explained using negations of legality: this layer is not allowed to call on that layer, these modules are not supposed to use those modules directly, etc. These rules can be said to be of a different sign, meaning that they are negative whereas the design patterns' rules are positive, which implies that architectural patterns are more so defined by the absence of certain relationships than by their presence. Nevertheless, as is discussed later on, architectural patterns can make use of "positive" rules in order to enforce that a collection of architectural elements actually constitutes an architectural pattern, as opposed to being merely a set of elements that happen not to break any rules.

In short, design patterns are usually small, specific, common and strict, whereas architectural patterns are far more grand, vague, uncommon and loosely defined. Additionally, architectural patterns might be better described using bans, instead of requirements, than design patterns. With architectural patterns sufficiently distinguished from design patterns, the following section will look at the specific definition of architectural patterns within HUSACCT's framework of available rules and modules.

4.2 Pattern Definition

As has already been noted, architectural patterns could be defined using HUSACCT's terminology of rules and exceptions. This is, however, not entirely clear cut. HUSACCT essentially links two graphs: the one defined by the realised architecture and the one defined by the user. We wish to replace that defined architecture with an architectural pattern in order to look for

Table 4.1: The 7 relation rule types, split into two sub-categories.

Is not allowed to use
Is not allowed to back-call (layers)
Is not allowed to skip-call (layers)
Is allowed to use
Is only allowed to use
Is the only module allowed to use
Must use

that pattern in the realised architecture. The realised architecture contains the software units (which may themselves have sub-units, so it is a special kind of graph that allows for this) linked by software dependencies. The pattern contains pattern modules linked by rules about the legality of dependencies. When software units are mapped unto the pattern modules, these dependencies may or may not conflict with these rules.

HUSACCT contains the following rule types in its framework: “Is not allowed to use”, “Is only allowed to use”, “Is the only module allowed to use”, “Must use” and three kinds of convention (inheritance, naming and visibility). These conventions are most likely not relevant to any architectural pattern, although names could play a role in identifying pattern modules. There is also the “Is allowed to use” rule type, but this is only used to define an exception to one of the aforementioned rules.

The other rule types, which of course each refer to two modules, are not completely mutually exclusive. In fact, three of these rule types can be equivalent in particular contexts. The exception is the “Must use” rule type, which cannot be precisely expressed using the other three. This type of rule should be a trivial part of any pattern definition, since modules are not part of the same pattern if there are no correct dependencies between them at all. Counting the number of violations is, therefore, not enough. The seven relation rule types are depicted in Table 4.1.

There are three additional rule types that are only available within HUSACCT’s intended architecture definition functionality when a specific module type has been chosen. These are: “Is not allowed to skip-call” and “Is not allowed to back-call” for layer modules and the façade convention for component modules. This latter convention simply enforces that an interface module should act as the interface (or façade) for a component-type module. These rule types may therefore also be used within pattern definitions, but only if the respective module types are used within the pattern.

To illustrate how some rule types are equivalent, let us once again take the 3-Layered pattern as an example, because of its simplicity and straightforwardness.

There are three layers and the following rules are how one would naturally express the pattern in terms of skip- and back-calls:

Rule set 1, Skip-calls and back-calls:

1. *Layer 1 is not allowed to skip-call to Layer 3.*
2. *Layer 2 is not allowed to back-call to Layer 1.*
3. *Layer 3 is not allowed to back-call to Layer 2.*
4. *Layer 3 is not allowed to back/skip-call to Layer 1.*

This formulation is already a step up from the generic “skip- and back-calls are not allowed,” but HUSACCT would require a more precise formulation still, in terms of its rule types, in order to be able to work with such a pattern definition in general. The fact that HUSACCT’s layer-type modules possess hierarchical levels that constrain dependencies already suggests one way of defining the N-Layered pattern. However, we want to be able to use other module types as well, e.g. in order to use component-type modules as layers. Which means we have to translate this set to one consisting of rules that do not rely on hierarchical levels.

A translation to “Is not allowed to use” rules would result in the following set of architectural rules for this pattern:

Rule set 2, “Is not allowed to use”:

1. *Layer 2 is not allowed to use Layer 1.*
2. *Layer 3 is not allowed to use Layer 2.*
3. *Layer 3 is not allowed to use Layer 1.*
4. *Layer 1 is not allowed to use Layer 3.*

This set of rules could also be formulated in terms of the “Is only allowed to use” rule type, namely as follows:

Rule set 3, “Is only allowed to use”:

1. *Layer 1 is only allowed to use Layer 2.*
2. *Layer 2 is only allowed to use Layer 3.*
3. *Layer 3 is not allowed to use Layer 1.*
4. *Layer 3 is not allowed to use Layer 2.*

This third rule set appears to signify the exact same pattern as the previous two. However, what happens when this set is used as a pattern definition within a system’s architecture? Let us imagine the rest of the architectural elements of this system as a single module, the *Remainder*.

The Remainder:

All architectural elements that are not part of the architectural pattern under consideration. These elements are not sub-modules of any of the pattern’s modules and are not mapped to the pattern. The Remainder is a collective term that depicts the non-pattern architecture as a single module. As such, the Remainder is also with respect to some pattern.

According to the first set of rules, the Remainder can call on any of the layers within the 3-Layered pattern and any of these layers can call upon the Remainder. This is depicted in Figure 4.5. “Must use” rules have been added as well, in order to make sure that the layers depend on each other the *right* way. Each layer *must* use the next. Otherwise, it would not be much of a 3-Layered pattern.

To graphically depict the patterns, we adopt a UML-like syntax. Expanded packages represent architectural elements, i.e. pattern modules and the Remainder. Collapsed packages indicate sub-modules. Legal dependencies are represented by dashed lines and directed associations figure as the architectural rules. Additional symbols are used to indicate the rule type being employed, as depicted in Table 4.2. Conflicting rules result in exceptions. HUSACCT

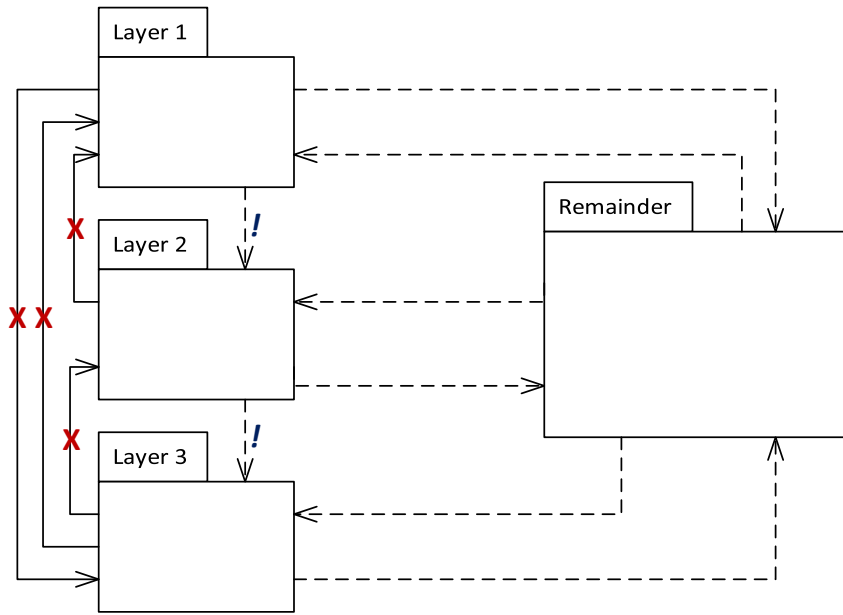


Figure 4.5: The 3-Layered pattern and the Remainder, defined with “Is not allowed to use” rules and showing allowed dependencies as dashed lines.

Table 4.2: This legend presents the different rule types for the provisional diagram notation used here.

“Is not allowed to use”	——— X ———>
“Is only allowed to use”	——— O ———>
“Is the only module allowed to use”	——— ———>
“Is allowed to use”	----->
“Must use”	----- ! ----->

allows for rule exceptions by specifying the module that is exempt from the given rule (Pruijt et al., 2014). So if “Module A is the only module allowed to use Module B” and “Module C must use Module B” are both true, it is implied that Module C is an exception to the first rule. In essence, the first rule should be understood as “Module A is the only module allowed to use Module B, except for Module C”.

If one were to use the third rule set, which makes use of the “Is only allowed to use” rule type, the situation changes to what is shown in Figure 4.6. Only Layer 3 is allowed to use the Remainder now, because the other two layers are only allowed to use the layer below themselves. But the Remainder, i.e. the rest of the system, can still make use of any of these layers, without restriction. Evidently, equivalent rule types turn out to be not so equivalent at all when additional architectural elements come into play.

Let us now consider the third type of rule, “Is the only module allowed to use.” The same initial set of architectural rules would then be transformed into:

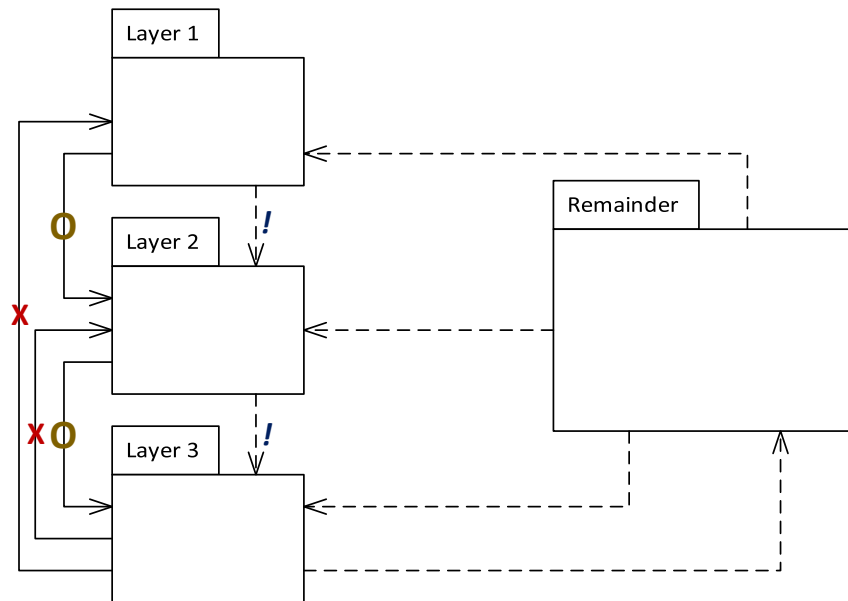


Figure 4.6: *The 3-Layered pattern and the Remainder, defined with “Is only allowed to use” rules.*

Rule set 4, “Is the only module allowed to use”:

1. *Layer 1 is the only module allowed to use Layer 2.*
2. *Layer 2 is the only module allowed to use Layer 3.*
3. *Layer 3 is not allowed to use Layer 1.*

This is slightly more concise, but it is almost trivial to see that this changes the situation in a profound manner when one includes a Remainder, as is apparent from Figure 4.7. Now, the Remainder is only able to utilise Layer 1, while any of the layers can call on the Remainder.

These supposedly equivalent sets of architectural rules create very specific nuances when it comes to the interaction with the rest of the architecture, all while the inherent rules of the N-Layered pattern, skip-call and back-call illegality, are correctly observed.

One might even prefer a hybrid form, such as the one displayed in Figure 4.8 and in the following rule set 5.

Rule set 5, Hybrid example:

1. *Layer 1 is the only module allowed to use Layer 2.*
2. *Layer 2 is only allowed to use Layer 3.*
3. *Layer 3 is not allowed to use Layer 1.*
4. *Layer 1 is not allowed to use Layer 3.*

This last interpretation of the 3-Layered pattern creates a scenario where the Remainder can call upon Layers 1 and 3, but not 2, while Layer 2 is the only layer that cannot communicate with the Remainder. This version essentially isolates Layer 2, while leaving a lot of freedom to

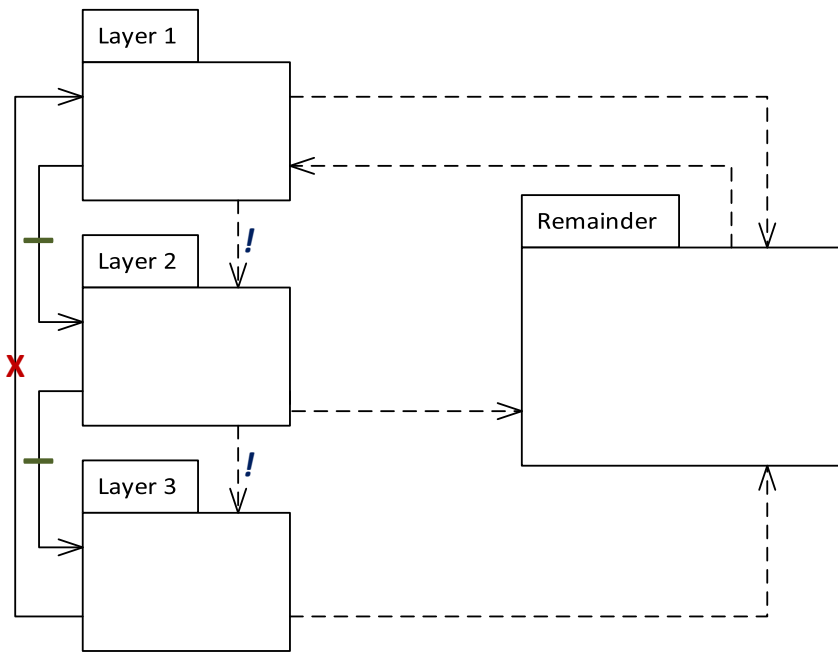


Figure 4.7: The 3-Layered pattern and the Remainder, defined with “Is the only module allowed to use” rules.

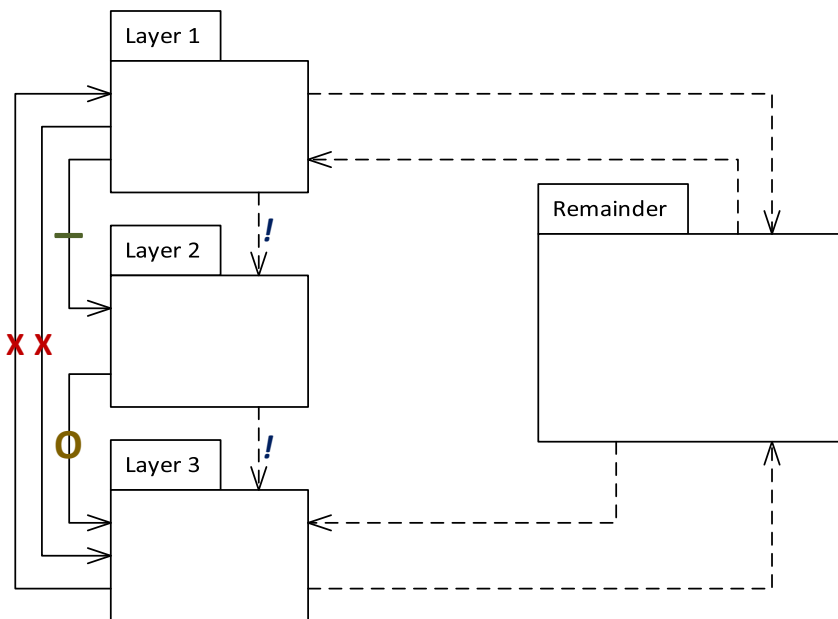


Figure 4.8: The 3-Layered pattern and the Remainder, defined with two different rule types (see rule set 5).

the other two layers.

Layer-type modules are special within HUSACCT, because they automatically receive rules that refer to their respective hierarchical numbers. This means that the back- and skip-call rules are actually implied by creating modules as layers. This implied rule set, generated by HUSACCT, is equivalent to the “Is not allowed to use” rule set from before. Whether this interpretation of the N-Layered pattern would be desirable is left up to the user.

4.3 Pattern Catalogue

In Section 4.2, it was described how architectural patterns can be defined in terms of HUSACCT's architecture rules, with respect to the various rule types that the ACC tool uses as part of its SRMA support. In the current section, the lessons learnt from Section 4.2 are applied to form a collection of architectural patterns for subsequent usage: a pattern catalogue, as it were.

Within the Java realisation of HUSACCT's reconstruction facilities, an abstract pattern class is used as a generic realisation of the architectural pattern. This source code is displayed in its entirety in Code Sample B.1, but can be summarised as follows:

```
// Abstract pattern class, parent of all architectural pattern classes.
public abstract class Pattern {

    protected ArrayList<SoftwareUnitDTO> unitsToMap;
    protected ArrayList<ModuleDTO> patternModules;

    // Add modules to the intended architecture in line with an architectural
    // pattern.
    protected abstract void defineModules();

    // Add rules to the intended architecture that apply to the pattern modules
    // and form part of the pattern.
    protected abstract void defineRules();

    protected abstract void defineMustUseRules();

    // Map specific SoftwareUnitDTOs from the analysed application to the defined
    // pattern modules.
    public abstract void mapPattern(ArrayList<String> patternNames);

    public abstract void mapPatternAllowingAggregates(Map<Integer, ArrayList<
        String>> patternUnitNames);

}
```

Code Sample 4.1: The abbreviated source code for the abstract Pattern class.

The abstract class contains methods to add rules (with or without exceptions) and a series of protected attributes, which are mostly services needed by the child classes. The patterns thus depend heavily on inheritance. Abstract child classes represent general pattern (N-Layered, MVC, Broker), whereas their children are actual interpretations of these patterns. The abstract children typically define “Must use” rules, pattern modules and the methods that allow for the mapping of software units. If such pattern classes are to be created dynamically in the future, the decorator pattern may be a useful design pattern. But since this would not add anything at the moment and since this hypothetical future realisation would likely require rewriting, or at least refactoring, anyway, two layers of inheritance was considered sufficient in terms of Object-Oriented design.

Each pattern subsequently created must fulfil the requirements of this abstract class. These include the definition of pattern modules within the intended architecture, much like a human user would do manually in HUSACCT's *Define intended architecture* window. When modules are created, they require certain specifications, depending on and including the module type.

For example, a layer-type module requires an integer argument known as the *hierarchical level*. This argument determines whether this layer would be the top layer (*level* = 1), the one below (*level* = 2) and so on. Similarly, a component-type module demands an interface, which is a particular class that fulfils HUSACCT's façade convention for the rest of the component. The

word *component* is used for all sorts of concepts within IT and beyond, but it has a very specific meaning here: a component is a kind of module that is only accessed through an interface class. HUSACCT requires this interface be defined when a component is added.

N-Layered Pattern

As explained in Section 4.2, several interpretations of the N-Layered pattern are possible, where it was shown that pattern definitions relying on the various rule types may not be equivalent as long as there is a Remainder. None of these seemed to match the idea of layered architectures significantly better than the others, so they are all deemed valid variants of this architectural pattern. The naming of these pattern definitions is not particularly fascinating, one might simply list a few (there are many more possibilities) of them as follows:

- **N-Layered pattern (Layered Types):** *equivalent to Complete Freedom, but with actual layer-type modules. Corresponds to Rule Set 1 and Code Sample B.7.*
- **N-Layered pattern (Complete freedom):** *there are no restrictions with regard to the Remainder. Corresponds to Figure 4.5, Rule Set 2 and Code Sample B.4.*
- **N-Layered pattern (Free Remainder):** *the Remainder can call on any layer. Corresponds to Figure 4.6, Rule Set 3 and Code Sample B.5.*
- **N-Layered pattern (Restricted Remainder):** *the Remainder cannot call on any layer. Corresponds to Figure 4.7, Rule Set 4 and Code Sample B.6.*
- **N-Layered pattern (Isolated internal layers):** *intermediary layers are never called by and can themselves not call on the Remainder. Corresponds to Figure 4.8, Rule Set 5, and Code Samples 4.3 and B.3.*

As mentioned earlier, 3-Layered, 4-Layered and 5-Layered patterns are all essentially the same and convey the same idea of strict layering. The N-Layered pattern is, therefore, the exact same pattern as far as the realisation is concerned and the value of N is merely an argument for the pattern class' constructor. The different interpretations of the N-Layered pattern are implemented as subclasses of an abstract N-Layered pattern class (Code Sample 4.2 for the summary, B.2 for the full code), itself a child of the abstract pattern class. As per example, the N-Layered (Isolated internal layers) pattern definition source code is displayed in Code Sample B.3 and summarised in Code Sample 4.3.

```
public abstract class LayeredPattern extends Pattern {
    // The abstract class for all N-Layered patterns. Constructor and mapping
    // methods can be defined here, as well as "MustUse" rules.

    @Override
    protected void defineModules() {
        for (int i = 1; i <= numberOfModules; i++) {
            defineService.addModule("Layer" + i, "**", "Subsystem", i, null);
        }
    }

    @Override
    protected void defineMustUseRules() {
        for (int i = 1; i < numberOfModules; i++)
            addSingleRule("Layer" + (i + 1), "Layer" + i, "MustUse", null);
    }
}
```

```

@Override
public void mapPattern(ArrayList<String> mapping) {
    ArrayList<SoftwareUnitDTO> temp = new ArrayList<>(1);
    for (int i = 1; i <= mapping.size(); i++) {
        temp.add(analyseService.getSoftwareUnitByUniqueName(mapping.get(i - 1)));
        defineService.editModule("Layer" + i, "Layer" + i, i, temp);
        temp.clear();
    }
}

@Override
public void mapPatternAllowingAggregates(Map<Integer, ArrayList<String>>
    patternUnitNames) {
    ArrayList<SoftwareUnitDTO> temp = new ArrayList<>();
    for (int i = 0; i < patternUnitNames.size(); i++) {
        for (int j = 0; j < patternUnitNames.get(i).size(); j++) {
            temp.add(analyseService.getSoftwareUnitByUniqueName(patternUnitNames.get(i).get(j)));
        }
        defineService.editModuleWithAggregation("Layer" + (i + 1), "Layer" + (i + 1), i + 1, temp);
        temp.clear();
    }
}
}

```

Code Sample 4.2: The abbreviated source code for the abstract class of the N-Layered pattern.

```

public class LayeredPattern_IsolatedInternalLayers extends LayeredPattern {
    // The version of the N-Layered pattern in which the internal layers are
    // isolated from the Remainder.

    @Override
    protected void defineRules() {
        for (int i = 1; i < numberOfModules; i++) {
            addSingleRule("Layer" + i, "Layer" + numberOfModules, "IsNotAllowedToUse",
                null);
            if (i == 1) {
                addSingleRule("Layer" + 2, "Layer" + 1, "IsTheOnlyModuleAllowedToUse",
                    null);
            } else {
                addSingleRule("Layer" + (i + 1), "Layer" + i, "IsOnlyAllowedToUse", null);
            }
        }
    }

    @Override
    protected void defineModules() {
        for (int i = 1; i <= numberOfModules; i++) {
            defineService.addModule("Layer" + i, "**", "Subsystem", i, null);
        }
    }
}

```

Code Sample 4.3: The abbreviated source code for one interpretation of the N-Layered pattern, namely the case with isolated internal layers.

MVC Pattern

The Model-View-Controller (hereafter MVC) pattern is considered to be a design pattern by some, but it also exists on a system level. As such, it can be used to separate graphics/presentation/UI logic from the underlying model logic by restricting communication between the two. A Controller acts as a mediator between them, with the essential restriction that Model cannot use Controller (Bass et al., 2012).

Multiple interpretations of MVC exist, since many interpretations and adaptations of this pattern occur in the field. For example, sometimes the View incorporates the concerns of the Controller, leaving only two modules in this MVC version. Also, whereas the rule that Model cannot depend on Controller seems rather well accepted, it is unclear whether Model can depend on View and vice versa. If it cannot, there is a clear separation of concerns. However, a common interpretation of the pattern states that View can perform a state query on Model and that Model can update View (e.g. Bass et al. (2012)). One way of solving this is to distinguish between allowing a dependency to exist and requiring it to.

The MVC pattern is often confused with the MVP (Model-View-Presenter) pattern (Qureshi & Sabir, 2014) (Potel, 1996). View and Model are completely decoupled in MVP. One would be inclined to suspect an instance of MVC when these three module names appear in, for example, Java package definitions. However, it might actually be more akin to MVP dependency-wise. Two possible interpretations of these patterns can be formulated as follows:

Rule set 6, interpretation of classic MVC:

1. *Controller must use Model.*
2. *Controller must use View.*
3. *View must use Model.*
4. *Model is not allowed to use Controller.*
5. *If Model is not allowed to use View, violations should only be change updates.*
6. *If View is not allowed to use Controller, violations should all be triggered by user actions.*

Rule set 7, interpretation of MVP:

1. *Presenter must use Model.*
2. *Presenter must use View.*
3. *View is not allowed to use Model.*
4. *Model is not allowed to use View.*
5. *Model is not allowed to use Presenter.*

We focus on the classic MVC pattern. It is clear that different interpretations of MVC would lead to several SRMA pattern definitions for this pattern as well. In fact, such ambiguity should be expected for all architectural patterns and this is precisely what we want to address with our research.

The most obvious three versions are a) complete freedom with regards to the Remainder for

all three modules, and b) only the Controller module is allowed to have dependencies going to and from the Remainder and c) Model takes on this interface role. The second can be deemed more obvious than to have the same arrangement with either the View- or Model-module, because the Controller is supposed to be the counterpart that is in control of the other two, as the name implies. It would, therefore, make sense that the Controller would operate as an interface for the whole pattern. Albeit perfectly reasonable that one would consider Model to be more suitable for this position, since one can make the argument that Controller *merely* controls the interaction between Model and View, whereas Model houses all the model data.

Furthermore, would this mean that the entire pattern should be viewed as a component, with the Controller as its interface? If not, should this pattern be defined so that Model and View are isolated from the Remainder, or not? These different interpretations imply, once again, distinct possible pattern definitions.

```
public abstract class MVCPattern extends Pattern {
    // The abstract class for all Model-View-Controller patterns. Constructor and
    // mapping methods can be defined here, as well as "MustUse" rules and
    // the modules themselves.

    @Override
    protected void defineModules() {
        defineService.addModule("Model", "**", "Subsystem", 1, null);
        defineService.addModule("View", "**", "Subsystem", 1, null);
        defineService.addModule("Controller", "**", "Subsystem", 1, null);
    }

    @Override
    protected void defineMustUseRules() {
        addSingleRule("Model", "Controller", "MustUse", null);
        addSingleRule("Model", "View", "MustUse", null);
        addSingleRule("View", "Controller", "MustUse", null);
    }

    @Override
    public void mapPattern(ArrayList<String> mapping) {
        ArrayList<SoftwareUnitDTO> temp = new ArrayList<>(1);
        temp.add(analyseService.getSoftwareUnitByUniqueName(mapping.get(0)));
        defineService.editModule("Model", "Model", 1, temp);
        temp.clear();
        temp.add(analyseService.getSoftwareUnitByUniqueName(mapping.get(1)));
        defineService.editModule("View", "View", 1, temp);
        temp.clear();
        temp.add(analyseService.getSoftwareUnitByUniqueName(mapping.get(2)));
        defineService.editModule("Controller", "Controller", 1, temp);
        temp.clear();
    }

    @Override
    public void mapPatternAllowingAggregates(Map<Integer, ArrayList<String>>
        patternUnitNames) {
        ArrayList<SoftwareUnitDTO> temp = new ArrayList<>();
        for (int j = 0; j < patternUnitNames.get(0).size(); j++) {
            temp.add(analyseService.getSoftwareUnitByUniqueName(patternUnitNames.get(0).get(j)));
        }
        defineService.editModuleWithAggregation("Model", "Model", 1, temp);
        temp.clear();
        for (int j = 0; j < patternUnitNames.get(1).size(); j++) {
            temp.add(analyseService.getSoftwareUnitByUniqueName(patternUnitNames.get(1).get(j)));
        }
    }
}
```

```

    }
    defineService.editModuleWithAggregation("View", "View", 1, temp);
    temp.clear();
    for (int j = 0; j < patternUnitNames.get(2).size(); j++) {
        temp.add(analyseService.getSoftwareUnitByUniqueName(patternUnitNames.get(2).get(j)));
    }
    defineService.editModuleWithAggregation("Controller", "Controller", 1, temp);
    ;
}
}

```

Code Sample 4.4: *The shortened source code for the abstract Model-View-Controller pattern class*

There can, obviously, be many different interpretations of this pattern and a few of these were realised. The abstract class is depicted in Code Sample B.8 (with a brief impression of the most important methods in Code Sample 4.4) and the following list describes its subclasses. The abstract class defines the modules and the “Must use” rules, as well as the mapping methods. Of course, one could come up with many more interpretations and the previously mentioned distinction with the MVP pattern means that there are similar versions possible for MVC-inspired patterns as well. Let us not go into all of these non-classic MVC patterns too, as it would be rather tedious and unenlightening.

- **MVC Pattern (Controller Interface):** *Both the Model- and the View-module are inaccessible to the Remainder, nor can the Remainder be used by them. Corresponding to Figure 4.9 and Code Sample 4.5.*
- **MVC Pattern (Model Interface):** *Both Controller and View are inaccessible to the Remainder, nor can the Remainder be used by them. Corresponding to Figure B.4 and Code Sample B.12.*
- **MVC Pattern (Complete Freedom):** *There are no restrictions with regard to the Remainder. Corresponding to Figure B.1 and Code Sample B.9.*
- **MVC Pattern (Free Remainder):** *Model and View are accessible to the Remainder, but they are themselves not able to circumvent Controller. Corresponding to Figure B.2 and Code Sample B.10.*
- **MVC Pattern (Restricted Remainder):** *Model and View can depend on the Remainder, but they are inaccessible for that Remainder. Corresponding to Figure B.3 and Code Sample B.11.*

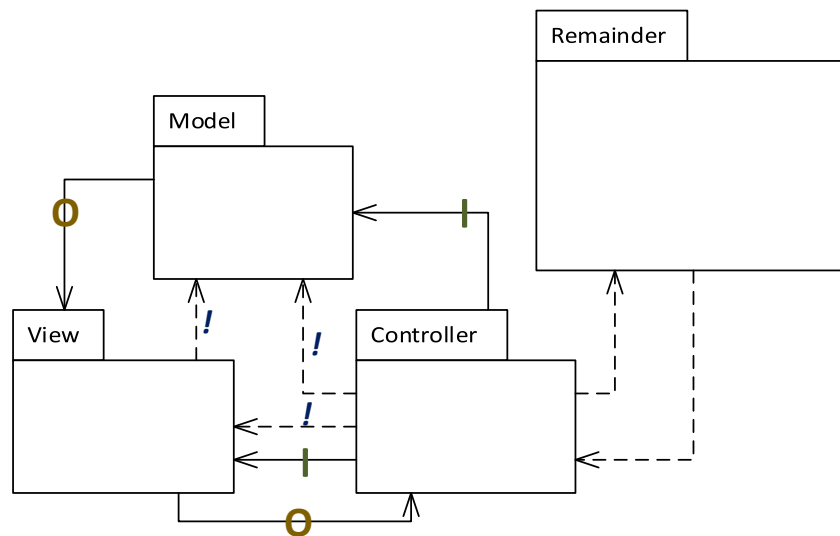


Figure 4.9: The Controller Interface interpretation of the classic MVC pattern.

```
package husacct.analyse.task.reconstruct.patterns;

public class MVCPattern_ControllerInterface extends MVCPattern {

    public MVCPattern_ControllerInterface() {
        // TODO Auto-generated constructor stub
    }

    @Override
    protected void defineRules() {
        addSingleRule("Controller", "View", "IsOnlyAllowedToUse", "Model");
        addSingleRule("View", "Model", "IsOnlyAllowedToUse", null);
        addSingleRule("Model", "Controller", "IsTheOnlyModuleAllowedToUse", "View");
        addSingleRule("View", "Controller", "IsTheOnlyModuleAllowedToUse", "Model");
    }
}
```

Code Sample 4.5: The source code for the Model-View-Controller (Controller Interface) pattern class.

Broker Pattern

Let us discuss one more common architectural pattern: the Broker pattern. Part of the conceptual essence of this pattern is that a set of modules which require several services from other modules are placed in one particular pattern module. This module, called *Client(s)* or *Requester(s)*, makes use of the necessary services within the module that is often called *Service(s)*, but also *Provider(s)*. In this thesis, they are referred to as *Requester* and *Provider*.

Regardless of this naming, the fundamental idea is that these two are separated by a module called *Broker*. As its name suggests, this module brokers between the two and, by doing so, shields the two from each other. The Requester software units might require many different actions from the Provider software units, such as various data processing tasks, depending on different possible types of input. The Broker prevents the need for the Requester's software units to all be aware of the numerous services and when to call on which. All the service selection logic can be handled in one central location, the Broker, with the Requester simply communicating

its needs. In summation, the Broker pattern is an exemplary embodiment of the separation of concerns, one of the pillars of object-oriented design principles.

As with the Model-View-Controller pattern, it is not totally unambiguous how the ideas of the Broker pattern are to be interpreted. One relatively obvious way to look at it might be that the Broker pattern is an alteration of this Model-View-Controller pattern, since Requester and Provider do not talk just like Model and View do not, with the difference being that not Broker but Requester seems to be the most suitable interface with the rest of the architecture. After all, the Requester requests a service from the Broker based on some functionality that has come before. It is the Requester that asks for something, not the Broker that decides that the Client ought to be doing something.

```
public abstract class BrokerPattern extends Pattern {
    @Override
    protected void defineMustUseRules() {
        addSingleRule("Provider", "Broker", "MustUse", null);
        addSingleRule("Broker", "Requester", "MustUse", null);
    }

    @Override
    protected void defineModules() {
        defineService.addModule("Broker", "**", "SubSystem", 1, null);
        defineService.addModule("Provider", "**", "SubSystem", 1, null);
        defineService.addModule("Requester", "**", "SubSystem", 1, null);
    }

    @Override
    public void mapPattern(ArrayList<String> mapping) {
        IAnalyseService analyseService = ServiceProvider.getInstance().
            getAnalyseService();
        ArrayList<SoftwareUnitDTO> temp = new ArrayList<>(1);
        temp.add(analyseService.getSoftwareUnitByUniqueName(mapping.get(0)));
        defineService.editModule("Broker", "Broker", 1, temp);
        temp.clear();
        temp.add(analyseService.getSoftwareUnitByUniqueName(mapping.get(1)));
        defineService.editModule("Provider", "Provider", 1, temp);
        temp.clear();
        temp.add(analyseService.getSoftwareUnitByUniqueName(mapping.get(2)));
        defineService.editModule("Requester", "Requester", 1, temp);
    }

    @Override
    public void mapPatternAllowingAggregates(Map<Integer, ArrayList<String>>
        patternUnitNames) {
        IAnalyseService analyseService = ServiceProvider.getInstance().
            getAnalyseService();
        ArrayList<SoftwareUnitDTO> temp = new ArrayList<>();
        for (int i = 0; i < patternUnitNames.size(); i++) {
            for (int j = 0; j < patternUnitNames.get(i).size(); j++) {
                temp.add(analyseService.getSoftwareUnitByUniqueName(patternUnitNames.get(i).get(j)));
            }
            if (i == 0)
                defineService.editModuleWithAggregation("Broker", "Broker", 1, temp);
            else if (i == 1)
                defineService.editModuleWithAggregation("Provider", "Provider", 1, temp);
            ;
            else if (i == 2)
                defineService.editModuleWithAggregation("Requester", "Requester", 1,
                    temp);
        }
    }
}
```

```

        temp.clear();
    }
}
}

```

Code Sample 4.6: The source code summary for the abstract Broker pattern class.

The abstract pattern class for the Broker pattern is depicted in Code Sample 4.6 (B.13 for all methods and attributes) and various interpretation are presented in the following list:

- **Broker Pattern (Requester Interface):** Both the Broker- and the Provider-module are inaccessible to the Remainder, nor can the Remainder be used by them. Corresponding to Figure 4.10 and Code Sample 4.7.
- **Broker Pattern (Complete Freedom):** There are no restriction with regard to the Remainder. Corresponding to Figure B.5 and Code Sample B.14.
- **Broker Pattern (Free Remainder):** Broker and Provider are accessible to the Remainder, but they are themselves not able to bypass Requester. Corresponding to Figure B.6 and Code Sample B.15.
- **Broker Pattern (Restricted Remainder):** Broker and Provider can depend on the Remainder, but they are themselves inaccessible for this Remainder. Corresponding to Figure B.7 and Code Sample B.16.

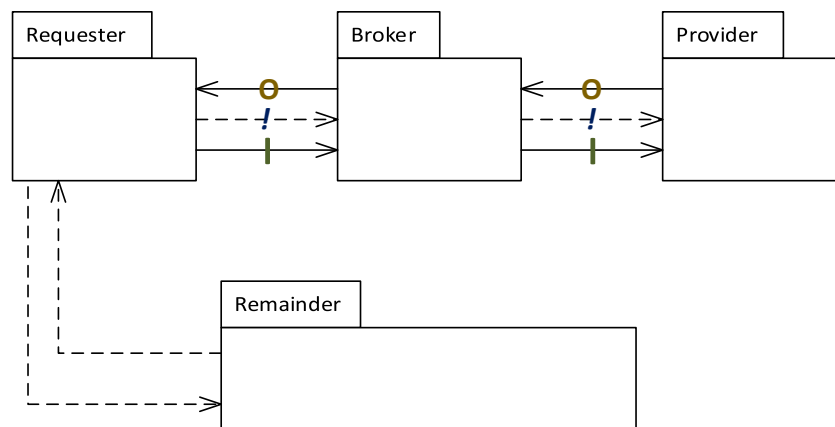


Figure 4.10: The Requester Interface interpretation of the Broker pattern.

```

package husacct.analyse.task.reconstruct.patterns;

public class BrokerPattern_RequesterInterface extends BrokerPattern {
    // In this interpretation, the Requester acts as an interface for the other
    // two modules.
    public BrokerPattern_RequesterInterface() {
        // TODO Auto-generated constructor stub
    }

    @Override
    protected void defineRules() {
        addSingleRule("Requester", "Broker", "IsOnlyAllowedToUse", "Provider");
        addSingleRule("Provider", "Broker", "IsTheOnlyModuleAllowedToUse", null);
        addSingleRule("Broker", "Provider", "IsOnlyAllowedToUse", null);
    }
}

```

```
    addSingleRule("Broker", "Requester", "IsTheOnlyModuleAllowedToUse", "
    Provider");
  }
}
```

Code Sample 4.7: The source code for the Broker (Requester Interface) pattern class.

Other patterns might be formulated in the same way, let alone additional interpretations of the three mentioned patterns, but these patterns were deemed enough to make the point that architectural patterns can be defined by means of the SRMA supporting architectural elements as used by HUSACCT. It leads one to think of how to interpret each pattern and how it ought to be used by being forced to contemplate and choose between the different rule types. New patterns can be added, even entirely novel pattern that not yet exist in literature, in the manner demonstrated here. Figure 4.11 shows an overview of the class hierarchy.

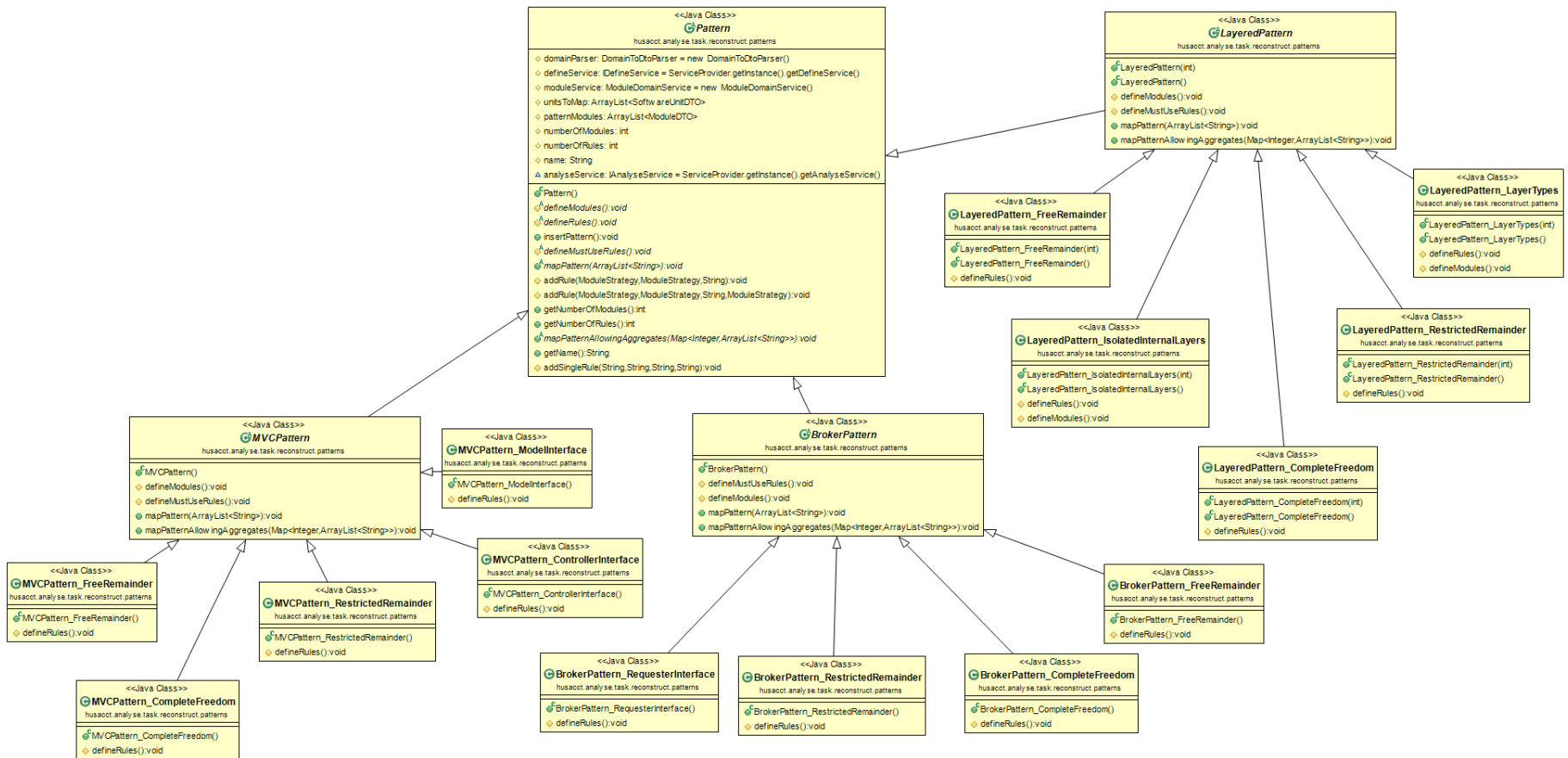


Figure 4.11: The class diagram of the realised architectural patterns within the Reconstruct package, generated using the ObjectAid UML Explorer plug-in (www.objectaid.com) for the Eclipse IDE.

Chapter 5

Proposed Approach

Now that the essential concepts have been described and pattern definitions within the framework of HUSACCT's SRMA support have been shown, the envisioned approach by which these architectural patterns might be discovered is provided. This is where the bulk of the literature study can be presented, since most of it relates to existing approaches related to this thesis.

5.1 Patterns in SAR

The researched literature has influenced the choice of terminology used here. Although *architectural pattern* is the preferred term in the rest of the thesis, the terminology of each particular paper is temporarily adopted when discussing that paper as part of the literature study in this section.

In order to provide an overview of related literature concerning architectural patterns, the evolution of this concept and its gradual adoption into the field of Software Architecture Reconstruction is illustrated here. An early example of architectural patterns being used in scientific literature is by Abowd, Allen and Garlan (1993). They introduce the notion of architectural styles, which consist of precise syntactic and semantic descriptions of both static and dynamic characteristics. A deep understanding of a style and its constraints can then be used for formal analysis of that style, studying potential sub-styles and comparing it with other styles. The paper's appendix briefly explains the Z language, which is the mathematical notation they use to define styles and which is strongly based on predicate logic. Similar notations are used by other papers to express the various components, connectors and constraints of patterns.

A paper by Garlan, Allen and Ockerbloom (1994) is also strongly concerned with architectural styles, identifying two categories: a) idioms and patterns, and b) reference models. This distinction is not very well accepted and seems to be related to the poorly accepted distinction between styles and patterns. Next to describing a few of these styles, such as Pipe-and-Filter, this paper introduces a system for the developments of style-specific architecture development environments.

Harris, Reubenstein and Yeh (1995) use architectural style representations in reverse engineering by building both a style library and a query library to analyse a C/Unix type system. The discovery of styles is then a top-down approach, based on existing documentation or clustering techniques for hypothesised styles. Performing a similar approach twenty years later would inevitably demand a more Object-Oriented perspective, as the researchers themselves anticipate.

At the beginning of the current century, Paakki, Karhinen, Gustafsson, Nenonen and Verkamo (2000) wrote a paper on architectural pattern mining, in which they manage to discover design patterns by translating UML diagrams to Prolog (predicate logic) statements and treating it as an arc consistency (AC-3) constraint satisfaction problem in the Maisa tool. They connect this

to software metrics, hoping to assist the design phase of a system. Their approach would struggle with inexact matches and pattern violations, as they restrict themselves to those patterns with a precise structure, making it not entirely ideal for the purposes of this thesis.

Another thing worth noting about the paper by Paakki et al. (2000) is the use of the term *architectural patterns* early in the text to refer to design patterns, architectural patterns, anti-patterns and idioms. And yet, while the rest of the paper does refer to architectural patterns, the only examples are of design patterns like Abstract Factory.

Another approach from early in the decade that makes use of architectural styles is called MAP (Mapping Architecture for Product lines) (Stoermer & O'Brien, 2001). MAP is a method with many steps and it is only the *Qualification* step, as it is called, that relies on architectural styles and design patterns. Since these architectural styles consist of well-documented architectures, their effect on Quality Attributes is known. There is no automation here, though, as it relies on interviews with architects and developers. It indicates what an automated technique might add to existing methods like this, presumably because interviews are notoriously costly in terms of time spent.

Pinzger and Gall (2002) extend an existing architecture recovery framework by including patterns (architectural, design and code patterns). They hope to find instances of these patterns by starting at the source code level and working their way up. ESPaRT (Enhanced String Pattern Recognition Tool) is used for this, as it has been extended by the researchers to allow for pattern specification in XML. This relies on what they call *hot-spots*, which are essentially heuristic indicators of a pattern instance in the code. They manage to show that the application uses a Client-Server pattern, but to do this they had to refine their socket pattern search query a few times and steer the algorithm to this conclusion. Their work is very useful, although rather code-specific and it might not be so successful in a large application with many unrelated patterns.

Sartipi presents an extensive pattern-based and data mining approach to SAR that makes use of both clustering and editing-cost-based graph-matching techniques (Sartipi, 2003). Although it is a rather sophisticated and admirable approach, with a strong mathematical background and promising results, this complexity may also turn out to be its weakness. It requires iterative user input that seems to make this approach quite dense for everyday use, though this may well be unavoidable for SAR. At least the usage of patterns adds to a level of intuitive understanding.

AcmeStudio is a tool that incorporates architectural style as well. Developed by Schmerl and Garlan (2004), it is aimed at the design phase of an application and allows for new styles to be defined by the user. Although it may not be relevant in terms of architecture reconstruction or pattern recovery, the paper does provide some interesting insights into the definition of such patterns. If a user wishes to design a system, he can select a particular architectural style to help him in the process. This is a way of speeding up the architect's work. AcmeStudio relies on Acme, as the name implies, which they describe as a style-neutral Architecture Description Language (ADL). They document two successful case studies, one with Ford Motor Company and one with NASA Jet Propulsion Lab.

Dynamic (runtime) analysis has been used to discover "architecture styles", which is an unusual variation of the term. The DiscoTect system uses state machines to do this, each one being specifically tailored to a particular model (Yan, Garlan, Schmerl, Aldrich & Kazman, 2004). This is a complex task and is aimed at dynamic discovery rather than static recovery, but an interesting indication of (relatively) recent developments nonetheless.

In a paper by J. S. Kim and Garlan (2006), architectural styles are analysed automatically for properties such as consistency and compatibility. This is done by, again, writing a style in the Acme ADL. Using a set of relatively straightforward translation rules, this is then turned into Alloy (a language based on predicate logic) and fed to the Alloy Analyzer. Within the Analyzer,

one can check whether various properties of styles hold according to the logical description.

A 2010 journal publication, successor to this 2006 workshop paper, investigates several styles (J. S. Kim & Garlan, 2010). It is a wonderfully detailed paper with a rich appendix, showing much of what would be needed to reproduce their approach. The connection with architecture recovery is not clear-cut, however.

Dillon, Wu and Chang (2007) provide an investigation of architectural styles concerning SOC-based (Service-Oriented Computing) systems in order to derive a reference architecture. It is an example of the continuing interest in these styles/patterns during the course of the decade.

A relatively rich ontological framework for architectural styles is presented by Pahl, Giesecke and Hasselbring (2009). Acme is used as the ADL once more and translated to predicate logic. This style ontology may also be extended to incorporate metrics such as Quality Attributes. Papers such as this can be used to collect the logical representations of architectural patterns by whomever might be interested in such expressions.

A systematic literature review on architecture reconstruction was presented by Ducasse and Pollet (2009), in which they mention architectural patterns and styles several times in their attempts to build an ontology of various methods. Many of the publications mentioned in this section were described or at least mentioned by them.

This concludes the overview of the published literature related to architectural pattern recovery. More papers have been published, including in the years since 2009, but the main ideas have been mentioned here.

Design Patterns in SAR

Because design patterns share so many characteristics with architectural patterns, scientific literature aimed at detecting these lower-level constructions can be highly relevant. Taking into account the voluminous literature in this sub-domain, the evolution of design pattern detection is given here in a brief overview.

Shortly after the introduction of design patterns by the GoF in 1994, Shull, Melo and Basili (1996) released a paper influenced by these ideas. It illustrates a method in which Object-Oriented Design Patterns can be discovered in an existing application as a step of the overall method. This method, called BACKDOOR (Backwards Architecting Concerned with Knowledge Discovery of OO Relationships), was not at all automated. Tool support was envisioned for future development, however. It relies heavily on a base of reference patterns, which they initially filled with design patterns from the GoF publication.

Actual recovery of design patterns was presented in a paper from 1998 (Antoniol, Fiutem & Cristoforetti, 1998). From an AOL (Abstract Object Language) representation of the code, the recovery process extracts class metrics. These consist of the number of attributes and operations, which can be public, private or protected, and the number of relations of various types. The approach reduces the search space significantly, since metric, structural and delegation constraints can be used to minimise the candidate set. The referred paper does not show how their approach increases system understanding or how the re-engineer would use the tool, but one can imagine the merits without a concrete example.

A more interactive and iterative recognition algorithm is presented by Niere, Schafer, Wadsack, Wendehals and Welsh (2002). They identify a main problem with other design pattern recovery approaches resulting from the variability of design pattern realisation: either the approach restricts itself to a level of granularity (such as call graphs and naming conventions) and thus struggles with false positives for the re-engineer to sort out, or the approach considers fine

behaviour such as data flows and subsequently struggles with scalability issues.

Their solution is to define patterns with respect to a program's ASG (Abstract Syntax Graph) with graph transformation rules, supported by the FUJABA (From UML to Java And Back Again) environment. This allows the re-engineer to enter new patterns using a notation much like UML, which would presumably be familiar. A combination of bottom-up and top-down steps is paused so that the re-engineer can annotate the ASG or stop the algorithm if analysis is unlikely to prove useful. This method both reduces the number of false positives and increases the scalability to large systems.

An example of an approach in which static and dynamic analysis were combined is presented in Heuzeroth, Holl, Hogstrom and Lowe (2003). The former is performed with Recoder, a Java analyser that constructs an AST (Abstract Syntax Tree). Event generators are then added by the researchers. Independent of naming conventions, which is something the researchers point out, the algorithm finds a large set of candidate pattern instances for a given design pattern. They reason that this set can be whittled down using expert knowledge, static data flow analysis or dynamic analysis.

The dynamic analysis for each pattern is a different algorithm that checks the runtime usage of class instances and whether they conform to the given design pattern. Although the number of candidates from the static analysis can be large, it should result in significantly less effort for the dynamic analysis to restrict itself to the candidate set than if it were to analyse the whole application. This approach leads to a high ratio of false negatives to false positives, since pattern instances can only be confirmed if they are executed during the dynamic analysis.

Guéhéneuc, Sahraoui and Zaidi (2004) use heuristics to cut down the search space in their search for micro-architectures similar to design motifs, which are the solutions provided by design patterns. They use quantitative signatures of classes to create roles fingerprints of design motifs. Metrics such as size, number of children and level of cohesion help them to remove true negatives from the search space. From a repository that contains 15 design motifs, they extract metrics and feed them to a rule learner. After some editing of these rules, fingerprints are integrated with the constraint-based tool PTIDEJ. They manage to achieve a search space reduction between 69.00% and 89.15% using this method, which is impressive.

Another approach to detecting design patterns was provided by Tsantalis, Chatzigeorgiou, Stephanides and Halkidis (2006), where the issue was tackled as a graph similarity problem. The fact that design patterns often employ some inheritance relation is exploited to limit the search space, an example of heuristics increasing the effectiveness of an algorithm in this field. No other heuristics were necessary, since the paper presents a rather successful attempt at finding design patterns in these graphs by translating them to matrices and feeding them to a similarity scoring algorithm. The approach resulted in a perfect precision and an impressive recall value of design pattern instances.

Inexact matching is an important advantage, since design patterns may not be realised exactly. Depending on the developers, variations or violations of the patterns may be built into the architecture. Another paper where this is taken into account is based on template matching from computer vision (Dong et al., 2008). Similarity scores are calculated based on matrix representations, where the process is supported in a tool prototype.

By 2009, design pattern mining was already mature enough that Dong, Zhao and Peng (2009) considered it time for a systematic literature review of its own. Published in the same year as the SLR by Ducasse and Pollet (2009), this review provides a more in-depth overview and classification of tools and approaches for finding design patterns, which is extremely useful for the purposes of this thesis. Various techniques are discussed, some of which have been mentioned in this literature section. A noteworthy theme in this SLR is the problem of experimental comparison: how to contrast or benchmark the various approaches and tools?

Arcelli Fontana and Zanoni (2011) present a seemingly elaborate design pattern detection tool, called MARPLE (Metrics and Architecture Reconstruction Plug-in for Eclipse), which was not mentioned in the literature review by Dong et al. (2009) despite the fact that it was already described in an earlier workshop publication (Arcelli, Tosi, Zanoni & Maggioni, 2008). The reason for this might be that the actual process is not described in great detail, since the papers focus on the design of the whole tool. It detects design pattern subcomponents using static analysis to build an AST, after which classification algorithms map patterns onto candidates. It seems that only a few design patterns are currently known by the detection algorithm and the tool is unfinished.

With this overview of related literature in mind, the next section provides this thesis' novel approach to finding architectural patterns using HUSACCT.

5.2 Process-Deliverable Diagram

The proposed method is displayed in Figure 5.1, in which the Process-Deliverable Diagram notation was used to display both the process side and the deliverable side. Activities are displayed on the left and their output is shown on the right. Activities with a white shadow, as it were, consist of additional sub-activities explained elsewhere. If this shadow is black, the sub-activities are outside the scope of this document. For more information on the PDD notation, see Van de Weerd and Brinkkemper (2008). PDDs should always be accompanied by an activity table and a concept table, which are both situated in the appendices (A.1 and A.2).

The two main steps of the process are shown, as well as the sub-steps that currently refer to the combination of decomposition based on Java package definitions and pattern matching based on the straightforward analysis of all permutations of a particular set of modules to a specific architectural pattern. Similar or vastly different approaches might be found that could, hopefully, fit into either main method of this approach, or perhaps function as a combination of both steps. In this regard, the approach is essentially modular and built with future improvements in mind.

Taking into account that the aim of architectural patterns is to break down the system into modules along the lines of object-oriented design principles, how the research tool identifies the modules themselves is of great importance. This will be discussed in the next section.

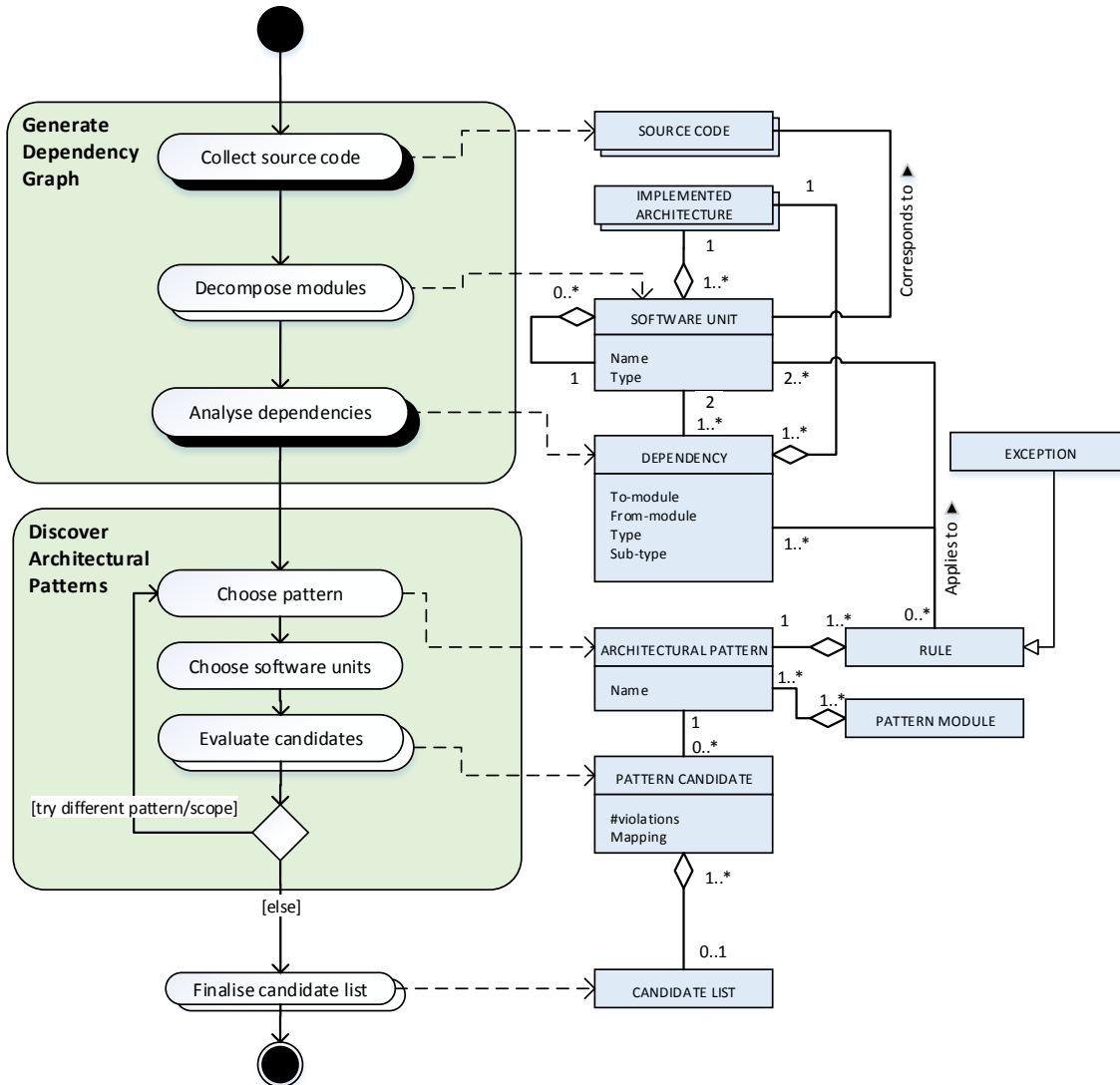


Figure 5.1: The proposed method.

5.3 Modular Decomposition

One possibility of realising an architectural pattern recovery method would be to rely on automatic layering. Such techniques have already been realised by others (Goldstein, Segall & Carmel, 2015)(Sarkar, Maskeri & Ramachandran, 2009)(Alvine, El Boussaidi & Mili, 2014), although not necessarily in such a way that one could be immediately implemented within HUSACCT itself. Using HUSACCT, whether as input to an automatic layering tool or by extending it with such functionality, would make it a different approach from others. Thanks to HUSACCT's rich semantics in terms of various types of modules, dependencies and rules, the layering technique could be influenced with filter and severity settings. For example, particular dependency types might be ignored or, conversely, deemed more important than others. However, architectural patterns include more than simple layers.

Evidently, it has already been made possible by others to analyse a system and come up with, e.g. a 4-Layered pattern and 263 violations thereof (skip-calls and back-calls). It would be valuable if this could be further improved. Making the automatic layering ignore a certain type of violation is not necessarily useful, since it would merely result in either a different layering solution or a different number of violations, if not both. There appears to be no particular reason why such a solution would be an improvement upon the unaltered one, unless manual inspection were to imply a far more plausible case for the intended architecture that makes perfect sense for some reason (e.g. based on naming conventions).

Building on this, it could be useful to apply a filtering algorithm *after* an automatic layering step, which would not affect the modular decomposition but would reduce the number of dependencies, possibly including violations. Although the resulting architecture might suddenly contain several orphaned packages due to this, it could give insight into the common type of violations and, thus, hint at development decisions concerning architecture rules. It is well possible that an architectural pattern can be identified due to a particular variation on a more basic pattern resulting in very few violations.

Either act of filtration or severity ranking, before or after the layering, provides a rather limited approach to architectural pattern recovery: it is restricted to the application of a strict layering pattern. Even as far as static modular patterns go, there is more variety than strict layering. As the field of software architecture develops, and it becomes easier to recognise and reuse architectural patterns, there will likely be an increase in the number of pattern variations. Being able to detect more sophisticated patterns would itself be more interesting to re-engineers, especially since the filtration and severity ranking are a bit obvious and would not necessarily prove advantageous.

For this reason, the reconstruction method proposed here incorporates automatic layering in the method activity of *decomposition* (essentially defining the software units). Other decomposition approaches should be available as well, such as clustering based on other criteria or the unadulterated copying of the source code directory structure. The approach currently used by HUSACCT, taking package definition as a modular decomposition to be edited by the user, is deemed to be sufficient for the purposes of this thesis and is depicted in Figure 5.2.

In HUSACCT, a pattern could be represented as a set of modules, rules and exceptions. The way to test for conformance with such a pattern would be to attempt all possible mappings (limited in scope to deal with the combinatorial explosion) to the dependency graph as extracted from the decomposition step and choose between the most optimal solutions. Decomposition might be preceded by filtration, which would currently be useless in HUSACCT, or followed by it, which would be identical to attempting a variety of the used pattern instead. In both cases, severity ranking might be used as well, as a weaker version of filtering. This would allow one to find situations where the architecture rules are not very strict for particular types of dependencies.

With this decision, with the choice for HUSACCT as a framework as opposed to building a separate system and with the completion of a method proposal, only one verdict remained before development could begin: the selection of a pattern matching algorithm to be realised.

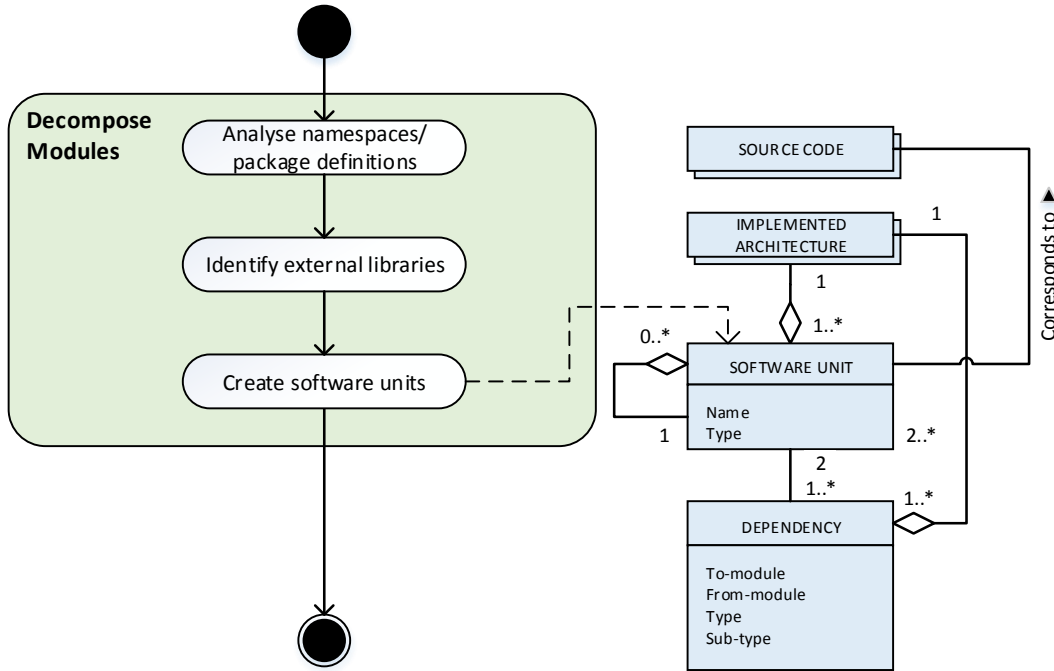


Figure 5.2: A PDD of the modular decomposition using namespaces (C#) or package definitions (Java).

5.4 Pattern Matching

Let us look at some existing approaches from the scientific literature.

Examples of pattern matching techniques include the one described in the paper by Tsantalis et al. (2006), where similarity scoring is chosen over inexact graph matching based on edit distance, because the authors feel this is more likely to give useful results. In other words, they make the point that edit distance can be rather misleading. Their use of similarity matrices is quite straightforward. The most difficult aspect would be to implement the creation of such matrices based on the HUSACCT Data Transfer Objects, but there appears to be no reason why this would be too complicated.

Sarkar et al. (2009) try to identify layers within an architecture. This is actually more in line with the decomposition step of the approach presented in this proposal, but they present it as though it were pattern recovery. This is really only true as long as that pattern is some simple layered architecture, but this paper is mentioned here nonetheless.

A similar algorithm has been partially built into HUSACCT by Leo Pruijt, based on Goldstein et al. (2015), which relies on packages. This may have to be completed to study its effect on the decomposition when compared with packages/namespaces.

A third example is one of template matching (Dong et al., 2008). This is very much the same as the similarity scoring presented by Tsantalis et al. (2006), since it is also based on matrices and similarity scores. Dong et al. (2008) also note this and point out the subtle differences, so one might think of this later paper as an improvement of the former.

Following the research questions presented in Section 2.1, it is now time to investigate how to express the fitness of a particular candidate in terms that HUSACCT understands: dependencies that are either violations or correct observances of the pattern’s architectural rules.

5.4.1 Fitness Function

The fitness of a particular pattern candidate is more than merely the number of violations of all the pattern’s architectural rules. Counting these violations is obviously essential, but this number would introduce some problems if it were the sole measure of fitness.

For example, a candidate that has a mapping of three software units to the three modules of a Model-View-Controller pattern would have a specific number of violations associated with it, based on the rules of the Model-View-Controller pattern. If this number turns out to be zero, i.e. there are no violations, then one would be inclined to think that this is a particularly good candidate. However, the fact that there are no violations in this case could be caused by the three software units having no direct dependencies connecting them to one another. If there are no dependencies, how could there be any violations? That the reported number of violations is zero is therefore trivial and inconsequential.

Would one want this candidate to receive a good fitness score, in light of this insight? Probably not, as it seems highly unlikely that a Model-View-Controller pattern would be implemented with its elements not relying on each other in the manner for which this pattern is intended. To the contrary, this seems to be a terrible candidate for the Model-View-Controller pattern. One of the software units could, theoretically, be in its correct place, but the whole mapping deserves a disastrous fitness score more than a perfect one.

But how to arrange this? After all, HUSACCT possesses a “Must use” rule type that applies to this situation. It requires that one module use another, which is what is needed here. There is a problem with this, however, and it is illustrated by determining this new fitness score that includes these “Must use” rules of the Model-View-Controller pattern.

Using the “Must use” rules, there would be one violation due to the software unit assigned to the Model not using the one assigned to Controller and one violation for the reverse. The same goes for View and Controller, so that is a total of four violations for this candidate. Four may be negligible next to the number of dependencies within this candidate.

Does that make this a terrible candidate? At least Model does not use View and vice versa, so it is not as bad as it could have been. Adding “Must use” rules to the pattern definition and counting their violations like HUSACCT would for any rule type is simply not sufficient, though. Thus, the following solution is implemented instead.

“Must use” rule violations ought to count more heavily towards a poor fitness score. It is a worse type of violation to any architectural pattern than any other type of rule being broken. A Model-View-Controller pattern candidate whose modules do not depend on one another simply is not a Model-View-Controller pattern instance. This type of violation should be ranked so severely that the current realisation simply rejects such pattern candidates all together. If the layers of an N-Layered pattern do not use each other in the allowed (required) way, then these layers do not constitute an N-Layered pattern. If even one pattern module does not rely on another despite the fact that it is supposed to, that candidate should perhaps be rejected without even looking at any other dependencies. This has the added benefit of speeding up computation, since HUSACCT can then swiftly move on to the subsequent candidate.

“Must use” heuristic:

If a “Must use” type rule of an architectural pattern is violated by a pattern candidate for that pattern, then this candidate can be discarded without further contemplation on any of

its other architectural rules and how they may or may not be violated by the candidate's dependencies.

This heuristic is more extreme than giving this type of violation a large weight, but it is a powerful way of speeding up computation.

Having established that the fitness function should default to a minimum value based on this heuristic, another concern needs to be addressed: the total number of dependencies. It is not difficult to see that a small amount of violations cannot be the only variable of a fitness expression, even after the heuristic has been applied.

The number of “good” dependencies, those that are supposed to be there, should be relatively high for a candidate to be a good fit. If a “Must use” requirement is met thanks to a single dependency, then this is not as strong of an argument for a pattern instance as a hundred dependencies would be. However, if one only looks at the absolute number of dependencies, then the candidates with the highest fitness scores will be those that show a high coupling between their software units. If this absolute number of dependencies be less important, these strongly coupled software units might be included in the pattern or might be aggregated within the same pattern module.

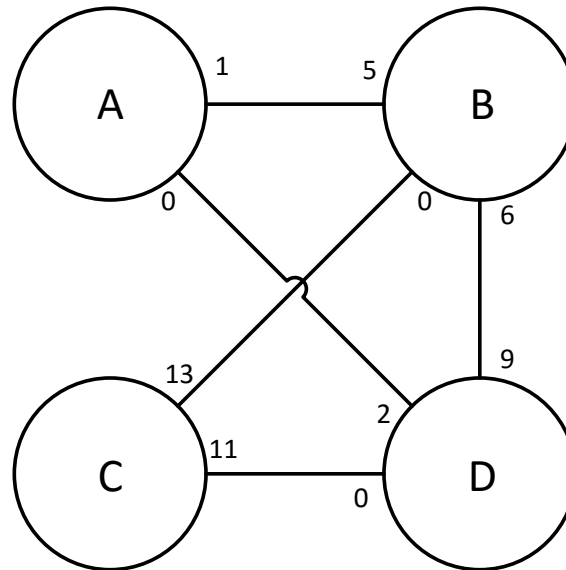


Figure 5.3: *Four imaginary software units and their mutual dependencies.*

For example, take software units A, B, C and D. They are connected through a number of dependencies, as depicted in Figure 5.3. This is another notation, where software units are represented as circular elements with bidirectional dependencies. In this figure, there are 5 dependencies from A to B and there is 1 from B to A.

If this set were passed to the pattern matching algorithm and were used to attempt the 3-Layered pattern while allowing for unassigned units, there would be $P(4,3) = 24$ possible candidates in the non-aggregation and $N(4,3) = 60$ in the aggregation cases. Two of these possible candidates are displayed in Figure 5.4. In candidate *a*, the following mapping has been used: $A \mapsto \text{Layer 1}$, $B \mapsto \text{Layer 2}$, $C \mapsto \text{Layer 3}$, $D \mapsto \text{Remainder}$. This means that D is not assigned to any of the pattern's modules, which has been explicitly allowed for before. Candidate *b* has the following mapping: $A \mapsto \text{Layer 1}$, $B \mapsto \text{Layer 2}$, $C \mapsto \text{Remainder}$, $D \mapsto \text{Layer 3}$.

Given these two candidates, which are both possible instances of the 3-Layered pattern, which one deserves a higher fitness score? The rules of this pattern are:

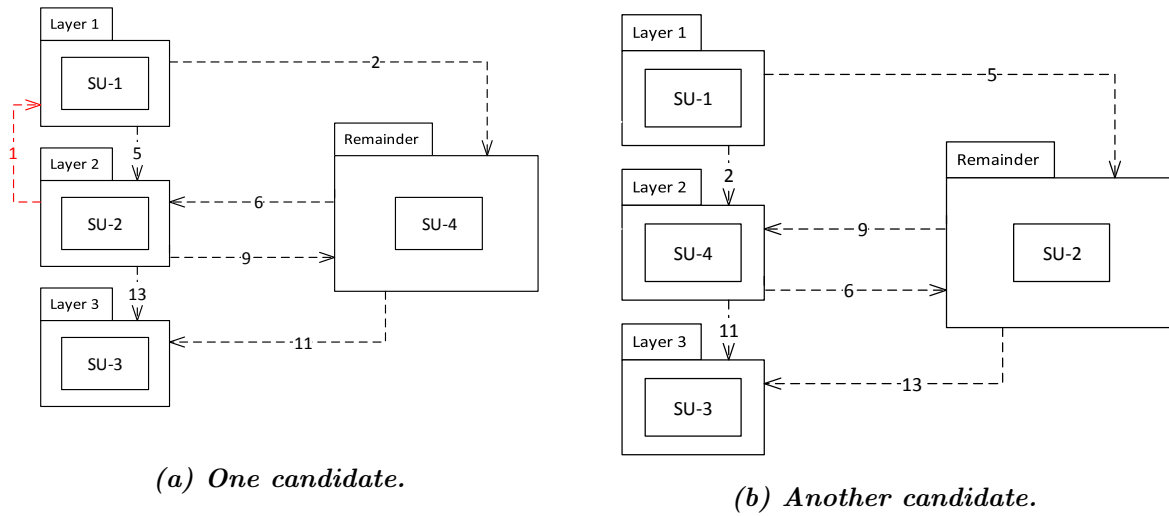


Figure 5.4: Two candidates for the 3-Layered pattern. See the set below for the architectural rules.

- 1: Layer 1 must use Layer 2.
- 2: Layer 2 must use Layer 3.
- 3: Layer 2 is not allowed to use Layer 1.
- 4: Layer 3 is not allowed to use Layer 2.
- 5: Layer 3 is not allowed to use Layer 1.
- 6: Layer 1 is not allowed to use Layer 3.

Given these rules, the conformance check results in the violation that is marked red in Figure 5.4a. Candidate *a* has only one single violation, which is a back-call from Layer 2 to Layer 1 due to software unit 2 having a dependency to software unit 1.

Candidate *b*, on the other hand, has absolutely no violations. Regardless of whether either of these is truly the best pattern candidate for these options and ignoring the question of whether there truly is a 3-Layered pattern instance here for now, these candidates require certain fitness scores.

Based on the number of violations, one would judge candidate *b* to be better than candidate *a*. Although, one might be inclined to state that candidate *a* is better, based on the number of correct dependencies. These “good” dependencies are simply those that comply with the “Must use” rules.

18 out of a total of 19 intra-pattern dependencies are in accordance with the “Must use” rules for candidate *a*, while there are only 13 required dependencies for candidate *b*, albeit that this is the total amount of intra-pattern dependencies in the candidate. Which candidate is better? Which value is more important, the absolute number of violations or the relative number compared to the desirable dependencies? Or would an expression that involves both values be, in fact, ideal?

Perhaps there exist many possible fitness functions that would do the trick. One of them, which will be used for the rest of this thesis, is inspired by the well-known F-measure from statistics. This F-measure (or F-score) is a harmonic mean of Precision and Recall, thereby

describing the accuracy of a test (Sokolova, Japkowicz & Szpakowicz, 2006). Both Precision and Recall should be as high as possible (maximum of both is 1) in order for the F-measure itself to be maximal (also 1). Similarly, our fitness function can be formulated in such a way that we use such a harmonic mean.

Let us first define the variables at our disposal:

- $D \in \mathbb{N}$: # dependencies
The number of dependencies between the pattern modules and with the Remainder.
- $M \in \mathbb{N}$: # “Must use” dependencies
The number of dependencies between pattern modules that are in line with that pattern’s “Must use” rules. These are essential dependencies, since these are the ones that are explained by the existence of the pattern.
- $V \in \mathbb{N}$: # violations
The number of violating dependencies. Violation of a “Must use” rule implies that $M = 0$ for that particular rule.

There are essentially two ratios that ought to be optimised. On the one hand, there is the number of dependencies explained by the pattern relative to the total number of dependencies of that pattern candidate: $\frac{M}{D}$, which should be minimised. On the other hand, we have the number of violations relative to the number of dependencies. Since $M \neq 0$ implies that the number of violations due to “Must use” rules is equal to 0 and since we want both ratios to vary between 0 and 1 for the sake of the harmonic mean, we should subtract M from D for this latter expression: $\frac{V}{D-M}$. This leads us to the following fitness function for $f \in \mathbb{R}$:

$$f(D, V, M, \beta) = (1 + \beta^2) \frac{(1 - \frac{V}{D-M}) \cdot \frac{M}{D}}{\beta^2 \cdot (1 - \frac{V}{D-M}) + \frac{M}{D}} \quad (5.1)$$

The variable $\beta \in \mathbb{R}$ controls the relative importance of one ratio over the other by controlling the relative weight of one expression in the mean. To establish this fitness function with both ratios equally valuable, take $\beta = 1$. The fitness function can then be reduced to:

$$f(D, V, M, 1) = 2 \frac{(1 - \frac{V}{D-M}) \cdot \frac{M}{D}}{(1 - \frac{V}{D-M}) + \frac{M}{D}} \quad (5.2)$$

Other fitness functions might tend towards results where there is more or less aggregation, larger or smaller pattern candidates, more or less emphasis on “Must use” affirmations. This would have to be extensively studied in future work, in order to learn when a particular function is best suited.

In summation, (5.1) is a fitness function such that $range(f) \subseteq [0, 1]$. We thereby maximise the number of dependencies that can be explained by the pattern and minimise the number of dependencies that violate the architectural rules of that pattern simultaneously. This is only one possible fitness function and although other functions could lead to more desirable candidates receiving a higher fitness score, this expression is deemed sufficiently satisfactory for the time being. Future work may include experimentation with and improvement of such functions.

5.5 Running Example Continued

In our earlier example, we looked at HUSACCT and how its ACC functionality can be used for SAR. A small group of software units from a package hierarchy were considered and these were mapped to the layers of a 3-Layered pattern. Now we will continue with this example, but with the addition of some yet to be fully realised HUSACCT functionality.

What if one could tell HUSACCT to insert an architectural pattern into the defined architecture? With the click of a button, one could have all three layers associated rules appear, instead of having to do this by hand. Layer-type modules automatically receive rules on skip-and back-calls when added, this is already part of HUSACCT's functionality concerning layers, but architectural patterns use many different types of rules that would otherwise have to be added manually. If one plans on adding a Broker pattern, all it takes is one click and there it is: a series of empty modules and architectural rules tying them together that collectively form a Broker pattern. As long as HUSACCT be familiar with this pattern, adding it to the defined architecture would be a breeze.

But what about the mapping? Adding a software unit to an empty pattern module is already a matter of a few mouse clicks so it should be rather straightforward for HUSACCT to do this for us. The software units are selected and allowed to be mapped. Then, HUSACCT takes care of the rest: it assigns each unit to a module and can even ask for validation to take place.

So there it is, "we" selected some software units from the package hierarchy and chose an architectural pattern to attempt, after which HUSACCT automatically tells us about the violations of the pattern's rules that incur for one particular mapping of these software units to the pattern modules. This particular mapping is what is referred to as a (pattern) candidate.

Apart from making our selection of software units and choosing an architectural pattern to which these are to be assigned, we have other decisions to make. First of all, is it desirable that the automatic mapper assigns multiple software units to the same pattern module, or do we want to get a candidate for which each module contains only one software unit?

If we allow for such aggregation of multiple units per module, we have to make a decision on a subsequent consideration: do we want this candidate to contain all the software units from our selection, or would we be fine with ending up with a candidate in which some of our selected units were left out of the pattern and remain unassigned?

The reason why these decisions have to be made is that the obvious next step is concerned with attempting multiple candidates. We do not just want some random mapping of software units to pattern modules, just so that HUSACCT can track down the violations that it associates with this particular candidate, whatever it may be. Instead, we want HUSACCT to then attempt another candidate, and another, and yet another, until it can tell us with some certainty that it has found a candidate that fits so well that it might just be a genuine pattern instance.

Once we have provided HUSACCT with these options (software units, an architectural pattern, whether to allow for aggregation and if so, whether to demand that all software units be used), it can attempt a series of candidates for us. But which candidates do we want it to consider? Surely, we can come up with hundreds if not thousands of ways to map a selection of software units to even a simple architectural pattern? Their exact number is addressed in the next chapter. The automatic mapper goes through each and every one of these candidates and HUSACCT keeps tracks of which candidates performed particularly well. The matter of how to express this fitness and compare the candidates is addressed in Section 5.4, but suffice it to say for now that we remember a top N of candidates once all candidates have been evaluated. The resulting Process-Deliverable Diagram which specifies the *Evaluate candidates* activity further is shown in Figure 5.5.

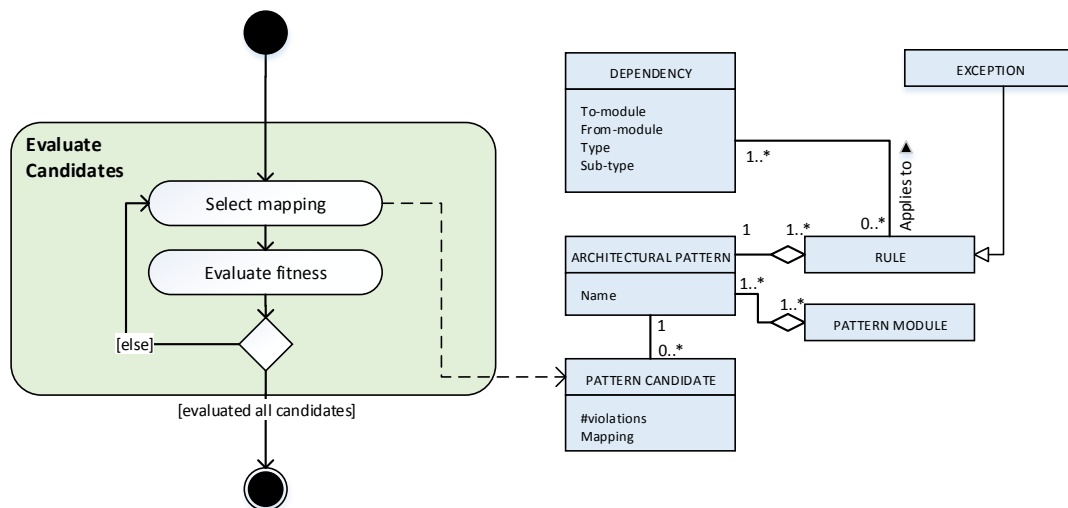


Figure 5.5: A PDD brute force approach to the Evaluate candidates activity.

Chapter 6

Search Algorithms

This chapter deals with the two approaches that were considered in this thesis: brute force and genetic algorithms. The former is the more straightforward albeit tedious approach, whereas the latter is the more convoluted yet innovative way of solving this particular problem.

6.1 Number of Candidates

Development was initially focussed on the *brute force* approach, which is to simply attempt all permutations of eligible packages mapped to the vacant module slots in an architectural pattern. This term is used in computer science and cryptography, among others, and is essentially an exhaustive search. Let us calculate the number of candidates for this brute force approach.

With n the number of software units that we are currently considering for a particular pattern and k the number of modules within that pattern, the total number of candidates N is quite trivial if we make the following assumption: each module can only store one software unit. In other words, the mapping has to be one-to-one. In this case, the number of candidates is simply the multiplication of the number of combinations in which to arrange n into groups of k and the number of times each such group can be ordered. The ordering refers to the labelling of the pattern modules. This is depicted in (6.1).

$$\begin{aligned} N(n, k) &= k! \frac{n!}{k!(n-k)!} \\ &= \frac{n!}{(n-k)!} \\ &= k! \cdot \binom{n}{k} \end{aligned} \tag{6.1}$$

In the case of a Model-View-Controller pattern, for example, a set of say 10 packages would result in $N(3, 10) = 720$ possible mappings. It is unclear at this stage how long this computation would take, so there is no reason to discard this approach as too computationally expensive beforehand. Even if this were to take several hours, it would still be faster than a human could do the same thing. As long as the results are useful, so is the computation. More sophisticated techniques, aimed at lessening the computation time without sacrificing too much accuracy, are possible. Inexact graph matching, for example. These would constitute possible extensions after the brute force mapper has been realised.

However, what happens when multiple software units can be mapped to the same pattern module? This is not unreasonable, since there is no particular reason why a single software unit ought to correspond to a single pattern module. In that scenario, which we call the aggregate

case, the number of candidates drastically increases. This is shown in (6.3), for which the expression shown in (6.2) is needed. The number of ways in which n can be distributed over k groups, allowing for aggregates and requiring that each group ends up with a value of at least one, is called the Stirling Number of the Second Kind. This has to be multiplied by the factorial of the number of groups in order to correct for the fact that these are labelled groups.

$$\begin{aligned}
 S_2(n, k) &= \left\{ \begin{matrix} n \\ k \end{matrix} \right\} \\
 &= \frac{1}{k!} \sum_{j=0}^k (-1)^{k-j} j^n \binom{k}{j}
 \end{aligned} \tag{6.2}$$

$$\begin{aligned}
 N^a(n, k) &= k! \cdot \left\{ \begin{matrix} n \\ k \end{matrix} \right\} \\
 &= \frac{k!}{k!} \sum_{j=0}^k (-1)^{k-j} j^n \binom{k}{j} \\
 &= \sum_{j=0}^k (-1)^{k-j} j^n \frac{k!}{j!(k-j)!}
 \end{aligned} \tag{6.3}$$

The number of possible candidates in this aggregate case, $N^a(n, k)$, is still not the whole story, though. It assumes that each and every one of the software units are mapped to one of the pattern modules. This may not always be the case, since it can be of added benefit if software units are left out of the pattern for a particular candidate, if that benefits the fitness score. So there will always be an additional module, one that is allowed to be empty but not required to be. This increases the number of possible candidates further, which can be calculated by adding the number of cases for $k + 1$ groups to expression (6.3). This results in expression (6.4), where $N^{a,r}(n, k)$ is the number of possible mappings in case of aggregation and remaining software units not assigned to the pattern.

Note that $N(n, k) = N^r(n, k)$, i.e. the non-aggregation case assumes the existence of a remaining portion of software units, as long as $n \leq k$. This additional group is bound to be an aggregation if $n > k + 1$. Additionally, do note that the existence of a Remainder here implies that a portion of the selected software units is not mapped to the pattern, whereas there may already be a Remainder in the sense that this selection does not necessarily encompass all available software units. Our approach puts a lot of emphasis on the Remainder by incorporating it in the pattern discovery process, dependencies with software units outside of the selection are therefore implied to be of no consequence. Whenever software units are of interest, they have to be included in the selection and thus increase the number of pattern candidates.

These expansions and rewritings do not result in more elegant expressions, but provide a better indication of how quickly these numbers would grow with increases in n and k . To further illustrate the combinatorial explosion that these expressions cause, Table 6.1 shows the various numbers of N for several values of k and n . These numbers range into the order of magnitude $10^{12} - 10^{14}$, which is comparable to the number of fish on Earth, cells in the human body or

stars in the Andromeda galaxy.

$$\begin{aligned}
N^{a,r}(n, k) &= k! \cdot \{n\}_k + (k+1)! \cdot \{n\}_{k+1} \\
&= \frac{k!}{k!} \sum_{j=0}^k (-1)^{k-j} j^n \binom{k}{j} + \frac{(k+1)!}{(k+1)!} \sum_{j=0}^{k+1} (-1)^{k+1-j} j^n \binom{k+1}{j} \\
&= \sum_{j=0}^k (-1)^{k-j} j^n \binom{k}{j} + \sum_{j=0}^k (-1)^{k+1-j} j^n \binom{k+1}{j} + (-1)^{k+1-(k+1)} (k+1)^n \binom{k+1}{k+1} \\
&= \sum_{j=0}^k \left((-1)^{k-j} j^n \binom{k}{j} + (-1)(-1)^{k-j} j^n \binom{k+1}{j} \right) + (k+1)^n \tag{6.4} \\
&= \sum_{j=0}^k \left((-1)^{k-j} j^n \left(\binom{k}{j} - \binom{k+1}{j} \right) \right) + (k+1)^n \\
&= \sum_{j=0}^k \left((-1)^{k-j} \frac{j^n}{j!} \left(\frac{k!}{(k-j)!} - \frac{(k+1)!}{(k+1-j)!} \right) \right) + (k+1)^n
\end{aligned}$$

Table 6.1: An illustration of how quickly the number of candidates becomes astoundingly large. $k = 3$ could represent a Model-View-Controller pattern and $k = 4$ a 4-Layered pattern, for example.

n, k	N	N^a	$N^{a,r}$
3,3	6	6	6
4,3	24	36	60
5,3	60	150	390
6,3	120	540	2,100
7,3	210	1,806	10,206
8,3	336	5,796	46,620
9,3	504	18,150	204,630
10,3	720	55,980	874,500
15,3	990	14,250,606	1,030,793,406
20,3	1,320	3,483,638,676	1,089,054,420,300
4,4	24	24	24
5,4	120	240	360
6,4	360	1,560	3,360
7,4	840	8,400	25,200
8,4	1,680	40,824	166,824
9,4	3,024	186,480	1,020,600
10,4	5,040	818,520	5,921,520
15,4	32,760	1,016,542,800	26,308,573,200
20,4	143,640	1,085,570,781,624	90,990,301,641,624

In the next section, the brute force approach that does actually evaluate each and every candidate in this table will be described.

6.2 Realisation of the Brute Force Approach

The choice of aggregation versus non-aggregation leads to distinct methods. Code Sample D.1 in the Appendix shows the actual Java code, but the algorithms can be reduced to the same simple pseudo code description (Algorithm 1). The methods are very similar: a mapping of software units to pattern modules is taken from a mapping generator class, this mapping is placed in HUSACCT's defined architecture and the validation of this architecture is used to determine a fitness score. Rather than keeping track of all results, only the top candidates are stored in the aggregation case, preventing memory problems. When there are no mappings left to evaluate, the algorithm ends. The best results are presented and the very best is again placed in HUSACCT's architecture. Methods and mapping generator vary depending on whether it be the aggregation case or not, and the aggregation mapping generator needs to be informed if there is a Remainder, but this is what the code boils down to.

Algorithm 1 A summarised version of the brute force approach.

```

while mappingGenerator.hasNext() do
    mapping  $\leftarrow$  mappingGenerator.next()
    candidate  $\leftarrow$  mapPattern(mapping)
    fitness  $\leftarrow$  determineFitness(validate(candidate))
    repopulateTopScores(candidate, fitness)
end while

```

In the non-aggregation case, there is no choice for a Remainder or not. If the selection of software units is larger than the number of pattern modules, there has to be a Remainder. In the aggregation case, however, this becomes a deliberate choice. If so decided, the algorithm has to run again, but with an additional pattern module that represents the Remainder. The top candidates may then consist of both Remainder and non-Remainder pattern candidates.

6.3 Genetic Algorithms

With the brute force approach completed, or at least its first realisation functioning as a proof of concept, it is time to move on to a more sophisticated approach. The method envisioned in this thesis so far is one of guided automation, with a user deciding on a selection of software units (discovered using a user-selected approach, e.g. charting the package hierarchy) and an architectural pattern to be attempted.

Furthermore, the user has to decide whether multiple software units are allowed to be assigned to the same pattern module and whether some of the selected software units are allowed to be excluded from the mapping as part of what we have dubbed the Remainder, i.e. the architectural elements that do not correspond to any pattern modules or their sub-modules given a specific candidate.

The brute force approach is itself fully automated: once equipped with all the necessary information and a fitness function by which to evaluate pattern candidates, the algorithm simply dictates that all candidates within the given search space must be evaluated. Thorough though it may be, this exhaustive search is quite costly in terms of computation time. Albeit still faster than any human could use HUSACCT in this way, faster many times over, it can be said to be inherently wasteful. Presumably, a great many of the candidates that come about in the process are hopelessly unfit and therefore form an unnecessary drain on computer resources. In this section, a possible tactic by which to address this wanton consumption of precious time and energy is discussed, a tactic that uses a very specific type of search technique: a genetic

algorithm. This is not the only possible technique, but it is a technique that can be applied in this situation due to the existence of a fitness function. Genetic algorithms come with a broad field of scientific research and the knowledge that these algorithms sometimes turn out to be very powerful. In addition to these reasons, the choice of this technique is in line with the exploratory part of exploratory design research, as we were rather interested to see where this train of thought would take us.

Evolutionary computation is a sub-field of artificial intelligence that relies heavily on Darwinian evolution as the basis of its algorithms. A particular subset of the evolutionary algorithms are known as genetic algorithms, which allow for the optimisation of specific search problems using a heuristic aimed at mimicking natural selection, finding a useful solution to a problem by allowing some kind of genetic code to evolve (Goldberg & Holland, 1988). Genetic algorithms rely on a population of “individuals”, essentially programmed chromosomes designed to code for possible solutions to the problem at hand, and an expression of their suitability via some sort of fitness function. By allowing this population to reproduce, as it were, according to this individual fitness and randomly mutate along the way, local optima tend to be found in a much shorter timespan than would be expected of an exhaustive search. In order for this to be true, the genetic encoding and the fitness function must, of course, be appropriate.

The identification of architectural patterns within dependency graphs can be thought of as a graph partitioning problem, where nodes are placed in one of the pattern’s modules or kept out of the pattern. Genetic algorithms, a subcategory of evolutionary computing, have been used for graph partitioning problems for many years now. For example, a hybrid, steady-state genetic algorithm was applied to a k-way partitioning problem by Thang Nguyen Bui and Byung Ro Moon (1996).

Another approach from around that time emphasised a genetic search in inexact graph matching instead of graph partitioning, but this can also be related to pattern matching (Cross, Wilson & Hancock, 1997). Inexact graph matching is called error-correcting graph isomorphism in Yuan-Kai Wang, Kuo-Chin Fan and Jorng-Tzong Horng (1997), but it is clear that the topic has received a lot of attention in this context.

In 1999, a paper was published that described a genetic algorithm not for finding particular sub-graphs, but for the problem of finding an optimal decomposition of software architecture (Doval, Mancoridis & Mitchell, 1999). This is essentially a genetic alternative to k-means clustering, automatic layering or any other such clustering approach. As such, genetic algorithms could be applied to the first (decomposition) step of our architectural pattern discovery approach instead of the second step, as well.

An explanation of the various considerations when using genetic algorithms for a graph partitioning problem can be found in the paper by J. Kim, Hwang, Kim and Moon (2011). Such characteristics include normalisation, optimisation and representation. They provide an overview of related approaches that anyone who uses genetic algorithms might find useful. An important consideration for our purposes, however, is that a dependency graph is essentially an edge-weighted directional graph and not all algorithms discussed in papers such as this apply to our circumstances.

As with most of such techniques, the idea of genetic algorithms is that the discovery of local optima will eventually lead to convergence on a global optimum. The analogy with (neo-)Darwinian evolution can be interpreted in many ways, leading to a variety of implementations of the fundamental observation that biological evolution results in powerful solutions through the harsh evaluation of many random solutions.

Genetic algorithms, or evolutionary computing in general, are not a method of modelling natural evolution itself. Rather, it is an attempt at solving very specific kinds of problems. It involves stochastic searches that use populations of solutions in a way that is inspired by the

mechanisms behind biological evolution.

In general term, the process is as follows: 1) generate a random population, 2) evaluate the individuals of this population, 3) evolve the population, 4) repeat steps 2 and 3 until termination criteria are met.

The evolution of the population is itself a process of several steps. Usually, a combination of mutation and crossover operations (recombination between “mates”) is used. The new population is then evaluated, with some selection criterion determining which individuals are allowed to “survive”, as it were, into the next generation.

In nature, DNA copies often contain a number of differences from the original in their genetic code. Mutation is the cause of small copying errors, which are just minor changes in individual genes that may or may not have profound effects. Chromosomal crossovers are more severe recombinations of whole sections of DNA that occur during the production of sperm and egg cells in sexual reproduction.

Let us, for the sake of illustration, consider a simple but useful example. There is an array of binary integers of length four, and a fitness function f that is merely the value of the integer that is represented by the binary integers.

$$f = \sum_i x_i \cdot 2^i \quad (6.5)$$

If the maximum value of i is 4, then $f \in [0, 15]$. If one generates a random initial population, it might look something like the example in Table 6.2.

Table 6.2: The initial population.

Chromosome	f
1001	9
0101	5
1010	10
1000	8
0010	2
0111	7
Mean	~ 6.83

Using this population, tournament selection is applied when pairing up individuals. This is a way of increasing the likelihood that fit individuals produce offspring. There is a random factor at work here, so the resulting parents in Table 6.3 are just an example of what the result might be.

Table 6.3: The selected population marked with crossover positions.

Chromosome	f
100/1	9
101/0	10
1/000	8
1/010	10
01/11	7
10/01	9
Mean	~ 8.83

The forward slashes indicate the positions at which single-point crossovers are about to take place. In this illustrative scenario, crossover can take place at any point of the chromosome, but only if another chromosome wants to cross over in the same spot (which determines the pairing). This results in the recombined chromosomes of Table 6.4.

Table 6.4: The selected population after crossover and marked for mutation.

Chromosome	f
1000	8
1011	11
1000	8
1010	10
0101	5
1011	11
Mean	~ 8.83

Subsequently, mutation occurs at the boldfaced, randomly chosen positions. Since these genes are binary, 1s change to 0s and 0s change to 1s. The result is displayed in Table 6.5, together with the new fitness scores. Due to the randomness at work here, there is no guarantee that the optimal solution (gene: 1111, $f = 15$) will be found. However, it is very likely that this solution or the ones close to it would soon dominate the population. See Tables 6.6 and 6.7 for an example of the next two generations.

Table 6.5: The resulting new generation.

Chromosome	f
1010	10
1001	9
0010	2
1100	12
0100	4
1001	9
Mean	~ 7.67

Table 6.6: The third generation.

Parent chromosomes	f		Child chromosomes	f
10/01	9	1001	1101	13
10/01	9	1010	1011	11
1/100	10	1001	1011	11
1/001	9	1100	1110	14
110/0	12	1100	0100	4
101/0	11	1010	1011	11
Mean:	10		Mean:	~ 10.67

Table 6.7: The fourth generation.

Parent chromosomes	f		Child chromosomes	f
1/101	13	1110	1010	10
1/110	14	1101	1100	12
11/10	14	1111	1101	13
10/11	11	1010	0010	2
110/1	13	1101	1001	9
101/1	11	1011	1010	10
Mean:	~ 12.67		Mean:	~ 9.33

Evidently, the average fitness has increased thanks to this combination of tournament selection, chromosomal crossover and random mutation. Other variations of these procedures may yield somewhat different results and this thesis is not the place to go into all of these considerations. Suffice it to say that, when performed over a number of generations, this algorithm will result in increasingly fitter populations. The point of genetic algorithms is that this can be a fast and easy approach at finding good solutions. Fitness might decrease for a particular generation, as seems to have happened with the fourth generation in the example, depending on selection criteria and randomisation. Given enough time, though, fitness tends to increase more than it might sometimes decrease.

Many considerations apply when using genetic algorithms. Selection, crossover and mutation can all be performed in a variety of ways and the fitness function itself is far from clear cut. This is why evolutionary computing is taught as a complete master course at Utrecht University and undoubtedly at other institutions as well.

Instead of formulating a particular implementation beforehand or discussing the effects of various techniques and parameters through extensive experimentation, this thesis serves primarily as a series of proofs of concept. The aim is to show that it is possible to treat the reconstruction of software architecture as a genetic local search problem, so that future research can find optimal approaches to this.

6.4 Genetic Code

Although there may be other ways to encode pattern candidates in a fashion suitable for genetic algorithms, the following structure is used in this thesis.

In humans, each ordinary cell contains a full genome, constructed out of DNA molecules and separated into 24 pairs of what are called chromosomes. The two members of each pair come from the two parents of the person. Other species, especially other mammals, have a very similar genetic structure, albeit that the amount to chromosomes varies greatly.

In our genetic algorithm, however, things are significantly more concise: a chromosome represents a single pattern candidate. It is one individual from the total population and its genetic code contains information on its mapping of software units to the pattern modules (and possibly the Remainder, if allowed). To keep things relatively simple and straightforward, a legend is used for the software units, meaning that the selection of software units is indexed and integers can be used to refer to these units, as place holders for their unique names.

A biological chromosome consists of genes, which are portions of the DNA that code for something. In living cells, these blocks of code describe protein structure or gene expression. In our genetic algorithm, each gene codes for the mapping of a software unit, so that means that there must be as many genes per chromosome as there are software units in the user-provided selection.

Each gene has an allele, which is the term for the actual value of a particular gene. In biology, these are series of nitrogen-bases that correspond to particular amino-acids in groups of three. In our algorithm, things are again slightly simpler. Whatever the architectural pattern to which the user is attempting to map the software units, the integer 0 as an allele stands for a mapping to the Remainder, i.e. not to any of the pattern's modules. If all software units are supposed to be assigned to the pattern, then none of the genes is allowed to have an allele of value 0.

Other integer values for such an allele logically correspond to other mappings. For example, in the case of a Model-View-Controller pattern, one might code the three pattern modules as the integers 1, 2 and 3, respectively. Following this, a chromosome may look like the one presented in Table 6.8. An allele of value 1 represents a mapping of that particular software unit to the Model-module, 2 to the View-module and a value of 3 means that the software unit is assigned to the Controller-module. As such, all software units are assigned to the pattern if there be a non-zero value for each gene. The chromosome in Table 6.8 therefore corresponds to the schematic in Figure 6.1, with each software unit assigned to its appropriate place with deference to the displayed legend.

Table 6.8: *An example of a chromosome for 7 software units.*

Genes:	SU-1	SU-2	SU-3	SU-4	SU-5	SU-6	SU-7
Alleles:	2	0	2	1	3	1	0

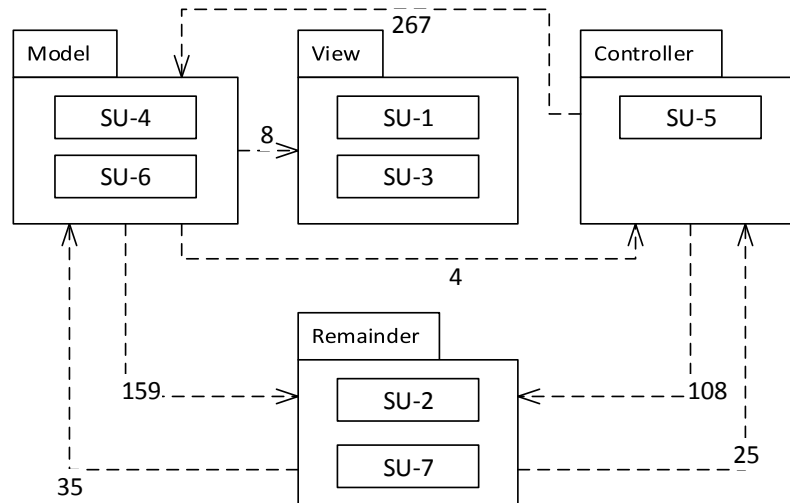


Figure 6.1: *The mapping of 7 software units to the MVC pattern & Remainder resulting from the chromosome in 6.8.*

Let us look at a more concrete example of this genetic encoding. We take the Model-View-Controller (Complete Freedom) pattern as our designated target (i.e. the pattern we are attempting). The following diagram results from HUSACCT's analysis of its own source code and the architectural elements in it are nothing but the unadulterated units taken from the package hierarchy (Figure 6.2).

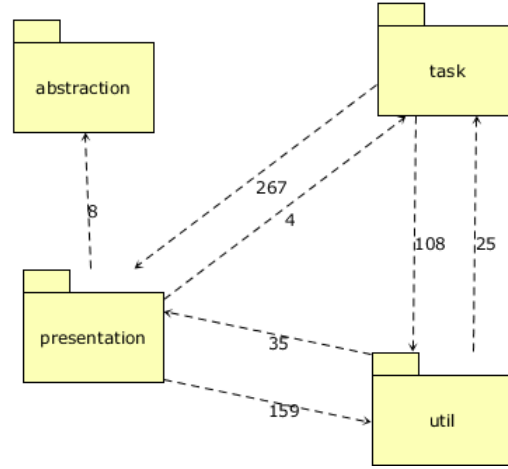


Figure 6.2: Analysis results of HUSACCT's own source code (v1.0) within its Graphics subsystem.

Using the legends of Tables 6.12a and 6.12b, we can code this in the fashion of the chromosome shown in Table 6.10.

Table 6.9: The legends for this candidate

Model	View	Controller	Remainder
1	2	3	0

(a)

SU-1	SU-2	SU-3	SU-4
Abstraction	Presentation	Task	Util

(b)

Table 6.10: An example of a chromosome for 4 software units mapped to the MVC pattern.

SU-1	SU-2	SU-3	SU-4
2	2	1	3

The resulting violations can be read from the diagram in Figure 6.3, which is how the situation would look if performed manually within the HUSACCT tool. This particular candidate has associated with it a particular fitness score. The calculation of the fitness score is presented in (6.6).

$$\begin{aligned}
 f(D, V, M, 1) &= 2 \frac{(1 - \frac{V}{D-M}) \cdot \frac{M}{D}}{(1 - \frac{V}{D-M}) + \frac{M}{D}} \\
 &= 2 \frac{(1 - \frac{108}{267+35+25+108+4+159-(267+35+25)}) \cdot \frac{267+35+25}{267+35+25+108+4+159}}{(1 - \frac{108}{267+35+25+108+4+159-(267+35+25)}) + \frac{267+35+25}{267+35+25+108+4+159}} \\
 &= 2 \frac{(1 - \frac{108}{271}) \cdot \frac{327}{598}}{(1 - \frac{108}{271}) + \frac{327}{598}} \\
 &\approx 0.5728
 \end{aligned} \tag{6.6}$$

Let us now consider a single, random mutation of this chromosome. One particular gene, say the second gene on the chromosome, undergoes a random mutation. Its current allele value is 2, so its possible values as a result of the mutation are 1, 3 and, since we allow for assignment

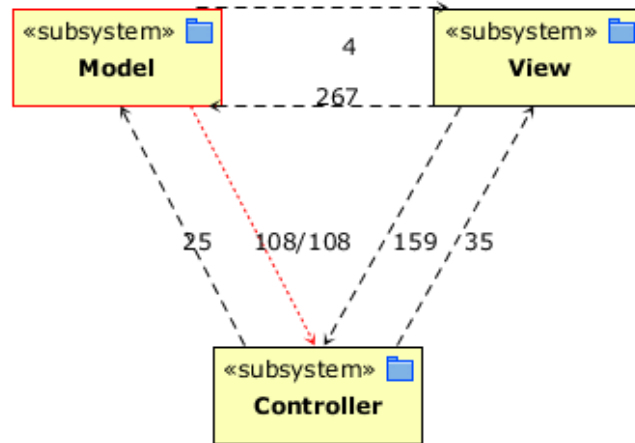


Figure 6.3: The resulting architecture diagram as if validation were manually called within HUSACCT. Model is not supposed to use Controller, so the dependencies between Presentation and Task result in violations (coloured red).

to the Remainder, 0. Mutation ought to be completely random, like a biological mutation in a creature’s DNA resulting in an allele that differs from the original as though it were a copying error. Let us say, for the sake of this example, that this mutation occurs in such a way that the allele value transitions from 2 to 3. The new chromosome is shown in Table 6.11.

Table 6.11: The chromosome post-mutation.

SU-1	SU-2	SU-3	SU-4
2	3	1	3

Since this new chromosome represents a new pattern candidate, we look at Figure 6.4 for a new visual depiction of the current mapping. We can tell, given the unchanged legend, that software unit 2 has changed positions. The allele value of the second gene, which does the encoding for software unit 2’s position, changed so that the software unit by the name of Presentation has been moved from the View- to the Controller-module. The resulting pattern candidate has a slightly different mapping from the one before the mutation and its fitness should differ by an amount that depends on the software dependencies between Presentation and the other software units. The number of dependencies that violate the “Is not allowed to use” rule between Model and Controller has increased. However, there are no dependencies from View to Model now, which is a violation of the “Must use” rule. This candidate is therefore immediately excluded, even though equation 6.6 could easily be applied to these new values.

In the manner described here, pattern candidates can change by randomly mutating the genes that code for the mapping of individual software units. This way of translating the pattern matching problem to a format suitable for usage with a genetic algorithm is elegant but not trivial, nor is it unambiguously unique. For it may be tempting to structure the chromosomes the other way around: each gene could stand for a pattern module and the allele values would then correspond to software units. This may seem more intuitive, since one is placing the software units into different groups and this type of sorting is more similar to a visual representation like in Figure 6.1.

However, allele values would be more complex in this scheme in the aggregation case. In Tables 6.12a and 6.12b, the same two candidates for the 3-Layered pattern are depicted for

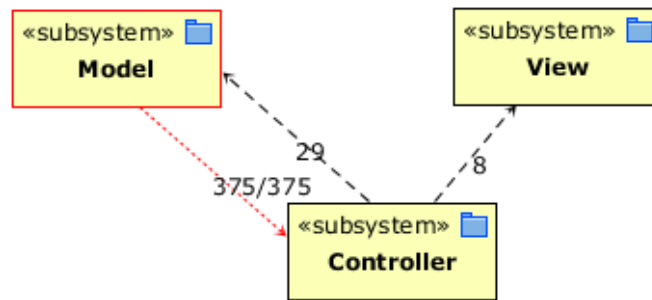


Figure 6.4: The new architecture diagram.

Table 6.13: A quick overview of the genetic structure.

Chromosome	Single Pattern Candidate
Gene	Single Software Unit
Allele	Corresponding Pattern Module

a group of five software units. Because multiple software units are assigned to the pattern modules, each gene needs to hold multiple indices (integers) in order to record which software units go where in the mapping. Not that this would be impossible to do, but this supposedly more intuitive way of structuring the genetic code makes the genes themselves unnecessarily complex. It is easier, both in terms of source code and conceptual understanding, to have the genes represent software units and have the allele values refer to the pattern modules via simple integers.

Table 6.12: The legends for this candidate. In both tables, the integers on the lower rows are indices for the pattern modules.

SU-1	SU-2	SU-3	SU-4	SU-5
1	2	1	3	2

(a)

Layer 1	Layer 2	Layer 3	Remainder
1,3	2,5	4	/

(b)

To conclude, Table 6.13 shows the set-up used to encode the mapping of pattern candidates in this approach. This approach results in a specification of the second activity of the overall PDD (Figure 5.1), displayed in Figure 6.5. In the next section, the general gist of the way this genetic algorithm and encoding were realised is briefly described.

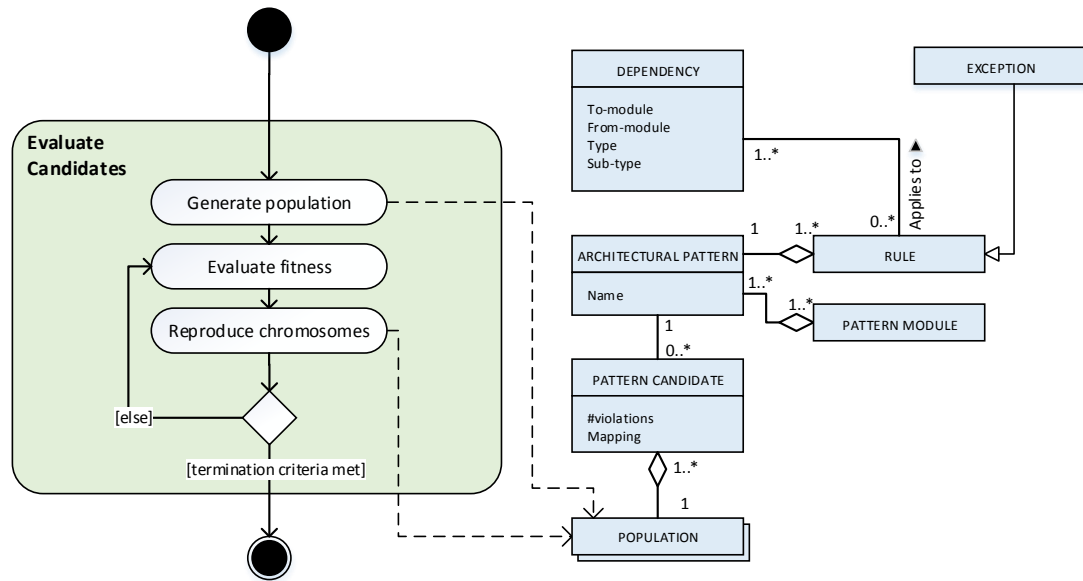


Figure 6.5: A PDD of the genetic approach to the Evaluate candidates activity.

6.5 Realisation of the Genetic Approach

The idea of using a genetic approach was conceived relatively late in the project and thus did not allow for extensive experimentation or analysis. As such, an existing library for genetic algorithms was used in combination with the existing code. Specifically, JGAP (Java Genetic Algorithms Package) was chosen, due to its modular nature and accessible documentation¹. JGAP was simply added to the Eclipse project as an external JAR file, leaving only the essentials that still had to be implemented: the fitness function and the specification of chromosomes. JGAP requires these two classes to be written, but allows for many more aspects of its configuration to be changed or expanded. In the future, we can use smarter fitness functions or more elaborate genetic operators to take the place of the default JGAP configuration. By default, the mutation rate is set to one in twelve genes randomly changing to another allowed value and crossovers takes place at random points in the chromosomes at a rate of 35% of the population size. Future experimentation might result in more optimal values.

The genetic algorithm was applied to the aggregation case only (with or without Remainder), primarily because there is little reason to use the algorithm for a small amount of candidates, when the non-aggregation brute force approach would probably be satisfactory. The Java class of the genetic algorithm is presented in Code Sample D.6, but it is very easy to summarise. The default JGAP configuration is loaded, chromosomes are constructed using IntegerGenes (JGAP objects) and a randomised population of size 100 is generated. Evolution subsequently takes place using JGAP's classes. The algorithm currently runs for a fixed number of generations, but more sophisticated termination criteria, like detecting convergence of the population, are expected to be part of future work.

Although validation and the calculation of a fitness score can be done with the same code

¹The JGAP website: <http://jgap.sourceforge.net/>, overview of JGAP v3.4.4: <http://greppcode.com/project/rep01.maven.org/maven2/net.sf.jgap/jgap/>.

as in the brute force approach, JGAP requires that a fitness function class be overwritten. This class is presented in Code Sample 6.1.

```
package husacct.analyse.task.reconstruct.genetic;

import org.jgap.FitnessFunction;
import org.jgap.Gene;
import org.jgap.IChromosome;

import husacct.analyse.task.reconstruct.ReconstructArchitecture;
import husacct.analyse.task.reconstruct.patterns.Pattern;

public class GeneticFitnessFunction extends FitnessFunction {

    private static final long serialVersionUID = 1L;
    public final static int MAX_BOUND = 4000;
    private ReconstructArchitecture reconstruct;
    private Pattern pattern;

    public GeneticFitnessFunction(Pattern currentPattern, ReconstructArchitecture
        reconstructArchitecture) {
        reconstruct = reconstructArchitecture;
        pattern = currentPattern;
    }

    // Determine fitness value for given a chromosome (pattern candidate). Higher
    // fitness should mean a better candidate throughout this class.

    public double evaluate(IChromosome a_subject) {
        // Take care of the fitness evaluator. It could either be weighting higher
        // fitness values higher (e.g.DefaultFitnessEvaluator). Or it could
        // weigh lower fitness values higher, because the fitness value is seen as a
        // defect rate (e.g. DeltaFitnessEvaluator)
        boolean defaultComparation = a_subject.getConfiguration().
            getFitnessEvaluator().isFitter(2, 1);
        if (defaultComparation == false) {
            return 1.0;
        }
        Gene[] genes = a_subject.getGenes();
        int[] alleles = new int[genes.length];
        for (int i = 0; i < genes.length; i++) {
            alleles[i] = (int) genes[i].getAllele();
        }
        double fitness = reconstruct.getFitnessScore(pattern, alleles);
        // System.out.println("Fitness score: " + fitness);
        return Math.max(0.0d, fitness);
    }

    public static int getMaxBounds() {
        return MAX_BOUND;
    }
}
```

Code Sample 6.1: The fitness function class for the genetic approach.

We thus have a brute force approach that can incorporate aggregation and a Remainder, but does not have to. We also have a genetic approach that uses aggregation and can use a Remainder, achieved with JGAP and using our integer-based chromosomes to encode the mappings. Both of these rely on packages/namespaces for the modular decomposition step at the moment, but there is no reason why these algorithms could not be applied to modules

derived from a clustering technique. With this realisation sufficiently explained for the purposes of this thesis, let us apply these algorithms to several existing software systems and draw some conclusions for the sake of future improvements.

Chapter 7

Evaluation

Finding architectural patterns using architectural rules and a dependency graph by means of an ACC tool was never intended to be *the* solution to the problem of software architecture reconstruction. Rather, it was always envisioned as another tool on the tool belt, a novel way of doing SAR that could also be used in ACC, since a piece of architecture documentation could be treated as though it were a pattern. Whenever such exploratory design research is performed in an academic context, the question of evaluation has to come up at a point that is often a bit early in the process.

In this case, evaluation would have to be relatively preliminary as well, due to the time constraints of a master thesis. In truth, this project is far from over and the concrete results (pattern definitions in SRMAs, brute force and genetic approaches to pattern discovery that can be used for both ACC and SAR) are only the beginning of what could be a complete method of architecture reconstruction that incorporates dependency-violation techniques together with techniques such as nomenclature (names of packages, classes and methods) and a strong human component (unless artificial intelligence would be incorporated).

As such, it seems too early for any validation process involving professional software architects in large case studies. This is something that would be rather interesting indeed, but would be far more worthwhile once this dependency-based approach has been incorporated in a larger ACC/SAR method. The same goes for extensive verification of the approach, since it makes more sense to do this type of evaluation once a more elaborate analysis can be performed.

Nevertheless, insights can already be gained from some illustrative runs on existing software systems. Even if we were to know for a fact that a certain system contained a particular pattern, we would still face the problem of not knowing whether it was flawlessly implemented. As such, we are better off studying a few architectures that allow us to identify the current limitations of our approach whilst reserving proper validation for a more mature approach in the future.

First off, a rather clear and concise example will be discussed with the Jasper architecture.

7.1 Jasper

Jasper (Yet Another Smart Process EditoR) is a software tool for process editing using petri nets (Van Hee, Oanea, Post, Somers & Van der Werf, 2006). Its source code (C#) contains 16.398 lines of code amongst 5 packages and 55 classes, according to HUSACCT. It is a relatively useful system to test our approach on, since it is quite small. However, as is apparent from Figure 7.1, the four existing packages do not unambiguously suggest a particular architectural pattern and its package hierarchy is rather flat. In fact, none of these packages contains any sub-packages, only classes.

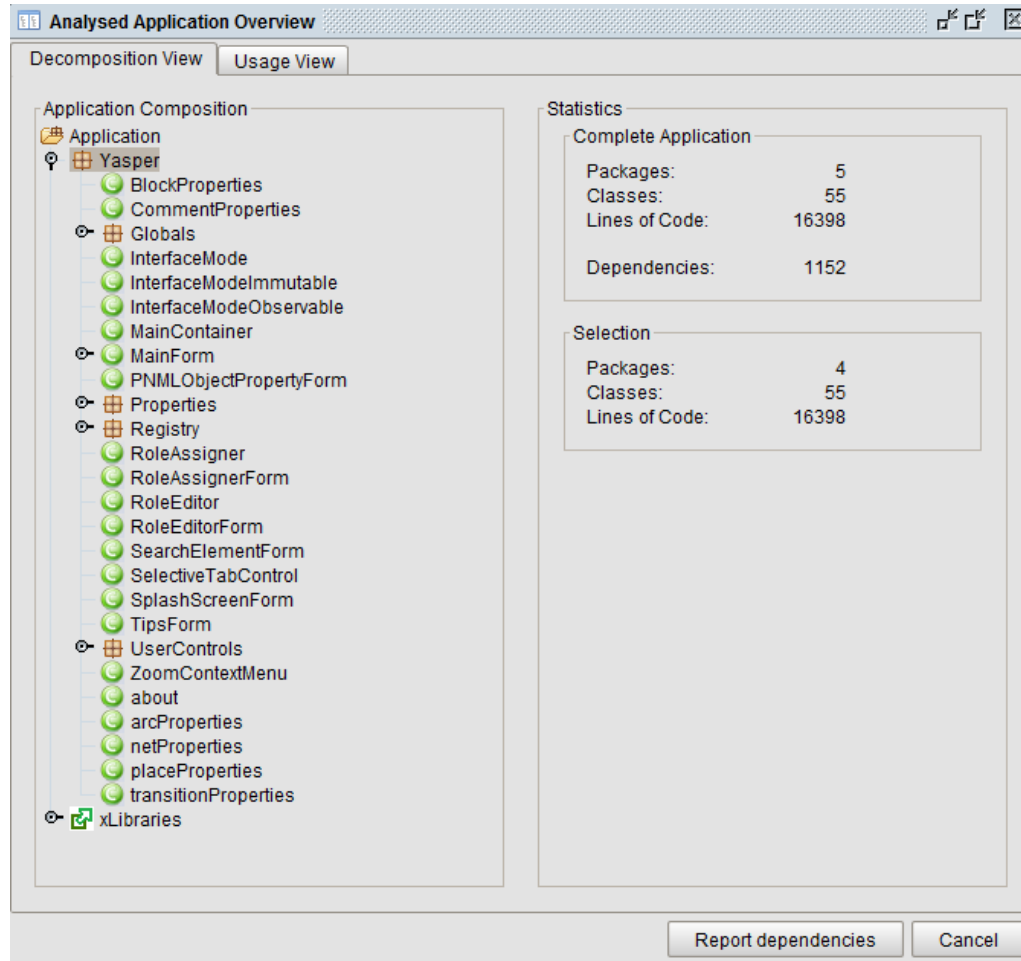


Figure 7.1: The package hierarchy of the Jasper system.

Thinking like someone trying to reconstruct the architecture of this system, the most likely initial hypothesis could be the N-Layered pattern. For the sake of simplicity, let us start with packages only (no individual classes) in a brute force search for the Complete Freedom form of the N-Layered pattern, which is the one defined solely by “Is not allowed to use” rules. It makes little sense to apply a non-aggregation brute force run here, since there are only 390 candidates for aggregation *and* remainder when $n = 4$ and $k = 3$. However, for the sake of demonstration, we will run all five cases (brute force: non-aggregation, aggregation with/without Remainder *and* genetic with/without remainder). Also for the sake of demonstration, let us look at the dependencies between these four packages (Figure 7.2).

Because a 3-Layered solution seems so incredibly obvious for these packages, we expect to see this pattern candidate in all of the following runs. The $N(n = 4, k = 3) = 24$ candidates of the non-aggregation case on the Jasper packages for this pattern take only 0.2 seconds (rough average of several runs) to evaluate¹. Note that the required computation time is not just a

¹Hardware used for these evaluations:

Processor: Intel Core i5-4210U CPU @ 1.70 GHz, 2401 MHz, 2 Cores, 4 Logical Processors

RAM: 8 GB

OS: Microsoft Windows 10 Pro x64

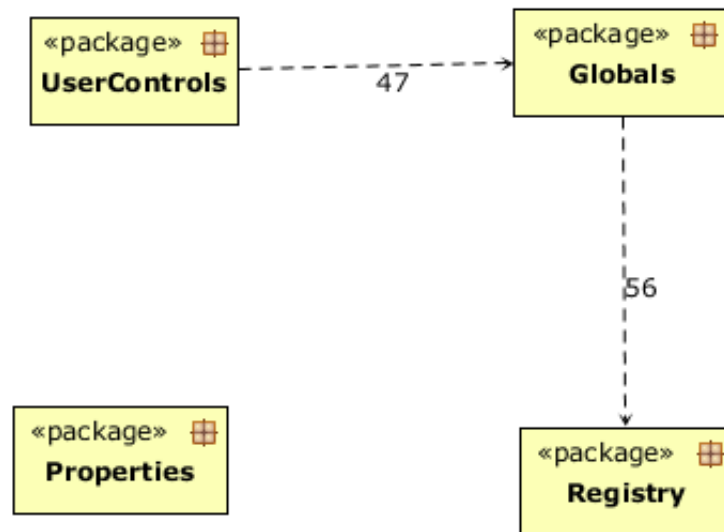


Figure 7.2: Dependencies among the root packages of the Yasper system. If this is to be a 3-Layered system, it is obvious how these packages should be mapped. Due to Properties being an independent package, it will not add anything to the fitness score of any pattern.

function of the number of pattern candidates, but also of the number of dependencies that need to be checked and the number of candidates that can be evaluated quickly due to the “Must use” heuristic. Also, rules that employ exceptions seem to slow HUSACCT down a lot.

As expected, the following results are given, followed by the best candidate being placed in the architecture (Figure 7.3). From left to right, the output shows the mapping of the pattern candidates. *UserControls* is mapped to Layer 1, *Globals* to 2 and *Registry* to 3. Although the algorithm is supposed to print the top-10 candidates, there is only a single pattern candidate that does not violate the “Must use” rules of this pattern and thus receives a non-zero score. None of the other candidates has even a single dependency for each of the three “Must use” rules, as is already perfectly clear from looking at Figure 7.2.

```

Candidate number: 21
Candidate number: 22
Candidate number: 23
Candidate number: 24
INFO [AWT-EventQueue-0] (ReconstructArchitecture.java:189) - Done

Best 10 candidates with non-zero fitness score:
Fitness score: 1.0. Mapping: [Yasper.UserControls, Yasper.Globals, Yasper.Registry]
This last mapping was selected for the intended architecture by default.
Elapsed time: 0.237 seconds.
  
```

Figure 7.3: Eclipse console output of the non-aggregation run on the Yasper packages, looking for the Complete Freedom variety of the 3-Layered pattern.

Doing the same thing for the aggregation case gives us $N^a(n = 4, k = 3) = 36$ pattern candidates. Since this is without a Remainder, all four packages have to be used in each candidate. We thus expect to see three perfect candidates distinguished from each other by the mapping of *Properties*, since we know that package not to add anything in terms of dependencies. The console output confirms our expectations (Figure 7.4).

```
Candidate number: 36
INFO [AWT-EventQueue-0] (ReconstructArchitecture.java:302) - Done

Best 10 candidates with non-zero fitness score:
Fitness: 1.0 --> Mapping: [[Yasper.Properties, Yasper.UserControls], [Yasper.Globals], [Yasper.Registry]]
Fitness: 1.0 --> Mapping: [[Yasper.UserControls], [Yasper.Globals, Yasper.Properties], [Yasper.Registry]]
Fitness: 1.0 --> Mapping: [[Yasper.UserControls], [Yasper.Globals], [Yasper.Properties, Yasper.Registry]]
INFO [AWT-EventQueue-0] (ReconstructArchitecture.java:330) - This last mapping was selected for the intended architecture by default.
Elapsed time: 0.306 seconds.
```

Figure 7.4: Eclipse console output of the aggregation run on the Yasper packages, looking for the Complete Freedom variety of the 3-Layered pattern.

Allowing for a Remainder means the aggregation approach runs twice: once with $k = 3$ and once with $k = 4$. One would thus expect to see the same three candidates as before receiving a perfect score, plus whatever candidates receive a non-zero fitness score in the $k = 4$ case. Because Yasper’s packages are so poorly related to each other, we already know that there should be but a single additional candidate. The output (Figure 7.5) once again confirms our insights. These $N^{a,r}(n = 4, k = 3) = 60$ candidates only took approximately 0.4 seconds to evaluate.

```
Candidate number: 60
INFO [AWT-EventQueue-0] (ReconstructArchitecture.java:302) - Done

Best 10 candidates with non-zero fitness score:
Fitness: 1.0 --> Mapping: [[Yasper.UserControls], [Yasper.Globals], [Yasper.Registry]]
Fitness: 1.0 --> Mapping: [[Yasper.Properties, Yasper.UserControls], [Yasper.Globals], [Yasper.Registry]]
Fitness: 1.0 --> Mapping: [[Yasper.UserControls], [Yasper.Globals, Yasper.Properties], [Yasper.Registry]]
Fitness: 1.0 --> Mapping: [[Yasper.UserControls], [Yasper.Globals], [Yasper.Properties, Yasper.Registry]]
INFO [AWT-EventQueue-0] (ReconstructArchitecture.java:330) - This last mapping was selected for the intended architecture by default.
Elapsed time: 0.39 seconds.
```

Figure 7.5: Eclipse console output of the aggregation + Remainder run on the Yasper packages, looking for the Complete Freedom variety of the 3-Layered pattern.

Finally, let us apply the genetic algorithm to these last two cases, to see if it comes up with the same results. Due to the small number of candidates, there should be no real threat of the algorithm getting caught in a local optimum. Both these runs took just over a second, but this is not very illustrative. Both runs used a population size of 100, evolved over 5 generations, which is more than enough time to evaluate every possible candidate. Hence, this only serves to demonstrate the lack of any irregularities when running the genetic algorithm. We see the same results as before (Figure 7.6 without and Figure 7.7 with Remainder).

```

Begin evolution iteration 4
End evolution iteration 4
Total evolution time: 1392 ms
Presenting the ordering of software units mapped in the chromosomes:

1: Jasper.Globals
2: Jasper.Properties
3: Jasper.Registry
4: Jasper.UserControls

The best (and unique) chromosomes are printed here. If their number is
This would mean that the algorithm has converged on a small number of s

Chromosome 1:
2231
Fitness value: 1.0
Placing best candidate in defined architecture...
Best candidate was successfully mapped and validated.
Chromosome 2:
2331
Fitness value: 1.0
Chromosome 3:
2131
Fitness value: 1.0
Elapsed time: 1.508 seconds.

```

Figure 7.6: Eclipse console output of a genetic run on the Jasper packages, still looking for the Complete Freedom version of the 3-Layered pattern.

Yasper packages & classes

To make this more interesting and more challenging, let us do this again with more software units. This time, we will include the individual root classes together with the four packages, which results in 26 software units. Let us also change the particular version of the N-Layered pattern from Complete Freedom to Isolated Internal Layers, just to see if the results continue to make perfect sense. With 3-Layered (Isolated Internal Layers), Layer 2 should have no dependencies with the Remainder going either way. Figure 7.8 depicts the results of the simple (i.e. no aggregation) brute force approach.

This scenario illustrates precisely when an approach like our genetic one becomes a necessity. The brute force cases with aggregation would require a very large number of candidates ($N^a(n = 26, k = 3) = 2,541,664,501,740$), which is exactly when one ought to switch to the genetic approach instead. Hence, Figure 7.9 shows the Eclipse output for the genetic approach on the 26 software units for the 3-Layered (Isolated Internal Layers) pattern without a Remainder, and Figure 7.10 likewise for the case with a Remainder.

In either genetic case, there are too many solutions with a perfect score of 1.0 to warrant listing them all. Far more important than the exact chromosomes that end up in the final generation is the lesson that can be learnt from this. Namely, that given enough software units (26, in this example), it may not be difficult to find relatively good pattern candidates. That is, with enough software units to distribute amongst the pattern modules, there can be many solutions that do not violate any architectural rules whilst affirming the “Must use” rules. These scenarios might require more sophisticated fitness functions or subsequent filtering in order to make sense of all the results, perhaps eliminating those candidates that, though be they in compliance with the rules, are not very interesting.

Suffice it to say that many good candidates can be found, but their number can grow quite

```
Total evolution time: 1124 ms
Presenting the ordering of software units mapped in the chromosomes:

1: Jasper.Globals
2: Jasper.Properties
3: Jasper.Registry
4: Jasper.UserControls

The best (and unique) chromosomes are printed here. If their number is p
This would mean that the algorithm has converged on a small number of so

Chromosome 1:
2031
Fitness value: 1.0
Placing best candidate in defined architecture...
Best candidate was successfully mapped and validated.
Chromosome 2:
2331
Fitness value: 1.0
Chromosome 3:
2231
Fitness value: 1.0
Chromosome 4:
2131
Fitness value: 1.0
Elapsed time: 1.26 seconds.
```

Figure 7.7: Eclipse console output of a genetic run (including Remainder) on the Jasper packages.

large when dealing with individual classes rather than packages. Essentially, such problems boil down to the first step in Figure 5.1, the modular decomposition of the as-is architecture. We have relied on namespaces for this evaluation, but a namespace hierarchy such as Jasper’s may be exactly the sort of hierarchy when a clustering approach would be a great contributor.

Next, we will look at a system with a slightly more accommodating hierarchy.

```

Candidate number: 15600
INFO [AWT-EventQueue-0] (ReconstructArchitecture.java:189) - Done

Best 100 candidates with non-zero fitness score:
Fitness score: 0.009084027252081723. Mapping: [Yasper.MainContainer, Yasper.MainForm, Yasper.InterfaceMode]
Fitness score: 0.027546047416871364. Mapping: [Yasper.MainForm, Yasper.TipsForm, Yasper.Globals]
Fitness score: 0.03088803088803089. Mapping: [Yasper.MainContainer, Yasper.MainForm, Yasper.TipsForm]
Fitness score: 0.0311284046692607. Mapping: [Yasper.MainContainer, Yasper.MainForm, Yasper.about]
Fitness score: 0.04413657818917428. Mapping: [Yasper.MainContainer, Yasper.MainForm, Yasper.Registry]
Fitness score: 0.04633204633204632. Mapping: [Yasper.MainContainer, Yasper.MainForm,
Yasper.SplashScreenForm]
Fitness score: 0.05363984674329502. Mapping: [Yasper.MainContainer, Yasper.MainForm, Yasper.ZoomContextMenu]
Fitness score: 0.08426733084267338. Mapping: [Yasper.TipsForm, Yasper.Globals, Yasper.Registry]
Fitness score: 0.11513637985275522. Mapping: [Yasper.MainForm, Yasper.UserControls, Yasper.ZoomContextMenu]
Fitness score: 0.13608247422680414. Mapping: [Yasper.PNMLObjectPropertyForm, Yasper.UserControls,
Yasper.ZoomContextMenu]
Fitness score: 0.13967638005581304. Mapping: [Yasper.BlockProperties, Yasper.PNMLObjectPropertyForm,
Yasper.UserControls]
Fitness score: 0.1513761467889908. Mapping: [Yasper.netProperties, Yasper.UserControls,
Yasper.ZoomContextMenu]
Fitness score: 0.1540581650827494. Mapping: [Yasper.netProperties, Yasper.UserControls, Yasper.Globals]
Fitness score: 0.1605647449650495. Mapping: [Yasper.arcProperties, Yasper.PNMLObjectPropertyForm,
Yasper.UserControls]
Fitness score: 0.16930232558139538. Mapping: [Yasper.placeProperties, Yasper.UserControls, Yasper.Globals]
Fitness score: 0.17051636191089495. Mapping: [Yasper.netProperties, Yasper.PNMLObjectPropertyForm,
Yasper.UserControls]
Fitness score: 0.17288177470306904. Mapping: [Yasper.PNMLObjectPropertyForm, Yasper.UserControls,
Yasper.Globals]
Fitness score: 0.17378425080227106. Mapping: [Yasper.placeProperties, Yasper.PNMLObjectPropertyForm,
Yasper.UserControls]
Fitness score: 0.1872463768115942. Mapping: [Yasper.BlockProperties, Yasper.PNMLObjectPropertyForm,
Yasper.SelectiveTabControl]
Fitness score: 0.19290040304330022. Mapping: [Yasper.transitionProperties, Yasper.PNMLObjectPropertyForm,
Yasper.UserControls]
Fitness score: 0.19999999999999998. Mapping: [Yasper.placeProperties, Yasper.UserControls,
Yasper.ZoomContextMenu]
Fitness score: 0.21942716258643605. Mapping: [Yasper.transitionProperties, Yasper.UserControls,
Yasper.Globals]
Fitness score: 0.2279431482971306. Mapping: [Yasper.arcProperties, Yasper.PNMLObjectPropertyForm,
Yasper.SelectiveTabControl]
Fitness score: 0.24419734427291373. Mapping: [Yasper.placeProperties, Yasper.PNMLObjectPropertyForm,
Yasper.SelectiveTabControl]
Fitness score: 0.24473579658323405. Mapping: [Yasper.netProperties, Yasper.PNMLObjectPropertyForm,
Yasper.SelectiveTabControl]
Fitness score: 0.255758538522637. Mapping: [Yasper.transitionProperties, Yasper.PNMLObjectPropertyForm,
Yasper.SelectiveTabControl]
Fitness score: 0.27577674878936764. Mapping: [Yasper.MainForm, Yasper.UserControls, Yasper.Globals]
Fitness score: 0.283430334478648. Mapping: [Yasper.MainContainer, Yasper.MainForm, Yasper.Globals]
Fitness score: 0.31275067412929686. Mapping: [Yasper.UserControls, Yasper.Globals, Yasper.Registry]
Fitness score: 0.3286573146292585. Mapping: [Yasper.transitionProperties, Yasper.UserControls,
Yasper.ZoomContextMenu]
Fitness score: 0.4007592597217713. Mapping: [Yasper.MainContainer, Yasper.MainForm, Yasper.UserControls]
Fitness score: 0.5189200040148549. Mapping: [Yasper.MainForm, Yasper.Globals, Yasper.Registry]
Fitness score: 0.7514450867052023. Mapping: [Yasper.InterfaceModeObservable, Yasper.InterfaceModeImmutable,
Yasper.InterfaceMode]
This last mapping was selected for the intended architecture by default.
Elapsed time: 37.319 seconds.

```

Figure 7.8: Eclipse console output of the non-aggregation run on the Yasper packages and classes, looking for the Isolated Internal Layers variety of the 3-Layered pattern. $N(n = 10, k = 3) = 15,600$ candidates.

```

Total evolution time: 33482 ms
Presenting the ordering of software units mapped in the
chromosomes:

1: Jasper.BlockProperties
2: Jasper.CommentProperties
3: Jasper.Globals
4: Jasper.InterfaceMode
5: Jasper.InterfaceModeImmutable
6: Jasper.InterfaceModeObservable
7: Jasper.MainContainer
8: Jasper.MainForm
9: Jasper.PNMLObjectPropertyForm
10: Jasper.Properties
11: Jasper.Registry
12: Jasper.RoleAssigner
13: Jasper.RoleAssignerForm
14: Jasper.RoleEditor
15: Jasper.RoleEditorForm
16: Jasper.SearchElementForm
17: Jasper.SelectiveTabControl
18: Jasper.SplashScreenForm
19: Jasper.TipsForm
20: Jasper.UserControls
21: Jasper.ZoomContextMenu
22: Jasper.about
23: Jasper.arcProperties
24: Jasper.netProperties
25: Jasper.placeProperties
26: Jasper.transitionProperties

The best (and unique) chromosomes are printed here. If their
number is particularly small, it is because the population
contained many duplicates.
This would mean that the algorithm has converged on a small
number of solutions. These should indicate at least local
optima, if not the global optimum.

Chromosome 1:
1232112222323331232222111
Fitness value: 1.0
Placing best candidate in defined architecture...
Best candidate was successfully mapped and validated.
Chromosome 2:
123211222332333123222211
Fitness value: 1.0
Chromosome 3:
1222112222323331232223211
Fitness value: 1.0
Chromosome 4:
223211222232333123222211
Fitness value: 1.0
Chromosome 5:
122211222232333123222211
Fitness value: 1.0
Chromosome 6:
12321122223233312322221
Fitness value: 1.0
Chromosome 7:
112211222232333123222211
Fitness value: 1.0
. . .
Elapsed time: 33.646 seconds.

```

Figure 7.9: Eclipse console output of genetic run on the Jasper packages and classes, showing only the first 7 out of 62 perfect candidates. Ran for 20 generation and population size 100.

```

Total evolution time: 96125 ms
Presenting the ordering of software units mapped in the
chromosomes:

1: Jasper.BlockProperties
2: Jasper.CommentProperties
3: Jasper.Globals
4: Jasper.InterfaceMode
5: Jasper.InterfaceModeImmutable
6: Jasper.InterfaceModeObservable
7: Jasper.MainContainer
8: Jasper.MainForm
9: Jasper.PNMLObjectPropertyForm
10: Jasper.Properties
11: Jasper.Registry
12: Jasper.RoleAssigner
13: Jasper.RoleAssignerForm
14: Jasper.RoleEditor
15: Jasper.RoleEditorForm
16: Jasper.SearchElementForm
17: Jasper.SelectiveTabControl
18: Jasper.SplashScreenForm
19: Jasper.TipsForm
20: Jasper.UserControls
21: Jasper.ZoomContextMenu
22: Jasper.about
23: Jasper.arcProperties
24: Jasper.netProperties
25: Jasper.placeProperties
26: Jasper.transitionProperties

The best (and unique) chromosomes are printed here. If
their number is particularly small, it is because the
population contained many duplicates.
This would mean that the algorithm has converged on a small
number of solutions. These should indicate at least local
optima, if not the global optimum.

Chromosome 1:
12203030213121102100321001
Fitness value: 1.0
Placing best candidate in defined architecture...
Best candidate was successfully mapped and validated.
Chromosome 2:
20211100213020002122321001
Fitness value: 1.0
Chromosome 3:
10321020223121102122321001
Fitness value: 1.0
Chromosome 4:
11333020203121112222321001
Fitness value: 1.0
Chromosome 5:
11203020223121122122011101
Fitness value: 1.0
Chromosome 6:
12203030233121102102321011
Fitness value: 1.0
Chromosome 7:
11203020223121030202221111
Fitness value: 1.0
. . .
Elapsed time: 96.281 seconds.

```

Figure 7.10: Eclipse console output of genetic run (including Remainder) on the Jasper packages and classes, showing only the first 7 out of 81 perfect candidates. Ran for 20 generation and population size 100.

7.2 aTunes

aTunes is an open-source, cross-platform music player and library manager. Some of its features include automatic tag editing, UI customisation and podcast support². This system is interesting to us due to its hierarchy structure, the fact that it is open-source and as an example of a system where we would expect an MVC-like pattern or style. We rely on the publicly available version 3.0.x branch instead of the latest code, due to the latest possibly still being under development.

The root of aTunes can be seen in Figure 7.11. Although it is not completely apparent from this image, this *package* hierarchy, for it is Java code, is a lot more structured than Jasper's. Both the GUI and Kernel packages contain sub-packages, with all root packages containing decent amounts of individual classes. As such, aTunes is far more structured along the lines of these four packages than Jasper is along its namespaces. This can also be observed from the higher numbers of dependencies between the packages in Figure 7.12.

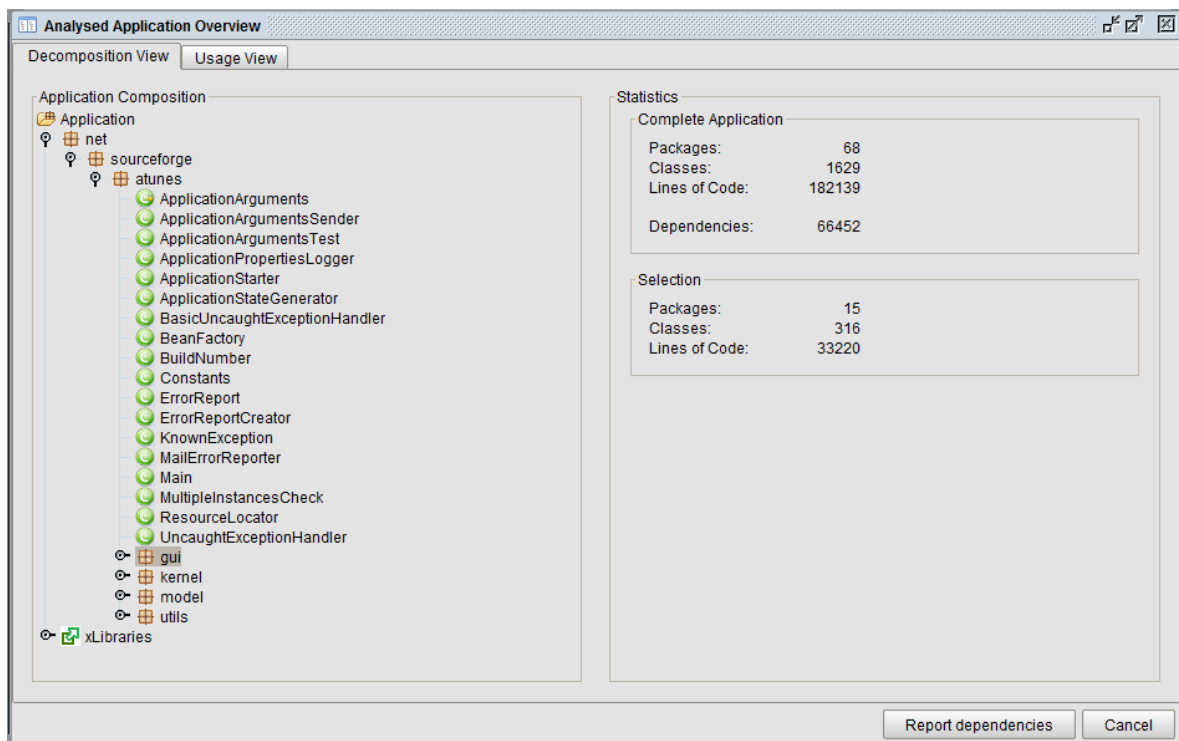


Figure 7.11: The root of the aTunes package hierarchy, as viewed from HUSACCT's Analysed Application Overview.

The names of these four packages very strongly suggest an MVC/MVP type separation. One might suspect an N-Layered style instead, since the same concerns might just as well be separated into layers, but the number of mutual dependencies already suggests that this is unlikely. Since this is a preliminary evaluation that functions partially as a demonstration, in addition to a means by which to discover what our future work should entail, we will simply start looking for MVC/MVP pattern within these root packages and leave it at that.

We would expect the following to be the actual instance of this pattern, given the pattern names: Model = Model, GUI = View and Controller either or both Kernel and Utils.

²The aTunes website: <http://www.atunes.org/>, SourceForge: <http://sourceforge.net/projects/atunes/>, FossHub: <http://www.fosshub.com/aTunes.html>.

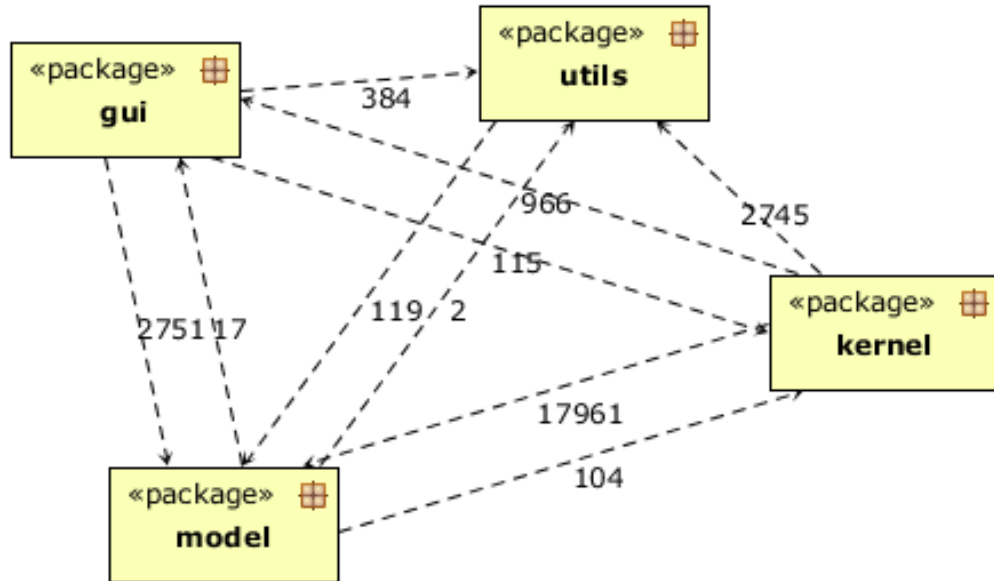


Figure 7.12: The dependencies in the root of the aTunes package hierarchy, as viewed with HUSACCT’s Implemented architecture diagram window and excluding all individual classes.

Let us start by searching for the Complete Freedom interpretation of the MVC-pattern. This is the one where the only rule besides the “Must use” rules says that Model cannot use Controller, thus leaving no constraints with regards to the Remainder. Since we do not expect too many pattern candidates ($N^{a,r}(n = 4, k = 3) = 60$, again), we will simply run the aggregation case including Remainder. Figure 7.13 shows the results. The top 20 candidates were requested.

Although the fitness score varies amongst the top results, this is still a relatively substantial amount of strong candidates. Since the mappings that are produced here show Model, View and Controller in this very order, some of the candidates do not make much sense from an MVC perspective. For example, due to the fact that GUI should so obviously be the View if this be MVC, any candidate where GUI does not take the second position in the mapping seems unlikely. Assuming that some form of MVC (or MVP, of course) was implemented correctly, can we apply a particular interpretation of the MVC pattern such that these more reasonable mappings come out on top?

Figure 7.14 shows the same results for the Model Interface interpretation of MVC, defined by Figure B.4 in Section 4.3. This interpretation does have consequences for the Remainder. It takes much longer to search for this pattern interpretation, because it uses rule exceptions in addition to regular rules.

Since a case can also be made for the Controller being the interface with the Remainder for the MVC pattern, let us run the same analysis for the MVC interpretation that was displayed in Figure 4.9 in Section 4.3. The results are shown in Figure 7.15

What can one conclude from these evaluations using aTunes? There are not very many legitimate pattern candidates for these three interpretations of MVC when run on the four root packages of aTunes. It turns out that one can find several candidates with a high fitness score

```

Candidate number: 60
INFO [AWT-EventQueue-0] (ReconstructArchitecture.java:302) - Done

Best 20 candidates with non-zero fitness score:
Fitness: 0.7179222303382978 --> Mapping: [[net.sourceforge.atunes.model], [net.sourceforge.atunes.gui,
net.sourceforge.atunes.utils], [net.sourceforge.atunes.kernel]]
Fitness: 0.7227803179449632 --> Mapping: [[net.sourceforge.atunes.utils], [net.sourceforge.atunes.gui],
[net.sourceforge.atunes.kernel, net.sourceforge.atunes.model]]
Fitness: 0.7706768944381706 --> Mapping: [[net.sourceforge.atunes.gui, net.sourceforge.atunes.model],
[net.sourceforge.atunes.utils], [net.sourceforge.atunes.kernel]]
Fitness: 0.8508080174113313 --> Mapping: [[net.sourceforge.atunes.gui], [net.sourceforge.atunes.model],
[net.sourceforge.atunes.kernel, net.sourceforge.atunes.utils]]
Fitness: 0.8704051515964583 --> Mapping: [[net.sourceforge.atunes.utils], [net.sourceforge.atunes.kernel],
[net.sourceforge.atunes.gui]]
Fitness: 0.8952702183695589 --> Mapping: [[net.sourceforge.atunes.model], [net.sourceforge.atunes.gui],
[net.sourceforge.atunes.kernel, net.sourceforge.atunes.utils]]
Fitness: 0.89569112434638 --> Mapping: [[net.sourceforge.atunes.model], [net.sourceforge.atunes.utils],
[net.sourceforge.atunes.kernel]]
Fitness: 0.9028601325427276 --> Mapping: [[net.sourceforge.atunes.utils], [net.sourceforge.atunes.model],
[net.sourceforge.atunes.kernel]]
Fitness: 0.9103082055893321 --> Mapping: [[net.sourceforge.atunes.gui], [net.sourceforge.atunes.model],
[net.sourceforge.atunes.kernel]]
Fitness: 0.9131813861249999 --> Mapping: [[net.sourceforge.atunes.gui], [net.sourceforge.atunes.model,
net.sourceforge.atunes.utils], [net.sourceforge.atunes.kernel]]
Fitness: 0.9140674040519081 --> Mapping: [[net.sourceforge.atunes.model], [net.sourceforge.atunes.kernel],
[net.sourceforge.atunes.gui, net.sourceforge.atunes.utils]]
Fitness: 0.9158521686010996 --> Mapping: [[net.sourceforge.atunes.utils], [net.sourceforge.atunes.kernel,
net.sourceforge.atunes.model], [net.sourceforge.atunes.gui]]
Fitness: 0.9169778547242682 --> Mapping: [[net.sourceforge.atunes.gui, net.sourceforge.atunes.utils],
[net.sourceforge.atunes.model], [net.sourceforge.atunes.kernel]]
Fitness: 0.9670807095868765 --> Mapping: [[net.sourceforge.atunes.model], [net.sourceforge.atunes.kernel],
[net.sourceforge.atunes.gui]]
Fitness: 0.9676265719494782 --> Mapping: [[net.sourceforge.atunes.model], [net.sourceforge.atunes.kernel,
net.sourceforge.atunes.utils], [net.sourceforge.atunes.gui]]
Fitness: 0.9702790409799104 --> Mapping: [[net.sourceforge.atunes.model, net.sourceforge.atunes.utils],
[net.sourceforge.atunes.kernel], [net.sourceforge.atunes.gui]]
Fitness: 0.978783151326053 --> Mapping: [[net.sourceforge.atunes.utils], [net.sourceforge.atunes.model],
[net.sourceforge.atunes.gui]]
Fitness: 0.9861529199277543 --> Mapping: [[net.sourceforge.atunes.utils], [net.sourceforge.atunes.gui],
[net.sourceforge.atunes.kernel]]
Fitness: 0.9923966347235345 --> Mapping: [[net.sourceforge.atunes.utils], [net.sourceforge.atunes.gui,
net.sourceforge.atunes.model], [net.sourceforge.atunes.kernel]]
Fitness: 0.9949922797646371 --> Mapping: [[net.sourceforge.atunes.utils], [net.sourceforge.atunes.model],
[net.sourceforge.atunes.gui, net.sourceforge.atunes.kernel]]
INFO [AWT-EventQueue-0] (ReconstructArchitecture.java:330) - This last mapping was selected for the intended
architecture by default.
Elapsed time: 18.109 seconds.

```

Figure 7.13: Eclipse output of aTunes aggregation + Remainder brute force for Complete Freedom MVC.

in these cases and that one MVC interpretation does not help us much more than the others. There are additional MVC/MVP interpretations left for us to attempt, with even more versions still conceivable, but the lesson we can draw from the analysis of aTunes is clear. If one is so sure that GUI should be assigned to View, or that Model undoubtedly is the MVC Model, then this information needs to be fed to the algorithm. Whether it be based on the package names alone, or on an analysis of syntax and/or semantics of these packages, this would aid a great deal. It would whittle down both the number of candidates to be considered and the number of top candidates presented to the user.

Next, we will have a look at architectures with higher numbers of packages in the root.

```

Candidate number: 60
INFO [AWT-EventQueue-0] (ReconstructArchitecture.java:302) - Done

Best 30 candidates with non-zero fitness score:
Fitness: 0.21377152716223544 --> Mapping: [[net.sourceforge.atunes.kernel, net.sourceforge.atunes.utils],
[net.sourceforge.atunes.model], [net.sourceforge.atunes.gui]]
Fitness: 0.21926006528835693 --> Mapping: [[net.sourceforge.atunes.utils], [net.sourceforge.atunes.gui],
[net.sourceforge.atunes.model]]
Fitness: 0.23793671117614779 --> Mapping: [[net.sourceforge.atunes.utils], [net.sourceforge.atunes.gui,
net.sourceforge.atunes.kernel], [net.sourceforge.atunes.model]]
Fitness: 0.26023459954944456 --> Mapping: [[net.sourceforge.atunes.utils], [net.sourceforge.atunes.kernel],
[net.sourceforge.atunes.gui, net.sourceforge.atunes.model]]
Fitness: 0.26798070874397145 --> Mapping: [[net.sourceforge.atunes.gui, net.sourceforge.atunes.utils],
[net.sourceforge.atunes.kernel], [net.sourceforge.atunes.model]]
Fitness: 0.6436759648452426 --> Mapping: [[net.sourceforge.atunes.kernel, net.sourceforge.atunes.model],
[net.sourceforge.atunes.utils], [net.sourceforge.atunes.gui]]
Fitness: 0.7337911352893961 --> Mapping: [[net.sourceforge.atunes.utils], [net.sourceforge.atunes.gui],
[net.sourceforge.atunes.kernel, net.sourceforge.atunes.model]]
Fitness: 0.8704051515964583 --> Mapping: [[net.sourceforge.atunes.utils], [net.sourceforge.atunes.kernel],
[net.sourceforge.atunes.gui]]
Fitness: 0.9028601325427276 --> Mapping: [[net.sourceforge.atunes.utils], [net.sourceforge.atunes.model],
[net.sourceforge.atunes.kernel]]
Fitness: 0.9056513514101198 --> Mapping: [[net.sourceforge.atunes.model], [net.sourceforge.atunes.utils],
[net.sourceforge.atunes.kernel]]
Fitness: 0.9158521686010996 --> Mapping: [[net.sourceforge.atunes.utils], [net.sourceforge.atunes.kernel,
net.sourceforge.atunes.model], [net.sourceforge.atunes.gui]]
Fitness: 0.9161527358614355 --> Mapping: [[net.sourceforge.atunes.model], [net.sourceforge.atunes.kernel],
[net.sourceforge.atunes.gui, net.sourceforge.atunes.utils]]
Fitness: 0.9190974398534304 --> Mapping: [[net.sourceforge.atunes.gui], [net.sourceforge.atunes.model],
[net.sourceforge.atunes.kernel, net.sourceforge.atunes.utils]]
Fitness: 0.9273092172891478 --> Mapping: [[net.sourceforge.atunes.gui], [net.sourceforge.atunes.model],
[net.sourceforge.atunes.kernel]]
Fitness: 0.9282290507575109 --> Mapping: [[net.sourceforge.atunes.gui], [net.sourceforge.atunes.model,
net.sourceforge.atunes.utils], [net.sourceforge.atunes.kernel]]
Fitness: 0.9335284370897979 --> Mapping: [[net.sourceforge.atunes.gui, net.sourceforge.atunes.utils],
[net.sourceforge.atunes.model], [net.sourceforge.atunes.kernel]]
Fitness: 0.9745677452563113 --> Mapping: [[net.sourceforge.atunes.model], [net.sourceforge.atunes.kernel],
[net.sourceforge.atunes.gui]]
Fitness: 0.9751080024686278 --> Mapping: [[net.sourceforge.atunes.model], [net.sourceforge.atunes.kernel,
net.sourceforge.atunes.utils], [net.sourceforge.atunes.gui]]
Fitness: 0.9778158737933428 --> Mapping: [[net.sourceforge.atunes.model, net.sourceforge.atunes.utils],
[net.sourceforge.atunes.kernel], [net.sourceforge.atunes.gui]]
Fitness: 0.978783151326053 --> Mapping: [[net.sourceforge.atunes.utils], [net.sourceforge.atunes.model],
[net.sourceforge.atunes.gui]]
Fitness: 0.9859326940474036 --> Mapping: [[net.sourceforge.atunes.model], [net.sourceforge.atunes.gui],
[net.sourceforge.atunes.kernel, net.sourceforge.atunes.utils]]
Fitness: 0.9861529199277543 --> Mapping: [[net.sourceforge.atunes.utils], [net.sourceforge.atunes.gui],
[net.sourceforge.atunes.kernel]]
Fitness: 0.9863081901916854 --> Mapping: [[net.sourceforge.atunes.gui, net.sourceforge.atunes.model],
[net.sourceforge.atunes.utils], [net.sourceforge.atunes.kernel]]
Fitness: 0.9923966347235345 --> Mapping: [[net.sourceforge.atunes.utils], [net.sourceforge.atunes.gui,
net.sourceforge.atunes.model], [net.sourceforge.atunes.kernel]]
Fitness: 0.994586162598642 --> Mapping: [[net.sourceforge.atunes.model], [net.sourceforge.atunes.gui],
[net.sourceforge.atunes.kernel]]
Fitness: 0.9949922797646371 --> Mapping: [[net.sourceforge.atunes.utils], [net.sourceforge.atunes.model],
[net.sourceforge.atunes.gui, net.sourceforge.atunes.kernel]]
Fitness: 0.9951745671302866 --> Mapping: [[net.sourceforge.atunes.model], [net.sourceforge.atunes.gui,
net.sourceforge.atunes.utils], [net.sourceforge.atunes.kernel]]
Fitness: 0.9952657973921765 --> Mapping: [[net.sourceforge.atunes.model, net.sourceforge.atunes.utils],
[net.sourceforge.atunes.gui], [net.sourceforge.atunes.kernel]]
Fitness: 0.9970890148613452 --> Mapping: [[net.sourceforge.atunes.model], [net.sourceforge.atunes.utils],
[net.sourceforge.atunes.gui]]
Fitness: 0.9974397935183065 --> Mapping: [[net.sourceforge.atunes.model], [net.sourceforge.atunes.utils],
[net.sourceforge.atunes.gui, net.sourceforge.atunes.kernel]]
INFO [AWT-EventQueue-0] (ReconstructArchitecture.java:330) - This last mapping was selected for the intended
architecture by default.
Elapsed time: 140.152 seconds.

```

Figure 7.14: Eclipse output of aTunes aggregation + Remainder brute force for Model Interface MVC.

```

Candidate number: 60
INFO [AWT-EventQueue-0] (ReconstructArchitecture.java:302) - Done

Best 30 candidates with non-zero fitness score:
Fitness: 0.23793671117614779 --> Mapping: [[net.sourceforge.atunes.utils], [net.sourceforge.atunes.gui,
net.sourceforge.atunes.kernel], [net.sourceforge.atunes.model]]
Fitness: 0.23870760327921553 --> Mapping: [[net.sourceforge.atunes.kernel], [net.sourceforge.atunes.model],
[net.sourceforge.atunes.gui]]
Fitness: 0.26023459954944456 --> Mapping: [[net.sourceforge.atunes.utils], [net.sourceforge.atunes.kernel],
[net.sourceforge.atunes.gui, net.sourceforge.atunes.model]]
Fitness: 0.2604073714839961 --> Mapping: [[net.sourceforge.atunes.kernel, net.sourceforge.atunes.utils],
[net.sourceforge.atunes.model], [net.sourceforge.atunes.gui]]
Fitness: 0.26798070874397145 --> Mapping: [[net.sourceforge.atunes.gui, net.sourceforge.atunes.utils],
[net.sourceforge.atunes.kernel], [net.sourceforge.atunes.model]]
Fitness: 0.6436759648452426 --> Mapping: [[net.sourceforge.atunes.kernel, net.sourceforge.atunes.model],
[net.sourceforge.atunes.utils], [net.sourceforge.atunes.gui]]
Fitness: 0.7337911352893961 --> Mapping: [[net.sourceforge.atunes.utils], [net.sourceforge.atunes.gui],
[net.sourceforge.atunes.kernel, net.sourceforge.atunes.model]]
Fitness: 0.8704051515964583 --> Mapping: [[net.sourceforge.atunes.utils], [net.sourceforge.atunes.kernel],
[net.sourceforge.atunes.gui]]
Fitness: 0.9028601325427276 --> Mapping: [[net.sourceforge.atunes.utils], [net.sourceforge.atunes.model],
[net.sourceforge.atunes.kernel]]
Fitness: 0.9056513514101198 --> Mapping: [[net.sourceforge.atunes.model], [net.sourceforge.atunes.utils],
[net.sourceforge.atunes.kernel]]
Fitness: 0.9158521686010996 --> Mapping: [[net.sourceforge.atunes.utils], [net.sourceforge.atunes.kernel,
net.sourceforge.atunes.model], [net.sourceforge.atunes.gui]]
Fitness: 0.9161527358614355 --> Mapping: [[net.sourceforge.atunes.model], [net.sourceforge.atunes.kernel],
[net.sourceforge.atunes.gui, net.sourceforge.atunes.utils]]
Fitness: 0.9190974398534304 --> Mapping: [[net.sourceforge.atunes.gui], [net.sourceforge.atunes.model],
[net.sourceforge.atunes.kernel, net.sourceforge.atunes.utils]]
Fitness: 0.9273092172891478 --> Mapping: [[net.sourceforge.atunes.gui], [net.sourceforge.atunes.model],
[net.sourceforge.atunes.kernel]]
Fitness: 0.9282290507575109 --> Mapping: [[net.sourceforge.atunes.gui], [net.sourceforge.atunes.model,
net.sourceforge.atunes.utils], [net.sourceforge.atunes.kernel]]
Fitness: 0.9335284370897979 --> Mapping: [[net.sourceforge.atunes.gui, net.sourceforge.atunes.utils],
[net.sourceforge.atunes.model], [net.sourceforge.atunes.kernel]]
Fitness: 0.9745677452563113 --> Mapping: [[net.sourceforge.atunes.model], [net.sourceforge.atunes.kernel],
[net.sourceforge.atunes.gui]]
Fitness: 0.9751080024686278 --> Mapping: [[net.sourceforge.atunes.model], [net.sourceforge.atunes.kernel,
net.sourceforge.atunes.utils], [net.sourceforge.atunes.gui]]
Fitness: 0.9778158737933428 --> Mapping: [[net.sourceforge.atunes.model, net.sourceforge.atunes.utils],
[net.sourceforge.atunes.kernel], [net.sourceforge.atunes.gui]]
Fitness: 0.978783151326053 --> Mapping: [[net.sourceforge.atunes.utils], [net.sourceforge.atunes.model],
[net.sourceforge.atunes.gui]]
Fitness: 0.9859326940474036 --> Mapping: [[net.sourceforge.atunes.model], [net.sourceforge.atunes.gui],
[net.sourceforge.atunes.kernel, net.sourceforge.atunes.utils]]
Fitness: 0.9861529199277543 --> Mapping: [[net.sourceforge.atunes.utils], [net.sourceforge.atunes.gui],
[net.sourceforge.atunes.kernel]]
Fitness: 0.9863081901916854 --> Mapping: [[net.sourceforge.atunes.gui, net.sourceforge.atunes.model],
[net.sourceforge.atunes.utils], [net.sourceforge.atunes.kernel]]
Fitness: 0.9923966347235345 --> Mapping: [[net.sourceforge.atunes.utils], [net.sourceforge.atunes.gui,
net.sourceforge.atunes.model], [net.sourceforge.atunes.kernel]]
Fitness: 0.994586162598642 --> Mapping: [[net.sourceforge.atunes.model], [net.sourceforge.atunes.gui],
[net.sourceforge.atunes.kernel]]
Fitness: 0.9949922797646371 --> Mapping: [[net.sourceforge.atunes.utils], [net.sourceforge.atunes.model],
[net.sourceforge.atunes.gui, net.sourceforge.atunes.kernel]]
Fitness: 0.9951745671302866 --> Mapping: [[net.sourceforge.atunes.model], [net.sourceforge.atunes.gui,
net.sourceforge.atunes.utils], [net.sourceforge.atunes.kernel]]
Fitness: 0.9952657973921765 --> Mapping: [[net.sourceforge.atunes.model, net.sourceforge.atunes.utils],
[net.sourceforge.atunes.gui], [net.sourceforge.atunes.kernel]]
Fitness: 0.9970890148613452 --> Mapping: [[net.sourceforge.atunes.model], [net.sourceforge.atunes.utils],
[net.sourceforge.atunes.gui]]
Fitness: 0.9974397935183065 --> Mapping: [[net.sourceforge.atunes.model], [net.sourceforge.atunes.utils],
[net.sourceforge.atunes.gui, net.sourceforge.atunes.kernel]]
INFO [AWT-EventQueue-0] (ReconstructArchitecture.java:330) - This last mapping was selected for the intended
architecture by default.
Elapsed time: 116.204 seconds.

```

Figure 7.15: Eclipse output of aTunes aggregation + Remainder brute force for Controller Interface MVC.

7.3 SweetHome3D

SweetHome3D is a Java application that allows its users to draw plans for houses for Windows, Mac OS, Linux and Solaris. It lets people apply their interior designs to such plans and view the results in three dimensions, furniture and all³. It is an open-source project that has had some time to mature (over ten years). We used the source code freely available on the download page of SweetHome3D's website, which is version 5.1.

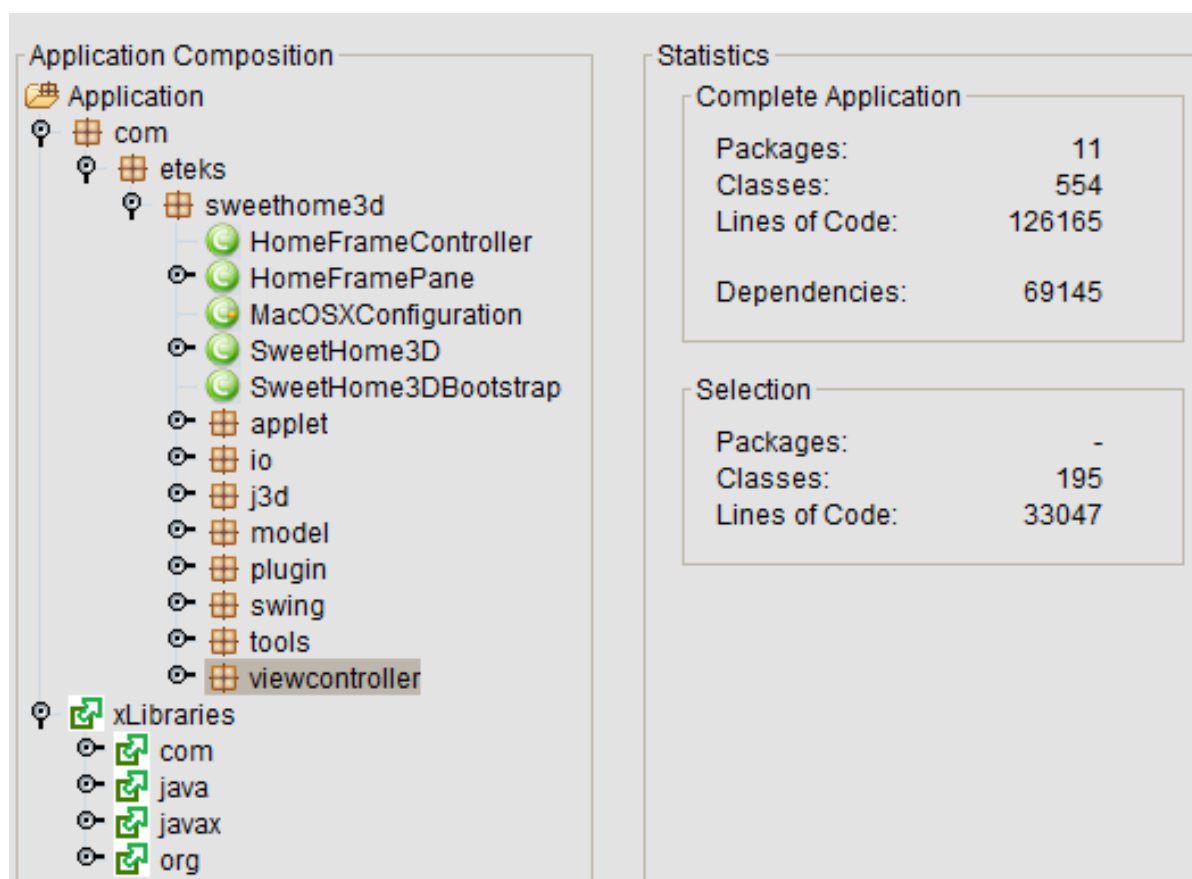


Figure 7.16: The SweetHome3D package hierarchy root.

This architecture is neither very flat nor very vertical. Figure 7.16 shows eight root packages, none of which has any sub-packages, although they all have plenty of individual classes. The developer's forum describes a combination of 3-Layered with MVC (http://www.sweethome3d.com/support/forum/viewthread_thread,3067). This is difficult to verify by looking at the package dependencies in Figure 7.17.

³SweetHome3D website: <http://www.sweethome3d.com/>, SourceForge: <http://sourceforge.net/projects/sweethome3d/>, Developer's forum: <http://www.sweethome3d.com/support/forum/>.

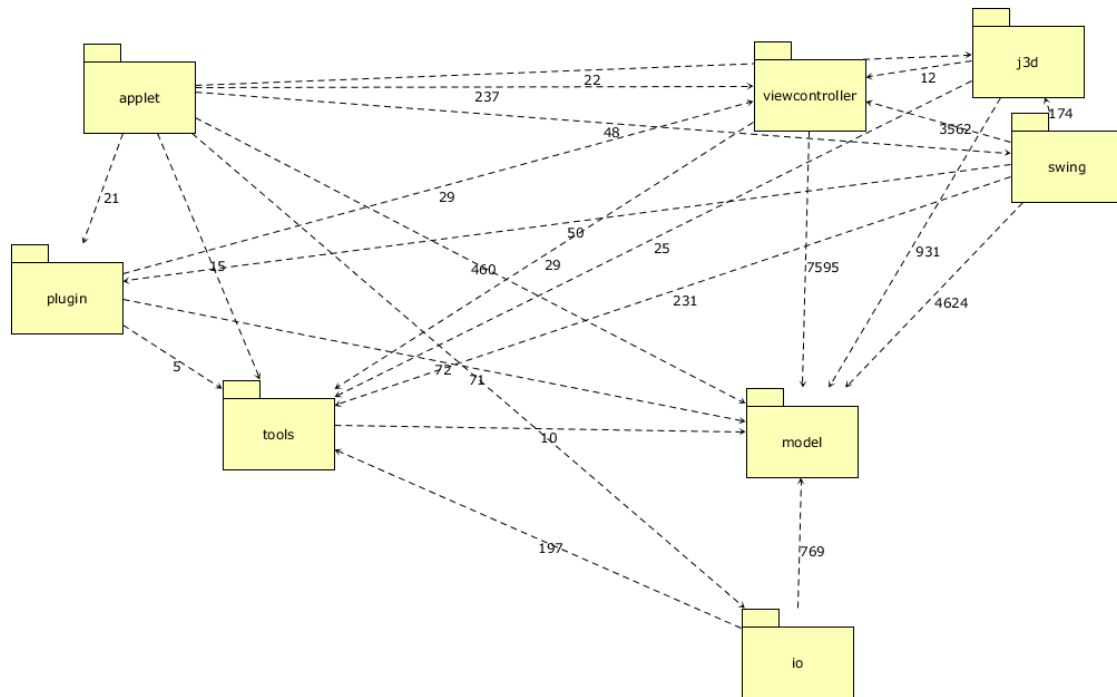


Figure 7.17: The dependencies between SweetHome3D's packages.

However, there are several JUnit tests within the source code that check the package dependencies. Since we actually have the source code, we can import it into Eclipse and run these JUnit tests (Code Sample D.7). There were no assertion errors. From this, we can get a good idea of the intended architecture.

From these sources, several things become rather apparent. There is a particular kind of MVC pattern here, an interpretation derived from classic MVC with View and Controller merged into one. Model cannot, and does not, depend on this Viewcontroller.

In terms of layering, there is definitely a presentation layer and a business logic layer, judging by both the dependencies and the forum description. However, the persistence layer that is supposedly embodied by the IO package appears to completely contradict the idea of the N-Layered pattern. If we assume this is the third layer, then it does not even adhere to the rule that “Layer 2 must use Layer 3”, which is part of the reason why the intended pattern does not appear as a candidate in the Complete Freedom 3-Layered (genetic with Remainder, 20 generations) search of Figure 7.18. If we would take the persistence layer to be Layer 2 or even 1, there would still be no strict layering among these three layers, since the whole problem is that what is called the business logic layer receives dependencies from both other layers and does itself not depend on anything.

```

End evolution iteration 29
Total evolution time: 426341 ms
Presenting the ordering of software units mapped in the chromosomes:

1: com.eteks.sweethome3d.applet
2: com.eteks.sweethome3d.io
3: com.eteks.sweethome3d.j3d
4: com.eteks.sweethome3d.model
5: com.eteks.sweethome3d.plugin
6: com.eteks.sweethome3d.swing
7: com.eteks.sweethome3d.tools
8: com.eteks.sweethome3d.viewcontroller

The best (and unique) chromosomes are printed here. If their number is
This would mean that the algorithm has converged on a small number of s

Chromosome 1:
02231022
Fitness value: 0.9160348800369358
Placing best candidate in defined architecture...
Best candidate was successfully mapped and validated.
Elapsed time: 426.492 seconds.

```

Figure 7.18: Eclipse output for Complete Freedom 3-Layered with SweetHome3D's root packages.

From this evaluation thus far, we can already draw the conclusion that more patterns need to be defined if one is to use an approach similar to ours in the field. But this insight is rather trivial and obvious, so let us see if we can learn a bit more from this system. Code Sample 7.1 shows a pattern class for this particular interpretation of the 3-Layered pattern, where the central layer receives dependencies from layers 1 and 2. This is also depicted in Figure 7.19. Because the business logic layer is only supposed to have Model in it and because Viewcontroller is one module, we could treat the Layering-MVC combination as a single pattern. However, if we define this Model-Viewcontroller as a pattern in and of itself, it would look like Code Sample 7.2 and Figure 7.20.

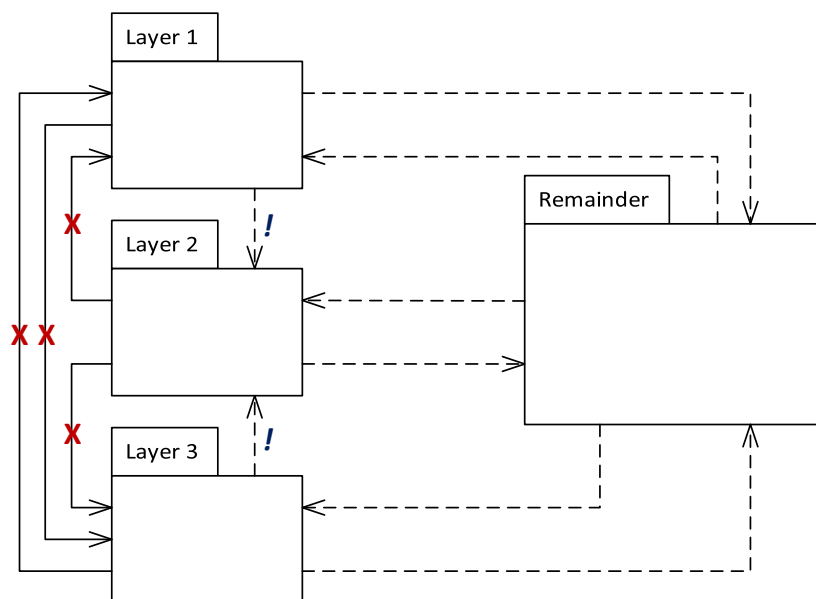


Figure 7.19: The layering as seems to be employed by SweetHome3D.

```

package husacct.analyse.task.reconstruct.patterns;

public class CentralisedLayering extends LayeredPattern_CompleteFreedom {

    public CentralisedLayering() {
        numberOfModules = 3;
        name = "Centralised Layering";
    }

    @Override
    protected void defineRules() {
        addSingleRule("Layer" + 1, "Layer" + 2, "IsNotAllowedToUse", null);
        addSingleRule("Layer" + 3, "Layer" + 2, "IsNotAllowedToUse", null);
        addSingleRule("Layer" + 3, "Layer" + 1, "IsNotAllowedToUse", null);
        addSingleRule("Layer" + 1, "Layer" + 3, "IsNotAllowedToUse", null);
    }

    @Override
    protected void defineMustUseRules() {
        addSingleRule("Layer" + 2, "Layer" + 1, "MustUse", null);
        addSingleRule("Layer" + 2, "Layer" + 3, "MustUse", null);
    }
}

```

Code Sample 7.1: A pattern that corresponds to SweetHome3D's layering structure.

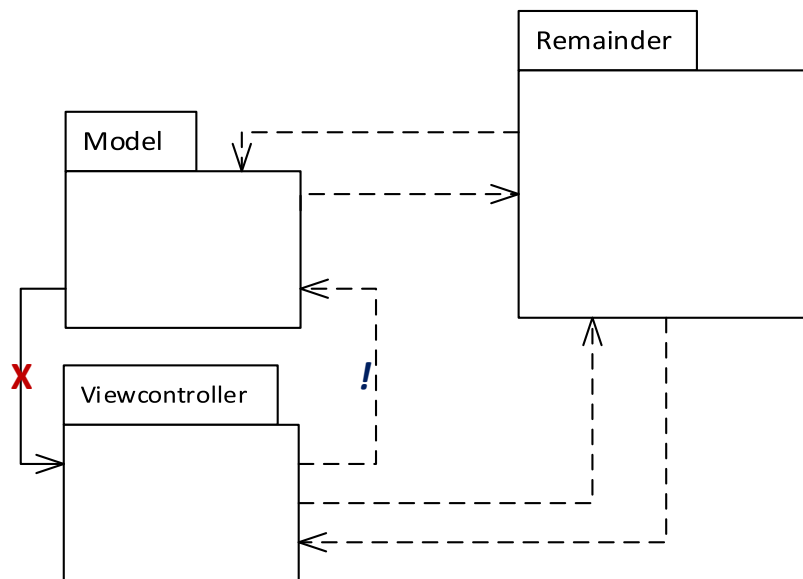


Figure 7.20: An interpretation of SweetHome3D's Model-Viewcontroller pattern.

```

package husacct.analyse.task.reconstruct.patterns;

import java.util.ArrayList;
import java.util.Map;

import husacct.common.dto.SoftwareUnitDTO;

```



```

public class Model_Viewcontroller extends Pattern {

    public Model_Viewcontroller() {
        numberOfModules = 2;
        name = "M-VC";
    }

    @Override
    protected void defineModules() {
        defineService.addModule("Model", "**", "Subsystem", 1, null);
        defineService.addModule("Viewcontroller", "**", "Subsystem", 1, null);
    }

    @Override
    protected void defineRules() {
        addSingleRule("Viewcontroller", "Model", "IsNotAllowedToUse", null);
    }

    @Override
    protected void defineMustUseRules() {
        addSingleRule("Model", "Viewcontroller", "MustUse", null);
    }

    @Override
    public void mapPattern(ArrayList<String> mapping) {
        ArrayList<SoftwareUnitDTO> temp = new ArrayList<>(1);
        temp.add(analyseService.getSoftwareUnitByUniqueName(mapping.get(0)));
        defineService.editModule("Model", "Model", 1, temp);
        temp.clear();
        temp.add(analyseService.getSoftwareUnitByUniqueName(mapping.get(1)));
        defineService.editModule("Viewcontroller", "Viewcontroller", 1, temp);
        temp.clear();
    }

    @Override
    public void mapPatternAllowingAggregates(Map<Integer, ArrayList<String>>
        patternUnitNames) {
        ArrayList<SoftwareUnitDTO> temp = new ArrayList<>();
        for (int j = 0; j < patternUnitNames.get(0).size(); j++) {
            temp.add(analyseService.getSoftwareUnitByUniqueName(patternUnitNames.get(0).get(j)));
        }
        defineService.editModuleWithAggregation("Model", "Model", 1, temp);
        temp.clear();
        for (int j = 0; j < patternUnitNames.get(1).size(); j++) {
            temp.add(analyseService.getSoftwareUnitByUniqueName(patternUnitNames.get(1).get(j)));
        }
        defineService.editModuleWithAggregation("Viewcontroller", "Viewcontroller",
            1, temp);
        temp.clear();
    }
}

```

Code Sample 7.2: A Complete Freedom interpretation of a Model-Viewcontroller pattern.

One could phrase SweetHome3D's architecture as follows: the style is a 3-Layered architecture in which the middle layer is called by both of the other layers, with a Model-Viewcontroller pattern (where Model does not have any outgoing dependencies) distributed over the business

```

End evolution iteration 29
Total evolution time: 109273 ms
Presenting the ordering of software units mapped in the
chromosomes:

1: com.eteks.sweethome3d.applet
2: com.eteks.sweethome3d.io
3: com.eteks.sweethome3d.j3d
4: com.eteks.sweethome3d.model
5: com.eteks.sweethome3d.plugin
6: com.eteks.sweethome3d.swing
7: com.eteks.sweethome3d.tools
8: com.eteks.sweethome3d.viewcontroller

The best (and unique) chromosomes are printed here. If their
number is particularly small, it is because the population
contained many duplicates.
This would mean that the algorithm has converged on a small
number of solutions. These should indicate at least local optima,
if not the global optimum.

Chromosome 1:
21212210
Fitness value: 1.0
Placing best candidate in defined architecture...
Best candidate was successfully mapped and validated.
Chromosome 2:
20010201
Fitness value: 1.0
Chromosome 3:
20001010
Fitness value: 1.0
Chromosome 4:
20110010
Fitness value: 1.0
Chromosome 5:
21200010
Fitness value: 1.0
Chromosome 6:
21211000
Fitness value: 1.0
. . .

Chromosome 69:
21000200
Fitness value: 1.0
Elapsed time: 109.399 seconds.

```

Figure 7.21: A genetic (with Remainder) run on SweetHome3D’s packages for the Model-Viewcontroller pattern, showing only a portion of the results.

logic and presentation layer, respectively. The first of these is not very difficult to find thanks to the relatively large numbers of dependencies along the lines of the “Must use” rules for both of these. A high number of “Must use” affirmations results in good fitness scores, because it explains all these dependencies with the existence of a pattern. The smaller pattern is much harder to find, due to the large number of candidates that are equally good. Without semantic input, it is very difficult to discover the pattern. This example raises an issue concerning pattern-based architectures that is further developed in Section 8.5.

Note also how Figure 7.22 depicts a solution similar to the one we anticipated (except for the placement of Tools), but upside down in the sense that Layer 1 is Layer 3 and vice versa. Because of the way the pattern is defined, this does indeed seem just as reasonable. Without semantic constraints, this sort of confusion is to be expected.

Finally, we will analyse an early version of HUSACCT itself.

```
End evolution iteration 29
Total evolution time: 424834 ms
Presenting the ordering of software units mapped in the
chromosomes:

1: com.eteeks.sweethome3d.applet
2: com.eteeks.sweethome3d.io
3: com.eteeks.sweethome3d.j3d
4: com.eteeks.sweethome3d.model
5: com.eteeks.sweethome3d.plugin
6: com.eteeks.sweethome3d.swing
7: com.eteeks.sweethome3d.tools
8: com.eteeks.sweethome3d.viewcontroller

The best (and unique) chromosomes are printed here. If their
number is particularly small, it is because the population
contained many duplicates.
This would mean that the algorithm has converged on a small
number of solutions. These should indicate at least local
optima, if not the global optimum.

Chromosome 1:
01320323
Fitness value: 0.971235879278368
Placing best candidate in defined architecture...
Best candidate was successfully mapped and validated.
Elapsed time: 424.991 seconds.
```

Figure 7.22: A genetic (with Remainder) run on *SweetHome3D*'s packages for the *Centralised Layering* pattern.

7.4 HUSACCT

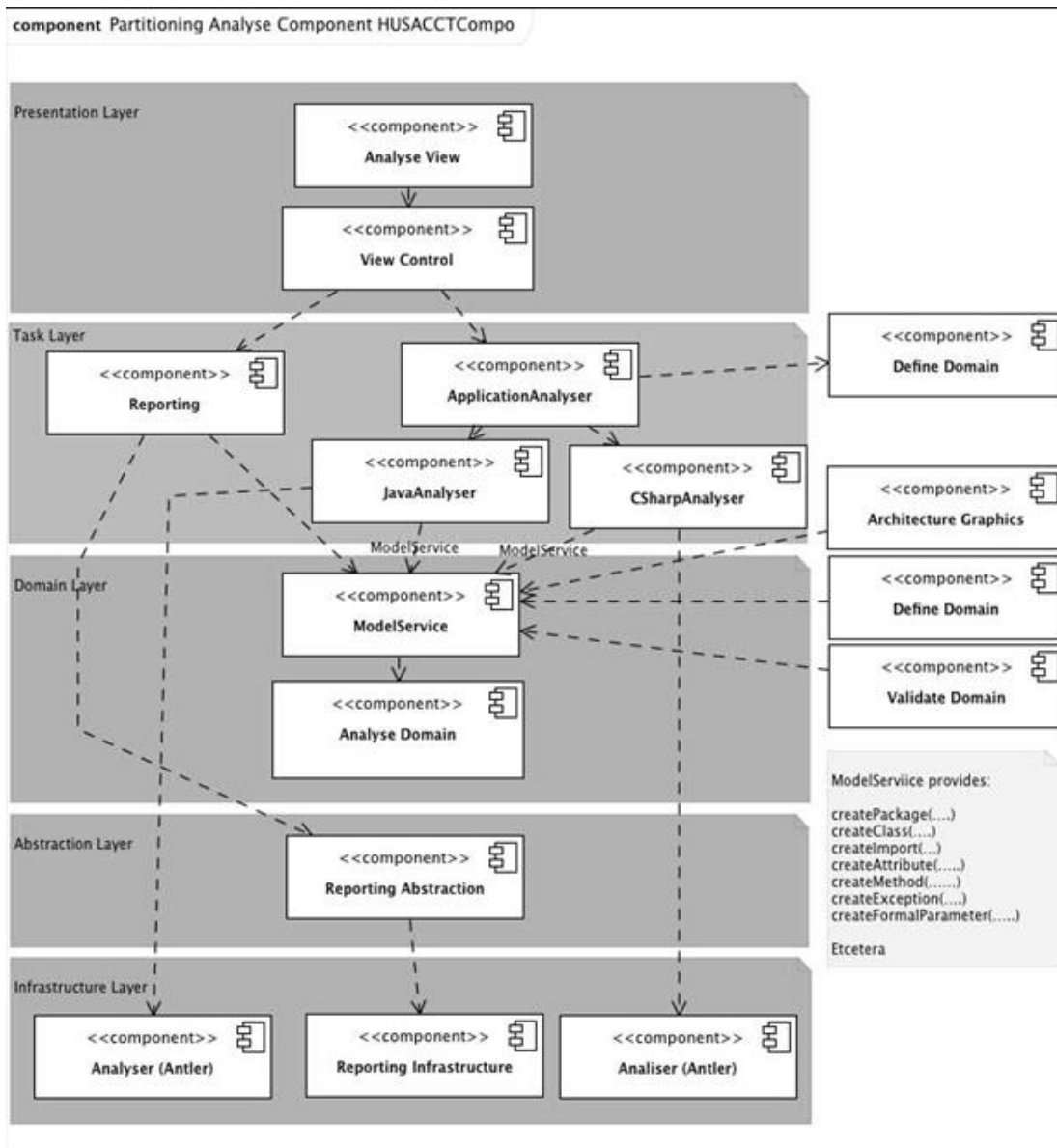


Figure 7.23: The intended architecture of HUSACCT's Analyse component.

Although the thesis has focussed mostly on SAR uses for pattern discovery, there is also a reason for ACC to apply such an approach, due to the equivalence described in Section 3.1. SweetHome3D already showed how we can easily add new patterns to search for and what the limitations are when doing so. In order to demonstrate this more fully and perhaps learn a thing or two for the sake of future work, let us look at an early version of HUSACCT. HUSACCT is open-source, like the other systems, and we know its intended architecture.

So let us look at part of HUSACCT's architecture that is not merely a particular pattern, but a more complex structure. We take the Analyse component of HUSACCT v1.0, like we did in Section 3.2. Figure 7.23 shows the intended architecture for this component. Although this

is a version of layering that might not be considered an established pattern, we can define it as though it were (Code Sample 7.3).

```
package husacct.analyse.task.reconstruct.patterns;

import java.util.ArrayList;
import java.util.Map;

import husacct.common.dto.SoftwareUnitDTO;

public class HUSACCT_Analyse extends Pattern {
    public HUSACCT_Analyse() {
        numberOfModules = 5;
    }

    @Override
    protected void defineModules() {
        defineService.addModule("PresentationLayer", "**", "Subsystem", 1, null);
        defineService.addModule("TaskLayer", "**", "Subsystem", 1, null);
        defineService.addModule("DomainLayer", "**", "Subsystem", 1, null);
        defineService.addModule("AbstractionLayer", "**", "Subsystem", 1, null);
        defineService.addModule("InfrastructureLayer", "**", "Subsystem", 1, null);
    }

    @Override
    protected void defineRules() {
        addSingleRule("TaskLayer", "PresentationLayer", "IsOnlyAllowedToUse", null);
        addSingleRule("PresentationLayer", "TaskLayer", "IsNotAllowedToUse", null);
        addSingleRule("PresentationLayer", "DomainLayer", "IsNotAllowedToUse", null);
        ;
        addSingleRule("PresentationLayer", "InfrastructureLayer", "IsNotAllowedToUse", null);
        addSingleRule("TaskLayer", "InfrastructureLayer", "IsNotAllowedToUse", null);
        ;
        addSingleRule("DomainLayer", "InfrastructureLayer", "IsNotAllowedToUse", null);
        addSingleRule("TaskLayer", "DomainLayer", "IsNotAllowedToUse", null);
        addSingleRule("AbstractionLayer", "TaskLayer", "IsTheOnlyModuleAllowedToUse", null);
        addSingleRule("InfrastructureLayer", "TaskLayer", "IsTheOnlyModuleAllowedToUse", "AbstractionLayer");
        addSingleRule("InfrastructureLayer", "AbstractionLayer", "IsOnlyAllowedToUse", null);
    }

    @Override
    protected void defineMustUseRules() {
        addSingleRule("TaskLayer", "PresentationLayer", "MustUse", null);
        addSingleRule("DomainLayer", "TaskLayer", "MustUse", null);
        addSingleRule("AbstractionLayer", "TaskLayer", "MustUse", null);
        addSingleRule("InfrastructureLayer", "TaskLayer", "MustUse", null);
        // addSingleRule("InfrastructureLayer", "AbstractionLayer", "MustUse", null);
    }

    @Override
    public void mapPattern(ArrayList<String> patternNames) {
        ArrayList<SoftwareUnitDTO> temp = new ArrayList<>(1);
        temp.add(analyseService.getSoftwareUnitByUniqueName(patternNames.get(0)));
        defineService.editModule("PresentationLayer", "PresentationLayer", 1, temp);
        temp.clear();
        temp.add(analyseService.getSoftwareUnitByUniqueName(patternNames.get(1)));
    }
}
```

```

        defineService.editModule("TaskLayer", "TaskLayer", 1, temp);
        temp.clear();
        temp.add(analyseService.getSoftwareUnitByUniqueName(patternNames.get(2)));
        defineService.editModule("DomainLayer", "DomainLayer", 1, temp);
        temp.clear();
        temp.add(analyseService.getSoftwareUnitByUniqueName(patternNames.get(3)));
        defineService.editModule("AbstractionLayer", "AbstractionLayer", 1, temp);
        temp.clear();
        temp.add(analyseService.getSoftwareUnitByUniqueName(patternNames.get(4)));
        defineService.editModule("InfrastructureLayer", "InfrastructureLayer", 1,
temp);
    }

    @Override
    public void mapPatternAllowingAggregates(Map<Integer, ArrayList<String>>
patternUnitNames) {
        ArrayList<SoftwareUnitDTO> temp = new ArrayList<>();
        for (int j = 0; j < patternUnitNames.get(0).size(); j++)
            temp.add(analyseService.getSoftwareUnitByUniqueName(patternUnitNames.get
(0).get(j)));
        defineService.editModuleWithAggregation("PresentationLayer", "
PresentationLayer", 1, temp);
        temp.clear();
        for (int j = 0; j < patternUnitNames.get(1).size(); j++)
            temp.add(analyseService.getSoftwareUnitByUniqueName(patternUnitNames.get
(1).get(j)));
        defineService.editModuleWithAggregation("TaskLayer", "TaskLayer", 1, temp);
        temp.clear();
        for (int j = 0; j < patternUnitNames.get(2).size(); j++)
            temp.add(analyseService.getSoftwareUnitByUniqueName(patternUnitNames.get
(2).get(j)));
        defineService.editModuleWithAggregation("DomainLayer", "DomainLayer", 1,
temp);
        temp.clear();
        for (int j = 0; j < patternUnitNames.get(3).size(); j++)
            temp.add(analyseService.getSoftwareUnitByUniqueName(patternUnitNames.get
(3).get(j)));
        defineService.editModuleWithAggregation("AbstractionLayer", "
AbstractionLayer", 1, temp);
        temp.clear();
        for (int j = 0; j < patternUnitNames.get(4).size(); j++)
            temp.add(analyseService.getSoftwareUnitByUniqueName(patternUnitNames.get
(4).get(j)));
        defineService.editModuleWithAggregation("InfrastructureLayer", "
InfrastructureLayer", 1, temp);
    }
}

```

Code Sample 7.3: HUSACCT's Analyse component.

When we ran the brute force approach on this example, no suitable candidates were found. When we commented out the “Must use” rules between the Abstraction and Infrastructure layers, the candidates portrayed in Figure 7.24 were produced. Upon closer inspection, it turned out that the Infrastructure layer did not contain the JXL Excel Export library that was supposed to be called by the Abstraction layer, as the documentation described. This is exactly what ACC is for, the documentation did not comply with reality. There appears to be some inconsistency in the documentation regarding these modules and our approach pointed it out to us rather quickly, which is a good sign.

Note that the caption under the figure states that we ran the brute force approach including a

Remainder, but that no such candidates were evaluated. That is because the number of software units was exactly equal to the number of pattern modules.

```

Candidate number: 120
HERE STARTS THE REMAINDER LOOP.
INFO [AWT-EventQueue-0] (ReconstructArchitecture.java:302) - Done

Best 30 candidates with non-zero fitness score:
Fitness: 1.0 --> Mapping: [[husacct.analyse.presentation], [husacct.analyse.task], [husacct.analyse.abstraction],
[husacct.analyse.domain], [husacct.analyse.infrastructure]]
Fitness: 1.0 --> Mapping: [[husacct.analyse.presentation], [husacct.analyse.task], [husacct.analyse.abstraction],
[husacct.analyse.infrastructure], [husacct.analyse.domain]]
Fitness: 1.0 --> Mapping: [[husacct.analyse.presentation], [husacct.analyse.task], [husacct.analyse.domain],
[husacct.analyse.abstraction], [husacct.analyse.infrastructure]]
Fitness: 1.0 --> Mapping: [[husacct.analyse.presentation], [husacct.analyse.task], [husacct.analyse.domain],
[husacct.analyse.infrastructure], [husacct.analyse.abstraction]]
Fitness: 1.0 --> Mapping: [[husacct.analyse.presentation], [husacct.analyse.task],
[husacct.analyse.infrastructure], [husacct.analyse.abstraction], [husacct.analyse.domain]]
Fitness: 1.0 --> Mapping: [[husacct.analyse.presentation], [husacct.analyse.task],
[husacct.analyse.infrastructure], [husacct.analyse.domain], [husacct.analyse.abstraction]]
INFO [AWT-EventQueue-0] (ReconstructArchitecture.java:330) - This last mapping was selected for the intended
architecture by default.
Elapsed time: 43.171 seconds.

```

Figure 7.24: The results of the brute force (aggregation + Remainder) run on the Analyse component.

Finally, we look at the packages and classes contained within the software units in the previous run, to see if we get the same layering. There are 23 software units, in that case, which would result in $N^{a,r}(n = 23, k = 5) \approx 7.3 \cdot 10^{17}$ pattern candidates. Obviously, we will use the genetic approach. Some of the results are presented in Figure 7.25.

There are two problems with these results. First of all, there are a lot of strong candidates. This is a powerful lesson to be drawn from the evaluations: a large number of software units can make it too easy to find candidates with high fitness functions, resulting in the need to somehow filter the results. Secondly, these software units do not really seem to correspond to the components in Figure 7.23. There was no need to perform any sort of clustering or manual grouping in the case with Analyse’s packages, but the sub-packages and -classes do not cooperate as nicely as we would like. This, again, goes to show that the modular decomposition step is critical: improving the input for either the brute force or the genetic approach would increase the odds of an improved output.

```

Presenting the ordering of software units mapped in the chromosomes:

1: husacct.analyse.abstraction.export
2: husacct.analyse.domain.AnalyseDomainServiceImpl
3: husacct.analyse.domain.IAnalyseDomainService
4: husacct.analyse.domain.IModelCreationService
5: husacct.analyse.domain.IModelPersistencyService
6: husacct.analyse.domain.IModelQueryService
7: husacct.analyse.domain.famix
8: husacct.analyse.infrastructureantlr
9: husacct.analyse.presentation.AnalyseDebuggingFrame
10: husacct.analyse.presentation.AnalyseInternalFrame
11: husacct.analyse.presentation.AnalyseUIController
12: husacct.analyse.presentation.ApplicationStructurePanel
13: husacct.analyse.presentation.DependencyPanel
14: husacct.analyse.presentation.DependencyTableModel
15: husacct.analyse.presentation.ExportDependenciesDialog
16: husacct.analyse.presentation.FileDialog
17: husacct.analyse.presentation.Regex
18: husacct.analyse.presentation.SoftwareTreeCellRenderer
19: husacct.analyse.presentation.ThreadedDependencyExport
20: husacct.analyse.task.AnalyseControllerServiceImpl
21: husacct.analyse.task.DependencyExportController
22: husacct.analyse.task.IAnalyseControlService
23: husacct.analyse.task.analyser

The best (and unique) chromosomes are printed here. If their number is particularly small, it is
because the population contained many duplicates.
This would mean that the algorithm has converged on a small number of solutions. These should
indicate at least local optima, if not the global optimum.

Chromosome 1:
52223203110200244542052
Fitness value: 1.0
Placing best candidate in defined architecture...
Best candidate was successfully mapped and validated.
Chromosome 2:
32443203110201243502052
Fitness value: 1.0
Chromosome 3:
22253223110204244522052
Fitness value: 1.0
Chromosome 4:
42253323110203243532002
Fitness value: 1.0
Chromosome 5:
32243323110200244342052
Fitness value: 1.0
Chromosome 6:
42223203110200254532052
Fitness value: 1.0
Chromosome 7:
22253323110200244522202
Fitness value: 1.0
. . .
Chromosome 89:
22423203110200224422052
Fitness value: 1.0
Elapsed time: 2216.668 seconds.

```

Figure 7.25: *The results of the genetic (aggregation + Remainder) run on the Analyse component's sub-units.*

7.5 Lessons Learnt

With these final lessons, we conclude the evaluation section. In the future, elaborate case studies involving actual software architects and their systems would be necessary to evaluate a more mature version of this thesis' approach. But for the moment, the preliminary evaluations are fruitful in that they point out the weaknesses and strengths of both the brute force and the genetic approach to pattern discovery using an ACC tool like HUSACCT.

The degree to which a pattern is difficult to find is likely to be related to the number of modules and the number of rules. Compare the evaluation of SweetHome3D's Model-Viewcontroller with HUSACCT's layering, for example. This is almost trivial, more constraints result in fewer pattern candidates.

In conclusion, the key takeaways from these evaluations can be summarised as follows. Nomenclature (semantics) can be a vital aid to our approach, it could influence the fitness of candidates or reduce the number of candidates. A high number of software units makes it more likely that there are many candidates with high fitness scores, making it more difficult to decide on a pattern instance. This may also be remedied using nomenclature, if not by applying a more sophisticated modular decomposition step.

In the next chapter, our vision of a more mature approach is described.

Chapter 8

GEAR

The essence of this thesis can be formulated as follows: an Architecture Compliance Checking software tool, such as HUSACCT, can be made to search for architectural patterns within object-oriented source code, to aid the performance of Software Architecture Reconstruction. It is a guided, semi-automated process, in which a re-engineer decides on the architectural pattern(s) and the software units to use in the search of that/those pattern(s). This can be done due to the fact that the concepts of ACC and SAR have a significant overlap, since verifying a hypothesised architecture (or a portion thereof) is equivalent to a compliance check with architecture documentation.

In theory, anything could be considered a pattern in this approach. There is no reason why a whole technical architecture could not be entered into the approach as though it were one of these architectural patterns. With such a high number of architectural elements, or pattern modules, and a number of selected software units that is at least equal to the number of elements, the number of possible pattern candidates would be astoundingly massive. The computation would, therefore, be terrible costly. However, it could be done.

From this insight, one can conclude that not only can an ACC tool be used to perform SAR, an SAR approach that searches for specific architectures (patterns, in this case) also leads to new ACC approaches. One might enter portions of the architecture as described by the architecture documentation into the pattern editor, as described in the previous section, and look for those. Using the various techniques and considerations that have been mentioned in the various sections of this master thesis, the best possible mappings could thus be found. Especially if this would involve semantic constraints on the names of software units or if some measure of syntactic analysis were included to identify which candidates might be more promising than others.

We envision GEAR, Guided Evolutionary Architecture-Reconstruction. *Guided* because the user has to be kept in the loop for the moment. *Evolutionary* because we employ techniques from evolutionary computing. *Architecture reconstruction* because the ultimate goal is SAR, while ACC is only a way station¹. This chapter will describe parts of GEAR, as an exploration of possible future work.

8.1 User Interface Design

So far, the reconstruction processes proposed in this thesis have only been discussed in abstract terms. Because these are exploratory techniques aimed at designing future research and providing proofs of concept, the level of formalisation is relatively low. However, all this design

¹And the hyphen so one cannot misunderstand it as the guided reconstruction of evolutionary architecture.

research is based, in abstract terms, on ACC tools and, concretely, on HUSACCT. Therefore, this section is aimed at providing a rudimentary design for the concrete implementation of the reconstruction approach within HUSACCT, with regards to a user interface.

HUSACCT's menus and screens display a consistent style, so it speaks for itself that any future developments should be in line with this choice of colours, fonts and other stylistic elements. This thesis has little to do with visual design, so the mock-ups shown in this section are intended to be plain examples of what future implementations could or should look like.

Figure 8.1 shows a screen from the current HUSACCT version, called *Define intended architecture*. This is where, in the ACC process, the re-engineer would inform the tool of the architecture described in the architecture documentation. Modules of the various types can be added, rules can be formulated and exceptions thereof can be specified. Perhaps most importantly, software units that were found in the source code analysis can be assigned to defined modules. Only then can validation result in certain dependencies being marked as violations, if they exist in spite of the rules defined here.

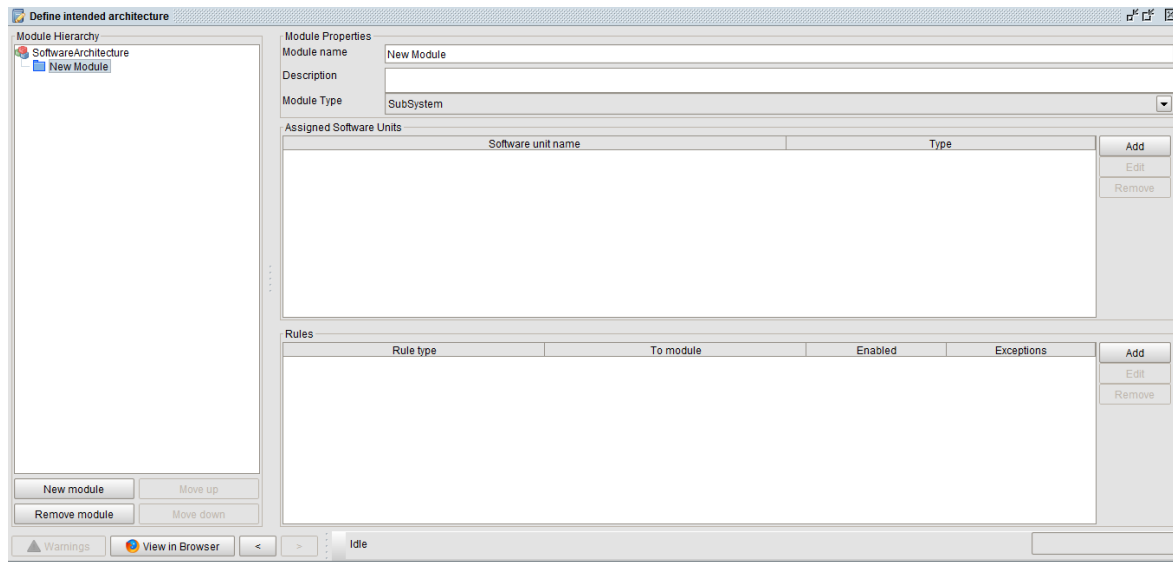


Figure 8.1: *Define intended architecture window.*

One useful but somewhat trivial expansion of this functionality would be to add some menu options involved with the insertion of architectural patterns. A simple button, called *Insert pattern*, for instance. This could even include patterns that are counted among the design patterns rather than architectural patterns, although these are probably problematic due to their rules and specifications not complying with HUSACCT's SRMA elements.

Inserting a pattern would be nothing more than adding multiple modules and rules at a time. Removing a whole pattern from the intended architecture with the click of a button would demand that HUSACCT also be aware of this set of modules constituting a pattern. So that would make things slightly less trivial. Still, this functionality would not be very interesting at all, from an academic perspective at least, but may contribute to further improvement of the software tool.

At the moment of writing, all reconstruction functionality is called by a single menu item, the *Reconstruct Architecture* item under the *Analyse implemented architecture* tab on the menu bar of HUSACCT. This is, obviously, a temporary solution. Decisions about reconstruction technique, software unit selection or fitness function have to be made within the source code, which is ridiculous in terms of usability. No efforts were made to create a suitable user interface,

beyond this design section, because this would be of no academic interest. Also, a master thesis is limited by a comparatively brief time span, most of which tends to be (and indeed was) taken up by literature study and achieving a sufficiently high degree of familiarity with the subject matter. Any remaining time needs to be invested in exploratory implementation, not into considerations of user friendliness or stylistic concerns. Nevertheless, some visual design can be useful to illustrate a vision of future development.

The future of HUSACCT's reconstruction functionality can be envisioned as follows. The menu button that now calls upon the Reconstruct Architecture class and whatever methods are programmed to run, should really open a reconstruction screen. A basic mock-up, devoid of style, of this window is shown in Figure 8.2.

Reconstruction	
<p>Approach</p> <p>Selected software units:</p> <p>4 units selected</p> <p>Selected pattern:</p> <p>N-Layered (isolated internal layers)</p> <p>Selected fitness function:</p> <p>Default</p> <p>Select pattern matching technique:</p> <p>Brute force</p>	<p>Options</p> <p>Allow aggregation <input checked="" type="checkbox"/></p> <p>Allow remainder <input checked="" type="checkbox"/></p> <p>Termination criteria</p> <p>None (brute force)</p>

Figure 8.2: A very basic mock up of a user interface that conveys the idea of what a future implementation could be like.

Obviously, this image implies a set of additional menus and/or windows. These will be discussed one by one.

Select pattern:

Selecting a pattern can be kept rather simple. How to define new patterns or alter existing ones, however, is not as trivial and will be discussed in the next section.

Select software units:

The selection of software units is an important choice for both the brute force approach and the genetic algorithm (and presumably for any other method that might be employed by HUSACCT in the future), since it determines the size and scope of the search space. The current realisation

has no way of dealing with sub-modules, i.e. one software unit cannot be the child of another within the same pattern candidate.

The selection of software units is no trivial matter. Throughout this thesis and within the current exploratory realisation, the package hierarchy (technically the namespace hierarchy in the case of C# source code) is used. However, as has already been alluded to in previous sections, other methods could be employed instead. Clustering methods, for example, could construct a custom hierarchy of software units based on degree of coupling. A K-means algorithm would be one example thereof, or some sort of hierarchical clustering where the optimal number of clusters is determined instead of user-provided. Regardless of what technique might be applied here, several choices will have to be made. See Figure 8.3 for an example of what this might look like. This is in the case of the package hierarchy being relied on, but this window ought to adapt to whatever method be selected. In the case of clustering, distance measures would have to be specified, for instance.

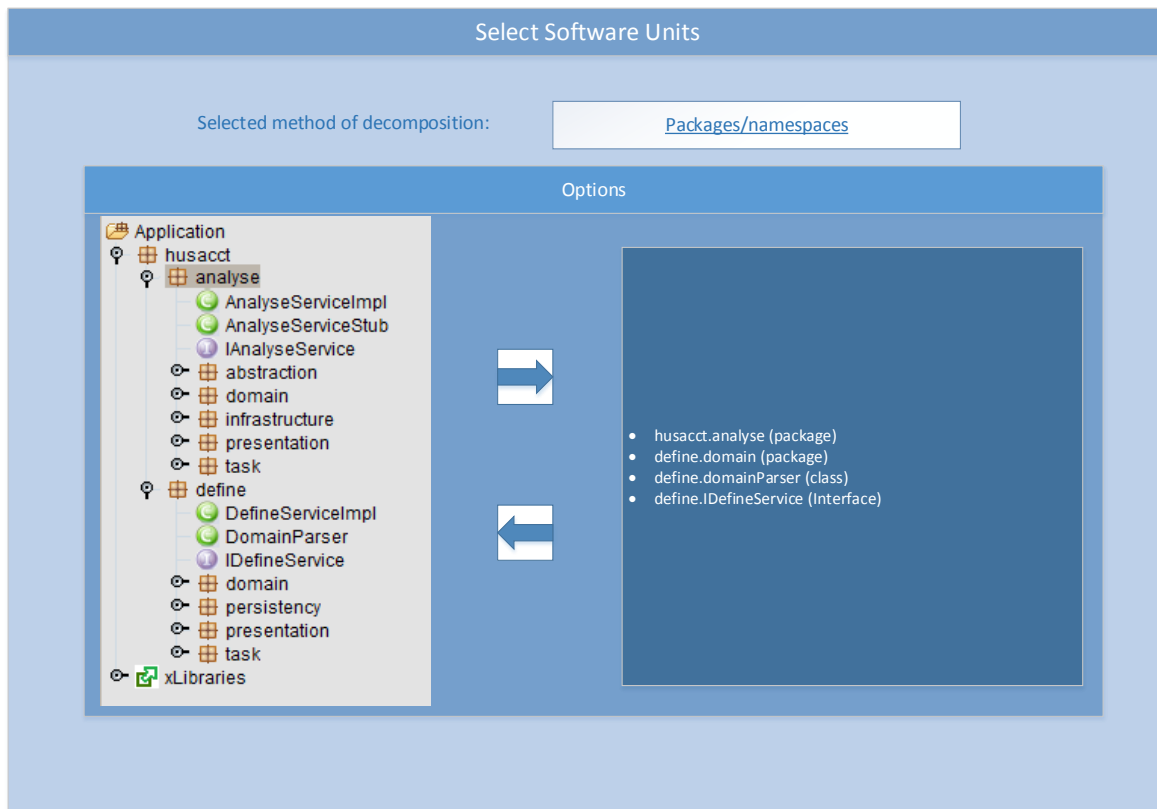


Figure 8.3: Selecting a technique of modular decomposition and choosing software units.

Select fitness function:

Different fitness functions might point a reconstruction algorithm towards different results. One function may promote simple pattern candidates with very few violations, while another function could tend to be maximised by solutions that contain large aggregates of software modules with many “good” dependencies. Which result is more favourable is unclear and, thus, remains up to the user.² Hence, Figure 8.4.

²This is not a coy way of stating that it has not yet been investigated, since there truly seems to be no way of telling beforehand which type of solution would work best for the architecture.

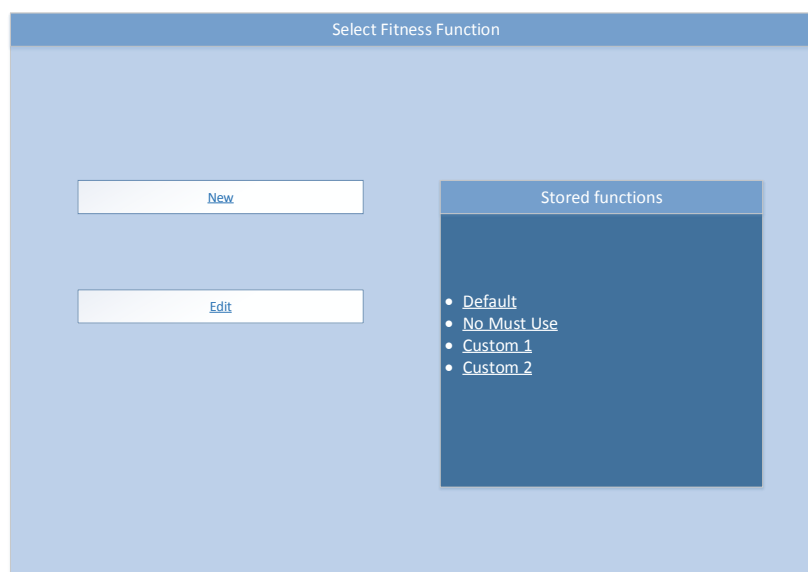


Figure 8.4: Selecting an existing fitness function or editing one (adjusting weights per rule type).

Ideally, the user would be able to edit or provide new fitness functions. This would require an interface for describing such functions, some functionality that allows mathematical expressions to be transcribed into usable code (Java). What sort of construction might fit best here is not known and not the topic of this research, at least not within the scope of the thesis itself.

Select pattern matching technique:

Selecting a method could be rather straightforward, at least as far as can be foreseen at this point. Figure 8.5 shows a small menu that might be displayed, with brief descriptions of the advantages and disadvantages of each technique.

The selection of a technique would then determine the options displayed in Figure 8.5. Some of these would be the same for both the genetic and brute force approaches, such as whether to aggregate software units, and some would be relevant to only one, such as termination criteria (brute force implies all candidates are attempted).

Once all these choices have been made, the algorithm that was selected is allowed to commence. As such, the options that are given in these windows should provide the parameters that are hard-coded into the reconstruction classes at this time.

While pattern matching is being performed, some sort of output and/or animation could be shown. This could be useful for the re-engineer and not just entertaining, since this would give him/her an idea of what is happening. It can be highly informative to observe if the genetic algorithm is converging, for example, even before termination criteria have been met. What exactly should be shown would be the concern of future development, as it does little to illustrate the proposed method for the purposes of this text.

Since the algorithm, whether brute force or otherwise, would be run for one or several architectural patterns, a series of top $n \in \mathbb{N}$ candidates ought to be both displayed as results and saved in case the re-engineer wants to try something else. For this reason, these results should look something like Figure 8.6.

This screen shows the top candidates in terms of their mapping. Filter options allow the re-engineer to view only candidates for a specific pattern, in case several have been attempted.

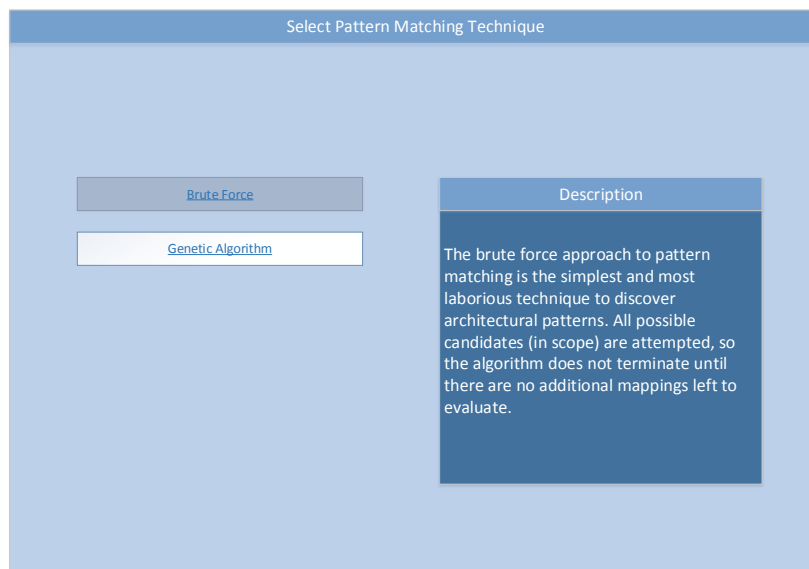


Figure 8.5: Selecting a pattern matching technique. Obviously, this list and others could be extended by additional techniques.

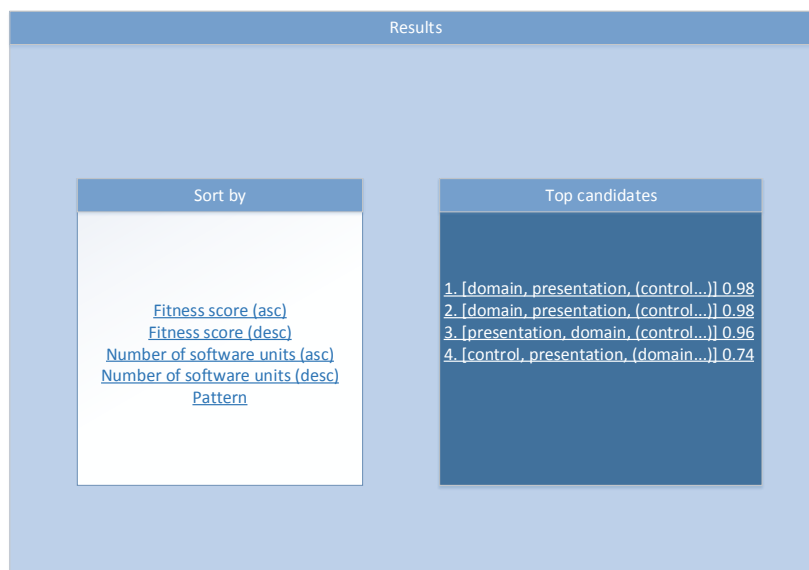


Figure 8.6: The best results displayed as an interactive list.

It is also possible to affect the sorting of the candidates, in terms of fitness values but also number of software units.³

Perhaps it would also be fruitful if double clicking (or otherwise opening) a specific pattern candidate were to provide a quick view of the candidate in terms of an architecture diagram. If the re-engineer clicks on the very best candidate in the list order to study it more closely, something like Figure 8.7 should appear.

This would give the re-engineer a quick impression of the pattern candidate and why it

³The number of software units per candidate would be equal for all candidates if a Remainder were not allowed, obviously.

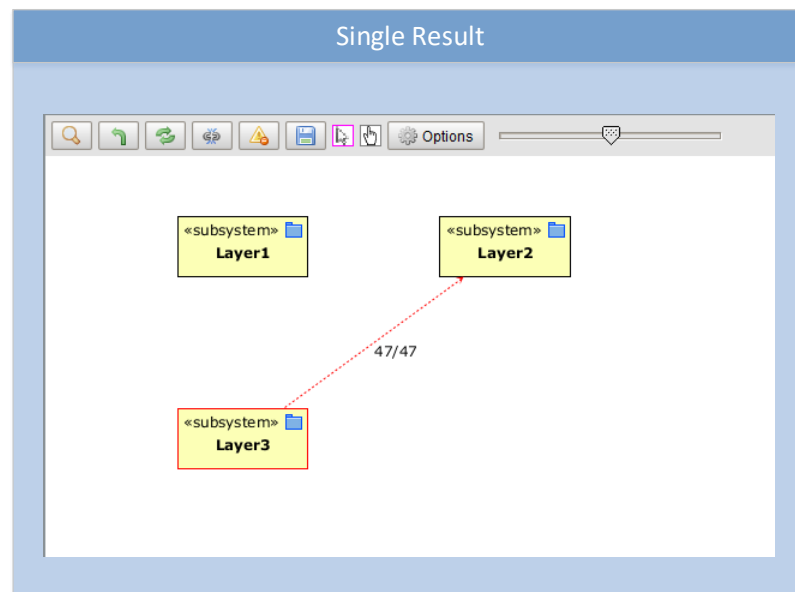


Figure 8.7: Studying a single result more closely.

was considered a particularly good mapping given the provided software units. Perhaps he/she would notice something valuable, such as the name of these software units relating to the pattern modules in either a good (the software unit in the Model-module is actually named *Model*) or bad (the software units in the View-module have nothing to do with graphics or user interfaces) way.

8.2 Pattern Editor

A pattern catalogue was presented in Section 4.3, in which several varieties of the Broker, Model-View-Controller, and the N-Layered patterns were described. It is not inconceivable that a re-engineer might be interested in an additional variety of one of these architectural patterns, with a slightly altered interpretation of the pattern’s rules or altered module types of its elements.

At the moment of writing, adding such a new pattern specification to the proof-of-concept implementation would require that someone alter the source code ⁴ and create a new pattern class. This may be rather easy and relatively simple, it would only entail a few straightforward lines of code, but this is unusable in an end-user application.

Furthermore, varieties of patterns can grow increasingly convoluted at the hands of experienced software architects. Figure 8.8 shows an example of a novel pattern which is, essentially, a mixture of two others. The three layers form a 3-Layered (Complete Freedom) pattern, but it contains several exceptions to the architectural rules conventionally associated with this pattern, because it is merged with the MVC (Controller Interface) pattern.

It would still be relatively convenient to create a pattern specification in which dependencies of a specific type are exempt from an architectural rule. In fact, this can be interpreted as simply an additional rule. It is also quite straightforward to exempt a module rather than a rule type. The more exceptions result in additional pattern definitions, though, the greater the need for another way to add new patterns to HUSACCT.

⁴Available on GitHub: <https://github.com/HUSACCT>.

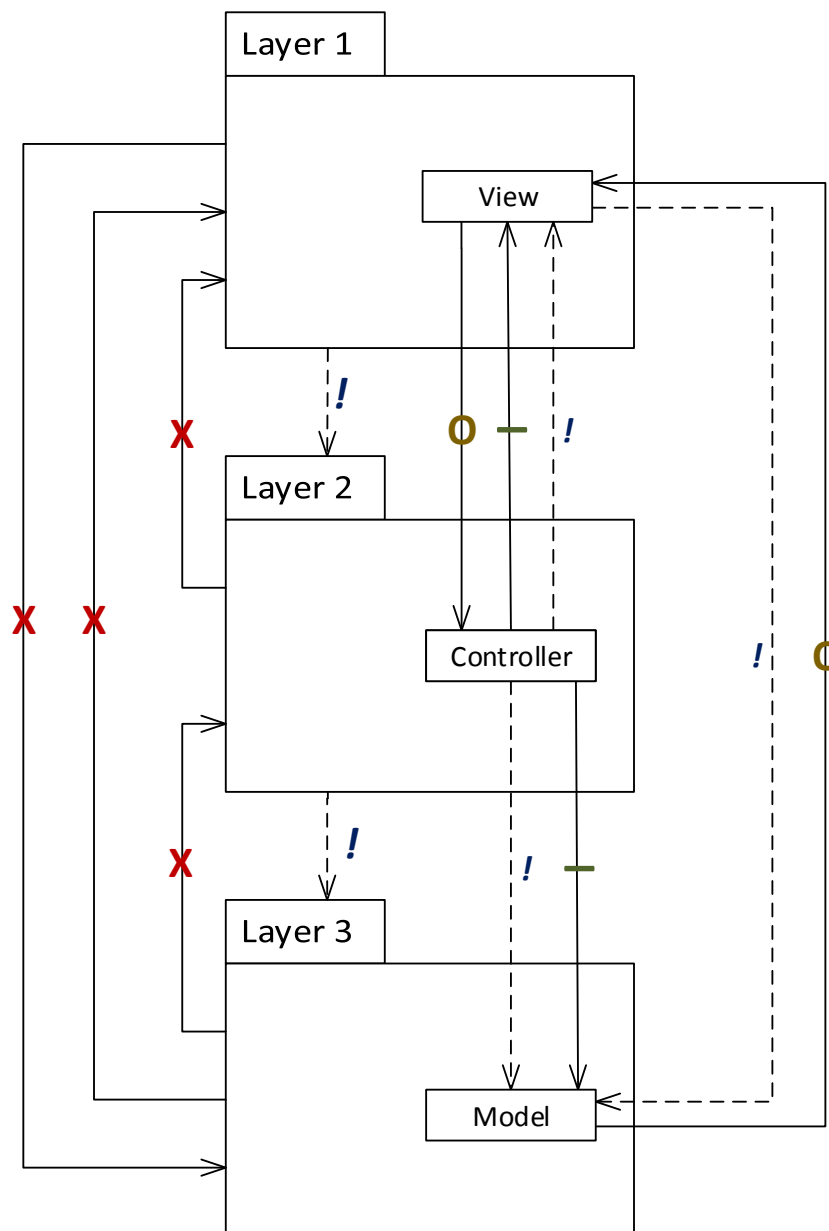


Figure 8.8: A combination of MVC and the 3-Layered patterns..

Moreover, the novel pattern in Figure 8.8 shows a blend of a 3-Layered pattern and a Model-View-Controller pattern, in which the latter's rules form the exceptions to the former pattern's rules. This too could be implemented in the source code directly, though the number of interpretations of this large pattern would warrant many more such code classes.

The problem is, of course, neither the amount of code nor its duplication, especially when correctly using design patterns to realise all this. It actually is the tediousness of building all these pattern specifications and the fact that this is utterly out of the question outside of the academic contact. HUSACCT cannot be used to discover architectural patterns if the addition of novel patterns entails programming and compilation of HUSACCT itself, easy though the programming may be.

For these reasons, a future realisation of this thesis' HUSACCT extensions ought to include

some sort of pattern editor. This project is not the time and place to go into the inner workings of HUSACCT's Famix model, suffice it to say that architectural patterns should be defined, edited, merged, and stored using some technique by which a re-engineer can avoid using code or any knowledge of the underlying mechanisms.

Figure 8.9 shows another user interface design, this one intended to provide a brief illustration of what such an editor might be like. The user can work with an intuitive interface to adapt existing patterns and create new varieties, or start from nothing and specify entirely new ways of structuring architectural elements of a technical architecture.

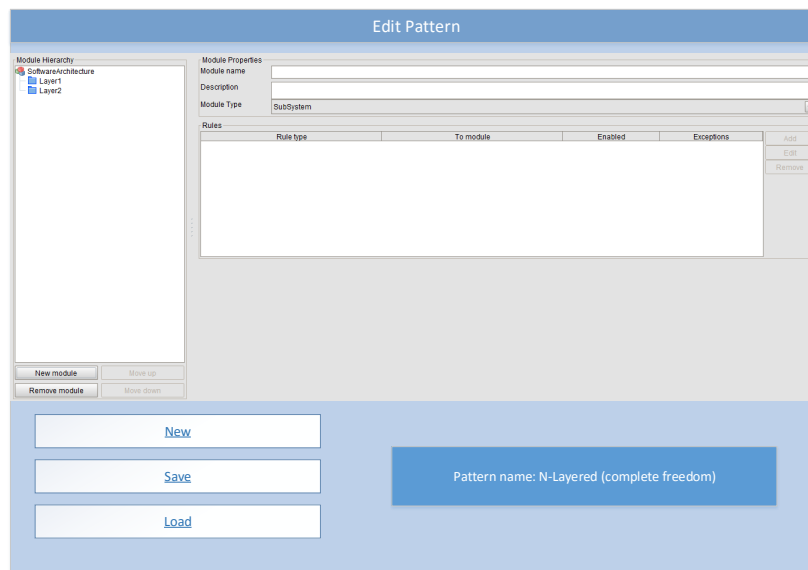


Figure 8.9: Simple example of a possible pattern editor.

Although not yet attempted during this research project, adding patterns through some sort of editor cannot be too difficult. Ideally, patterns can be created or edited in a separate pattern editor, to be subsequently used to define intended architectures. This would speed up the process of copying architecture documentation as well, if an ACC process happens to involve patterns, but this is the least interesting and most trivial implication of these ideas.

8.3 Custom Fitness Function

Similar to the aforementioned pattern editor, a fitness function can be a custom-made (made by the user) formula based on violations, correct observances and weights per rule type. As suggested earlier, different fitness functions are likely to result in different pattern candidates in the list of top candidates. Therefore, fitness functions might be associated with specific candidate characteristics and with specific architectural patterns, or at least the weight factors might be.

It is not necessarily true that “Is not allowed to user” rules are equal in importance to “Is the only module allowed to use” rules for a particular pattern. For example, in the N-Layered pattern version that was characterised by the internal layers being isolated from the Remainder, the “Is not allowed to use” rules were the ones that banned back-calls from the lowest layer to the uppermost. It is therefore not unreasonable to make one type of rule count more towards a poor fitness score if violated than another rule type. This too can be left to the discretion of the re-engineer who uses this approach.

Figure 8.10 shows another rudimentary interface design, showcasing how a re-engineer might adjust the various weights for a particular fitness function. Altering the fitness function itself would probably be more complicated to realise and may be unnecessary. If the user could choose from a selection of fitness functions, different formulations using dependency counts, it may well be sufficient. If not, there would have to be an interface through which a mathematical expression can be used to form a novel fitness function.

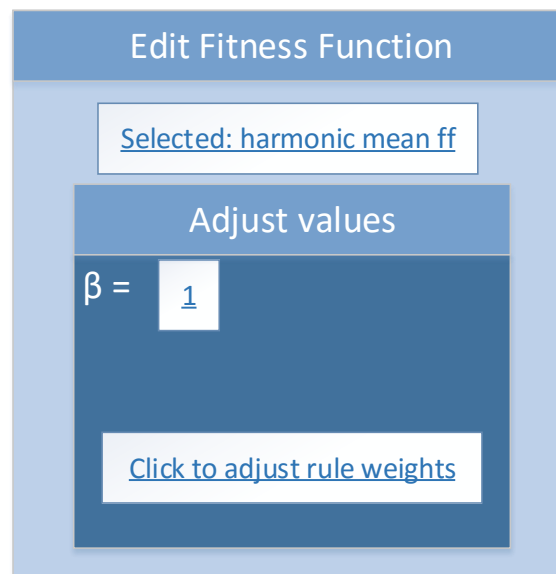


Figure 8.10: A basic illustration to communicate the idea of a future fitness function editor.

The current fitness function, as described in Section 5.4, does not involve a variation of weights per rule type. HUSACCT, however, does rank violation by severity. This is why it would make sense that rule weights be adjusted here, were a future fitness function to take advantage of this as of yet unused functionality of HUSACCT.

8.4 Additional Constraints

Our approach to architectural pattern discovery has focussed mainly on software dependencies: does this module depend on that module and does that agree with the suspected pattern? As we have seen in Chapter 7, dependency analysis is useful but also limited. Although reasonable pattern candidates can show up in the top results, so can rather unreasonable ones. As noted in Section 7.2, this problem might be mitigated by semantic analysis.

The names of software units should definitely give us a clue about the architectural pattern if there is something as obvious as with SweetHome3D's MVC modules. Not only can we then be quite certain about the sort of pattern, even if the precise interpretation is still an open question, we can also be sure of part of the candidate's mapping. Fixing this during the process of architectural pattern discovery is an obvious future improvement.

However, what one would truly want in such cases is for a pattern-based SAR approach to take such things into account as part of its algorithm. Also, this can be far more elaborate than just package/namespace or class names.

Take the semantic clustering by Kuhn, Ducasse and Girba (2007), for example. They take a Latent Semantic Indexing approach to information retrieval from source code that results the

grouping of source code into topics. This could be used in our approach to provide the software units, in the Modular Decomposition step, but should also affect the candidate evaluation process. The topics hint at the function of the software units, after all. Another example would be Saeidi, Hage, Khadka and Jansen (2015), who developed an interactive topic modelling environment called ITMViz that incorporates documented domain knowledge to refine the topic models. We would have to link patterns and pattern modules to such topic names, thus making certain candidates more likely than others. Nomenclature might also be involved in the choice of crossovers, to provide one example where it is not the fitness function that is altered. This topic demands much more future work.

Another thing entirely is syntactic analysis. In this context, this seems more typical of the search for design patterns, where there are particular actions like object instantiation that might hint at certain patterns. In the case of system level architectural patterns, this seems less applicable. However, the combination of semantic and syntactic analysis can be very applicable, such as when a module instantiates many GUI-related objects and thus appears to be related to some kind of presentation pattern module. This could then also have an effect on candidate evaluation, though much research would still have to be performed if this is to be realised.

8.5 Pattern-based Architectures

“Experienced architects typically think of creating an architecture as a process of selecting, tailoring, and combining patterns” (Bass et al., 2012). We saw that SweetHome3D is an example of this in Section 7.3. We thus want to be able to recognise multiple patterns in the same architecture.

Although one would not expect to see a number of architectural patterns in the same order of magnitude as the number of design patterns within a given architecture, systems may include more than one architectural pattern. The exact interpretations of these patterns may result in subtle constraints or even contradictions. This makes explicit definition of patterns even more important, especially during a process like ACC or pattern recognition.

To illustrate this point, let us take the example of an architecture with an MVC pattern. Figure 8.11 shows the “Complete Freedom” variety of MVC. This interpretation uses only the “Is not allowed to use” rule between Model and Controller besides its “Must use” rules. While MVC is the overall style, there is a 3-Layered pattern within the Model module. This is quite reasonable and plausible, as Model might contain business logic and data access functionality separated from the rest (let us call Layer 1 “communication”) through layering. It begs the question, however, of who can access whom in this case.

Based on the architectural rules of these two patterns alone, are the lower layers of Model allowed to have dependencies to and from View and Controller? And, if there happens to be any, what about the Remainder outside the MVC pattern? From the perspective of this 3-Layered pattern, View and Controller are part of the Remainder and therefore the particular pattern definition has a subtle effect. If we were to apply the “Isolated Internal Layers” version of the N-Layered pattern (Figure 4.8), the data access layer would be allowed to have dependencies to and from View and Module. The “Free Remainder” interpretation used in Figure 8.11 would allow dependencies of these outside modules with the data access (in both directions) and business logic (one way) layers, which may also not be desirable.

Probably, the architect wanted to isolate the lower layers from the other MVC modules. Thus, the “Restricted Remainder” interpretation (two “Is the only module allowed to use” rules and one “Is not allowed to use”) ought to be combined with rules that prohibit the data access layer from accessing anything but those modules that it should require. This may be solved

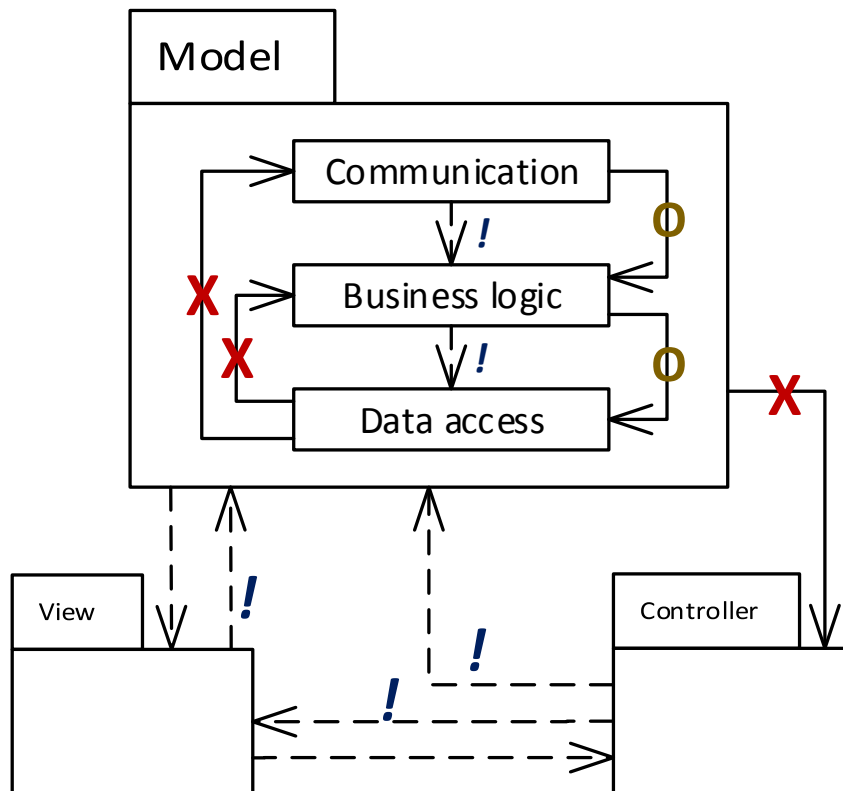


Figure 8.11: A combination of Complete Freedom MVC (style) and Free Remainder 3-Layered (pattern), which fails to isolate the lower Model layers.

with some of the relational rules, or by turning Model into a component type with either the communication layer or an additional element within Model functioning as the interface for this component, thereby calling upon the façade convention that was briefly mentioned in Section 2.

If these combinations raise such questions, then pattern recognition within pattern-based architectures needs to take into account that identifying overall style first could result in a different conclusion from an approach to pattern recognition starting with the smaller pattern.

Chapter 9

Conclusion and Discussion

This master thesis has looked at Architecture Compliance Checking, Software Architecture Reconstruction and pattern recognition/discovery through the lens of allowed and disallowed software dependencies between elements of static, technical, modular architectures. The Hogeschool Utrecht Software Architecture Compliance Checking Tool supports several architectural elements commonly found in Semantically Rich Modular Architectures. The definition of architectural patterns within such a rich “language” results in subtle problems and nuances of interpretation. Not only are architectural patterns so different from design patterns that distinct approaches have to be utilised in order to discover such patterns; not only are patterns like Model-View-Controller so open to interpretation that several versions can be formulated that all seem to grasp the same fundamental ideas; not only is there the problem of determining which software units to work with; in addition to all that, there is the problem of the Remainder. With the examples provided in this thesis, we have illustrated that patterns have an open world, which needs to be *made* explicit; whereas a complete architecture is closed and thus all rules *are* explicit. Assumptions and consequences, such as those involving the Remainder, will thereby become apparent.

The thesis describes several related approaches to pattern discovery in modular architectures. The approach that is conceptually the simplest is the brute force approach, which is to simply try every possible way the selection of software units, which we consistently take from package definitions (Java) or namespace names (C#) throughout the thesis, can be assigned to the pattern modules. These mappings may allow for aggregation, assigning multiple software units to the same pattern module. They may also allow for a Remainder, by which we mean that not each and every software unit within the selection has to be assigned to a pattern module at all. This allows the selection of software units to be relatively crude, as opposed to being a set of packages that ought to correspond one-to-one to the pattern modules.

Next to this, we have developed the initial version of an evolutionary approach to this problem, which employs a genetic algorithm provided by the JGAP framework. Although the thesis has not incorporated extensive experimentation with the effects of adjusting the genetic operators (mutation rates and crossovers) or the effects of more sophisticated ways of tackling the algorithm on convergence rates, we have shown that there is promise in this line of research.

We thus envision GEAR, Guided Evolutionary Architecture-Reconstruction, as a hypothetical future incarnation of our work. There is much research yet to be performed, questions to be answered and code to be (re)written; but we believe one or more publications can already come out of this thesis’ foundations.

Let us now go through the research questions and discuss whether we have truly answered them.

SQ1: *How are architectural patterns used in Architecture Compliance Checking and Software Architecture Reconstruction?*

This question has been addressed in the literature study in Section 5.1. Architectural patterns are used by some researchers as a starting point for architectural design or for reconstruction approaches. We have paid less attention to ACC in literature, since it more of a secondary concern in this thesis, than to SAR, but the two are so related that many papers are relevant to both. Many promising techniques from the literature seem to be based either on predicate logic or on edit-cost graph matching.

SQ2: *How can the individual software modules that correspond to an architectural pattern instance be correctly identified in an application's source code?*

This is mostly answered in Section 5.3, where it is written that this thesis opts for the package/namespace hierarchy. This choice was quite reasonable, since one would not want to waste too much time looking into clustering techniques based on coupling/nomenclature or automatic layering approaches when this is not the main research topic. But these two alternatives, and especially the former, ought to be part of future endeavours. Package hierarchies have proved themselves to be useful at times, such as when an MVC pattern shows itself through package names, but this advantage will not always be present. We saw in the evaluations (Chapter 7) that a high amount of software units can make things difficult, which may be solved with clustering.

SQ3: *How can candidate architectural patterns be identified within a software system?*

The answer to this question is mostly expressed in the two search algorithms, see Chapter 6. The discovery of architectural patterns using an ACC tool depends strongly on the definition of such patterns in that tool's terms. Because HUSACCT is a tool with extensive SRMA support and because patterns are open architectures within a larger system if they do not describe the entire architecture's style, these definitions form a large portion of the thesis.

SQ4: *How can the fitness of a candidate architectural pattern be expressed?*

The fitness function is described in Section 5.4. We have based ourselves on several variables, such as the number of "Must use" affirmations, and used a harmonic mean. Future studies could investigate what the effects of other such fitness functions might be. One function may force most software units into the pattern, whilst another may tend towards small-scaled solutions and thus large Remainders.

SQ5: *How can multiple candidate architectural patterns be compared with one another?*

Implicitly, we have mostly looked at two characteristics: fitness score and believability. The fitness score part is obvious, but the top results are nothing more than those candidates with a high fitness score anyway. Whether a pattern candidate is likely to be a pattern instance depends on whether one can accept it as such, i.e. is it plausible? This largely depends on the package/class names, at the moment. However, as was noted in Section 8.4 and in Section 7.5, this may be exactly when the dependency-derived fitness score ought to be combined with nomenclature analysis to express how one candidate is more likely than another.

SQ6: *How can the best candidate architecture patterns be chosen from a list of candidates?*

This question is much tougher to answer than the previous one, similar though it may seem. Comparing candidates is one thing, but choosing one implies that there are no more candidates of sufficient standing to compare it with. But there may be other pattern definitions that have not been attempted, consequences for subsequent pattern searches when a candidate is chosen and the big question: how do we stop ourselves from settling on the wrong pattern candidate only because architectural erosion has made it *seem* that this is the best candidate? Without verification from the original software architect, do we really now this to be a genuine pattern instance? We saw in Chapter 7 that it is very much possible to find several pattern candidates with perfect scores, based on our fitness function.

Without delving into epistemological philosophy, one can only answer this question in a practical sense. Architectural pattern discovery for the sake of ACC or SAR has a very clear purpose. High amounts of architectural erosion will always make pattern discovery practically impossible and it is likely that there will always be some degree of uncertainty when it comes to any architectural pattern candidate. However, when provided with enough tools, we can make an educated guess. The ACC-based approach described in this thesis is one such tool and others have been mentioned, such as the analysis of syntax or semantics rather than dependencies.

Next to phonology, the study of pronunciation, language can be said to have three aspects: semantics, syntax and pragmatics. Semantics is the vocabulary, the words we choose and what they mean. Syntax is the grammar, the way we construct sentences to convey meaning. And pragmatics is intention, the meaning of our sentences and what we really use them for.

One might be able to see a similar distinction in our take on pattern discovery in the context of reconstructing software architectures. Semantic analysis would be studying the names of packages, classes and methods; looking at comments and searching for key words. This can give us an indication of which architectural pattern might have been applied.

Syntactic analysis would constitute looking at the code, at how it is functioning and what it tries to do. Even if the naming were not to tell us much, the operations within software units, such as data processing or user interaction handling, may give us a clue about the pattern. The distinction with semantics would be a bit more pronounced if we were also to look at object instantiation, to give one example.

Pragmatics would mean investigating the extent to which the source code complies with the ideas behind a pattern. It is the taking into account of what an architectural pattern is supposed to do: constraining the dependencies. It allows dependencies between some parts of the system and prohibits them between others, thus structuring the architecture in a way that promotes qualities such as maintainability and reusability.

This master thesis has been an attempt to contribute to the latter.

As has been described in several sections of this thesis, there is much work still to be done thanks to this project opening up several topics for future research and improvement. The lessons learnt from the preliminary evaluations point out several such topics. Semantics is already part of our approach in the sense that the names of software units play a strong role, but it can have a much larger role to play in modular decomposition and the determination of the fitness score. Incorporating names of packages, classes and perhaps even methods is by far the most important next step to take with this line of research, as became apparent in Chapter 7. Dependency analysis using SRMA-supporting ACC tools is very promising, but it is difficult to find *only* believable pattern candidates without looking at nomenclature.

In the future, we or other researchers may want to experiment with alternative fitness functions. The one described in Subsection 5.4.1 has been useful for the purposes of this thesis. As the evaluation in this thesis shows, a high fitness does not necessarily imply that the pat-

tern is a genuine instance. Extensive experimentation with and evaluation of different fitness functions using systems where the pattern instances are known would allow researchers to draw conclusions about the characteristics of suitable fitness functions.

This thesis explores the possibilities of a genetic approach using ACC to discover architectural patterns. Its practical relevance requires a different approach through many case studies. These may be set up in several ways. Software architecture re-engineers could incorporate pattern discovery in their reconstruction process, or architects could be asked to judge whether supposed pattern instances are genuine instances of whatever patterns were discovered. It seems reasonable to develop a mature version of our approach first, primarily by incorporating semantics, before delving into case studies in the field. It must be noted, however, that this line of research has a very strong practical component and must ultimately result in user friendly tooling that aids the practitioners of ACC and SAR, as opposed to remaining a theoretical exercise.

Next to nomenclature, fitness functions and ultimately case studies, the genetic approach demands much improvement. JGAP was used to speed up the process of implementing a genetic algorithm, where the default configuration was used. But this default configuration is likely to be sub-optimal. Nomenclature and fitness function are both strongly related with improvements in the genetic operators of the genetic approach. Names may be included in the determination of crossover mates and perhaps the fitness function can be made decomposable so as to speed up its evaluation. Experimentation with mutation rates, smart crossovers and sophisticated termination criteria may allow for fast convergence rates, quickly identifying the global optimum. All this would have to be preceded by extensive literature research delving into evolutionary computing, for which there was not enough time during this thesis. The ability to create new pattern candidate populations using such genetic operators opens up a whole field of study and a wide range of possibilities.

Finally, the fact that the approach is set up as a two step process, with modular decomposition preceding the actual search algorithm, allows for more future experimentation. Package hierarchies (or namespaces in the case of C#) do not have to be good indicators of modular architectures. Clustering techniques, whether based on nomenclature or metrics like coupling, seem rather promising in this regard. Future research will have to investigate the applicability of such techniques and when it is best to stick to the hierarchy, as we did.

Chapter 10

Acknowledgements

First and foremost, I would like to thank Jan Martijn van der Werf for the opportunity to collaborate with him in this project. He has taught me a lot about architecture, academia and the inner workings of the university. He was always willing to discuss an idea or read through my work, yet had no problem advising me when I just needed to be told what to do next.

Many thanks also to Leo Pruijt, who encouraged me to work with his tool (HUSACCT) and even set up the code for me. He provided valuable insights and constructive criticism during the project. Over the course of the thesis, I grew to understand and thus appreciate his work even more than I did when I was still familiarising myself with the topic.

Thanks to my second supervisor, Jurriaan Hage, for his thoughts and comments. Next to his serious contributions, his frequent visits to the end of the hall added to the pleasant, encouraging atmosphere created by Amir Saeidi, Sandor Spruit, Jan Strien and Michiel Meulendijk, who also have my gratitude.

Thanks to Sjaak Brinkkemper for his remarks during our discussions and to Sietse Overbeek for his patience when Jan Martijn and I spent far too long brainstorming in what is also his office.

Last but most certainly not least, thanks to my lovely girlfriend, Ana Gómez Rojo, for her constant support, patience and willingness to proofread what must have sometimes been chaotic writings.

References

- Abowd, G. D., Allen, R. & Garlan, D. (1993). Using Style to Understand Descriptions of Software Architecture. In *Proceedings of the 1st acm sigsoft symposium on foundations of software engineering* (pp. 9–20). Los Angeles, CA, USA: ACM. Retrieved from <http://dl.acm.org.proxy.library.uu.nl/citation.cfm?id=167055> doi: 10.1145/256428.167055
- Alvine, B. B., El Boussaidi, G. & Mili, H. (2014). Recovering software layers from object oriented systems. In *International conference on evaluation of novel approaches to software engineering (enase)*. Lisbon, Portugal: IEEE. Retrieved from http://ieeexplore.ieee.org.proxy.library.uu.nl/xpl/freeabs_all.jsp?arnumber=7077119&abstractAccess=no&userType=inst
- Antoniol, G., Fiutem, R. & Cristoforetti, L. (1998). Design pattern recovery in object-oriented software. In *Proceedings. 6th international workshop on program comprehension. iwpc'98 (cat. no.98tb100242)* (pp. 153–160). Ischia, Italy: IEEE. Retrieved from http://ieeexplore.ieee.org.proxy.library.uu.nl/xpl/freeabs_all.jsp?arnumber=693342&abstractAccess=no&userType=inst doi: 10.1109/WPC.1998.693342
- Arcelli, F., Tosi, C., Zanoni, M. & Maggioni, S. (2008). The MARPLE Project -A Tool for Design Pattern Detection and Software Architecture Reconstruction. In *1st international workshop on academic software development tools and techniques*.
- Arcelli Fontana, F. & Zanoni, M. (2011). A tool for design pattern detection and software architecture reconstruction. *Information Sciences*, 181(7), 1306–1324. Retrieved from <http://www.sciencedirect.com/science/article/pii/S0020025510005955#> doi: 10.1016/j.ins.2010.12.002
- Bass, L., Clements, P. & Kazman, R. (2012). *Software Architecture in Practice* (3rd ed.; P. Gordon, Ed.). Boston, Massachusetts, USA: Addison-Wesley Professional.
- Cross, A. D., Wilson, R. C. & Hancock, E. R. (1997, jun). Inexact graph matching using genetic search. *Pattern Recognition*, 30(6), 953–970. Retrieved from <http://linkinghub.elsevier.com/retrieve/pii/S0031320396001239> doi: 10.1016/S0031-3203(96)00123-9
- Dillon, T., Wu, C. & Chang, E. (2007). Reference Architectural Styles for Service-Oriented Computing. In K. Li, C. Jessphope, H. Jin & J.-L. Gaudiot (Eds.), *Network and parallel computing* (Vol. 4672, pp. 543–555). Springer Berlin Heidelberg. Retrieved from http://dx.doi.org/10.1007/978-3-540-74784-0_57 doi: 10.1007/978-3-540-74784-0_57
- Dong, J., Sun, Y. & Zhao, Y. (2008). Design pattern detection by template matching. In *Proceedings of the 2008 acm symposium on applied computing - sac '08* (p. 765). Cear , Brazil: ACCM. Retrieved from <http://dl.acm.org/citation.cfm?id=1363686.1363864> doi: 10.1145/1363686.1363864
- Dong, J., Zhao, Y. & Peng, T. (2009). a Review of Design Pattern Mining Techniques. *International Journal of Software Engineering and Knowledge Engineering*, 19(06), 823–

855. Retrieved from <http://www.worldscientific.com.proxy.library.uu.nl/doi/abs/10.1142/S021819400900443X> doi: 10.1142/S021819400900443X
- Doval, D., Mancoridis, S. & Mitchell, B. S. (1999). Automatic clustering of software systems using a genetic algorithm. *STEP '99. Proceedings Ninth International Workshop Software Technology and Engineering Practice*, 73–81. Retrieved from <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=798481> doi: 10.1109/STEP.1999.798481
- Ducasse, S. & Pollet, D. (2009). Software architecture reconstruction: A process-oriented taxonomy. *IEEE Transactions on Software Engineering*, 35(4), 573–591. Retrieved from <http://www.computer.org/csdl/trans/ts/2009/04/tts2009040573-abs.html> doi: 10.1109/TSE.2009.19
- Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1994). *Design patterns: elements of reusable object-oriented software* (1st ed.). Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. Retrieved from <http://dl.acm.org/citation.cfm?id=186897>
- Garlan, D., Allen, R. & Ockerbloom, J. (1994). Exploiting style in architectural design environments. In *2nd acm sigsoft symposium on foundations of software engineering* (pp. 175–188). New Orleans, LA, USA: ACM. Retrieved from <http://dl.acm.org.proxy.library.uu.nl/citation.cfm?id=195404> doi: 10.1145/195274.195404
- Goldberg, D. E. & Holland, J. H. (1988). No Title. *Machine Learning*, 3(2/3), 95–99. Retrieved from <http://link.springer.com/10.1023/A:1022602019183> doi: 10.1023/A:1022602019183
- Goldstein, M., Segall, I. & Carmel, M. (2015). Automatic and Continuous Software Architecture Validation. In *Ieee international conference on software engineering* (pp. 59–68). Retrieved from <http://2015.icse-conferences.org/component/content/article?id=123> doi: 10.1109/ICSE.2015.135
- Guéhéneuc, Y. G., Sahraoui, H. & Zaidi, F. (2004). Fingerprinting design patterns. In *Proceedings - working conference on reverse engineering, wcre* (pp. 172–181). IEEE. Retrieved from http://ieeexplore.ieee.org.proxy.library.uu.nl/xpl/freeabs_all.jsp?arnumber=1374317&abstractAccess=no&userType=inst doi: 10.1109/WCRE.2004.21
- Harris, D. R., Reubenstein, H. B. & Yeh, A. S. (1995). Reverse engineering to the architectural level. In *Proceedings of the 17th international conference on software engineering - icse '95* (pp. 186–195). New York, New York, USA: ACM Press. Retrieved from <http://dl.acm.org.proxy.library.uu.nl/citation.cfm?id=225032http://portal.acm.org/citation.cfm?doid=225014.225032> doi: 10.1145/225014.225032
- Heuzeroth, D., Holl, T., Hogstrom, G. & Lowe, W. (2003). Automatic design pattern detection. In *Mhs2003. proceedings of 2003 international symposium on micromechatronics and human science (ieee cat. no.03th8717)* (pp. 94–103). IEEE Comput. Soc. Retrieved from <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1199193> doi: 10.1109/WPC.2003.1199193
- Kim, J., Hwang, I., Kim, Y.-H. & Moon, B.-R. (2011). Genetic approaches for graph partitioning. In *Proceedings of the 13th annual conference on genetic and evolutionary computation - gecco '11* (p. 473). New York, New York, USA: ACM Press. Retrieved from <http://portal.acm.org/citation.cfm?doid=2001576.2001642> doi: 10.1145/2001576.2001642
- Kim, J. S. & Garlan, D. (2006). Analyzing architectural styles with alloy. In *Proceedings of the issta 2006 workshop on role of software architecture for testing and analysis - rosatea '06* (pp. 70–80). New York, New York, USA: ACM Press. Retrieved from <http://portal.acm.org/citation.cfm?id=1147259http://portal.acm.org/citation.cfm?doid=1147249.1147259> doi: 10.1145/1147249.1147259

- Kim, J. S. & Garlan, D. (2010, jul). Analyzing architectural styles. *Journal of Systems and Software*, 83(7), 1216–1235. Retrieved from <http://linkinghub.elsevier.com/retrieve/pii/S0164121210000336> doi: 10.1016/j.jss.2010.01.049
- Kuhn, A., Ducasse, S. & Girba, T. (2007, mar). Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 49(3), 230–243. Retrieved from <http://linkinghub.elsevier.com/retrieve/pii/S0950584906001820> doi: 10.1016/j.infsof.2006.10.017
- Niere, J., Schafer, W., Wadsack, J., Wendehals, L. & Welsh, J. (2002). Towards pattern-based design recovery. In *Proceedings of the 24th international conference on software engineering. icse 2002* (pp. 338–348). Orlando, FL, USA: ACM. Retrieved from <http://dl.acm.org.proxy.library.uu.nl/citation.cfm?id=581382> doi: 10.1145/581339.581382
- Paakki, J., Karhinen, A., Gustafsson, J., Nenonen, L. & Verkamo, a. I. (2000). Software metrics by architectural pattern mining. In *Proceedings of the international conference on software theory and practice 16th ifip world computer congress* (pp. 325–332). Retrieved from <http://www.cs.helsinki.fi/group/maisa/ifip2000.pdf>
- Pahl, C., Giesecke, S. & Hasselbring, W. (2009). Ontology-based modelling of architectural styles. *Information and Software Technology*, 51(12), 1739–1749. Retrieved from <http://dx.doi.org/10.1016/j.infsof.2009.06.001> doi: 10.1016/j.infsof.2009.06.001
- Pinzger, M. & Gall, H. (2002). Pattern-supported architecture recovery. In *Proceedings 10th international workshop on program comprehension* (pp. 53–61). IEEE. Retrieved from http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1021318&abstractAccess=no&userType=inst doi: 10.1109/WPC.2002.1021318
- Potel, M. (1996). *MVP : Model-View-Presenter The Taligent Programming Model for C++ and Java* (Tech. Rep.). Taligent, Inc. Retrieved from metrology.googlecode.com/svn-history/r350/trunk/doc/ebooks/mvp.pdf
- Pruijt, L. & Brinkkemper, S. (2014). A metamodel for the support of semantically rich modular architectures in the context of static architecture compliance checking. In *Proceedings of the first international conference on dependable and secure cloud computing architecture - dascca '14* (pp. 1–8). New York, New York, USA: ACM Press. Retrieved from <http://dl.acm.org/citation.cfm?doid=2578128.2578233> doi: 10.1145/2578128.2578233
- Pruijt, L., Koppe, C. & Brinkkemper, S. (2013, sep). Architecture Compliance Checking of Semantically Rich Modular Architectures: A Comparative Study of Tool Support. In *2013 ieee international conference on software maintenance* (pp. 220–229). IEEE. Retrieved from <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6676893> doi: 10.1109/ICSM.2013.33
- Pruijt, L., Köppe, C. & Brinkkemper, S. (2013). On the accuracy of Architecture Compliance Checking Support. In *Ieee international conference on program comprehension* (pp. 172–181). San Francisco, CA, USA. Retrieved from http://ieeexplore.ieee.org.proxy.library.uu.nl/xpl/freeabs_all.jsp?arnumber=6613845&abstractAccess=no&userType=inst doi: 10.1109/ICPC.2013.6613845
- Pruijt, L., Köppe, C., Van der Werf, J. M. E. & Brinkkemper, S. (2014). HUSACCT: Architecture Compliance Checking with Rich Sets of Module and Rule Types. In *Proceedings of the 29th acm/ieee international conference on automated software engineering - ase '14* (pp. 851–854). Vasteras, Sweden: ACM. Retrieved from <http://dl.acm.org/citation.cfm?doid=2642937.2648624> doi: 10.1145/2642937.2648624
- Pruijt, L. & van der Werf, J. M. E. (2015). Dependency Types and Subtypes in the Context of Architecture Reconstruction and Compliance Checking. In *Proceedings of the 2015*

- europa conference on software architecture workshops - ecsaw '15* (pp. 1–7). New York, New York, USA: ACM Press. Retrieved from <http://dl.acm.org/citation.cfm?doid=2797433.2797491> doi: 10.1145/2797433.2797491
- Qureshi, M. R. J. & Sabir, F. (2014). A Comparison of Model View Controller and Model View Presenter. *CoRR*, *abs/1408.5*. Retrieved from <http://arxiv.org/abs/1408.5786>
- Rozanski, N. & Woods, E. (2011). *Software Systems Architecture: Working with stakeholders using viewpoints and perspectives*. (2nd ed.). Addison-Wesley. Retrieved from <http://www.viewpoints-and-perspectives.info/>
- Saeidi, A. M., Hage, J., Khadka, R. & Jansen, S. (2015, may). ITMViz: Interactive Topic Modeling for Source Code Analysis. In *2015 IEEE 23rd international conference on program comprehension* (pp. 295–298). IEEE. Retrieved from <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7181462> doi: 10.1109/ICPC.2015.44
- Sarkar, S., Maskeri, G. & Ramachandran, S. (2009). Discovery of architectural layers and measurement of layering violations in source code. *Journal of Systems and Software*, *82*(11), 1891–1905. Retrieved from <http://dx.doi.org/10.1016/j.jss.2009.06.039> doi: 10.1016/j.jss.2009.06.039
- Sartipi, K. (2003). Software architecture recovery based on pattern matching. In *International conference on software maintenance, 2003. icism 2003. proceedings.* (pp. 293–296). IEEE. Retrieved from http://ieeexplore.ieee.org.proxy.library.uu.nl/xpl/freeabs_all.jsp?arnumber=1235434&abstractAccess=no&userType=inst doi: 10.1109/ICSM.2003.1235434
- Schmerl, B. & Garlan, D. (2004). AcmeStudio: supporting style-centered architecture development. In *26th international conference on software engineering* (pp. 704–705). IEEE. Retrieved from http://ieeexplore.ieee.org.proxy.library.uu.nl/xpl/freeabs_all.jsp?arnumber=1317497&abstractAccess=no&userType=inst doi: 10.1109/ICSE.2004.1317497
- Shalloway, A. & Trott, J. R. (2004). *Design Patterns Explained: A New Perspective on Object Oriented Design* (2nd ed.). Addison-Wesley. Retrieved from <http://www.pearson.ch/1471/9780321247148/Design-Patterns-Explained-A-New.aspx>
- Shull, F., Melo, W. L. & Basili, V. R. (1996). *An Inductive Method for Discovering Design Patterns From Object-Oriented Software Systems* (Tech. Rep.). University of Maryland. Retrieved from <http://drum.lib.umd.edu/handle/1903/799>
- Sokolova, M., Japkowicz, N. & Szpakowicz, S. (2006). Beyond accuracy, F-Score and ROC: A family of discriminant measures for performance evaluation. In A. Sattar & B.-h. Kang (Eds.), *Advances in artificial intelligence* (Vol. 4304, pp. 1015–1021). Springer Berlin Heidelberg. Retrieved from http://link.springer.com/10.1007/11941439_114http://dx.doi.org/10.1007/11941439_114 doi: 10.1007/11941439_114
- Stoermer, C. & O'Brien, L. (2001). MAP - mining architectures for product line evaluations. In *Proceedings working IEEE/IFIP conference on software architecture* (pp. 35–44). Amsterdam, The Netherlands: IEEE. Retrieved from http://ieeexplore.ieee.org.proxy.library.uu.nl/xpl/freeabs_all.jsp?arnumber=948405&abstractAccess=no&userType=inst doi: 10.1109/WICSA.2001.948405
- Taylor, R. N., Medvidović, N. & Dashofy, E. M. (2009). *Software Architecture: Foundations, Theory, and Practice*. Wiley. Retrieved from <http://eu.wiley.com/WileyCDA/WileyTitle/productCd-EHEP000180.html>
- Thang Nguyen Bui & Byung Ro Moon. (1996, jul). Genetic algorithm and graph partitioning. *IEEE Transactions on Computers*, *45*(7), 841–855. Retrieved from <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=508322> doi: 10.1109/12.508322
- Tsantalis, N., Chatzigeorgiou, A., Stephanides, G. & Halkidis, S. T. (2006). Design pat-

- tern detection using similarity scoring. *IEEE Transactions on Software Engineering*, 32(11), 896–909. Retrieved from http://ieeexplore.ieee.org.proxy.library.uu.nl/xpl/freeabs_all.jsp?arnumber=4015512&abstractAccess=no&userType=inst doi: 10.1109/TSE.2006.112
- Van de Weerd, I. & Brinkkemper, S. (2008, jul). Meta-Modeling for Situational Analysis and Design Methods. In M. R. Syed & S. N. Syed (Eds.), *Handbook of research on modern systems analysis and design technologies and applications* (pp. 35–54). IGI Global. Retrieved from <http://www.igi-global.com/chapter/handbook-research-modern-systems-analysis/21060> doi: 10.4018/978-1-59904-887-1
- Van Hee, K., Oanea, O., Post, R., Somers, L. & Van der Werf, J. M. E. (2006). Jasper: a tool for workflow modeling and analysis. In *Sixth international conference on application of concurrency to system design (acsd'06)* (pp. 279–282). IEEE. Retrieved from <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1640246> doi: 10.1109/ACSD.2006.37
- Yan, H. Y. H., Garlan, D., Schmerl, B., Aldrich, J. & Kazman, R. (2004). DiscoTect: a system for discovering architectures from running systems. In *Proceedings. 26th international conference on software engineering* (pp. 470–479). IEEE Computer Society. Retrieved from <http://dl.acm.org.proxy.library.uu.nl/citation.cfm?id=999450> doi: 10.1109/ICSE.2004.1317469
- Yuan-Kai Wang, Kuo-Chin Fan & Jorng-Tzong Horng. (1997). Genetic-based search for error-correcting graph isomorphism. *IEEE Transactions on Systems, Man and Cybernetics, Part B (Cybernetics)*, 27(4), 588–597. Retrieved from <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=604100> doi: 10.1109/3477.604100

Appendices

Appendix A

PDD Tables

The activity table describes all the activities and sub-activities of the various Process-Deliverable Diagrams (PDDs) used in the main body of the thesis in one concise overview. The same is true for the concept table, which is concerned with the deliverable side of the PDDs.

Table A.1: A quick overview of the genetic structure.

Activity	Sub-activity	Description
Collect source code		Source code must be processed for further analysis.
Decompose modules	Analyse namespaces/package definitions	Using the package hierarchy is a relatively simple and straightforward way to perform the modular decomposition step.
	Identify external libraries	HUSACCT filters the external libraries out from the rest of the software units.
	Create software units	Software units are generated as a hierarchical structure.
Analyse dependencies		The dependencies between the discovered software units must be processed and stored to be part of the dependency graph.
Choose pattern		An architectural pattern must be selected. This forms a hypothesis for the architecture.
Choose software units		The selected software units determine the scope of the search.
Evaluate Candidates	Select mapping	In the brute force approach, each possible mapping is selected so that each candidate can be evaluated.
	Evaluate fitness	The fitness score of each candidate is determined based on a given fitness function.
	Generate population	A genetic algorithm requires that a population of chromosomes be created.
	Reproduce chromosomes	The pattern candidates must reproduce and evolve, which involves mutation and crossover.
Finalise candidate list		A number of candidates is presented to the user as the top pattern candidates, from which a pattern instance may or may not be chosen.

Table A.2: The concept table for the various Process-Deliverable Diagrams used throughout this thesis.

Concept	Description
SOURCE CODE	The actual code of the system, presumably organised in a directory filled with class files.
DEPENDENCY GRAPH	Essentially the implemented architecture. The dependency graph consists of software units and their mutual dependencies.
SOFTWARE UNIT	The modules derived from analysing the source code, as distinct from the hypothesised modules in the patterns.
DEPENDENCY	A relationship between software units, see (Pruijt, Köppe & Brinkkemper, 2013), (Pruijt et al., 2014), (Pruijt & Brinkkemper, 2014) and (Pruijt & van der Werf, 2015) for detailed descriptions of these dependency types and the reasons why these were incorporated into the architectural elements.
RULE	An architectural rule, limiting the allowed dependencies between specific software units.
EXCEPTION	A particular exception of a defined rule.
ARCHITECTURAL PATTERN	The pattern that is being evaluated as a hypothesised piece of the architecture.
PATTERN MODULE	An architectural element of a pattern.
PATTERN CANDIDATE	A particular mapping of software units to pattern modules.
CANDIDATE LIST	A list of candidates with high fitness scores, intended as a top-N overview from which to determine possible pattern instance.

Appendix B

Additional Pattern Definitions

```
package husacct.analyse.task.reconstruct.patterns;

import java.util.ArrayList;
import java.util.Map;

import husacct.ServiceProvider;
import husacct.analyse.IAnalyseService;
import husacct.common.dto.ModuleDTO;
import husacct.common.dto.RuleDTO;
import husacct.common.dto.SoftwareUnitDTO;
import husacct.define.DomainToDtoParser;
import husacct.define.IDefineService;
import husacct.define.domain.module.ModuleStrategy;
import husacct.define.domain.services.ModuleDomainService;

// Abstract pattern class, parent of all architectural pattern classes.
public abstract class Pattern {

    protected DomainToDtoParser domainParser = new DomainToDtoParser();
    protected IDefineService defineService = ServiceProvider.getInstance().
        getDefineService();
    protected ModuleDomainService moduleService = new ModuleDomainService();
    protected ArrayList<SoftwareUnitDTO> unitsToMap;
    protected ArrayList<ModuleDTO> patternModules;
    protected int numberOfModules;
    protected int numberOfRules;
    protected String name;
    IAnalyseService analyseService = ServiceProvider.getInstance().
        getAnalyseService();

    // Add modules to the intended architecture in line with an architectural
    // pattern.
    protected abstract void defineModules();

    // Add rules to the intended architecture that apply to the pattern modules
    // and form part of the pattern.
    protected abstract void defineRules();

    // Insert a pattern into the intended architecture.
    public void insertPattern() {
        numberOfRules = 0;
        defineModules();
        defineMustUseRules();
        defineRules();
    }
}
```

```

}

protected abstract void defineMustUseRules();

// Map specific SoftwareUnitDTOs from the analysed application to the defined
// pattern modules.
public abstract void mapPattern(ArrayList<String> patternNames);

protected void addRule(ModuleStrategy moduleTo, ModuleStrategy moduleFrom,
    String ruleType) {
    defineService.addRule(new RuleDTO(ruleType, true, domainParser.parseModule(
        moduleTo), domainParser.parseModule(moduleFrom), new String[0], "", null,
        false));
    numberOfRules++;
}

protected void addRule(ModuleStrategy moduleTo, ModuleStrategy moduleFrom,
    String ruleType, ModuleStrategy exception) {
    defineService.addRuleWithException(
        new RuleDTO(ruleType, true, domainParser.parseModule(moduleTo),
        domainParser.parseModule(moduleFrom), new String[0], "", null, false),
        exception);
    numberOfRules += 2; // An exception is essentially an additional rule (e.g.
    // two "Is only allowed to use" rules coming from the same module is
    // simply one such rule plus an exception of that rule.)
}

public int getNumberOfModules() {
    return numberOfModules;
}

public int getNumberOfRules() {
    return numberOfRules;
}

public abstract void mapPatternAllowingAggregates(Map<Integer, ArrayList<
    String>> patternUnitNames);

public String getName() {
    return name;
}

/** Adds a rule to the pattern. Leave exceptionModule null or empty if you don
    't want an exception.
    *
    * @param moduleFrom
    * @param moduleTo
    * @param ruleType
    * @param exceptionModule */
protected void addSingleRule(String moduleTo, String moduleFrom, String
    ruleType, String exceptionModule) {
    if (exceptionModule == null || exceptionModule.isEmpty())
        addRule(moduleService.getModuleByLogicalPath(moduleTo), moduleService.
            getModuleByLogicalPath(moduleFrom), ruleType);
    else
        addRule(moduleService.getModuleByLogicalPath(moduleTo), moduleService.
            getModuleByLogicalPath(moduleFrom), ruleType,
            moduleService.getModuleByLogicalPath(exceptionModule));
}
}

```

Code Sample B.1: The source code for the abstract Pattern class.

```

package husacct.analyse.task.reconstruct.patterns;

import java.util.ArrayList;
import java.util.Map;

import husacct.common.dto.SoftwareUnitDTO;

public abstract class LayeredPattern extends Pattern {
    // The abstract class for all N-Layered patterns. Constructor and mapping
    // methods can be defined here, as well as "MustUse" rules.
    public LayeredPattern(int numberOfLayers) {
        numberOfModules = numberOfLayers;
        name = "Layered";
    }

    public LayeredPattern() {
        numberOfModules = 3;
        name = "Layered";
    }

    @Override
    protected void defineModules() {
        for (int i = 1; i <= numberOfModules; i++) {
            defineService.addModule("Layer" + i, "**", "Subsystem", i, null);
        }
    }

    @Override
    protected void defineMustUseRules() {
        for (int i = 1; i < numberOfModules; i++)
            addSingleRule("Layer" + (i + 1), "Layer" + i, "MustUse", null);
    }

    @Override
    public void mapPattern(ArrayList<String> mapping) {
        ArrayList<SoftwareUnitDTO> temp = new ArrayList<>(1);
        for (int i = 1; i <= mapping.size(); i++) {
            temp.add(analyseService.getSoftwareUnitByUniqueName(mapping.get(i - 1)));
            defineService.editModule("Layer" + i, "Layer" + i, i, temp);
            temp.clear();
        }
    }

    @Override
    public void mapPatternAllowingAggregates(Map<Integer, ArrayList<String>>
        patternUnitNames) {
        ArrayList<SoftwareUnitDTO> temp = new ArrayList<>();
        for (int i = 0; i < patternUnitNames.size(); i++) {
            for (int j = 0; j < patternUnitNames.get(i).size(); j++) {
                temp.add(analyseService.getSoftwareUnitByUniqueName(patternUnitNames.get(i).get(j)));
            }
            defineService.editModuleWithAggregation("Layer" + (i + 1), "Layer" + (i + 1), i + 1, temp);
            temp.clear();
        }
    }
}

```

```
}

```

Code Sample B.2: The source code for the abstract class of the N-Layered pattern.

```
package husacct.analyse.task.reconstruct.patterns;

public class LayeredPattern_IsolatedInternalLayers extends LayeredPattern {
    // The version of the N-Layered pattern in which the internal layers are
    // isolated from the Remainder.
    public LayeredPattern_IsolatedInternalLayers(int numberOfLayers) {
        super(numberOfLayers);
    }

    public LayeredPattern_IsolatedInternalLayers() {
        super();
    }

    @Override
    protected void defineRules() {
        for (int i = 1; i < numberOfModules; i++) {
            addSingleRule("Layer" + i, "Layer" + numberOfModules, "IsNotAllowedToUse",
                null);
            if (i == 1) {
                addSingleRule("Layer" + 2, "Layer" + 1, "IsTheOnlyModuleAllowedToUse",
                    null);
            } else {
                addSingleRule("Layer" + (i + 1), "Layer" + i, "IsOnlyAllowedToUse", null);
            }
        }
    }

    @Override
    protected void defineModules() {
        for (int i = 1; i <= numberOfModules; i++) {
            defineService.addModule("Layer" + i, "**", "Subsystem", i, null);
        }
    }
}
```

Code Sample B.3: The Isolated Internal Layers variety of the N-Layered pattern.

```
package husacct.analyse.task.reconstruct.patterns;

public class LayeredPattern_CompleteFreedom extends LayeredPattern {
    // The interpretation of the N-Layered pattern for which there exist no
    // restriction for the Remainder to use the layers.
    public LayeredPattern_CompleteFreedom(int numberOfLayers) {
        super(numberOfLayers);
    }

    public LayeredPattern_CompleteFreedom() {
        super();
    }

    @Override
    protected void defineRules() {
        for (int i = 1; i <= numberOfModules; i++) {

```

```

        for (int j = 1; j <= i; j++) {
            if (j < i)
                addSingleRule("Layer" + (i - j), "Layer" + i, "IsNotAllowedToUse",
null);
            if ((i + j + 1) <= numberOfModules)
                addSingleRule("Layer" + (i + j + 1), "Layer" + i, "IsNotAllowedToUse",
null);
        }
    }
}
}

```

Code Sample B.4: The Complete Freedom variety of the N-Layered pattern.

```

package husacct.analyse.task.reconstruct.patterns;

public class LayeredPattern_FreeRemainder extends LayeredPattern {
    // The version of the N-Layered pattern where the Remainder can all on any
    // layer, though all but the final layer cannot call on the Remainder.
    public LayeredPattern_FreeRemainder(int numberOfLayers) {
        super(numberOfLayers);
    }

    public LayeredPattern_FreeRemainder() {
        super();
    }

    @Override
    protected void defineRules() {
        for (int i = 1; i < numberOfModules; i++) {
            addSingleRule("Layer" + (i + 1), "Layer" + i, "IsOnlyAllowedToUse", null);
            addSingleRule("Layer" + i, "Layer" + numberOfModules, "IsNotAllowedToUse",
null);
        }
    }
}

```

Code Sample B.5: The Free Remainder variety of the N-Layered pattern.

```

package husacct.analyse.task.reconstruct.patterns;

public class LayeredPattern_RestrictedRemainder extends LayeredPattern {
    // An interpretation of the N-Layered pattern in which the layers can call
    // upon the Remainder, but not vice versa.
    public LayeredPattern_RestrictedRemainder(int numberOfLayers) {
        super(numberOfLayers);
    }

    public LayeredPattern_RestrictedRemainder() {
        super();
    }

    @Override
    protected void defineRules() {

```

```

    for (int i = 1; i < numberOfModules; i++) {
        addSingleRule("Layer" + (i + 1), "Layer" + i, "IsTheOnlyModuleAllowedToUse", null);
    }
    addSingleRule("Layer" + 1, "Layer" + numberOfModules, "IsNotAllowedToUse", null);
}
}

```

Code Sample B.6: The Restricted Remainder variety of the N-Layered pattern.

```

package husacct.analyse.task.reconstruct.patterns;

public class LayeredPattern_LayerTypes extends LayeredPattern {
    // This is the N-Layered pattern with actual layer-type modules and the
    // associated skip-call and back-call bans.
    public LayeredPattern_LayerTypes(int numberOfLayers) {
        super(numberOfLayers);
    }

    public LayeredPattern_LayerTypes() {
        super();
    }

    @Override
    protected void defineRules() {
        // Skip-call and back-call rules get added automatically, so there's no need
        // for them here.
    }

    @Override
    protected void defineModules() {
        for (int i = 1; i <= numberOfModules; i++) {
            defineService.addModule("Layer" + i, "**", "Layer", i, null);
        }
    }
}

```

Code Sample B.7: The Layer Types variety of the N-Layered pattern.

```

package husacct.analyse.task.reconstruct.patterns;

import java.util.ArrayList;
import java.util.Map;

import husacct.common.dto.SoftwareUnitDTO;

public abstract class MVCPattern extends Pattern {
    // The abstract class for all Model-View-Controller patterns. Constructor and
    // mapping methods can be defined here, as well as "MustUse" rules and
    // the modules themselves.
    public MVCPattern() {
        numberOfModules = 3;
        name = "MVC";
    }

    @Override
    protected void defineModules() {
        defineService.addModule("Model", "**", "Subsystem", 1, null);
    }
}

```

```

        defineService.addModule("View", "**", "Subsystem", 1, null);
        defineService.addModule("Controller", "**", "Subsystem", 1, null);
    }

    @Override
    protected void defineMustUseRules() {
        addSingleRule("Model", "Controller", "MustUse", null);
        addSingleRule("Model", "View", "MustUse", null);
        addSingleRule("View", "Controller", "MustUse", null);
    }

    @Override
    public void mapPattern(ArrayList<String> mapping) {
        ArrayList<SoftwareUnitDTO> temp = new ArrayList<>(1);
        temp.add(analyseService.getSoftwareUnitByUniqueName(mapping.get(0)));
        defineService.editModule("Model", "Model", 1, temp);
        temp.clear();
        temp.add(analyseService.getSoftwareUnitByUniqueName(mapping.get(1)));
        defineService.editModule("View", "View", 1, temp);
        temp.clear();
        temp.add(analyseService.getSoftwareUnitByUniqueName(mapping.get(2)));
        defineService.editModule("Controller", "Controller", 1, temp);
        temp.clear();
    }

    @Override
    public void mapPatternAllowingAggregates(Map<Integer, ArrayList<String>>
        patternUnitNames) {
        ArrayList<SoftwareUnitDTO> temp = new ArrayList<>();
        for (int j = 0; j < patternUnitNames.get(0).size(); j++) {
            temp.add(analyseService.getSoftwareUnitByUniqueName(patternUnitNames.get(0).get(j)));
        }
        defineService.editModuleWithAggregation("Model", "Model", 1, temp);
        temp.clear();
        for (int j = 0; j < patternUnitNames.get(1).size(); j++) {
            temp.add(analyseService.getSoftwareUnitByUniqueName(patternUnitNames.get(1).get(j)));
        }
        defineService.editModuleWithAggregation("View", "View", 1, temp);
        temp.clear();
        for (int j = 0; j < patternUnitNames.get(2).size(); j++) {
            temp.add(analyseService.getSoftwareUnitByUniqueName(patternUnitNames.get(2).get(j)));
        }
        defineService.editModuleWithAggregation("Controller", "Controller", 1, temp);
    }
}

```

Code Sample B.8: The source code for the abstract Model-View-Controller pattern class.

```

package husacct.analyse.task.reconstruct.patterns;

public class MVCPattern_CompleteFreedom extends MVCPattern {

    public MVCPattern_CompleteFreedom() {
        // TODO Auto-generated constructor stub
    }
}

```

```

@Override
protected void defineRules() {
    addSingleRule("Controller", "Model", "IsNotAllowedToUse", null);
}
}

```

Code Sample B.9: The Complete Freedom variety of the MVC pattern.

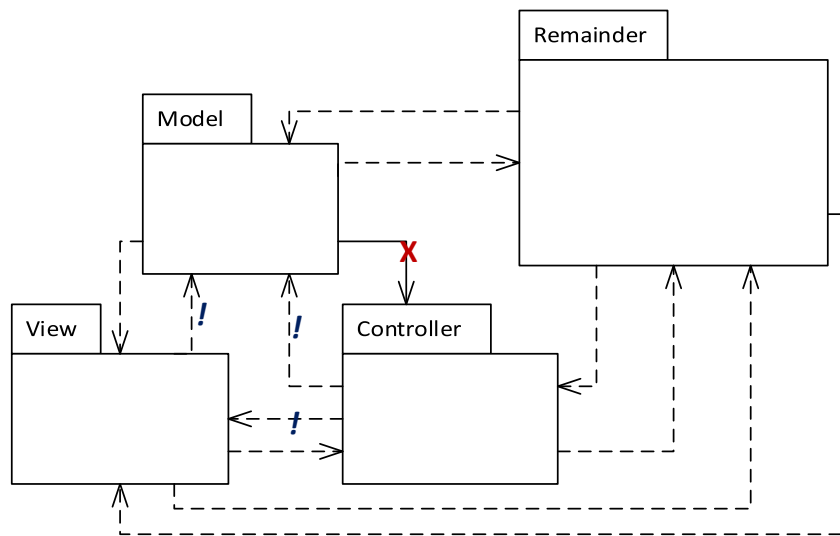


Figure B.1: The Complete Freedom interpretation of the classic MVC pattern.

```

package husacct.analyse.task.reconstruct.patterns;

public class MVCPattern_FreeRemainder extends MVCPattern {

    public MVCPattern_FreeRemainder() {
        // TODO Auto-generated constructor stub
    }

    @Override
    protected void defineRules() {
        addSingleRule("View", "Model", "IsOnlyAllowedToUse", null);
        addSingleRule("Model", "View", "IsOnlyAllowedToUse", "Controller");
        addSingleRule("Controller", "View", "IsOnlyAllowedToUse", "Model");
        addSingleRule("View", "Controller", "IsOnlyAllowedToUse", "Model");
        addSingleRule("Model", "Controller", "IsOnlyAllowedToUse", "View");
    }
}

```

Code Sample B.10: The Free Remainder variety of the MVC pattern.

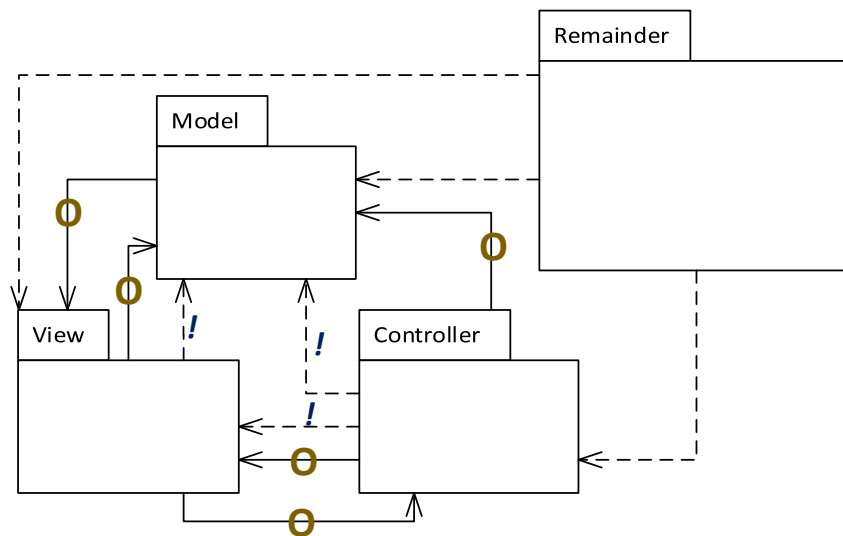


Figure B.2: The Free Remainder interpretation of the classic MVC pattern.

```
package husacct.analyse.task.reconstruct.patterns;

public class MVCPattern_RestrictedRemainder extends MVCPattern {

    public MVCPattern_RestrictedRemainder() {
        // TODO Auto-generated constructor stub
    }

    @Override
    protected void defineRules() {
        addSingleRule("View", "Model", "IsTheOnlyModuleAllowedToUse", "Controller");
        addSingleRule("Model", "View", "IsTheOnlyModuleAllowedToUse", "Controller");
        addSingleRule("Controller", "View", "IsTheOnlyModuleAllowedToUse", null);
        addSingleRule("View", "Controller", "IsTheOnlyModuleAllowedToUse", "Model");
        addSingleRule("Model", "Controller", "IsTheOnlyModuleAllowedToUse", "View");
    }
}
```

Code Sample B.11: The Restricted Remainder variety of the MVC pattern.

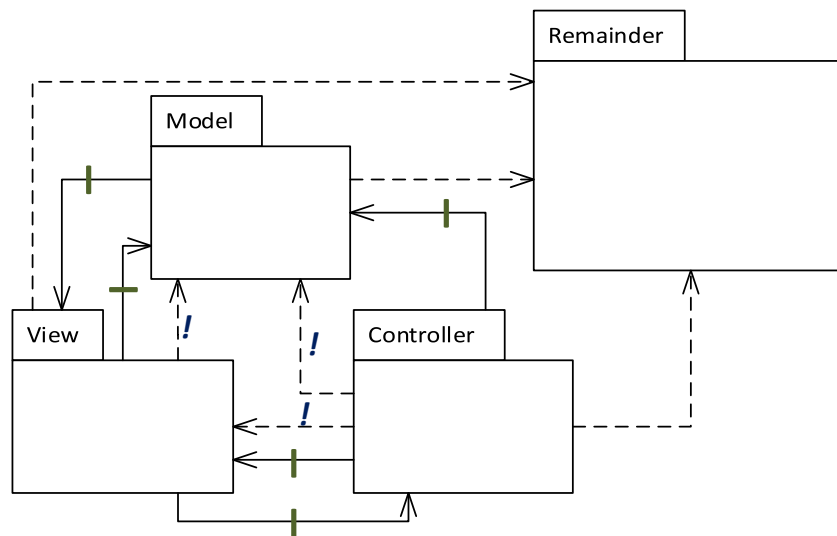


Figure B.3: The Restricted Remainder interpretation of the classic MVC pattern.

```
package husacct.analyse.task.reconstruct.patterns;

public class MVCPattern_ModelInterface extends MVCPattern {

    public MVCPattern_ModelInterface() {
        // TODO Auto-generated constructor stub
    }

    @Override
    protected void defineRules() {
        addSingleRule("View", "Model", "IsTheOnlyModuleAllowedToUse", "Controller");
        addSingleRule("View", "Controller", "IsTheOnlyModuleAllowedToUse", "Model");
        addSingleRule("Controller", "View", "IsTheOnlyModuleAllowedToUse", null);
        addSingleRule("Model", "View", "IsOnlyAllowedToUse", "Controller");
        addSingleRule("Model", "Controller", "IsOnlyAllowedToUse", "View");
    }
}
```

Code Sample B.12: The Model Interface variety of the MVC pattern.

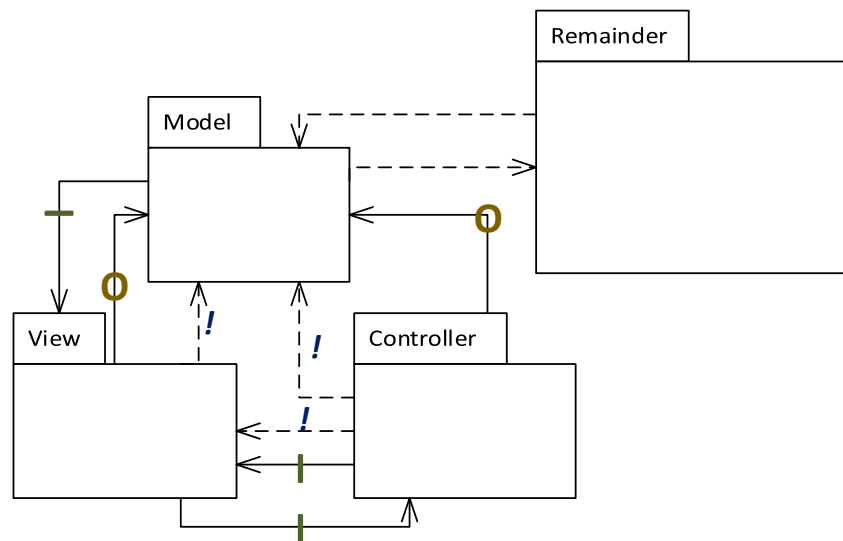


Figure B.4: The Model Interface interpretation of the classic MVC pattern.

```

package husacct.analyse.task.reconstruct.patterns;

import java.util.ArrayList;
import java.util.Map;

import husacct.ServiceProvider;
import husacct.analyse.IAnalyseService;
import husacct.common.dto.SoftwareUnitDTO;

public abstract class BrokerPattern extends Pattern {
    // In the Broker Pattern, the Requester calls on the Broker to use the correct
    // services of the Provider. The Provider may want to use the Broker
    // and the Broker could depend on the Requester, but this is not required. If
    // one of these three modules were to be an interface with respect to
    // the Remainder, i.e. the rest of the architecture, it would make most sense
    // of the Requester were placed in this role.
    public BrokerPattern() {
        numberOfModules = 3;
        name = "Broker";
    }

    @Override
    protected void defineMustUseRules() {
        addSingleRule("Provider", "Broker", "MustUse", null);
        addSingleRule("Broker", "Requester", "MustUse", null);
    }

    @Override
    protected void defineModules() {
        defineService.addModule("Broker", "**", "SubSystem", 1, null);
        defineService.addModule("Provider", "**", "SubSystem", 1, null);
        defineService.addModule("Requester", "**", "SubSystem", 1, null);
    }

    @Override
    public void mapPattern(ArrayList<String> mapping) {
        IAnalyseService analyseService = ServiceProvider.getInstance().
            getAnalyseService();
        ArrayList<SoftwareUnitDTO> temp = new ArrayList<>(1);
    }
}

```

```

        temp.add(analyseService.getSoftwareUnitByUniqueName(mapping.get(0)));
        defineService.editModule("Broker", "Broker", 1, temp);
        temp.clear();
        temp.add(analyseService.getSoftwareUnitByUniqueName(mapping.get(1)));
        defineService.editModule("Provider", "Provider", 1, temp);
        temp.clear();
        temp.add(analyseService.getSoftwareUnitByUniqueName(mapping.get(2)));
        defineService.editModule("Requester", "Requester", 1, temp);
    }

    @Override
    public void mapPatternAllowingAggregates(Map<Integer, ArrayList<String>>
        patternUnitNames) {
        IAnalyseService analyseService = ServiceProvider.getInstance().
            getAnalyseService();
        ArrayList<SoftwareUnitDTO> temp = new ArrayList<>();
        for (int i = 0; i < patternUnitNames.size(); i++) {
            for (int j = 0; j < patternUnitNames.get(i).size(); j++) {
                temp.add(analyseService.getSoftwareUnitByUniqueName(patternUnitNames.get(
                    i).get(j)));
            }
            if (i == 0)
                defineService.editModuleWithAggregation("Broker", "Broker", 1, temp);
            else if (i == 1)
                defineService.editModuleWithAggregation("Provider", "Provider", 1, temp);
            ;
            else if (i == 2)
                defineService.editModuleWithAggregation("Requester", "Requester", 1,
                    temp);
            temp.clear();
        }
    }
}

```

Code Sample B.13: The source code for the abstract Broker pattern class.

```

package husacct.analyse.task.reconstruct.patterns;

public class BrokerPattern_CompleteFreedom extends BrokerPattern {
    // The version of the Broker pattern in which there are no restrictions with
    // regard to the Remainder (so dependencies to and from are allowed).
    public BrokerPattern_CompleteFreedom() {
        // TODO Auto-generated constructor stub
    }

    @Override
    protected void defineRules() {
        addSingleRule("Requester", "Provider", "IsNotAllowedToUse", null);
        addSingleRule("Provider", "Requester", "IsNotAllowedToUse", null);
    }
}

```

Code Sample B.14: The source code for the Broker (Complete Freedom) pattern class.

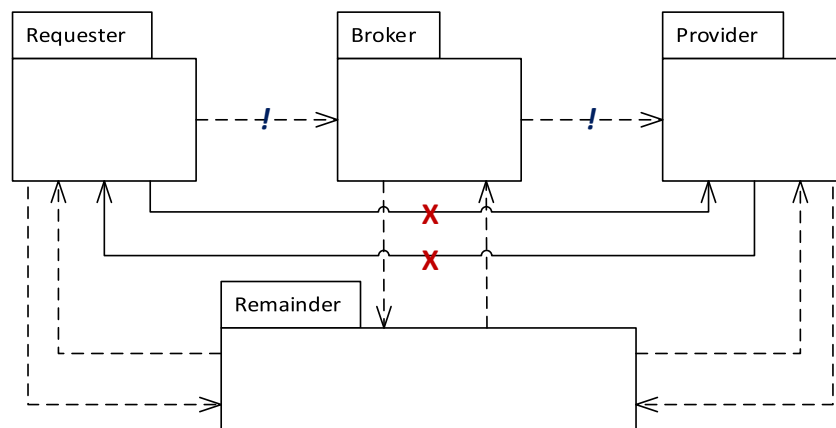


Figure B.5: The Requester Interface interpretation of the Broker pattern.

```
package husacct.analyse.task.reconstruct.patterns;

public class BrokerPattern_RestrictedRemainder extends BrokerPattern {
    // In this variety of the Broker Pattern, the pattern modules are free to call
    // on the Remainder.
    public BrokerPattern_RestrictedRemainder() {
        // TODO Auto-generated constructor stub
    }

    @Override
    protected void defineRules() {
        addSingleRule("Broker", "Requester", "IsTheOnlyModuleAllowedToUse", "
        Provider");
        addSingleRule("Requester", "Broker", "IsTheOnlyModuleAllowedToUse", null);
        addSingleRule("Provider", "Broker", "IsTheOnlyModuleAllowedToUse", null);
        addSingleRule("Broker", "Provider", "IsTheOnlyModuleAllowedToUse", "
        Requester");
    }
}
```

Code Sample B.15: The source code for the Broker (Free Remainder) pattern class.

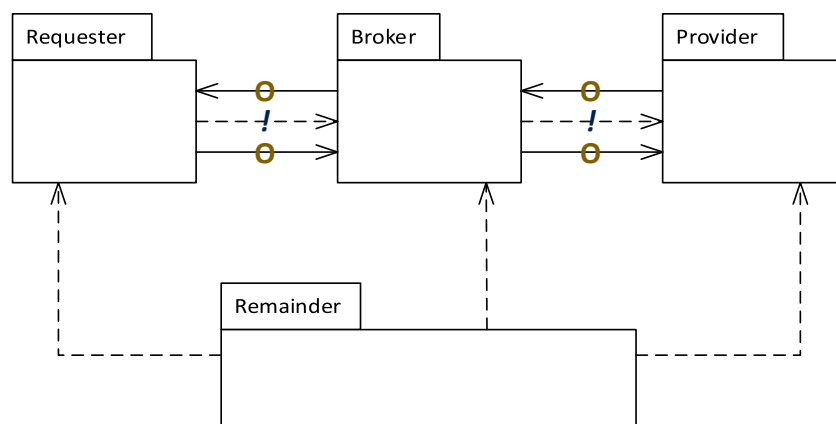


Figure B.6: The Requester Interface interpretation of the Broker pattern.

```

package husacct.analyse.task.reconstruct.patterns;

public class BrokerPattern_RestrictedRemainder extends BrokerPattern {
    // In this variety of the Broker Pattern, the pattern modules are free to call
    // on the Remainder.
    public BrokerPattern_RestrictedRemainder() {
        // TODO Auto-generated constructor stub
    }

    @Override
    protected void defineRules() {
        addSingleRule("Broker", "Requester", "IsTheOnlyModuleAllowedToUse", "
Provider");
        addSingleRule("Requester", "Broker", "IsTheOnlyModuleAllowedToUse", null);
        addSingleRule("Provider", "Broker", "IsTheOnlyModuleAllowedToUse", null);
        addSingleRule("Broker", "Provider", "IsTheOnlyModuleAllowedToUse", "
Requester");
    }
}

```

Code Sample B.16: The source code for the Broker (Restricted Remainder) pattern class.

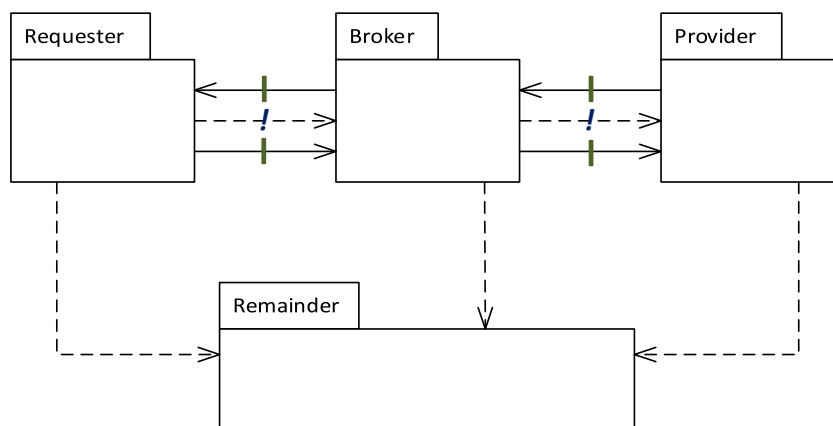


Figure B.7: The Requester Interface interpretation of the Broker pattern.

Appendix C

Class Diagram

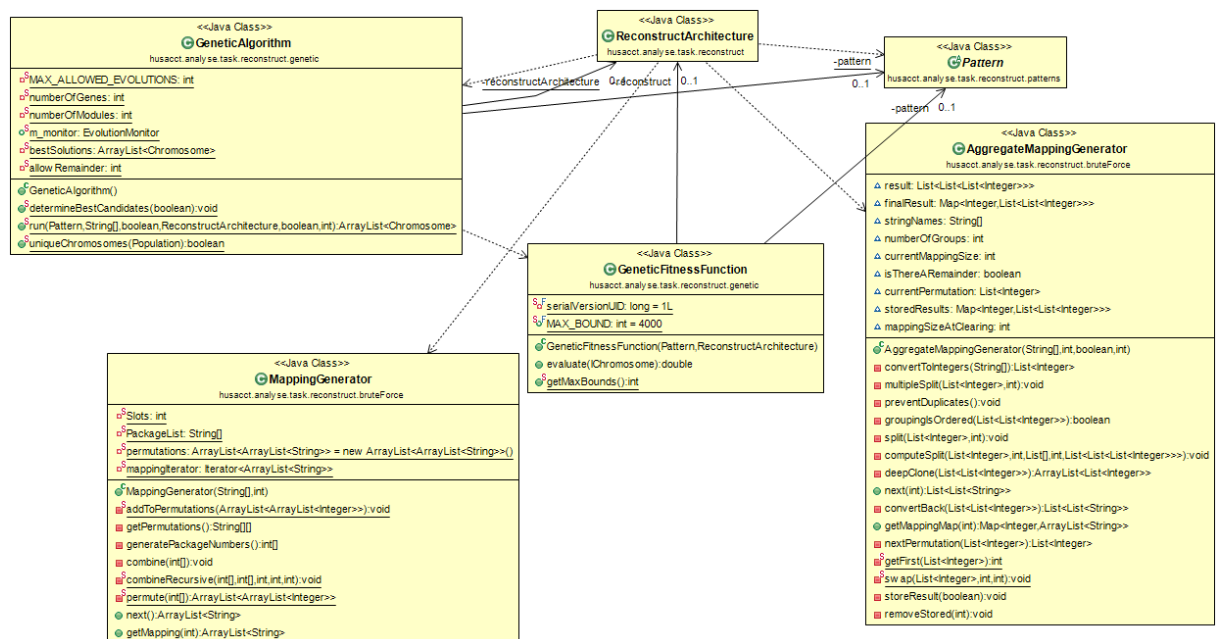


Figure C.1: The class diagram of the Reconstruct package, generated using the ObjectAid UML Explorer plug-in (www.objectaid.com) for the Eclipse IDE.

Appendix D

Additional Code Samples

```
package husacct.analyse.task.reconstruct;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Comparator;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.TreeMap;

import org.apache.log4j.Logger;
import org.jgap.Chromosome;
import org.jgap.Gene;

import husacct.ServiceProvider;
import husacct.analyse.IAnalyseService;
import husacct.analyse.domain.IModelQueryService;
import husacct.analyse.task.reconstruct.bruteForce.AggregateMappingGenerator;
import husacct.analyse.task.reconstruct.bruteForce.MappingGenerator;
import husacct.analyse.task.reconstruct.genetic.GeneticAlgorithm;
import husacct.analyse.task.reconstruct.patterns.LayeredPattern_CompleteFreedom;
import husacct.analyse.task.reconstruct.patterns.Pattern;
import husacct.common.dto.RuleDTO;
import husacct.common.dto.SoftwareUnitDTO;
import husacct.define.DomainToDtoParser;
import husacct.define.IDefineService;
import husacct.define.domain.module.ModuleStrategy;
import husacct.validate.IValidateService;

/** Software Architecture Reconstruction based on architectural patterns (i.e.
    NOT design patterns).
    *
    * @author Joeri Peters */
public class ReconstructArchitecture {

    private final Logger logger = Logger.getLogger(ReconstructArchitecture.class);
    private IModelQueryService queryService;
    private IDefineService defineService;
    private IValidateService validateService;
    private ArrayList<SoftwareUnitDTO> internalRootPackagesWithClasses;
    // The first packages starting from the project root that contain one or more
    // classes.
    // This is the selection of software units for the pattern search.
```

```

// External system variables
private String xLibrariesRootPackage = "xLibraries";
private ArrayList<SoftwareUnitDTO> xLibrariesMainPackages = new ArrayList<
    SoftwareUnitDTO>();
// Layer variables
private TreeMap<Integer, ArrayList<SoftwareUnitDTO>> layers = new TreeMap<
    Integer, ArrayList<SoftwareUnitDTO>>();
private int layerThreshold = 10; // Percentage of allowed violating
    dependencies (back-calls)
    // for adding layers.
private int skipCallThreshold = 10; // Percentage of allowed violating
    dependencies (skip-calls)
    // for partially merging layers.
int betaSquared = 1;
// Beta^2 affects the F-measure-inspired fitness function. betaSquared = 1
    means F1, although it
// is not actually the harmonic mean of recall
// and precision, but of various validation results.

/** The main method for the ReconstructArchitecture class. Currently requires
    hard-coded
    * modification instead of arguments.
    *
    * @param queryService */
@SuppressWarnings("unused")
public ReconstructArchitecture(IModelQueryService queryService) {
    long start = System.currentTimeMillis();
    this.queryService = queryService;
    defineService = ServiceProvider.getInstance().getDefineService();
    validateService = ServiceProvider.getInstance().getValidateService();
    identifyExternalSystems();
    determineInternalRootPackagesWithClasses(0);
    // determineInternalRootPackagesWithClassesIncludingClasses(0);
    // If source code is not well structured in a package hierarchy, including
    individual
    // classes in the root might help. This can make n way too
    // big, though, so be careful if you're taking the brute force approach.
    This may cause
    // memory issues.

    int numberOfTopCandidates = 10; // Only relevant for the brute force
    approach.
    boolean aggregation = true;
    boolean remainder = true; // Not relevant for brute force if aggregation =
    false.
    int generations = 10; // Only relevant for the genetic approach.

    int numberOfLayers = 3; // Only matters for N-Layered patterns.

    logger.info("Number of rules before applying patterns: " + defineService.
        getDefinedRules().length);
    Pattern currentPattern = null;
    currentPattern = new LayeredPattern_CompleteFreedom(numberOfLayers);
    // currentPattern = new LayeredPattern_FreeRemainder(numberOfLayers);
    // currentPattern = new LayeredPattern_IsolatedInternalLayers(numberOfLayers
    );
    // currentPattern = new LayeredPattern_LayerTypes(numberOfLayers);
    // currentPattern = new LayeredPattern_RestrictedRemainder(numberOfLayers);
    // currentPattern = new MVCPattern_CompleteFreedom();
    // currentPattern = new MVCPattern_ModelInterface();
    // currentPattern = new MVCPattern_ControllerInterface();

```

```

// currentPattern = new MVCPattern_FreeRemainder();
// currentPattern = new MVCPattern_RestrictedRemainder();
// currentPattern = new BrokerPattern_CompleteFreedom();
// currentPattern = new BrokerPattern_FreeRemainder();
// currentPattern = new BrokerPattern_RequesterInterface();
// currentPattern = new BrokerPattern_RestrictedRemainder();

// currentPattern = new CentralisedLayering();
// currentPattern = new Model_Viewcontroller();
// currentPattern = new HUSACCT_Analyse();

currentPattern.insertPattern(); // This adds modules and rules (including
exceptions if need
// be) to the intended architecture.
logger.info("Number of rules after applying patterns: " + defineService.
getDefinedRules().length);
if (currentPattern != null) {
    bruteForceApproach(currentPattern, remainder, aggregation,
numberOfTopCandidates);
    // geneticApproach(currentPattern, remainder, generations);
}
System.out.println("Elapsed time: " + ((System.currentTimeMillis() - start)
/ 1000.0) + " seconds.");
}

/** SAR approach using a genetic algorithm to find architectural pattern
candidates.
*
* @param currentPattern
* @param aggregates */
private void geneticApproach(Pattern currentPattern, boolean remainder, int
generations) {
    try {
        String[] patternUnitNames = new String[internalRootPackagesWithClasses.
size()];
        for (int i = 0; i < patternUnitNames.length; i++)
            patternUnitNames[i] = internalRootPackagesWithClasses.get(i).uniqueName;
        ArrayList<Chromosome> bestSolutions = new ArrayList<Chromosome>(10);
        ServiceProvider.getInstance().getControlService().setValidate(true);
        bestSolutions.addAll(GeneticAlgorithm.run(currentPattern, patternUnitNames
, true, this, remainder, generations)); // TODO:
                                                    // print
                                                    // best
                                                    // candidates

        // elsewhere.
        ServiceProvider.getInstance().getControlService().setValidate(false);
        for (int i = 0; i < bestSolutions.size(); i++) {
            if (bestSolutions.get(i).getFitnessValue() - 1 > 0.0) {
                Gene[] genes = bestSolutions.get(i).getGenes();
                System.out.println("Chromosome " + (i + 1) + ": ");
                for (int j = 0; j < genes.length; j++) {
                    System.out.print(genes[j].getAllele().toString());
                }
                System.out.println("\nFitness value: " + (bestSolutions.get(i).
getFitnessValue() - 1));
                if (i == 0) {
                    System.out.println("Placing best candidate in defined architecture
...");
                    int[] bestAlleles = new int[genes.length];
                    for (int j = 0; j < genes.length; j++) {

```

```

        bestAlleles[j] = (int) genes[j].getAllele();
    }
    placeBestSolutionInIntendedArchitecture(currentPattern, bestAlleles)
;
    }
}
} catch (Exception e) {
    e.printStackTrace();
}
}

/** The exhaustive search for architectural pattern candidates as an SAR
    approach.
    *
    * @param currentPattern
    * @param aggregates */
private void bruteForceApproach(Pattern currentPattern, boolean remainder,
    boolean aggregation, int numberOfTopCandidates) {
    String[] patternUnitNames = new String[internalRootPackagesWithClasses.size()];
    for (int i = 0; i < patternUnitNames.length; i++)
        patternUnitNames[i] = internalRootPackagesWithClasses.get(i).uniqueName;
    if (aggregation)
        aggregationBruteForce(currentPattern, numberOfTopCandidates,
            patternUnitNames, remainder);
    else
        simpleBruteForce(currentPattern, numberOfTopCandidates, patternUnitNames);
}

// This is the brute force (try everything!) approach without aggregation.
private void simpleBruteForce(Pattern currentPattern, int
    numberOfTopCandidates, String[] patternUnitNames) {
    MappingGenerator mapgen = new MappingGenerator(patternUnitNames,
        currentPattern.getNumberOfModules());
    ArrayList<String> patternNames = new ArrayList<String>(currentPattern.
        getNumberOfModules());
    double[][] candidateScores = new double[numberOfTopCandidates][2];
    double fitness = 0;
    double lowestTopFitness = 0;
    ServiceProvider.getInstance().getControlService().setValidate(true);
    patternNames = mapgen.next();
    int candidateNumber = 0;
    while (patternNames != null) {
        // This is where validation is requested and a top N is generated.
        currentPattern.mapPattern(patternNames);
        fitness = determineFitnessHarmonic(validatePatternCandidateSimple(
            patternNames));
        lowestTopFitness = keepScore(numberOfTopCandidates, candidateScores,
            fitness, lowestTopFitness, candidateNumber)[0];
        patternNames = mapgen.next();
        System.out.println("Candidate number: " + (candidateNumber + 1));
        candidateNumber++;
    }
    ServiceProvider.getInstance().getControlService().setValidate(false);
    sortCandidates(candidateScores);
    logger.info("Done \n");
    System.out.println("Best " + numberOfTopCandidates + " candidates with non-
    zero fitness score: ");
    ArrayList<String> bestMapping;
    for (int i = 0; i < numberOfTopCandidates; i++) {

```

```

        bestMapping = mapgen.getMapping((int) candidateScores[i][1]);
        if (i < numberOfTopCandidates - 1) {
            if (candidateScores[i][0] > 0.0)
                System.out.println("Fitness score: " + (candidateScores[i][0]) + ".
Mapping: " + bestMapping);
        } else
            System.out.println("Fitness score: " + (candidateScores[i][0]) + ".
Mapping: " + bestMapping);
    }
    bestMapping = mapgen.getMapping((int) candidateScores[numberOfTopCandidates
- 1][1]);
    currentPattern.mapPattern(bestMapping);
    System.out.println("This last mapping was selected for the intended
architecture by default.");
}

private double[] keepScore(int numberOfTopCandidates, double[][]
candidateScores, double fitness, double lowestTopFitness, int i) {
    if (fitness == -1)
        return new double[] { lowestTopFitness };
    candidateScores[0][0] = fitness;
    candidateScores[0][1] = i;
    sortCandidates(candidateScores);
    double oldCandidateNumber = candidateScores[0][1];
    if (oldCandidateNumber > 0) { // Worst of the top mappings needs to be
removed.
        return new double[] { candidateScores[1][0], oldCandidateNumber };
    } else {
        return new double[] { candidateScores[0][0], -1.0 }; // All top mappings
must be kept.
    }
}

// This is the aggregation case of the brute force approach.
private void aggregationBruteForce(Pattern currentPattern, int
numberOfTopCandidates, String[] patternUnitNames, boolean remainder) {
    AggregateMappingGenerator mapCal = new AggregateMappingGenerator(
patternUnitNames, currentPattern.getNumberOfModules(), false,
numberOfTopCandidates);
    Map<Integer, ArrayList<String>> patternMapping = new HashMap<Integer,
ArrayList<String>>();
    // Although this is the brute force approach and not the genetic, the same
genetic encoding
    // is used for the mapping here.
    numberOfTopCandidates++;
    double[][] candidateScores = new double[numberOfTopCandidates][2];
    double fitness = 0;
    double lowestTopFitness = 0;
    ServiceProvider.getInstance().getControlService().setValidate(true);
    int candidateNumber = 0;
    List<List<String>> map = new ArrayList<List<String>>();
    double[] scoreResult;
    map = mapCal.next(-2);
    while (true) {
        if (map == null)
            break;
        for (int i = 0; i < map.size(); i++)
            patternMapping.put(i, (ArrayList<String>) map.get(i));
        currentPattern.mapPatternAllowingAggregates(patternMapping);
        fitness = determineFitnessHarmonic(validatePatternCandidateAggregation(
patternMapping));
    }
}

```

```

        System.out.println("Candidate number: " + (candidateNumber + 1));
        if (fitness >= 0.0) {
            scoreResult = keepScore(numberOfTopCandidates, candidateScores, fitness,
lowestTopFitness, candidateNumber);
            if (scoreResult.length > 1) { // A mapping must be replaced.
                if (scoreResult[1] > -1) { // Unless he is stilling filling up the top
-10.
                    lowestTopFitness = scoreResult[0];
                    map = mapCal.next((int) scoreResult[1]);
                } else
                    map = mapCal.next(-1);
            } else // No mapping has to be replaced.
                map = mapCal.next(-1);
        } else
            map = mapCal.next(-2);
        candidateNumber++;
    }
    sortCandidates(candidateScores);

    // =====Moving
    on to include
    // Remainder

    AggregateMappingGenerator mapCalRemainder = null;
    int remainderNumber = candidateNumber;
    if (remainder) {
        System.out.println("HERE STARTS THE REMAINDER LOOP.");
        if (patternUnitNames.length >= currentPattern.getNumberOfModules() + 1) {
            mapCalRemainder = new AggregateMappingGenerator(patternUnitNames,
currentPattern.getNumberOfModules(), true, numberOfTopCandidates);
            map = mapCalRemainder.next(-2);
            while (true) {
                if (map == null)
                    break;
                for (int i = 0; i < map.size(); i++) {
                    patternMapping.put(i, (ArrayList<String>) map.get(i));
                }
                currentPattern.mapPatternAllowingAggregates(patternMapping);
                fitness = determineFitnessHarmonic(validatePatternCandidateAggregation
(patternMapping));
                System.out.println("Candidate number: " + (candidateNumber + 1));
                if (fitness >= 0.0) {
                    scoreResult = keepScore(numberOfTopCandidates, candidateScores,
fitness, lowestTopFitness, candidateNumber);
                    if (scoreResult.length > 1) { // A mapping must be replaced.
                        if (scoreResult[1] > -1) { // Unless he is stilling filling up the
// top-10.
                            if (scoreResult[1] < remainderNumber) {
                                lowestTopFitness = scoreResult[0];
                                map = mapCalRemainder.next(-1);
                            } else {
                                lowestTopFitness = scoreResult[0];
                                map = mapCalRemainder.next((int) scoreResult[1] -
remainderNumber);
                            }
                        } else
                            map = mapCalRemainder.next(-1);
                    } else // No mapping has to be replaced.
                        map = mapCalRemainder.next(-1);
                } else
                    map = mapCalRemainder.next(-2);
            }
        }
    }

```

```

        candidateNumber++;
    }
    sortCandidates(candidateScores);
}
}
ServiceProvider.getInstance().getControlService().setValidate(false);
logger.info("Done \n");
System.out.println("Best " + (numberOfTopCandidates - 1) + " candidates with
non-zero fitness score: ");

ArrayList<ArrayList<ArrayList<String>>> newBestMappings = new ArrayList<
ArrayList<ArrayList<String>>>(numberOfTopCandidates);
for (int i = 0; i < numberOfTopCandidates; i++) {
    newBestMappings.add(i, new ArrayList<ArrayList<String>>());
    if (i == 0 && remainder == false)
        continue;
    if (candidateScores[i][0] > 0) {
        if ((int) candidateScores[i][1] >= remainderNumber) {
            for (int j = 0; j < currentPattern.getNumberOfModules(); j++)
                newBestMappings.get(i).add(j, mapCalRemainder.getMappingMap((int)
candidateScores[i][1] - remainderNumber + 1).get(j));
        } else {
            for (int j = 0; j < currentPattern.getNumberOfModules(); j++)
                newBestMappings.get(i).add(j, mapCal.getMappingMap((int)
candidateScores[i][1] + 1).get(j));
        }
        if (i > 0)
            System.out.println("Fitness: " + candidateScores[i][0] + " --> Mapping
: " + newBestMappings.get(i).toString());
    }
}
if (newBestMappings.get(numberOfTopCandidates - 1).isEmpty()) {
    logger.info("No suitable candidate with non-zero fitness scores were found
");
} else {
    HashMap<Integer, ArrayList<String>> bestCandidate = new HashMap<Integer,
ArrayList<String>>();
    for (int i = 0; i < currentPattern.getNumberOfModules(); i++) {
        bestCandidate.put(i, newBestMappings.get(numberOfTopCandidates - 1).get(
i));
    }
    currentPattern.mapPatternAllowingAggregates(bestCandidate);
    logger.info("This last mapping was selected for the intended architecture
by default.");
}
}

/** Fitness scores are calculated for a chromosome. This allows for
aggregation (multiple SUs in
* one pattern module).
*
* @param pattern
* @param alleles
* @return */
public double getFitnessScore(Pattern pattern, int[] alleles) {
    Map<Integer, ArrayList<String>> patternUnitNames = new HashMap<Integer,
ArrayList<String>>();
    for (int i = 0; i < alleles.length; i++) {
        if (alleles[i] != 0) {
            ArrayList<String> temp = new ArrayList<String>(1);
            if (patternUnitNames.get(alleles[i] - 1) != null) {

```

```

        temp = patternUnitNames.get(alleles[i] - 1);
        temp.add(internalRootPackagesWithClasses.get(i).uniqueName);
        patternUnitNames.put(alleles[i] - 1, temp);
    } else {
        temp.add(internalRootPackagesWithClasses.get(i).uniqueName);
        patternUnitNames.put(alleles[i] - 1, temp);
    }
}
// else
// System.out.println("Remainder: " +
// internalRootPackagesWithClasses.get(i).uniqueName);
}
if (patternUnitNames.keySet().size() == pattern.getNumberOfModules()) {
    pattern.mapPatternAllowingAggregates(patternUnitNames);
    return determineFitnessHarmonic(validatePatternCandidateAggregation(
patternUnitNames)) + 1;
} else
    return -1;
}

// This calculates the fitness function, which is defined as follows:
//  $f(N_d, N_d-m, N_d-m, v, N_d, m, \beta) = 2 * (1 - N_d-m, v / N_d-m) * N_d, m / N_d) / (1 - N_d-m, v / N_d-m) + N_d, m / N_d)$ 
// This is the harmonic mean of the two ratios, meaning it is a balance
// between the number of
// violations over the number of non-essential
// dependencies and the number of essential dependencies explained by the
// pattern.
// It is inspired by the F-measure, although that is not what it is since this
// fitness score is
// not about true/false negatives/positives.
private double determineFitnessHarmonic(int[] validation) {
    if (validation == null)
        return -1; // In case of excluded candidate.
    double sum = 0;
    sum += validation[1];
    double nvScore = 1 - sum / (validation[0] - validation[2]); // nvScore is
// based on the
// relative number of
// violations.

    if (!(nvScore >= 0))
        nvScore = 1;
    double nmScore = (double) validation[2] / validation[0];
    try {
        return (1 + betaSquared) * nvScore * nmScore / (betaSquared * nvScore +
nmScore);
    } catch (Exception e) {
        if (betaSquared <= 0)
            System.out.println("Beta^2 is negative: " + betaSquared);
        else
            System.out.println("Unknown exception.");
        return 0.0;
    }
}

// This calculates the numbers necessary for determining the fitness score in
// the
// non-aggregation case.
private int[] validatePatternCandidateSimple(ArrayList<String> patternNames) {
    // int numberOfViolations = new int[7];

```



```

int numberOfViolations = 0;
int numberOfMustUseAffirmations = 0;
int i = 6;
validateService.checkConformance();
for (RuleDTO currentAppliedRule : defineService.getDefinedRules()) {
    i = determineCategoryIndex(currentAppliedRule.ruleTypeKey);
    if (i == 0) { // If the number of MustUse violations is zero, that's good.
        if (validateService.getViolationsByRule(currentAppliedRule).length != 0)
        {
            return null;
        } else {
            for (String name : patternNames) {
                if (currentAppliedRule.moduleFrom.logicalPath.equals(defineService.
getModule_BasedOnSoftwareUnitName(name).logicalPath)) {
                    for (String other : patternNames) {
                        if (currentAppliedRule.moduleTo.logicalPath
                            .equals(defineService.getModule_BasedOnSoftwareUnitName(
other).logicalPath)) {
                            numberOfMustUseAffirmations +=
getNumberOfDependenciesBetweenSoftwareUnits(name, other);
                        }
                    }
                }
            }
        } else if (i != 6)
        try {
            // numberOfViolations[i] +=
            // validateService.getViolationsByRule(currentAppliedRule).length;
            numberOfViolations += validateService.getViolationsByRule(
currentAppliedRule).length;
        } catch (Exception e) {
            System.out.println("Problem counting violations of: " +
currentAppliedRule);
        }
    }
    int[] results = new int[3];
    int numberOfDependencies = 0;
    ArrayList<String> remainderUnits = new ArrayList<String>(
internalRootPackagesWithClasses.size());
    for (SoftwareUnitDTO unit : internalRootPackagesWithClasses) {
        if (!patternNames.contains(unit.uniqueName))
            remainderUnits.add(unit.uniqueName);
    }
    for (String name : patternNames) {
        for (String otherName : remainderUnits) {
            numberOfDependencies += getNumberOfDependenciesBetweenSoftwareUnits(name
, otherName);
            numberOfDependencies += getNumberOfDependenciesBetweenSoftwareUnits(
otherName, name);
        }
        for (String otherPatternName : patternNames) {
            if (!name.equals(otherPatternName))
                numberOfDependencies += getNumberOfDependenciesBetweenSoftwareUnits(
name, otherPatternName);
        }
    }
    results[0] = numberOfDependencies;
    // The total number of relevant dependencies but "Must use".
    results[1] = numberOfViolations; // The total number of violations of the
above

```

```

        // dependencies, sorted per rule type.
        results[2] = numberOfMustUseAffirmations; // The total number of correct
        dependency
        // instances of the "Must use" rule type.
    }
    return results;
}

// This calculates the numbers necessary for determining the fitness score in
// the aggregation
// case.
private int[] validatePatternCandidateAggregation(Map<Integer, ArrayList<
String>> patternUnitNames) {
    // int[] numberOfViolations = new int[7];
    int numberOfViolations = 0;
    int numberOfMustUseAffirmations = 0;
    int i = 6;
    validateService.checkConformance();
    for (RuleDTO currentAppliedRule : defineService.getDefinedRules()) {
        i = determineCategoryIndex(currentAppliedRule.ruleTypeKey);
        if (i == 0) { // If the number of MustUse violations is zero, that's good.
            if (validateService.getViolationsByRule(currentAppliedRule).length != 0)
            {
                return null;
            }
            else {
                for (int j = 0; j < patternUnitNames.size(); j++) {
                    for (String name : patternUnitNames.get(j)) {
                        if (currentAppliedRule.moduleFrom.logicalPath.equals(defineService
.getModule_BasedOnSoftwareUnitName(name).logicalPath)) {
                            for (int k = 0; k < patternUnitNames.size(); k++) {
                                if (j != k) {
                                    for (String otherName : patternUnitNames.get(k)) {
                                        if (currentAppliedRule.moduleTo.logicalPath
.equals(defineService
.getModule_BasedOnSoftwareUnitName(otherName).logicalPath)) {
                                            numberOfMustUseAffirmations +=
getNumberOfDependenciesBetweenSoftwareUnits(name, otherName);
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
        else if (i != 6) {
            try {
                numberOfViolations += validateService.getViolationsByRule(
currentAppliedRule).length;
            } catch (Exception e) {
                System.out.println("Problem counting violations of: " +
currentAppliedRule);
            }
        }
    }
    boolean contains = false;
    ArrayList<String> remainderUnits = new ArrayList<String>(
internalRootPackagesWithClasses.size());
    for (SoftwareUnitDTO unit : internalRootPackagesWithClasses) {
        for (ArrayList<String> value : patternUnitNames.values()) {
            if (value.contains(unit.uniqueName)) {
                contains = true;
            }
        }
    }
}

```

```

        break;
    }
}
if (contains == false)
    remainderUnits.add(unit.uniqueName());
}
int[] results = new int[3];
int numberOfDependencies = 0;
for (int j = 0; j < patternUnitNames.size(); j++) {
    for (String name : patternUnitNames.get(j)) {
        for (int k = 0; k < patternUnitNames.size(); k++) {
            if (j != k) {
                for (String otherPatternName : patternUnitNames.get(k)) {
                    if (!name.equals(otherPatternName))
                        numberOfDependencies +=
getNumberOfDependenciesBetweenSoftwareUnits(name, otherPatternName);
                }
            }
        }
        for (String otherName : remainderUnits) {
            numberOfDependencies += getNumberOfDependenciesBetweenSoftwareUnits(
name, otherName);
            numberOfDependencies += getNumberOfDependenciesBetweenSoftwareUnits(
otherName, name);
        }
    }
}
results[0] = numberOfDependencies;
// The total number of relevant dependencies but "Must use".
results[1] = numberOfViolations; // The total number of violations of the
above
                                // dependencies, sorted per rule type.
results[2] = numberOfMustUseAffirmations; // The total number of correct
dependency
                                // instances of the "Must use" rule type.
return results;
}

//
// /** A particular pattern candidate is validated while allowing for multiple
// software units
// assigned to the same pattern module.
// *
// * @param patternUnitNames
// * @return */
// private int[][] validatePatternCandidateAllowingAggregates(Map<Integer,
// ArrayList<String>>
// patternUnitNames) {
// int[][] results = new int[2][6];
// validateService.checkConformance();
// int[] numberOfViolations = new int[6];
// int i = 6;
// for (RuleDTO currentAppliedRule : defineService.getDefinedRules()) {
// i = determineCategoryIndex(currentAppliedRule.ruleTypeKey);
// if (i == 0) {
// if (validateService.getViolationsByRule(currentAppliedRule).length != 0) {
// logger.info("Candidate was excluded due to violation of MustUse rule(s)");
// results[0] = null;
// results[1] = null;
// return results;
// }
}

```

```

// } else if (i != 6)
// numberOfViolations[i] += validateService.getViolationsByRule(
//     currentAppliedRule).length;
// }
// int[] totalNumberOfDependencies = new int[1];
// for (int j = 0; j < patternUnitNames.size(); j++) {
// for (String name : patternUnitNames.get(j)) {
// for (int k = 0; k < patternUnitNames.size(); k++) {
// for (String otherName : patternUnitNames.get(k)) {
// if (j != k) {
// if (name != otherName)
// totalNumberOfDependencies[0] += getNumberOfDependenciesBetweenSoftwareUnits
//     (name, otherName);
// }
// }
// }
// }
// }
// // logger.info("Number of dependencies within pattern: " +
//     totalNumberOfDependencies[0] + ",
// resulting in a total of "
// // + IntStream.of(numberOfViolations).sum() + " violations.");
// results[0] = totalNumberOfDependencies;
// results[1] = numberOfViolations;
// return results;
// }

/** Determine the rule type of a rule.
 *
 * @param currentRuleType
 * @return */
private int determineCategoryIndex(String currentRuleType) {
    if (currentRuleType == "MustUse")
        return 0;
    else if (currentRuleType.equalsIgnoreCase("IsNotAllowedToMakeBackCall"))
        return 1;
    else if (currentRuleType.equalsIgnoreCase("IsNotAllowedToMakeSkipCall"))
        return 2;
    else if (currentRuleType.equalsIgnoreCase("IsNotAllowedToUse"))
        return 3;
    else if (currentRuleType.equalsIgnoreCase("IsOnlyAllowedToUse"))
        return 4;
    else if (currentRuleType.equalsIgnoreCase("IsTheOnlyModuleAllowedToUse"))
        return 5;
    else
        return 6;
}

/** Sorting two candidates based on their fitness scores. This uses a custom
    Comparator.
 *
 * @param candidateScores */
private void sortCandidates(double[][] candidateScores) {
    Arrays.sort(candidateScores, new Comparator<double[]>() {

        @Override
        public int compare(double[] o1, double[] o2) {
            double fitness1 = o1[0];
            double fitness2 = o2[0];
            return Double.compare(fitness1, fitness2);
        }
    });
}

```

```

    });
}

private int getNumberOfDependenciesBetweenSoftwareUnits(String fromUnit,
String toUnit) {
    IAnalyseService analyseService = ServiceProvider.getInstance().
getAnalyseService();
    return analyseService.getDependenciesFromSoftwareUnitToSoftwareUnit(fromUnit
, toUnit).length;
}

/** Identify the external libraries within the source code. */
private void identifyExternalSystems() {
    // Create module "ExternalSystems"
    ArrayList<SoftwareUnitDTO> emptySoftwareUnitsArgument = new ArrayList<
SoftwareUnitDTO>();
    defineService.addModule("ExternalSystems", "**", "ExternalLibrary", 0,
emptySoftwareUnitsArgument);
    // Create a module for each childUnit of xLibrariesRootPackage
    int nrOfExternalLibraries = 0;
    for (SoftwareUnitDTO mainUnit : queryService.getChildUnitsOfSoftwareUnit(
xLibrariesRootPackage)) {
        xLibrariesMainPackages.add(mainUnit);
        ArrayList<SoftwareUnitDTO> softwareUnitsArgument = new ArrayList<
SoftwareUnitDTO>();
        softwareUnitsArgument.add(mainUnit);
        defineService.addModule(mainUnit.name, "ExternalSystems", "ExternalLibrary
", 0, softwareUnitsArgument);
        nrOfExternalLibraries++;
    }
    logger.info(" Number of added ExternalLibraries: " + nrOfExternalLibraries);
}

/** Determine which packages form the root of the source code hierarchy,
excluding single
* classes. If there is just a single package in that root, this package
becomes the root and
* packages are identified within it. */
private void determineInternalRootPackagesWithClasses(int level) {
    internalRootPackagesWithClasses = new ArrayList<SoftwareUnitDTO>();
    SoftwareUnitDTO[] allRootUnits = queryService.getSoftwareUnitsInRoot(); //
Get all root

                                // units
    for (SoftwareUnitDTO rootModule : allRootUnits) {
        if (!rootModule.uniqueName.equals(xLibrariesRootPackage)) { // Get all
root units that

                                // are not libraries
        for (String internalPackage : queryService.getRootPackagesWithClass(
rootModule.uniqueName)) {
            // Get root packages
            internalRootPackagesWithClasses.add(queryService.
getSoftwareUnitByUniqueName(internalPackage));
        }
    }
}

if (internalRootPackagesWithClasses.size() == 1) {
    // Temporal solution useful for HUSACCT20 test. To be improved!
    // E.g., classes in root are excluded from the process.
    String newRoot = internalRootPackagesWithClasses.get(0).uniqueName;
    internalRootPackagesWithClasses = new ArrayList<SoftwareUnitDTO>();
}

```

```

        for (SoftwareUnitDTO child : queryService.getChildUnitsOfSoftwareUnit(
newRoot)) {
            if (child.type.equalsIgnoreCase("package"))
                internalRootPackagesWithClasses.add(child);
        }
        if (level == 1) {
            ArrayList<SoftwareUnitDTO> temp = new ArrayList<SoftwareUnitDTO>();
            for (int i = 0; i < internalRootPackagesWithClasses.size(); i++) {
                newRoot = internalRootPackagesWithClasses.get(i).uniqueName;
                for (SoftwareUnitDTO child : queryService.getChildUnitsOfSoftwareUnit(
newRoot)) {
                    if (child.type.equalsIgnoreCase("package"))
                        temp.add(child);
                }
            }
            internalRootPackagesWithClasses = new ArrayList<SoftwareUnitDTO>(temp);
        }
    }
}

/** Determine which packages form the root of the source code hierarchy,
including single
* classes. If there is just a single package in that root, this package
becomes the root and
* packages are identified within it.
*
* @param level */
@SuppressWarnings("unused")
private void determineInternalRootPackagesWithClassesIncludingClasses(int
level) {
    internalRootPackagesWithClasses = new ArrayList<SoftwareUnitDTO>();
    SoftwareUnitDTO[] allRootUnits = queryService.getSoftwareUnitsInRoot(); //
    Get all root

                                // units
    for (SoftwareUnitDTO rootModule : allRootUnits) {
        if (!rootModule.uniqueName.equals(xLibrariesRootPackage)) { // Get all
root units that

                                // are not libraries
        for (String internalPackage : queryService.getRootPackagesWithClass(
rootModule.uniqueName)) { // Get

                                // root
                                // packages
            internalRootPackagesWithClasses.add(queryService.
getSoftwareUnitByUniqueName(internalPackage));
        }
    }
}
if (internalRootPackagesWithClasses.size() == 1) {
    // Temporal solution useful for HUSACCT20 test. To be improved!
    // E.g., classes in root are excluded from the process.
    String newRoot = internalRootPackagesWithClasses.get(0).uniqueName;
    internalRootPackagesWithClasses = new ArrayList<SoftwareUnitDTO>();
    for (SoftwareUnitDTO child : queryService.getChildUnitsOfSoftwareUnit(
newRoot)) {
        internalRootPackagesWithClasses.add(child);
    }
    if (level == 1) {
        ArrayList<SoftwareUnitDTO> temp = new ArrayList<SoftwareUnitDTO>();
        for (int i = 0; i < internalRootPackagesWithClasses.size(); i++) {
            newRoot = internalRootPackagesWithClasses.get(i).uniqueName;

```

```

        for (SoftwareUnitDTO child : queryService.getChildUnitsOfSoftwareUnit(
newRoot)) {
            temp.add(child);
        }
    }
    internalRootPackagesWithClasses = new ArrayList<SoftwareUnitDTO>(temp);
}
}

// At the end, you want the very best solution to actually be placed in the
// intended
// architecture again, so that's what happens here.
private void placeBestSolutionInIntendedArchitecture(Pattern currentPattern,
int[] bestAlleles) {
    Map<Integer, ArrayList<String>> patternUnitNames = new HashMap<Integer,
ArrayList<String>>();
    for (int i = 0; i < bestAlleles.length; i++) {
        if (bestAlleles[i] != 0) {
            ArrayList<String> temp = new ArrayList<String>(1);
            if (patternUnitNames.get(bestAlleles[i] - 1) != null) {
                temp = patternUnitNames.get(bestAlleles[i] - 1);
                temp.add(internalRootPackagesWithClasses.get(i).uniqueName);
                patternUnitNames.put(bestAlleles[i] - 1, temp);
            } else {
                temp.add(internalRootPackagesWithClasses.get(i).uniqueName);
                patternUnitNames.put(bestAlleles[i] - 1, temp);
            }
        }
    }
    if (patternUnitNames.keySet().size() == currentPattern.getNumberOfModules())
    {
        currentPattern.mapPatternAllowingAggregates(patternUnitNames);
        validatePatternCandidateAggregation(patternUnitNames);
        System.out.println("Best candidate was successfully mapped and validated.");
    }
    else
        System.out.println("Failed to map best candidate");
}

// private void identifyComponents() {
//
// }
//
// private void identifySubSystems() {
//
// }
//
// private void IdentifyAdapters() { // Here, and adapter is a module with a
// IsTheOnlyModuleAllowedToUse rule.
//
// }
//
// private void createModule() {
//
// }

@SuppressWarnings("unused")
private void createRule(ModuleStrategy moduleTo, ModuleStrategy moduleFrom,
String ruleType) {
    DomainToDtoParser domainParser = new DomainToDtoParser();

```

```

        defineService.addRule(new RuleDTO(ruleType, true, domainParser.parseModule(
            moduleTo), domainParser.parseModule(moduleFrom), new String[0], "",
            null, false));
    }

    /** Automatic layering algorithm, to be improved. */
    @SuppressWarnings("unused")
    private void identifyLayers() {
        // 1) Assign all internalRootPackages to bottom layer
        int layerId = 1;
        ArrayList<SoftwareUnitDTO> assignedUnits = new ArrayList<SoftwareUnitDTO>();
        assignedUnits.addAll(internalRootPackagesWithClasses);
        layers.put(layerId, assignedUnits);

        // 2) Identify the bottom layer. Look for packages with dependencies to
        // external systems only.
        identifyTopLayerBasedOnUnitsInBottomLayer(layerId);

        // 3) Look iteratively for packages on top of the bottom layer, et
        // cetera.
        while (layers.lastKey() > layerId) {
            layerId++;
            identifyTopLayerBasedOnUnitsInBottomLayer(layerId);
        }
        // mergeLayersPartiallyBasedOnSkipCallAvoidance();
        // Extra step to minimise the number of skip-calls by moving problematic
        // modules to lower layers.

        // 4) Add the layers to the intended architecture
        int highestLevelLayer = layers.size();
        if (highestLevelLayer > 1) {
            // Reverse the layer levels. The numbering of the layers within the
            // intended architecture is different: the highest level layer has
            // hierarchicalLevel = 1
            int lowestLevelLayer = 1;
            int raise = highestLevelLayer - lowestLevelLayer;
            TreeMap<Integer, ArrayList<SoftwareUnitDTO>> tempLayers = new TreeMap<
                Integer, ArrayList<SoftwareUnitDTO>>();
            for (int i = lowestLevelLayer; i <= highestLevelLayer; i++) {
                ArrayList<SoftwareUnitDTO> unitsOfLayer = layers.get(i);
                int level = lowestLevelLayer + raise;
                tempLayers.put(level, unitsOfLayer);
                raise--;
            }
            layers = tempLayers;
            for (Integer hierarchicalLevel : layers.keySet()) {
                defineService.addModule("Layer" + hierarchicalLevel, "**", "Layer",
                    hierarchicalLevel, layers.get(hierarchicalLevel));
            }
        }
        logger.info(" Number of added Layers: " + layers.size());
    }

    private void identifyTopLayerBasedOnUnitsInBottomLayer(int bottomLayerId) {
        ArrayList<SoftwareUnitDTO> assignedUnitsOriginalBottomLayer = layers.get(
            bottomLayerId);
        @SuppressWarnings("unchecked")
        ArrayList<SoftwareUnitDTO> assignedUnitsBottomLayerClone = (ArrayList<
            SoftwareUnitDTO>) assignedUnitsOriginalBottomLayer.clone();
        ArrayList<SoftwareUnitDTO> assignedUnitsNewBottomLayer = new ArrayList<
            SoftwareUnitDTO>();
    }

```



```

    ArrayList<SoftwareUnitDTO> assignedUnitsTopLayer = new ArrayList<
SoftwareUnitDTO>();
    for (SoftwareUnitDTO softwareUnit : assignedUnitsOriginalBottomLayer) {
        boolean rootPackageDoesNotUseOtherPackage = true;
        for (SoftwareUnitDTO otherSoftwareUnit : assignedUnitsBottomLayerClone) {
            if (!otherSoftwareUnit.uniqueName.equals(softwareUnit.uniqueName)) {
                int nrOfDependenciesFromSoftwareUnitToOther = queryService.
getDependenciesFromSoftwareUnitToSoftwareUnit(softwareUnit.uniqueName,
otherSoftwareUnit.uniqueName).length;
                int nrOfDependenciesFromOtherToSoftwareUnit = queryService
.getDependenciesFromSoftwareUnitToSoftwareUnit(otherSoftwareUnit.
uniqueName, softwareUnit.uniqueName).length;
                if ((nrOfDependenciesFromSoftwareUnitToOther > ((
nrOfDependenciesFromOtherToSoftwareUnit / 100) * layerThreshold)) {
                    rootPackageDoesNotUseOtherPackage = false;
                }
            }
        }
        if (rootPackageDoesNotUseOtherPackage) { // Leave unit in the lower
// layer
            assignedUnitsNewBottomLayer.add(softwareUnit);
        } else { // Assign unit to the higher layer
            assignedUnitsTopLayer.add(softwareUnit);
        }
    }
    if ((assignedUnitsTopLayer.size() > 0) && (assignedUnitsNewBottomLayer.size
() > 0)) {
        layers.remove(bottomLayerId);
        layers.put(bottomLayerId, assignedUnitsNewBottomLayer);
        bottomLayerId++;
        layers.put(bottomLayerId, assignedUnitsTopLayer);
    }
}

@SuppressWarnings("unused")
private void mergeLayersPartiallyBasedOnSkipCallAvoidance() {
    for (int currentLayerID = layers.size(); currentLayerID > 2; currentLayerID
--) {
        ArrayList<SoftwareUnitDTO> unitsInCurrentLayer = layers.get(currentLayerID
);
        ArrayList<SoftwareUnitDTO> unitsInCurrentLowerLayer;
        ArrayList<SoftwareUnitDTO> assignedUnitsNewLowerLayer = new ArrayList<
SoftwareUnitDTO>();
        ArrayList<SoftwareUnitDTO> assignedUnitsNewUpperLayer = new ArrayList<
SoftwareUnitDTO>();
        for (SoftwareUnitDTO softwareUnit : unitsInCurrentLayer) {
            int nrOfDependenciesFromSoftwareUnitToOtherWithinLayer = 0;
            int nrOfSkipCallsFromSoftwareUnitToOtherLayers = 0;
            for (SoftwareUnitDTO otherSoftwareUnit : unitsInCurrentLayer) {
                nrOfDependenciesFromSoftwareUnitToOtherWithinLayer += queryService
.getDependenciesFromSoftwareUnitToSoftwareUnit(softwareUnit.
uniqueName, otherSoftwareUnit.uniqueName).length;
            }
            for (int currentLowerLayerID = currentLayerID - 2; currentLowerLayerID <
0; currentLowerLayerID--) {
                unitsInCurrentLowerLayer = layers.get(currentLowerLayerID);
                for (SoftwareUnitDTO lowerSoftwareUnit : unitsInCurrentLowerLayer) {
                    if (!lowerSoftwareUnit.uniqueName.equals(softwareUnit.uniqueName)) {
                        nrOfSkipCallsFromSoftwareUnitToOtherLayers += queryService
.getDependenciesFromSoftwareUnitToSoftwareUnit(softwareUnit.
uniqueName, lowerSoftwareUnit.uniqueName).length;

```

```

        }
    }
    }
    if (nrOfSkipCallsFromSoftwareUnitToOtherLayers < ((
nrOfDependenciesFromSoftwareUnitToOtherWithinLayer / 100) *
skipCallThreshold)) {
        assignedUnitsNewUpperLayer.add(softwareUnit);
        // Keep in current layer.
    } else {
        assignedUnitsNewLowerLayer.add(softwareUnit);
        // Move to one layer below.
    }
}
layers.remove(currentLayerID);
if (assignedUnitsNewLowerLayer.size() > 0) {
    logger.info(" Number of modules moved downwards: " +
assignedUnitsNewLowerLayer);
    layers.put(currentLayerID, assignedUnitsNewUpperLayer);
    layers.get(currentLayerID - 1).addAll(assignedUnitsNewLowerLayer);
}
}
}
}
}

```

Code Sample D.1: The main reconstruction class. Currently has a very low cohesion and high coupling to other classes, in line with the fact that this is an experimental setup and not a finalised product.

```

package husacct.analyse.task.reconstruct.bruteForce;

import java.util.ArrayList;
import java.util.Iterator;

// This MappingGenerator provides you with all the permutations by which a group
// of size N can be arranged into groups of size r.
// This is simply a multiplication of the number of combinations in one respect,
// the number of ways N can be split into groups of size r,
// times the number of combinations in the other respect, namely the number of
// ways a group of size r can be arranged.
// These permutations can be used for a "brute force" approach to pattern
// matching, where all possible mappings are considered.

public class MappingGenerator {

    private static int Slots;
    private static String[] PackageList;
    private static ArrayList<ArrayList<String>> permutations = new ArrayList<
        ArrayList<String>>();
    private static Iterator<ArrayList<String>> mappingIterator;

    public MappingGenerator(String[] packageList, int numberOfModules) {
        Slots = numberOfModules;
        PackageList = packageList;
        getPermutations();
        mappingIterator = permutations.iterator();
    }

    private static void addToPermutations(ArrayList<ArrayList<Integer>> arrayList)
    {
        for (int i = 0; i < arrayList.size(); i++) {

```

```

        ArrayList<String> temp = new ArrayList<String>(Slots);
        for (int j = 0; j < Slots; j++) {
            temp.add(PackageList[arrayList.get(i).get(j)]);
        }
        permutations.add(temp);
    }
}

private String[][] getPermutations() {
    combine(generatePackageNumbers());
    String[][] permutationsArray = new String[permutations.size()][Slots];
    for (int i = 0; i < permutationsArray.length; i++) {
        permutationsArray[i] = permutations.get(i).toArray(new String[permutations
            .get(i).size()]);
    }
    return permutationsArray;
}

private int[] generatePackageNumbers() {
    int[] packageNumbers = new int[PackageList.length];
    for (int i = 0; i < PackageList.length; i++) {
        packageNumbers[i] = i;
    }
    return packageNumbers;
}

private void combine(int[] packageNumbers) {
    int[] combinations = new int[Slots];
    combineRecursive(packageNumbers, combinations, 0, 0, Slots);
}

private static void combineRecursive(int[] array, int[] result, int
    currentIndex, int level, int r) {
    if (level == r) {
        addToPermutations(permute(result));
        return;
    }
    for (int i = currentIndex; i < array.length; i++) {
        result[level] = array[i];
        combineRecursive(array, result, i + 1, level + 1, r);
        if (i < array.length - 1 && array[i] == array[i + 1]) {
            i++;
        }
    }
}

private static ArrayList<ArrayList<Integer>> permute(int[] combinations) {
    ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();
    result.add(new ArrayList<Integer>());
    for (int i = 0; i < combinations.length; i++) {
        ArrayList<ArrayList<Integer>> current = new ArrayList<ArrayList<Integer>
            >>();
        for (ArrayList<Integer> l : result) {
            for (int j = 0; j < l.size() + 1; j++) {
                l.add(j, combinations[i]);
                ArrayList<Integer> temp = new ArrayList<Integer>(l);
                current.add(temp);
                l.remove(j);
            }
        }
        result = new ArrayList<ArrayList<Integer>>(current);
    }
}

```

```

    }
    return result;
}

public ArrayList<String> next() {
    if (mappingIterator.hasNext()) {
        return (ArrayList<String>) mappingIterator.next();
    } else
        return null;
}

public ArrayList<String> getMapping(int i) {
    return (ArrayList<String>) permutations.get(i);
}
}

```

Code Sample D.2: The mapping generator for non-aggregation mappings of the brute force approach.

```

package husaccttest.analyse;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotNull;
import static org.junit.Assert.assertTrue;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

import javax.swing.plaf.synth.SynthSeparatorUI;

import org.junit.BeforeClass;
import org.junit.Test;

import husacct.analyse.task.reconstruct.bruteForce.MappingGenerator;

public class MappingGeneratorTest {

    private static int testSlots;
    private static String[] testPackageList;

    @BeforeClass
    public static void setUp() {
        testSlots = 3;
        testPackageList = new String[] { "package1", "package2", "package3", "package4", "package5", "package6", "package7" };
    }

    @Test
    public void testNumberOfPermutations() {
        MappingGenerator mg = new MappingGenerator(testPackageList, testSlots);
        List<String> singleMapping = new ArrayList<String>();
        int countN = 0;
        assertNotNull(mg.next());
        countN++;
        ArrayList<List<String>> mappingList = new ArrayList<List<String>>();
        while (true) {
            singleMapping = mg.next();

```

```

        if (singleMapping == null) {
            // System.out.println("NO MORE MAPPINGS LEFT");
            // System.out.println(countN);
            break;
        } else {
            assertTrue("MAPPING LIST CONTAINS DUPLICATES", !mappingList.contains(
singleMapping));
            mappingList.add(singleMapping);
            // System.out.println(singleMapping);
            countN++;
        }
    }
    assertEquals("INCORRECT NUMBER OF PERMUTATIONS: ", factorial(testPackageList
.length) / factorial(testPackageList.length - testSlots), countN);
}

private static int factorial(int n) {
    int fact = 1;
    for (int i = 1; i <= n; i++)
        fact *= i;
    return fact;
}
}

```

Code Sample D.3: Test code for the mapping generator.

```

package husacct.analyse.task.reconstruct.bruteForce;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collection;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

// Calculating all possible mappings in the aggregation case is a bit of a pain.
// The current implementation can undoubtedly be sped up, possibly
// by the use of sets instead of lists whenever we don't care about ordering.
// This I reserve for future improvements. The crucial step in slowing down
// the pattern matching approach is not the mapping generation, anyway. Earlier
// version of this mapper tended to run into heap space problem due to
// the large amounts of data being stored. Now, however, it works as follows:
// the given set of software unit names is converted to integers in a
// reverse ordering, e.g. (5,4,3,2,1). This list is then split into the number
// of groups equal to the amount of pattern modules (+1 in case of a
// Remainder). After that, the mapping is excluded if the integers within a
// single sublist (a pattern module) are not in order. TODO: rewrite this
// algorithm so that this step is unnecessary. After that, subsequent
// permutations of the original list are used to regroup the integers, only
// accepting ordered results and thus preventing duplicates. The permutation
// ends when the completely ordered list, (1,2,3,4,5), is found. Because
// this guarantees that no new mapping will be the same as an earlier one, there
// is no need to store all mappings in a variable. This circumvents the
// heap space problem. Perhaps there is a simpler way of doing this, I assume
// there is, but this will do for now. -- Joeri Peters
public class AggregateMappingGenerator {
    List<List<List<Integer>>> result;
    Map<Integer, List<List<Integer>>> finalResult;
    String[] stringNames;
    int numberOfGroups;
}

```

```

int currentMappingSize;
boolean isThereARemainder;
List<Integer> currentPermutation;
Map<Integer, List<List<Integer>>> storedResults;
int mappingSizeAtClearing;

public AggregateMappingGenerator(String[] packageNames, int
    numberOfPatternModules, boolean remainder, int numberOfTops) {
    numberOfGroups = numberOfPatternModules;
    isThereARemainder = remainder;
    if (remainder)
        numberOfGroups++; // Call this class once with and once without remainder
    to get all possible mappings.
    stringNames = packageNames;
    currentPermutation = convertToIntegers(stringNames);
    finalResult = new HashMap<Integer, List<List<Integer>>>();
    currentMappingSize = -1;
    mappingSizeAtClearing = 0;
    storedResults = new HashMap<Integer, List<List<Integer>>>(numberOfTops);
}

// Convert from strings to integers (makes things easier).
private List<Integer> convertToIntegers(String[] names) {
    List<Integer> list = new ArrayList<Integer>(names.length);
    for (int i = 0; i < names.length; i++)
        list.add(i, names.length - i - 1);
    return list;
}

// Takes a list of integers and returns all possible ways to group them
// according to the given number of groups. For instance, [1,2,3,4] can be
// arranged as [1],[2,3,4], as [1,2],[3,4] etc. The ordering within the groups
// has no effect.
private void multipleSplit(List<Integer> individualPermutation, int
    numberOfGroups) {
    split(individualPermutation, numberOfGroups);
    preventDuplicates();
}

// The current way of calculating all possible distributions results in many
// duplicates due to the fact that [1,2],[3,4] is not truly different
// from [1,2],[4,3]. This method prevents such duplicates.
private void preventDuplicates() {
    int index = currentMappingSize;
    for (List<List<Integer>> singleGrouping : result) {
        if (groupingIsOrdered(singleGrouping) != false) {
            finalResult.put(index, singleGrouping);
            index++;
        }
    }
}

private boolean groupingIsOrdered(List<List<Integer>> singleGrouping) {
    for (int j = 0; j < singleGrouping.size(); j++) {
        for (int i = 0; i < singleGrouping.get(j).size() - 1; i++) {
            if (singleGrouping.get(j).get(i) > singleGrouping.get(j).get(i + 1))
                return false;
        }
    }
    return true;
}

```

```

// Splitting a list into groups in all possible ways.
private void split(List<Integer> listSU, int numberOfGroups) {
    if (numberOfGroups <= 0 || numberOfGroups > listSU.size())
        throw new IllegalArgumentException("Invalid number of groups");
    result = new ArrayList<>();
    computeSplit(listSU, 0, new List[numberOfGroups], 0, result);
}

// Recursive method to generate splits.
@SuppressWarnings("unchecked")
private void computeSplit(List<Integer> listSU, int i, List[] combination, int
    j, List<List<List<Integer>>> resultTemp) {
    if (combination.length - j == 1) {
        combination[j] = listSU.subList(i, listSU.size());
        List<List<Integer>> temp = new ArrayList<>();
        temp.addAll((Collection<? extends List<Integer>>) new ArrayList<>(Arrays.
asList(combination)));
        if (!resultTemp.contains(temp))
            resultTemp.add(deepClone(temp));
    } else {
        for (int index = 0; index <= (listSU.size() - i) - (combination.length - j
); index++) {
            combination[j] = listSU.subList(i, i + index + 1);
            computeSplit(listSU, i + index + 1, combination, j + 1, resultTemp);
        }
    }
}

// Java always passes by value, but that value is a reference in the case of a
// nested list (essentially). Deep cloning creates a brand new list.
private ArrayList<List<Integer>> deepClone(List<List<Integer>> listOfLists) {
    ArrayList<List<Integer>> cloneList = new ArrayList<List<Integer>>(
listOfLists.size());
    ArrayList<Integer> clone;
    for (List<Integer> nestedList : listOfLists) {
        clone = new ArrayList<Integer>(nestedList.size());
        for (Integer element : nestedList)
            clone.add(element);
        cloneList.add(clone);
    }
    return cloneList;
}

public List<List<String>> next(int remove) {

    if (currentMappingSize == -1) { // The next permutation must not be called
the first time.
        currentMappingSize++;
        multipleSplit(currentPermutation, numberOfGroups);
    }
    while (currentMappingSize - mappingSizeAtClearing == finalResult.size()) {
        currentPermutation = nextPermutation(currentPermutation);
        if (currentPermutation != null) {
            multipleSplit(currentPermutation, numberOfGroups);
        } else {
            if (remove != -2) { // The previous call (if there was one), was not in
the top N.
                storeResult(true);
            }
            return null; // If not, there exist no more mappings and the brute force

```

```

        loop is complete.
    }
}
if (remove != -2) { // The previous call (if there was one), was not in the
top N.
    if (remove == -1) { // The previous call fills up the top N
        storeResult(true);
    } else { // The previous call replaces a top candidate.
        storeResult(true);
        removeStored(remove);
    }
}
}
return convertBack(finalResult.get(currentMappingSize));
}

private List<List<String>> convertBack(List<List<Integer>> next) {
    List<List<String>> mapping = new ArrayList<List<String>>(next.size());
    ArrayList<String> temp;
    for (int i = 0; i < next.size(); i++) {
        temp = new ArrayList<String>();
        for (int j = 0; j < next.get(i).size(); j++) {
            temp.add(stringNames[next.get(i).get(j)]);
        }
        mapping.add(i, temp);
    }
    if (isThereARemainder)
        mapping.remove(numberOfGroups - 1);
    currentMappingSize++;
    return mapping;
}

public Map<Integer, ArrayList<String>> getMappingMap(int i) {
    Map<Integer, ArrayList<String>> map = new HashMap<>();
    List<List<String>> bestOne = convertBack(storedResults.get(i - 1));
    for (int j = 0; j < bestOne.size(); j++) {
        ArrayList<String> temp = new ArrayList<String>();
        for (int k = 0; k < bestOne.get(j).size(); k++) {
            temp.add(bestOne.get(j).get(k));
        }
        map.put(j, temp);
    }
    return map;
}

private List<Integer> nextPermutation(final List<Integer> currentPermutation)
{
    int first = getFirst(currentPermutation);
    if (first == -1)
        return null; // no greater permutation
    int toSwap = currentPermutation.size() - 1;
    while (currentPermutation.get(first) <= currentPermutation.get(toSwap))
        --toSwap;
    swap(currentPermutation, first++, toSwap);
    toSwap = currentPermutation.size() - 1;
    while (first < toSwap)
        swap(currentPermutation, first++, toSwap--);
    return currentPermutation;
}

private static int getFirst(final List<Integer> currentPermutation) {
    for (int i = currentPermutation.size() - 2; i >= 0; --i)

```



```

        if (currentPermutation.get(i) > currentPermutation.get(i + 1))
            return i;
        return -1;
    }

    private static void swap(final List<Integer> currentPermutation, final int i,
        final int j) {
        final int tmp = currentPermutation.get(i);
        currentPermutation.set(i, currentPermutation.get(j));
        currentPermutation.set(j, tmp);
    }

    private void storeResult(boolean store) {
        if (store)
            storedResults.put(currentMappingSize - 1, finalResult.get(
                currentMappingSize - 1));
        if (currentMappingSize > 3000 && currentMappingSize == finalResult.size()) {
            finalResult.clear();
            mappingSizeAtClearing = currentMappingSize;
        }
    }

    private void removeStored(int i) {
        if (storedResults.containsKey(i))
            storedResults.remove(i);
        else
            System.out.println("ERROR!");
    }
}

```

Code Sample D.4: The mapping generator for the aggregation case of the brute force approach.

```

package husaccttest.analyse;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotNull;
import static org.junit.Assert.assertTrue;

import java.util.ArrayList;
import java.util.List;

import org.junit.Test;

import husacct.analyse.task.reconstruct.bruteForce.AggregateMappingGenerator;

public class AggregateMappingGeneratorTest {

    @Test
    public void nextTest() {
        List<ArrayList<List<String>>> mappingList = new ArrayList<ArrayList<List<
            String>>>();
        String[] names = new String[] { "A", "B", "C", "D", "E", "F" };
        int numberOfGroups = 3;
        AggregateMappingGenerator mappingCalculator = new AggregateMappingGenerator(
            names, numberOfGroups, false, 10);
        List<List<String>> singleMapping;
        int count = 0;
        singleMapping = mappingCalculator.next(-2);
    }
}

```

```

    assertNotNull(singleMapping);
    count++;
    while (true) {
        // if (count == 42)
        // singleMapping = mappingCalculator.next();
        // else
        singleMapping = mappingCalculator.next(-1);
        if (singleMapping == null) {
            // System.out.println("NO MORE MAPPINGS LEFT");
            // System.out.println("Count: " + count);
            break;
        } else {
            assertTrue("MAPPING LIST CONTAINS DUPLICATES", !mappingList.contains(
singleMapping));
            mappingList.add(deepClone(singleMapping));
            count++;
        }
    }
    assertEquals("INCORRECT NUMBER OF MAPPINGS", 540, count);
    // System.out.println(mappingCalculator.getMappingMap(42));
    assertNotNull(mappingCalculator.getMappingMap(42));
}

@Test
public void nextIncludingRemainderTest() {
    String[] names = new String[] { "package1", "package2", "package3", "
package4", "package5", "package6", "p7", "p8" };
    int numberOfGroups = 3;
    AggregateMappingGenerator mappingCalculatorRemainder = new
AggregateMappingGenerator(names, numberOfGroups, true, 10);
    List<List<String>> singleMapping;
    int count = 0;
    assertNotNull(mappingCalculatorRemainder.next(-2));
    count++;
    while (true) {
        if (count%2==0) {
            singleMapping = mappingCalculatorRemainder.next(-1);}
        else
            singleMapping = mappingCalculatorRemainder.next(count);
        if (singleMapping == null) {
            // System.out.println("NO MORE MAPPINGS LEFT");
            // System.out.println("Count: " + count);
            break;
        } else {
            // System.out.println(singleMapping);
            count++;
            // System.out.println("Count: " + count);
        }
    }
    assertEquals("INCORRECT NUMBER OF MAPPINGS", 40824, count);
    assertNotNull(mappingCalculatorRemainder.getMappingMap(42));
}

private ArrayList<List<String>> deepClone(List<List<String>> listOfLists) {
    ArrayList<List<String>> cloneList = new ArrayList<List<String>>(listOfLists.
size());
    ArrayList<String> clone;
    for (List<String> nestedList : listOfLists) {
        clone = new ArrayList<String>(nestedList.size());
        for (String element : nestedList)
            clone.add(element);
    }
}

```

```

        cloneList.add(clone);
    }
    return cloneList;
}
}

```

Code Sample D.5: The test code for the aggregate mapping.

```

package husacct.analyse.task.reconstruct.genetic;

import java.util.ArrayList;

import org.jgap.Chromosome;
import org.jgap.Configuration;
import org.jgap.FitnessFunction;
import org.jgap.Gene;
import org.jgap.Genotype;
import org.jgap.IChromosome;
import org.jgap.Population;
import org.jgap.audit.EvolutionMonitor;
import org.jgap.impl.DefaultConfiguration;
import org.jgap.impl.IntegerGene;

import husacct.analyse.task.reconstruct.ReconstructArchitecture;
import husacct.analyse.task.reconstruct.patterns.Pattern;

public class GeneticAlgorithm {

    private static int MAX_ALLOWED_EVOLUTIONS;
    private static int numberOfGenes;
    private static int numberOfModules;
    public static EvolutionMonitor m_monitor;
    private static ArrayList<Chromosome> bestSolutions;
    private static ReconstructArchitecture reconstructArchitecture;
    private static Pattern pattern;
    private static int allowRemainder;

    @SuppressWarnings("unchecked")
    public static void determineBestCandidates(boolean a_doMonitor) throws
        Exception {
        Configuration conf = new DefaultConfiguration();
        // Care that the fittest individual of the current population is always
        // taken to the next generation. With that, the population size may
        // exceed its original size by one.
        conf.setPreservFittestIndividual(true);
        conf.setKeepPopulationSizeConstant(false);
        // Set the fitness function we want to use, which is our
        // GeneticFitnessFunction. We construct it with the target amount of change
        // passed in to
        // this method.
        FitnessFunction myFunc = new GeneticFitnessFunction(pattern,
            reconstructArchitecture);
        conf.setFitnessFunction(myFunc);
        if (a_doMonitor) {
            // Turn on monitoring/auditing of evolution progress.
            m_monitor = new EvolutionMonitor();
            conf.setMonitor(m_monitor);
        }
        Gene[] sampleGenes = new Gene[numberOfGenes];
        for (int i = 0; i < numberOfGenes; i++) {
            sampleGenes[i] = new IntegerGene(conf, allowRemainder, numberOfModules);
        }
    }
}

```

```

    }
    IChromosome sampleChromosome = new Chromosome(conf, sampleGenes);
    conf.setSampleChromosome(sampleChromosome);
    conf.setPopulationSize(100);
    // Create random initial population of Chromosomes.
    Genotype population = Genotype.randomInitialGenotype(conf);
    long startTime = System.currentTimeMillis();
    for (int i = 0; i < MAX_ALLOWED_EVOLUTIONS; i++) {
        if (!uniqueChromosomes(population.getPopulation()))
            throw new RuntimeException("Invalid state in generation " + i);
        if (m_monitor != null) {
            System.out.println("Begin evolution iteration " + i);
            population.evolve(m_monitor);
            System.out.println("End evolution iteration " + i);
        } else {
            System.out.println("Begin evolution iteration " + i);
            population.evolve();
            System.out.println("End evolution iteration " + i);
        }
    }
    long endTime = System.currentTimeMillis();
    System.out.println("Total evolution time: " + (endTime - startTime) + " ms");
    ;
    bestSolutions = new ArrayList<Chromosome>(population.getPopulation().size());
    ;
    bestSolutions.addAll(population.getFittestChromosomes(population.
getPopulation().size()));
    for (int i = 0; i < bestSolutions.size(); i++) {
        for (int j = i + 1; j < bestSolutions.size(); j++) {
            if (bestSolutions.get(i).equals(bestSolutions.get(j))) {
                bestSolutions.remove(j);
                j--;
            }
        }
    }
}

}

}

public static ArrayList<Chromosome> run(Pattern currentPattern, String[]
softwareUnits, boolean monitor, ReconstructArchitecture reconstruct, boolean
remainder,
    int generations) throws Exception {
    if (softwareUnits.length < 2) {
        System.out.println("Too few software units. ");
    } else if (softwareUnits.length > GeneticFitnessFunction.getMaxBounds())
        System.out.println("Too many software units. ");
    else if (currentPattern.getNumberOfModules() > softwareUnits.length)
        System.out.println("Too few pattern modules. ");
    else {
        numberOfGenes = softwareUnits.length;
        numberOfModules = currentPattern.getNumberOfModules();
        reconstructArchitecture = reconstruct;
        pattern = currentPattern;
        MAX_ALLOWED_EVOLUTIONS = generations;
        if (remainder)
            allowRemainder = 0;
        else
            allowRemainder = 1; // Allowing a Remainder (software units not mapped
to any of the pattern's module) means that genes should have
// allele values starting with 0, otherwise they should start
with 1. Hence the integer value here.
    }
}

```

```

    }
    determineBestCandidates(monitor);
    System.out.println("Presenting the ordering of software units mapped in the
    chromosomes: \n");
    for (int i = 1; i <= softwareUnits.length; i++) {
        System.out.println(i + ": " + softwareUnits[i - 1]);
    }
    System.out.println(
        "\nThe best (and unique) chromosomes are printed here. If their number
        is particularly small, it is because the population contained many
        duplicates.");
    System.out.println(
        "This would mean that the algorithm has converged on a small number of
        solutions. These should indicate at least local optima, if not the global
        optimum.");
    System.out.println();
    return bestSolutions;
}

// Check that all chromosomes are unique
public static boolean uniqueChromosomes(Population a_pop) {
    for (int i = 0; i < a_pop.size() - 1; i++) {
        IChromosome c = a_pop.getChromosome(i);
        for (int j = i + 1; j < a_pop.size(); j++) {
            IChromosome c2 = a_pop.getChromosome(j);
            if (c == c2)
                return false;
        }
    }
    return true;
}
}

```

Code Sample D.6: The genetic algorithm class.

```

/**
 * PackageDependenciesTest.java
 */
package com.eteks.sweethome3d.junit;

import java.io.IOException;

import jdepend.framework.DependencyConstraint;
import jdepend.framework.JDepend;
import jdepend.framework.JavaPackage;
import jdepend.framework.PackageFilter;
import junit.framework.TestCase;

/**
 * Tests if dependencies between Sweet Home 3D packages are met.
 * @author Emmanuel Puybaret
 */
public class PackageDependenciesTest extends TestCase {
    /**
     * Tests that the package dependencies constraint is met for the analyzed
     * packages.
     */
    public void testPackageDependencies() throws IOException {
        PackageFilter packageFilter = new PackageFilter();
        // Ignore Java packages and Swing sub packages
        packageFilter.addPackage("java.*");
    }
}

```

```

packageFilter.addPackage("javax.swing.*");
// Ignore JUnit tests
packageFilter.addPackage("com.eteks.sweethome3d.junit");

JDepend jdepend = new JDepend(packageFilter);
jdepend.addDirectory("classes");

DependencyConstraint constraint = new DependencyConstraint();
// Sweet Home 3D packages
JavaPackage sweetHome3DModel = constraint.addPackage("com.eteks.sweethome3d.model");
JavaPackage sweetHome3DTools = constraint.addPackage("com.eteks.sweethome3d.tools");
JavaPackage sweetHome3DPlugin = constraint.addPackage("com.eteks.sweethome3d.plugin");
JavaPackage sweetHome3DViewController = constraint.addPackage("com.eteks.sweethome3d.viewcontroller");
JavaPackage sweetHome3DSwing = constraint.addPackage("com.eteks.sweethome3d.swing");
JavaPackage sweetHome3DJava3D = constraint.addPackage("com.eteks.sweethome3d.j3d");
JavaPackage sweetHome3DIO = constraint.addPackage("com.eteks.sweethome3d.io");
JavaPackage sweetHome3DApplication = constraint.addPackage("com.eteks.sweethome3d");
JavaPackage sweetHome3DApplet = constraint.addPackage("com.eteks.sweethome3d.applet");
// Swing components packages
JavaPackage swing = constraint.addPackage("javax.swing");
JavaPackage imageio = constraint.addPackage("javax.imageio");
JavaPackage imageioStream = constraint.addPackage("javax.imageio.stream");
// Java 3D
JavaPackage java3d = constraint.addPackage("javax.media.j3d");
JavaPackage vecmath = constraint.addPackage("javax.vecmath");
JavaPackage sun3dLoaders = constraint.addPackage("com.sun.j3d.loaders");
JavaPackage sun3dLoadersLw3d = constraint.addPackage("com.sun.j3d.loaders.lw3d");
JavaPackage sun3dUtilsGeometry = constraint.addPackage("com.sun.j3d.utils.geometry");
JavaPackage sun3dUtilsImage = constraint.addPackage("com.sun.j3d.utils.image");
JavaPackage sun3dUtilsUniverse = constraint.addPackage("com.sun.j3d.utils.universe");
JavaPackage sun3dExpSwing = constraint.addPackage("com.sun.j3d.exp.swing.JCanvas3D");
// XML
JavaPackage xmlParsers = constraint.addPackage("javax.xml.parsers");
JavaPackage xmlSax = constraint.addPackage("org.xml.sax");
JavaPackage xmlSaxHelpers = constraint.addPackage("org.xml.sax.helpers");
// JMF
JavaPackage jmf = constraint.addPackage("javax.media");
JavaPackage jmfControl = constraint.addPackage("javax.media.control");
JavaPackage jmfDataSink = constraint.addPackage("javax.media.datasink");
JavaPackage jmfFormat = constraint.addPackage("javax.media.format");
JavaPackage jmfProtocol = constraint.addPackage("javax.media.protocol");
// SunFlow
JavaPackage sunflow = constraint.addPackage("org.sunflow");
JavaPackage sunflowCore = constraint.addPackage("org.sunflow.core");
JavaPackage sunflowCoreLight = constraint.addPackage("org.sunflow.core.light");
JavaPackage sunflowCorePrimitive = constraint.addPackage("org.sunflow.core.

```

```

primitive");
JavaPackage sunflowImage = constraint.addPackage("org.sunflow.image");
JavaPackage sunflowMath = constraint.addPackage("org.sunflow.math");
JavaPackage sunflowSystem = constraint.addPackage("org.sunflow.system");
JavaPackage sunflowSystemUI = constraint.addPackage("org.sunflow.system.ui");
;
// iText for PDF
JavaPackage iText = constraint.addPackage("com.lowagie.text");
JavaPackage iTextPdf = constraint.addPackage("com.lowagie.text.pdf");
// FreeHEP Vector Graphics for SVG
JavaPackage vectorGraphicsUtil = constraint.addPackage("org.freehep.util");
JavaPackage vectorGraphicsSvg = constraint.addPackage("org.freehep.
graphicsio.svg");
// Batik for SVG path parsing
JavaPackage orgApacheBatikParser = constraint.addPackage("org.apache.batik.
parser");
// Java JNLP
JavaPackage jnlp = constraint.addPackage("javax.jnlp");
// Mac OS X specific interfaces
JavaPackage eawt = constraint.addPackage("com.applet.eawt");
JavaPackage eio = constraint.addPackage("com.applet.eio");

// Describe dependencies : model don't have any dependency on
// other packages, IO and View/Controller packages ignore each other
// and Swing components and Java 3D use is isolated in sweetHome3DSwing
sweetHome3DTools.dependsUpon(sweetHome3DModel);
sweetHome3DTools.dependsUpon(eio);

sweetHome3DPlugin.dependsUpon(sweetHome3DModel);
sweetHome3DPlugin.dependsUpon(sweetHome3DTools);
sweetHome3DPlugin.dependsUpon(sweetHome3DViewController);

sweetHome3DViewController.dependsUpon(sweetHome3DModel);
sweetHome3DViewController.dependsUpon(sweetHome3DTools);

sweetHome3DJava3D.dependsUpon(sweetHome3DModel);
sweetHome3DJava3D.dependsUpon(sweetHome3DTools);
sweetHome3DJava3D.dependsUpon(sweetHome3DViewController);
sweetHome3DJava3D.dependsUpon(java3d);
sweetHome3DJava3D.dependsUpon(vecmath);
sweetHome3DJava3D.dependsUpon(sun3dLoaders);
sweetHome3DJava3D.dependsUpon(sun3dLoadersLw3d);
sweetHome3DJava3D.dependsUpon(sun3dUtilsGeometry);
sweetHome3DJava3D.dependsUpon(sun3dUtilsImage);
sweetHome3DJava3D.dependsUpon(sun3dUtilsUniverse);
sweetHome3DJava3D.dependsUpon(imageio);
sweetHome3DJava3D.dependsUpon(sunflow);
sweetHome3DJava3D.dependsUpon(sunflowCore);
sweetHome3DJava3D.dependsUpon(sunflowCoreLight);
sweetHome3DJava3D.dependsUpon(sunflowCorePrimitive);
sweetHome3DJava3D.dependsUpon(sunflowImage);
sweetHome3DJava3D.dependsUpon(sunflowMath);
sweetHome3DJava3D.dependsUpon(sunflowSystem);
sweetHome3DJava3D.dependsUpon(sunflowSystemUI);
sweetHome3DJava3D.dependsUpon(xmlParsers);
sweetHome3DJava3D.dependsUpon(xmlSax);
sweetHome3DJava3D.dependsUpon(xmlSaxHelpers);
sweetHome3DJava3D.dependsUpon(orgApacheBatikParser);

sweetHome3DSwing.dependsUpon(sweetHome3DModel);
sweetHome3DSwing.dependsUpon(sweetHome3DTools);

```

```

sweetHome3DSwing.dependsUpon(sweetHome3DPlugin);
sweetHome3DSwing.dependsUpon(sweetHome3DViewController);
sweetHome3DSwing.dependsUpon(sweetHome3DJava3D);
sweetHome3DSwing.dependsUpon(swing);
sweetHome3DSwing.dependsUpon(imageio);
sweetHome3DSwing.dependsUpon(imageioStream);
sweetHome3DSwing.dependsUpon(java3d);
sweetHome3DSwing.dependsUpon(vecmath);
sweetHome3DSwing.dependsUpon(sun3dUtilsGeometry);
sweetHome3DSwing.dependsUpon(sun3dUtilsUniverse);
sweetHome3DSwing.dependsUpon(sun3dExpSwing);
sweetHome3DSwing.dependsUpon(jmf);
sweetHome3DSwing.dependsUpon(jmfControl);
sweetHome3DSwing.dependsUpon(jmfDataSink);
sweetHome3DSwing.dependsUpon(jmfFormat);
sweetHome3DSwing.dependsUpon(jmfProtocol);
sweetHome3DSwing.dependsUpon(iText);
sweetHome3DSwing.dependsUpon(iTextPdf);
sweetHome3DSwing.dependsUpon(vectorGraphicsUtil);
sweetHome3DSwing.dependsUpon(vectorGraphicsSvg);
sweetHome3DSwing.dependsUpon(jnlp);

sweetHome3DIO.dependsUpon(sweetHome3DModel);
sweetHome3DIO.dependsUpon(sweetHome3DTools);
sweetHome3DIO.dependsUpon(eio);

// Describe application and applet assembly packages
sweetHome3DApplication.dependsUpon(sweetHome3DModel);
sweetHome3DApplication.dependsUpon(sweetHome3DTools);
sweetHome3DApplication.dependsUpon(sweetHome3DPlugin);
sweetHome3DApplication.dependsUpon(sweetHome3DViewController);
sweetHome3DApplication.dependsUpon(sweetHome3DJava3D);
sweetHome3DApplication.dependsUpon(sweetHome3DSwing);
sweetHome3DApplication.dependsUpon(sweetHome3DIO);
sweetHome3DApplication.dependsUpon(swing);
sweetHome3DApplication.dependsUpon(imageio);
sweetHome3DApplication.dependsUpon(java3d);
sweetHome3DApplication.dependsUpon(eawt);
sweetHome3DApplication.dependsUpon(jnlp);

sweetHome3DApplet.dependsUpon(sweetHome3DModel);
sweetHome3DApplet.dependsUpon(sweetHome3DTools);
sweetHome3DApplet.dependsUpon(sweetHome3DPlugin);
sweetHome3DApplet.dependsUpon(sweetHome3DViewController);
sweetHome3DApplet.dependsUpon(sweetHome3DJava3D);
sweetHome3DApplet.dependsUpon(sweetHome3DSwing);
sweetHome3DApplet.dependsUpon(sweetHome3DIO);
sweetHome3DApplet.dependsUpon(swing);
sweetHome3DApplet.dependsUpon(java3d);
sweetHome3DApplet.dependsUpon(jnlp);

jdepend.analyze();

assertTrue("Dependency mismatch", jdepend.dependencyMatch(constraint));
}
}

```

Code Sample D.7: The JUnit tests for SweetHome3D's package dependencies.

Appendix E

WICSA/CompArch 2016 paper

Next is a paper submitted to the Young Researchers Forum track of the joint WICSA/CompArch 2016 conference in Venice. We are currently awaiting provisional acceptance.

The paper is focussed on the pattern definitions and the problems that arise with the Remainder, including the issues raised in Section 8.5. We thought it seemed wiser to reserve the search algorithms and the SAR aspects of this thesis for a later publication, possibly in combination with case studies, as well as improved code and formalisation.

Architectural Pattern Definition for Semantically Rich Modular Architectures

Joeri Peters, Jan Martijn E.M. van der Werf, Jurriaan Hage
Department of Information and Computing Sciences
Utrecht University
Utrecht, the Netherlands
J.G.T.Peters@students.uu.nl, {J.M.E.M.vanderWerf, J.Hage}@uu.nl

Abstract—Architectural patterns represent reusable design of software architecture at a high level of abstraction. They can be used to structure new applications and to recover the modular structure of existing systems. Techniques like Architecture Compliance Checking (ACC) focus on testing whether the realised artefacts adhere to the architecture. Typically, these techniques require a complete architecture as input. In this paper, we focus on defining architectural patterns in such a way that we can use ACC tools to recognise architectural pattern instances. This requires us to explicitly define architectural patterns in terms of allowed and disallowed software dependencies. We base ourselves on Semantically Rich Modular Architectures. Defining architectural patterns this way allows us to reason about them. For example, how patterns should be interpreted as incomplete architectures and how different interpretations affect the pattern recognition process. Recognising architectural patterns using ACC techniques also has great potential in architecture design and Software Architecture Reconstruction.

I. INTRODUCTION

“The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both” [1]. One can distinguish three types of structures within software architecture (static, dynamic and allocation) [1], or look at a categorisation based on views and viewpoints, such as functional, development, and concurrency [2]. This paper is restricted to static, technical architectures, the branch of software architecture that looks at the arrangement of source code into modules such as components and layers. We study their connecting dependencies without incorporating runtime behaviour.

Architectural patterns prevent software architects from reinventing the wheel in much the same way as design patterns do for developers. They establish the relationship between a context, a problem and a solution [1]. Examples of such patterns include Model-View-Controller, Broker and N-Layered patterns. Another concept related to architectural patterns is architectural *style* [3]. We consider the style to be the highest level architectural pattern when multiple patterns are applied, such as a system split into three layers (the overall style of the system) with a smaller Model-View-Controller pattern used within one or more of these layers.

This paper describes our ongoing research into pattern recognition using Architecture Compliance Checking (ACC). ACC works by checking whether realised artefacts, such as

source code, adhere to the intended architecture [4]. Static ACC is thus aimed at the analysis of software dependencies between static modules and their correspondence to architecture documentation. ACC relies on a proper and adequate architecture description, which is typically tool dependent.

“A semantically rich modular architecture includes modules of semantically different types, while a variety of rule types may constrain the modules” [5]. Design and validation of such Semantically Rich Modular Architectures (SRMAs) is possible with the ACC tool HUSACCT, since it provides support for such a diverse variety of architectural elements [6] [7]. These elements can be said to be HUSACCT’s architectural language.

HUSACCT is intended to describe a complete, as opposed to partial, architecture. As we are interested in the potential role of ACC to check whether an architectural pattern is present in source code, we want to study the possibilities this tool’s language provides to express architectural patterns in terms of allowed and disallowed dependencies. HUSACCT does distinguish between dependency types, such as method calls and variable accesses, but this is not relevant to our pattern definitions at this time. We thus focus on the following research question: “how can architectural patterns be expressed in HUSACCT’s terms and what are the consequences of alternative pattern definitions?”

The remainder of this paper is structured as follows: Section 2 explains the relevant portions of HUSACCT’s SRMA support, in Section 3 the consequences of pattern definitions for the allowed dependencies are investigated, Section 4 shows how these choices become critical when combining patterns, Section 5 lists some of the relevant literature and Section 6 portrays the possible future industrial impact of our work.

II. SRMA CONCEPTS

SRMAs combine various types of modules with rule types to express architectural elements and their constraints. This enhances expressiveness and supports architecture reasoning in terms comparable to regular language [5]. Using the module and rule types understood by HUSACCT, we want to express architectural patterns.

HUSACCT supports the following types of software modules: subsystems, layers, components, interfaces and external systems. Subsystems are modules with clear responsibilities.

TABLE I
THE 7 RELATION RULE TYPES, SPLIT INTO TWO SUB-CATEGORIES.

Is not allowed to use
Is not allowed to back-call (layers)
Is not allowed to skip-call (layers)
Is allowed to use
Is only allowed to use
Is the only module allowed to use
Must use

A layer has the additional property of a hierarchical level, which enforces strict layering as HUSACCT automatically adds rules banning skip-calls and back-calls to the relevant levels. Components are modules whose contents are hidden behind and accessed through an interface module. Finally, external systems represent libraries, modules that are not actually part of the system under consideration [5] [6].

The rule types can be placed in one of two categories: property rule types and relation rule types [5]. The former consists of conventions such as naming and inheritance. Only the façade convention is used in this paper. This convention states that interface Module A always had to act as the interface for component Module B.

The second category, relation rule types, consists of a further subdivision into: “Is not allowed to use” and “Is allowed to use” rules, which are themselves two basic rule types that can be used to express rules like “Module A is not allowed to use Module B”. Two rule types exist within the first subdivision (Back-call bans and Skip-call bans), which exist specifically for layer-type modules. Within the second subdivision, there are three: “Is only allowed to use”, “Is the only module allowed to use” and “Must use”. That makes a total of seven relation rule types, each tying two modules together and putting limitations on the dependencies allowed in the modular architecture. These rules are listed in Table I.

III. ARCHITECTURAL PATTERN DEFINITION

Architectural patterns, such as the N-Layered pattern (equivalent terms include Layered Architectures and N-Layers pattern), are not as precisely defined as the Gang-of-Four design patterns [8]. One can see them as two extremes on the same spectrum. The main characteristic of design patterns is that they are designed for solving recurrent problems on the level of the detailed design, e.g. source code, whereas architectural patterns exist on a system level. Consequently, design patterns appear more frequently within the same system, deal with far more specific concepts and are meant not to have any rule violations at all. Architectural patterns tend to focus more on which dependencies are *not* allowed. Design patterns specify the dependencies that *should* be implemented. As such, design patterns are more fit to be used as detailed design techniques similar to the tactics described by Bass et al. [1].

In this paper, we explore the possibility to express architectural patterns in terms of allowed and disallowed dependencies. Although at first sight this seems an easy exercise, in re-

ality defining patterns this way leaves room for interpretation. Consider the 3-Layered pattern. This pattern is commonly used as a primary separation of concerns within a software system, e.g. user interface, business logic and data access layers. Each layer is only supposed to make use of its own internal modules or those of the layer directly below.

There are three layers and the following rules are how one would ordinarily express the pattern in terms of skip- and back-calls:

Rule set 1, Skip-calls and back-calls:

- 1) *Layer i is not allowed to skip-call to Layer $j > i + 1$.*
- 2) *Layer i is not allowed to back-call to Layer $j < i$.*

The fact that HUSACCT’s layer-type modules possess hierarchical levels that constrain dependencies already suggests one way of defining the N-Layered pattern. However, we want to be able to use other module types as well, e.g. in order to use component-type modules as layers, which means we have to translate this set to one consisting of rules that do not rely on hierarchical levels.

A translation to “Is not allowed to use” rules would result in the following set of architectural rules for this pattern:

Rule set 2, “Is not allowed to use”:

- 1) *Layer 2 is not allowed to use Layer 1.*
- 2) *Layer 3 is not allowed to use Layer 2.*
- 3) *Layer 3 is not allowed to use Layer 1.*
- 4) *Layer 1 is not allowed to use Layer 3.*

The same can be achieved by replacing rules 1 and 4 with “Is the only module allowed to use” rules from one layer to the next, or by two “Is only allowed to use” rules whilst removing rule 4. Both would appear to signify the exact same pattern. One might even prefer a hybrid form, such as the one displayed in rule set 3:

Rule set 3, Hybrid example:

- 1) *Layer 1 is the only module allowed to use Layer 2.*
- 2) *Layer 2 is only allowed to use Layer 3.*
- 3) *Layer 3 is not allowed to use Layer 1.*
- 4) *Layer 1 is not allowed to use Layer 3.*

Let us compare the different rule sets to see what happens when such a set is used as a pattern definition within a system’s architecture. Imagine the rest of the architectural elements of this system as a single module, the *Remainder*, all architectural elements that are not part of the architectural pattern under consideration and not sub-modules of any of the pattern’s modules. As such, the Remainder is always with respect to some pattern.

According to rule set 2, nothings prevents the Remainder from calling on any of the layers within the 3-Layered pattern and any of these layers can call upon the Remainder. This is depicted in Figure 1. Rule set 3 forbids any dependencies between the Remainder and Layer 2, as shown in Figure 2. This creates a scenario where the Remainder is free to call upon Layers 1 and 3, but not 2, while Layer 2 is the only layer that cannot call upon the Remainder. This version essentially

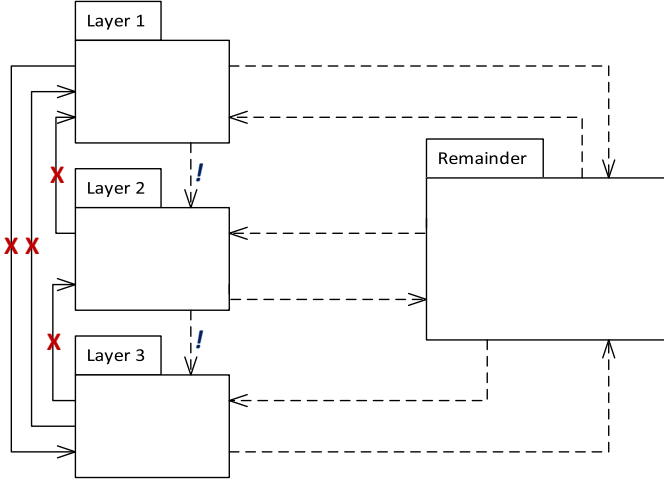


Fig. 1. The 3-Layered pattern and the Remainder, defined with “Is not allowed to use” rules (rule set 2) and showing allowed dependencies as dashed lines.

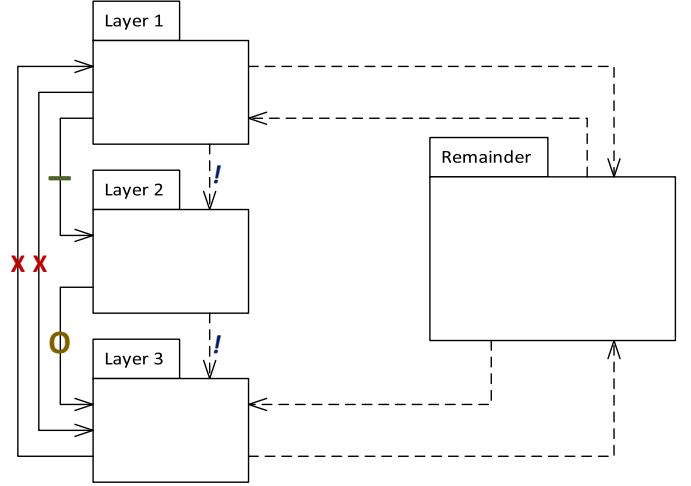


Fig. 2. The 3-Layered pattern and the Remainder, defined with three different rule types (rule set 3).

TABLE II
THIS LEGEND PRESENTS THE DIFFERENT RULE TYPES FOR THE
PROVISIONAL DIAGRAM NOTATION USED HERE.

“Is not allowed to use”	———X———>
“Is only allowed to use”	———O———>
“Is the only module allowed to use”	——— ———>
“Is allowed to use”	----->
“Must use”	-----!----->

isolates Layer 2 from the Remainder, while leaving a lot of freedom to the other two layers. In both cases, we have added “Must use” rules between the layers in order to enforce the correct usage of the pattern.

To graphically depict the patterns, we adopt a UML-like syntax. Expanded packages represent architectural elements, i.e. pattern modules and the Remainder. Collapsed packages indicate sub-modules. Legal dependencies are represented by dashed lines and directed associations figure as the architectural rules. Additional symbols are used to indicate the rule type being employed, as depicted in Table II. Conflicting rules result in exceptions. HUSACCT allows for rule exceptions by specifying the module that is exempt from the given rule [6].

At first sight, these rule sets seem to be equivalent. However, as an architectural pattern is an incomplete architecture, subtle differences emerge. Consider again the 3-Layered pattern.

More interpretations can be formulated using additional rule sets, combining the various rule types in different ways. We can thus identify the following interpretations of the N-Layered pattern:

- **N-Layered pattern (Complete freedom):** there are no restrictions with regard to the Remainder; hence the name. This corresponds to rule sets 1 and 2.
- **N-Layered pattern (Free Remainder):** the Remainder

cannot be called by any layer, but can itself depend on any of them. The rule set requires two “Is only allowed to use” rules from each layer to the next and two “Is not allowed to use” rules for the back-call bans.

- **N-Layered pattern (Restricted Remainder):** the Remainder cannot call on any layer. This is with “Is the only module allowed to use” rules from each layer to the next and an “Is not allowed to use” rule from Layer 3 to 1.
- **N-Layered pattern (Isolated internal layers):** intermediary layers are never called by and can themselves not call on the Remainder. This corresponds to rule set 3.

More varieties are conceivable, especially when different module-types are taken into account. This goes to show that even a pattern as simple as the N-Layered pattern allows for various different versions when the Remainder is taken into account. It shows that for pattern recognition, we need explicit pattern definitions. These same issues come up when one considers other architectural patterns and, for the sake of illustration, we will briefly go into one of them: the Model-View-Controller pattern.

Model-View-Controller

The Model-View-Controller (hereafter MVC) pattern is considered to be a design pattern by some, but it also exists on a system level. As such, it can be used to separate graphics/presentation/UI logic from the underlying model logic by restricting communication between the two. A Controller acts as a mediator between them, with the essential restriction that Model cannot use Controller [1].

Multiple interpretations of MVC exist, since many interpretations and adaptations of this pattern occur in the field. For example, sometimes the View incorporates the concerns of the Controller, leaving only two modules in this MVC version. Also, whereas the rule that Model cannot depend on Controller seems rather well accepted, it is unclear whether Model can depend on View and vice versa. If it cannot, there is a clear

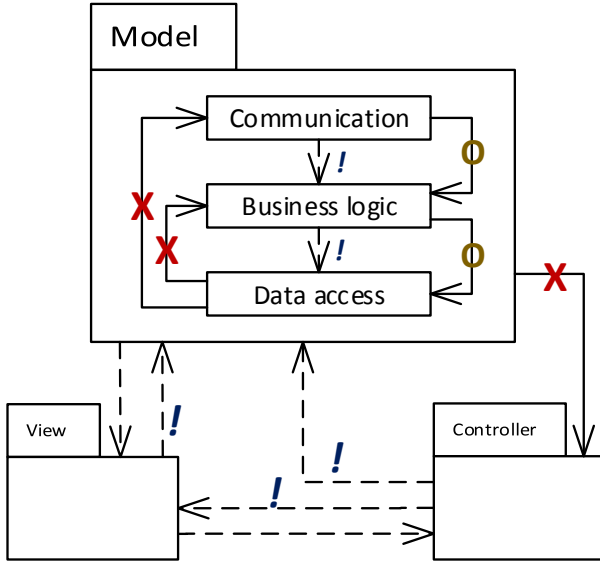


Fig. 4. A combination of Complete Freedom MVC (style) and Free Remainder 3-Layered (pattern), which fails to isolate the lower Model layers.

der” interpretation (two “Is the only module allowed to use” rules and one “Is not allowed to use”) ought to be combined with rules that prohibit the data access layer from accessing anything but those modules that it should require. This may be solved with some of the relational rules, or by turning Model into a component type with either the communication layer or an additional element within Model functioning as the interface for this component, thereby calling upon the façade convention that was briefly mentioned in Section 2.

If these combinations raise such questions, then pattern recognition within pattern-based architectures needs to take into account that identifying overall style first could result in a different conclusion from an approach to pattern recognition starting with the smaller pattern.

V. RELATED LITERATURE

This study of architectural pattern definition in HUSACCT’s terms is part of a larger project to recognise such patterns in actual source code. Other researchers have come up with definitions of patterns as well, though not always for the purposes of ACC support.

An early example of architectural patterns being used in scientific literature is by Abowd, Allen and Garlan [12]. They introduce the notion of architectural styles, which consist of precise syntactic and semantic descriptions of both static and dynamic characteristics. A deep understanding of a style and its constraints can then be used for formal analysis of that style, studying potential sub-styles and comparing it with other styles. The paper’s appendix briefly explains the Z language, which is the mathematical notation they use to define styles and which is strongly based on predicate logic.

Sartipi presents an extensive approach to software architecture reconstruction (SAR) based on patterns and data mining that makes use of both clustering and editing-cost-based

graph-matching techniques [13]. Although this has a strong mathematical background and promising results, it requires substantial user input, but this may well be unavoidable for SAR.

Developed by Schmerl and Garlan [14], AcmeStudio is aimed at the design phase of an application and allows for new styles to be defined by the user in terms of predicate logic. Although it may not be relevant in terms of architecture reconstruction or pattern recovery, the paper does provide some interesting insights into the definition of such patterns. AcmeStudio relies on Acme, as the name implies, which they describe as a style-neutral Architecture Description Language (ADL).

Kim and Garlan [15] [16] analyse architectural styles automatically for properties such as consistency and compatibility. This is done by, again, writing a style in the Acme ADL. Using a set of relatively straightforward translation rules, this is then turned into Alloy (a language based on predicate logic) and fed to the Alloy Analyzer. Within the Analyzer, one can check whether various properties of styles hold according to the logical description.

A relatively rich ontological framework for architectural styles is presented by Pahl, Giesecke and Hasselbring [17]. Acme is used as the ADL once more and translated to predicate logic. This style ontology may also be extended to incorporate metrics such as Quality Attributes. Papers such as this can be used to collect the logical representations of architectural patterns by whoever might be interested in such expressions.

For a systematic literature review on architecture reconstruction, we refer the reader to the 2009 publication by Ducasse and Pollet [18], in which they mention architectural patterns and styles several times in their attempts to build an ontology of various methods.

VI. INDUSTRIAL IMPACT

A precise definition of a specific architectural pattern undoubtedly aids the design process, by making it clear for everyone involved what exactly is meant by that pattern. Pattern definition within the context of Semantically Rich Modular Architectures allows us to do this more accurately. The possibility of substituting module types within each pattern definition is no trivial consequence of HUSACCT’s language, as the various module types imply certain architectural rules. However, the main consequences relate to the rule types at our disposal and how they change the effect an architectural pattern has when the rest of the architecture is taken into consideration. This observation and the resulting collection of subtly different interpretations of each architectural pattern do not only impact software architecture design that uses SRMAS, but also the related activities of ACC and SAR if they were to make use of such a rich language.

Architectural patterns may be useful in ACC if one is able to enter such a pattern into a software tool, as part of the intended architecture, for example. With ACC software tools, the intended architecture needs to be fed into the tool so that

it can be compared to the realised architecture found by the tool. In this way, we want to measure the degree to which the realised architecture complies with the intended architecture.

Not only ACC can benefit from a precise understanding of architectural patterns. We see potential use in SAR as well. Recognising the architectural patterns applied in the realised artefacts, and the order in which they have been applied, would result in better architecture documentation.

For example, a particular system's realised architecture can have the appearance of a layered structure with some MVC aspects. If so, these patterns would naturally form the ideas on which to base the initial reconstruction steps. A well-defined notion of such patterns helps to test the hypothesis, especially when ACC is used for this.

VII. CONCLUSIONS

With this paper, we show how HUSACCT's architectural elements, which are commonly found in Semantically Rich Modular Architectures, allow one to think about architectural pattern definitions in some precision. This raises many questions, such as what we mean when we say "MVC pattern" or what a layered architecture implies with regard to architectural elements that are not part of any layer. These questions, and the more deliberate choices of definitions that follow from them, may be useful for software architects to describe exactly what it is they mean by their applied patterns, but it can also be fruitful for purposes other than software design. Our own current and future research is focussed on the usage of these SRMA pattern definitions in ACC and SAR, where we want to detect instances of architectural patterns within existing architectures. The pattern definitions we choose and the order in which we detect such patterns can have profound effects on the allowed dependencies. It is therefore imperative that we make all rules, assumptions and consequences surrounding architectural patterns completely explicit.

Acknowledgement

We wish to express our thanks to our colleagues at the Department of Information and Computing Sciences of Utrecht University for their support. Special thanks to Leo Pruijt of the HU University of Applied Sciences for his many contributions and insights during our research, as well as his work involving SRMAs and HUSACCT, on which our research is founded.

REFERENCES

- [1] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 3rd ed., P. Gordon, Ed. Boston, Massachusetts, USA: Addison-Wesley Professional, 2012.
- [2] N. Rozanski and E. Woods, *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives*, 1st ed. Addison-Wesley, 2005, vol. 8, no. 2. [Online]. Available: <http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20&path=ASIN/0321112296>
- [3] R. N. Taylor, N. Medvidović, and E. M. Dashofy, *Software Architecture: Foundations, Theory, and Practice*. Wiley, 2009. [Online]. Available: <http://eu.wiley.com/WileyCDA/WileyTitle/productCd-EHEP000180.html>
- [4] J. Knodel and D. Popescu, "A Comparison of Static Architecture Compliance Checking Approaches," in *2007 Working IEEE/IFIP Conference on Software Architecture (WICSA'07)*. IEEE, jan 2007, pp. 12–12. [Online]. Available: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4077029>
- [5] L. Pruijt and S. Brinkkemper, "A metamodel for the support of semantically rich modular architectures in the context of static architecture compliance checking," in *Proceedings of the First International Conference on Dependable and Secure Cloud Computing Architecture - DASCCA '14*. New York, New York, USA: ACM Press, 2014, pp. 1–8. [Online]. Available: <http://dl.acm.org/citation.cfm?doid=2578128.2578233>
- [6] L. Pruijt, C. Köppe, J. M. Van der Werf, and S. Brinkkemper, "HUSACCT: Architecture Compliance Checking with Rich Sets of Module and Rule Types," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering - ASE '14*. Vasteras, Sweden: ACM, 2014, pp. 851–854. [Online]. Available: <http://dl.acm.org/citation.cfm?doid=2642937.2648624>
- [7] L. Pruijt and J. M. E. M. van der Werf, "Dependency Types and Subtypes in the Context of Architecture Reconstruction and Compliance Checking," in *Proceedings of the 2015 European Conference on Software Architecture Workshops - ECSAW '15*. New York, New York, USA: ACM Press, 2015, pp. 1–7. [Online]. Available: <http://dl.acm.org/citation.cfm?doid=2797433.2797491>
- [8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*, 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1994. [Online]. Available: <http://dl.acm.org/citation.cfm?id=186897>
- [9] M. R. J. Qureshi and F. Sabir, "A Comparison of Model View Controller and Model View Presenter," *CoRR*, vol. abs/1408.5, 2014. [Online]. Available: <http://arxiv.org/abs/1408.5786>
- [10] M. Potel, "MVP : Model-View-Presenter The Taligent Programming Model for C++ and Java," Taligent, Inc., Tech. Rep., 1996. [Online]. Available: metrology.googlecode.com/svn-history/r350/trunk/doc/ebooks/mvp.pdf
- [11] E. Puybaret, "SweetHome3D," 2015. [Online]. Available: <http://www.sweethome3d.com/>
- [12] G. D. Abowd, R. Allen, and D. Garlan, "Using Style to Understand Descriptions of Software Architecture," in *Proceedings of the 1st ACM SIGSOFT Symposium on Foundations of Software Engineering*. Los Angeles, CA, USA: ACM, 1993, pp. 9–20. [Online]. Available: <http://dl.acm.org.proxy.library.uu.nl/citation.cfm?id=167055>
- [13] K. Sartipi, "Software architecture recovery based on pattern matching," in *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings*. IEEE, 2003, pp. 293–296. [Online]. Available: http://ieeexplore.ieee.org.proxy.library.uu.nl/xpl/freeabs_all.jsp?arnumber=1235434&abstractAccess=no&userType=inst
- [14] B. Schmerl and D. Garlan, "AcmeStudio: supporting style-centered architecture development," in *26th International Conference on Software Engineering*. IEEE, 2004, pp. 704–705. [Online]. Available: http://ieeexplore.ieee.org.proxy.library.uu.nl/xpl/freeabs_all.jsp?arnumber=1317497&abstractAccess=no&userType=inst
- [15] J. S. Kim and D. Garlan, "Analyzing architectural styles with alloy," in *Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis - ROSATEA '06*. New York, New York, USA: ACM Press, 2006, pp. 70–80. [Online]. Available: <http://portal.acm.org/citation.cfm?id=1147259http://portal.acm.org/citation.cfm?doid=1147249.1147259>
- [16] —, "Analyzing architectural styles," *Journal of Systems and Software*, vol. 83, no. 7, pp. 1216–1235, jul 2010. [Online]. Available: <http://linkinghub.elsevier.com/retrieve/pii/S0164121210000336>
- [17] C. Pahl, S. Giesecke, and W. Hasselbring, "Ontology-based modelling of architectural styles," *Information and Software Technology*, vol. 51, no. 12, pp. 1739–1749, 2009. [Online]. Available: <http://dx.doi.org/10.1016/j.infsof.2009.06.001>
- [18] S. Ducasse and D. Pollet, "Software architecture reconstruction: A process-oriented taxonomy," *IEEE Transactions on Software Engineering*, vol. 35, no. 4, pp. 573–591, 2009. [Online]. Available: <http://www.computer.org/csdl/trans/ts/2009/04/ts2009040573-abs.html>