OBJECTED-ORIENTED DESIGN PATTERN DETECTION USING STATIC AND DYNAMIC ANALYSIS IN JAVA SOFTWARE

by Marcel Birkner

By virtue of submitting this document electronically, the author certifies that this is a true electronic equivalent of the copy of the thesis approved by the University of Applied Sciences Bonn-Rhein-Sieg for the award of the degree. No alteration of the content has occurred and if there are any minor variations in formatting, they are as a result of the conversion to Adobe Acrobat format (or similar software application).

Examination Committee Members:

1. Professor Manfred Kaul
2. Professor Rainer Herpers

Supervisor at York University:

1. Associate Professor Vassilios Tzerpos
# Table of Contents

Table of Contents iii

List of Figures viii

1 Introduction 4

1.1 Statement of the problem 4

1.2 Definition of terms 6

1.3 Research contribution 7

1.4 Thesis outline 8

2 Background 11

2.1 Literature overview 11

2.2 Software 18

2.2.1 Software Architecture Group (SWAG) 19

2.2.2 Eclipse 20
E.0.2 Adapter .................................................. 134
E.0.3 Bridge ................................................... 137
E.0.4 Builder .................................................. 139
E.0.5 Chain of Responsibility ................................. 142
E.0.6 Command ............................................... 144
E.0.7 Composite .............................................. 146
E.0.8 Decorator ............................................... 149
E.0.9 Factory Method ......................................... 152
E.0.10 Flyweight .............................................. 154
E.0.11 Interpreter ............................................. 156
E.0.12 Iterator ............................................... 158
E.0.13 Mediator ............................................... 160
E.0.14 Memento ............................................... 163
E.0.15 Observer ............................................... 166
E.0.16 Prototype ............................................. 171
E.0.17 Proxy ................................................ 173
E.0.18 Singleton ............................................. 175
E.0.19 State .................................................. 177
E.0.20 Strategy ............................................... 180
E.0.21 Template Method ..................................... 182
List of Figures

2.1 Spool environment ............................................. 14
3.1 UML for Adapter pattern detection ................................. 23
3.2 Detection process .............................................. 26
3.3 Facts in Rigi Standard Format .................................... 28
3.4 Output from QL after manipulating the factbase .................... 28
3.5 Lifting facts to class level with Grok. ......................... 29
3.6 Pseudocode example .............................................. 30
3.7 UML sequence diagram for Adapter design pattern .................. 32
3.8 Static analysis output shows the candidate instances .............. 36
3.9 Temporal restricting - next call in subtree .................... 38
3.10 Temporal restriction - next call in order ...................... 39
3.11 Temporal restriction - next call not in order .................. 40
3.12 Using object ids for more detailed description of design patterns. . 41
3.13 UML sequence diagram for the Command pattern .................. 45
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Overview of implemented detection process</td>
<td>49</td>
</tr>
<tr>
<td>4.2 Facts in Rigi Standard Format</td>
<td>51</td>
</tr>
<tr>
<td>4.3 QL script for the Adapter pattern</td>
<td>52</td>
</tr>
<tr>
<td>4.4 Probekit editor</td>
<td>54</td>
</tr>
<tr>
<td>4.5 Tip: Use test suites of software to create extensive dynamic facts</td>
<td>54</td>
</tr>
<tr>
<td>4.6 Excerpt from dynamic facts</td>
<td>55</td>
</tr>
<tr>
<td>4.7 Adapter candidate instances</td>
<td>57</td>
</tr>
<tr>
<td>4.8 DTD XML Schema for the dynamic definition</td>
<td>59</td>
</tr>
<tr>
<td>4.9 Dynamic definition of the Adapter design pattern</td>
<td>60</td>
</tr>
<tr>
<td>4.10 Dynamic definition conversion</td>
<td>62</td>
</tr>
<tr>
<td>4.11 Matching dynamic definition in dynamic facts</td>
<td>65</td>
</tr>
<tr>
<td>4.12 Dynamic definition template</td>
<td>66</td>
</tr>
<tr>
<td>4.13 UML class diagram for the Command pattern</td>
<td>70</td>
</tr>
<tr>
<td>4.14 QL script for Command design pattern</td>
<td>71</td>
</tr>
<tr>
<td>4.15 UML sequence diagram for the Command pattern</td>
<td>72</td>
</tr>
<tr>
<td>4.16 Ranking the output of PDE</td>
<td>75</td>
</tr>
<tr>
<td>5.1 Detailed results for PINOT. On the left side are the code examples</td>
<td>84</td>
</tr>
<tr>
<td>and the top represents the design patterns that we detected</td>
<td></td>
</tr>
</tbody>
</table>

ix
5.2 Detailed results for PDE (threshold=100%). On the left side are the code examples and the top represents the design patterns that we detected.

5.3 Detailed results for PDE (threshold=80%).

5.4 Detailed results for PDE (combined threshold of 80% and 100%).

5.5 Manually checked results from PDE for 22 design patterns.

5.6 Results from dynamic analysis.

5.7 JHotDraw results.

A.1 Simple PDE example.

A.2 Simple PDE example output.

A.3 Another PDE example.

A.4 Running PDE with the configuration for our experiments.

B.1 PDE input FactFile.

B.2 Static UML diagram of PDE.

C.1 Extract static facts from javex.

C.2 Javex synopsis.

C.3 Run Grok to extract uses and inheritance relations.

C.4 Grok procedure by Ric Holt.

C.5 Lift static facts to class level and extract uses and inherits relations.
C.6 Find classes with private constructor .......................... 121

D.1 Probekit code: method entry .......................................... 127
D.2 Probekit code: method exit ............................................ 128

E.1 Abstract Factory static definition ................................. 131
E.2 UML class diagram - Abstract Factory ......................... 132
E.3 UML class diagram - Adapter ................................. 134
E.4 Adapter design pattern UML sequence diagram .......... 136
E.5 UML class diagram - Bridge .......................................... 137
E.6 UML class diagram - Builder ................................. 139
E.7 Builder design pattern UML sequence diagram .......... 141
E.8 UML class diagram - Chain of Responsibility .......... 142
E.9 UML class diagram - Command ................................. 144
E.10 UML class diagram - Composite ......................... 147
E.11 UML class diagram - Decorator ................................. 150
E.12 UML class diagram - Factory Method ......................... 152
E.13 UML class diagram - Flyweight ................................. 154
E.14 UML class diagram - Interpreter ................................. 156
E.15 UML class diagram - Iterator ................................. 158
E.16 UML class diagram - Mediator ................................. 160
E.17 Mediator design pattern UML sequence diagram . . . . . . . . . . . . 162
E.18 UML class diagram - Memento . . . . . . . . . . . . . . . . . . . . . 163
E.19 Memento design pattern UML sequence diagram . . . . . . . . . . . 165
E.20 UML class diagram - Observer . . . . . . . . . . . . . . . . . . . . 167
E.21 Observer design pattern UML sequence diagram . . . . . . . . . . . 170
E.22 UML class diagram - Prototype . . . . . . . . . . . . . . . . . . . 171
E.23 UML class diagram - Proxy . . . . . . . . . . . . . . . . . . . . . . 173
E.24 UML class diagram - Singleton . . . . . . . . . . . . . . . . . . . 175
E.25 UML class diagram - State . . . . . . . . . . . . . . . . . . . . . . 177
E.26 UML class diagram - Strategy . . . . . . . . . . . . . . . . . . . 180
E.27 UML class diagram - Template Method . . . . . . . . . . . . . . . . 182
E.28 UML class diagram - Visitor . . . . . . . . . . . . . . . . . . . . . 185
E.29 Visitor design pattern UML sequence diagram . . . . . . . . . . . . 187
Erklärung

Ich versichere an Eides Statt, die von mir vorgelegte Arbeit selbständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

Ort       Datum       Unterschrift
Abstract

Design patterns abstract reusable object-oriented software design. Each pattern solves design problems that occur in every day software development. The detection of design patterns during the process of software reverse engineering can provide a better understanding of the software system. The latest tools rely on the abstract syntax tree representation of the source code for fact extraction.

Our approach uses static and dynamic code analysis to detect design pattern in Java applications. We use the roles, responsibilities and collaboration information of each design pattern to define static and dynamic definitions. The static definitions are used to find candidate instances during the static analysis. After the static analysis we validate the found candidate instances using the dynamic behavior of design patterns. For the dynamic analysis we instrument the Java bytecode of the application we are analyzing with additional code fragments and extract method traces from the running application. These method traces represent the dynamic facts of an application. We present several restrictions that are used to define design
patterns dynamically. After the dynamic validation we rank the results according on how good they match the dynamic definitions.

This thesis introduces a new approach in detection of object-oriented design patterns in Java applications. We test our approach using the 23 original GoF design patterns and analyze the results. Compared to other tools, our software achieves better results in detecting design patterns. The methods we choose for our approach work great in detecting patterns given the static and dynamic facts as input files.
1 Introduction

This chapter is an introduction to design patterns and the motivation behind detecting design patterns in software. We explain the research contribution and common vocabulary used throughout the thesis.

1.1 Statement of the problem

Software design patterns abstract reusable object-oriented software design. Each pattern solves design problems that occur in every day software development. The detection of design patterns during the process of software reverse engineering can provide a better and faster understanding of the software system.

The first time the term design pattern was related to software development was in the book Design Patterns - Elements of Reusable Object-Oriented Software written by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides (GHJV95). The group of authors is also known as the Gang-of-Four or GoF. They discuss object-oriented design techniques that are based on their experience as software
developers. They introduce 23 design patterns using a consistent format of information for each pattern to provide a uniform structure. Using the intent, trade-offs and graphical notations for design patterns, software engineers can decide which design pattern solves their design problems. It also makes it easier to discuss design problems and solutions with colleagues by using a common vocabulary. It is also useful documenting which design patterns have been used in a software so that other developers will get a better overview of the software without having to read the source code in detail.

This thesis is concerned with the problem of detecting software design patterns. We present an approach that will detect software design patterns using their static structure - as described in class diagrams - as well as their dynamic behavior. The results can be used to verify the implementation of design patterns that were specified before the implementation phase. Having this additional information can be crucial during software maintenance. If well-designed software is poorly documented then this good design might be broken by a different developer that needs to add more functionality. These changes might introduce new bugs and problems and lead to degradation of the software. Therefore, it is important to document these design choices to improve the understanding of the software.
1.2 Definition of terms

This section provides a list of definitions that will be used throughout this thesis. These definitions are meant to help the reader and establish a common vocabulary. The terms will be explained in more detail when they are first introduced, therefore it is not necessary to memorize them:

- static code analysis - examination of code without executing the program; provides understanding of the code structure.

- dynamic code analysis - executing the program and recording method calls; provides details about the behavior of the program.

- role (also participant) - class that has a specific characteristic/function in the context of a design pattern.

- intent - encapsulates the idea and objective of a design pattern.

- candidate instances - possible candidates for a design pattern. Each candidate instance is a set of classes. These classes possibly match the roles from a design pattern.

- Rigi Standard Format (RSF) - an RSF file consists of a sequence of triples, one triple on a line.
• Tuple Attribute Language - TA, the Tuple Attribute Language, is intended to allow convenient recording of information about certain types of graphs. This information includes (1) nodes and edges in the graph, and (2) attributes of these nodes and edges.

Example: two edges in the Tuple language.

○ calls P Q
○ references Q V

Graph representation: P \xrightarrow{calls} Q \xrightarrow{references} V

1.3 Research contribution

The purpose of this thesis is to present a reverse engineering approach that will detect software design patterns in Java source code. Our approach uses static and dynamic code analysis. We introduce several models that we use to remove false positives from our static analysis results. We are interested in which order methods have to be called and what objects are involved at run-time. We rank the results according to how well each candidate instance matches the dynamic definitions. We provide a tool that implements our approach. This tool can be downloaded from our website together with a library of static and dynamic definitions for the original GoF design patterns, see http://www.cse.yorku.ca/~mbirkner/
1.4 Thesis outline

Chapter 1 is an introduction to design patterns and the motivation behind detecting design patterns in software. We explain the research contribution and common vocabulary used throughout the thesis.

Chapter 2 presents background on design pattern detection. We show different approaches that have been documented so far in the literature and present the tools that have been developed. This thesis extends some of the work presented by Wang (Wan04) and Lebon (Leb06). Their work is related to the Eiffel programming language but some ideas and methods are reused. At the end of this chapter we introduce the tools that we use in our work.

Chapter 3 explains our approach in detecting design patterns, using static and dynamic analysis techniques. We introduce models that define design patterns statically and dynamically. We use the Adapter design pattern as a running example to present our approach and models as detailed as possible.

Chapter 4 presents our Pattern Detection Engine (PDE). This chapter ex-
plains the architecture of our tool and the models it uses. As a running example we will use the Command design pattern. We show how this tool is used, what kind of input it needs and what the results look like.

Chapter 5 explains the experiments we did to test the correctness, robustness and functionality of our tool. We use design pattern examples from the Applied Java Patterns book written by Stephen A. Stelting and Olav Maassen to show that we can detect design patterns correctly. We present the results from a comparison of our tool with another design pattern detection tool called PINOT.

Chapter 6 is the conclusion of our work and summarizes the achieved results. It gives an overview of possible future work.

Appendix A is the User Manual for PDE. It explains how to run the software in the Eclipse environment as well as how to use it to detect design patterns.

Appendix B contains the design documents for PDE. We describe the UML diagrams that represent the class structure of the tool.

Appendix C talks about the static analysis. It introduces the tools that were
developed by SWAG at the University of Waterloo and that we use in this thesis. We explain how we use Javex to extract static facts from Java class files. After that we show how we use Grok and QL to detect candidate instances of design patterns in these static facts.

Appendix D presents the details about the dynamic analysis. The first section gives an overview of Probekit from the Test and Performance Tools Platform Projec. We use Probekit to instrument the Java bytecode of the software we are testing to extract dynamic facts. These facts are later used as input for the Pattern Detection Engine.

Appendix E lists our static and dynamic definitions of all 22 GoF design patterns that we are able to detect with PDE. We extract these facts from UML class and sequence diagrams.
2 Background

This chapter presents the background of design pattern detection. We show different approaches that have been documented so far in the literature and present the tools that have been developed. This thesis extends some of the work that was done by Wang (Wan04) and Lebon (Leb06). Their work is related to the Eiffel programming language but some ideas and methods are reused here. At the end of this chapter we introduce the tools that we use in our work.

2.1 Literature overview

Detecting design patterns enhances program understanding as well as existing source code and documentation. There is a great deal of use in having this knowledge about a software system. The time developers need to understand a software system decreases if they have access to good documentation about the design of the application. There has been a lot of research in the area of design pattern detection
over the last years. Many different approaches have been developed and some of them were implemented in tools and tested with software systems to show their effectiveness. The following chapter will list the different approaches and their results.

One of the first papers about detecting design patterns was written by Kraemer and Prechelt in 1996. They introduced an approach detecting design information directly from C++ header files. This information is stored in a repository. The design patterns are expressed as PROLOG rules which are used to query the repository with the extracted information. Their work focused on detecting five structural design patterns: Adapter, Bridge, Composite, Decorator, and Proxy. The precision of their developed tool is 14 to 50 percent which they claim is acceptable for discovering design information. They suggest a more detailed approach that considers method call delegation during structural analysis which would lead to overall better results (KP96).

Another approach in detecting design patterns in C++ code was introduced by Espinoza, Esqueer and Cansino. They formulate a canonical model to represent design patterns. They developed a system called DEsign PAtterns Identification of C++ programs (DEPAIC++). This tool is composed by two modules that first transform the C++ to a canonical form and then recognize design patterns. It
is implemented in Java. They use the structural relationships between classes to identify design patterns in the source code. They tested their work with different software systems that used design patterns and were able to detect them using DEPAIC++. Their future work includes extending this tool as well as detecting design patterns in Java source code (EEC02).

Keller et al. present the SPOOL (Spreading Desirable Properties into the Design of Object-Oriented, Large Scale Software Systems) environment for the reverse engineering of design components based on the structural descriptions of design patterns for C++ software. SPOOL is a joint industry/university collaboration between Bell Canada and the University of Montreal. The project aims at both software comprehension and software design quality assessment. This environment supports both reverse and forward engineering of design patterns. The SPOOL environment consists of source code capturing tools, a design repository, and functionality for pattern-based design recovery and representation.

The source code capturing tools contain source code parser for C++, Java and Smalltalk that transform the code into an intermediate UML/CDIF representation that is used for later language independent processing. The design repository is used to provide centralized storage for the implemented design components, the source code models, and abstract design components. The pattern-based recovery
structures parts of class diagrams to resemble pattern diagrams. Figure 2.1 gives an overview of the architecture of the SPOOL environment (KSRP99).

Kyle Brown proposed in his thesis a reverse engineering tool called KT to detect the use of design patterns in Smalltalk programs. His work examines the static and dynamic structure of design patterns and determines the nature of what makes a design pattern detectable by automated means. It also outlines algorithms by which a small set of design patterns can be detected. His work shows that reverse engineering tools not only need to support static but also dynamic analysis in order to detect design patterns (Bro).
Seemann et al. present their ideas on how to recover design information from Java source code. They move step by step by deriving several layers of increasing abstraction. First, a compiler collects information about inheritance hierarchies and method call relations. The result of the initial parsing is a graph that is used for the design recovery process. This graph is subsequently transformed by grammar productions. They define criteria for the automatic detection of associations and aggregations between classes. After filtering the graph they detect the information about design patterns. A case study with the AWT package shows that their approach detects design patterns (SvG98).

Shi and Olsson introduce a new, fully automated pattern detection approach for Java source code which is based on the reclassification of design patterns by their intent. They reclassified the GoF patterns into five categories in the reverse engineering sense: language provided, driven by structural design, driven by behavioral design, domain specific and generic concepts. Their approach uses lightweight static analysis techniques to capture the intent of the software. They developed a Pattern INference and recOvery Tool called PINOT that implements this new approach. PINOT detects all original design patterns introduced by Gamma et al. that have concrete definitions driven by code structure or system behavior. They
provide results from detecting design patterns in Java AWT, JHotDraw, Swing, Apache Ant and other packages on their website: www.cs.ucdavis.edu/~shini/research/pinot/. We will compare our results in detecting design patterns with PINOT since this is one of the most mature tools we could find in the literature that is detecting design patterns in Java source code (SO06).

Another interesting approach is introduced with FUJABA at the University of Paderborn. FUJABA is a Tool Suite that contains a number of tools for software engineering and reverse engineering. Design patterns are defined as graph transformation rules, with respect to the abstract syntax graph (ASG) of a systems source code. The pattern rules are applied by an inference algorithm which is implemented by the InferenceEngine plug-in. After parsing the source code of the system into an ASG the inference engine is started by loading a catalogue of pattern rules. FUJABA allows the user to define patterns in a visual UML-like format. For running FUJABA the user only has to specify the location of the source code and then runs the pattern inference engine (NSW+01).

There is also a lot of research that explains the use of design patterns during software development and reverse engineering. The results of the pattern detection can be used for implementation verification (Ben) and automatic verification of de-
design patterns (BBS05), for understanding the behavior of Java programs (Sys00), for easing maintenance of Java programs (PUP97) and improving the communication between software developers (PRE97). Using design patterns during software development and maintenance results in better software design and less errors in the source code (PRE97).

The work of Wei Wang at the York University presents an approach in detecting design patterns in Eiffel source code. First, he introduces a design pattern diagram catalogue in BON (Business Object Notation). This catalogue includes all 23 patterns introduced by Erich Gamma (GHJV95). Each pattern includes both static and dynamic diagrams for the example implementations created by Jezequel et al (JTM99). These diagrams are used for understanding the structure and behavior of the patterns. They also assist in composing static and dynamic definitions used in pattern detection.

Besides the diagram catalogue, Wang developed the reverse engineering tool DPVK (Design Pattern Verification toolKit). This tool is used to detect pattern instances in Eiffel systems. In his work, he analyzes different patterns and examines Eiffel software in terms of both static structure and dynamic behavior.

Maurice Lebon continued the work of Wang. He introduced fine-grained rules to the model and extended the Design Pattern Verification toolKit with these addi-
tional rules. These additional rules are introduced to remove more false positives and to improve the results of the tool. His work includes detailed definitions for each GoF pattern that explain the use of all additional rules.

The previous mentioned research explains our motivation that led to this thesis. We believe that the information of design patterns in software will lead to better software understanding. It also helps with the software maintenance process and reduces errors in the source code when the legacy software is extended. The found design patterns can also be added to the documentation to make it easier for developers to understand different parts of the code easier and faster.

2.2 Software

In order to detect design patterns in Java software, we used several existing tools and technologies. In the following sections, we give a brief introduction to all tools and technologies, and explain how we deploy them in our work. Appendix C contains more detailed information about these tools and how we used them for our approach.
2.2.1 Software Architecture Group (SWAG)

For the static code analysis we make extensive use of three tools that are developed by the Software Architecture Group at the University of Waterloo. These three tools are Javex, Grok and QL.

Javex is a fact extractor for Java, that extracts facts from Java class files. It is a command line program that takes Java classes as input and outputs all facts in Tuple Attribute format (i.e: "calls P Q" or "references Q V"). These facts can be directly viewed using lsedit from SWAG or used for further analysis using Grok and QL.

Grok is a programming language designed for manipulating collections of binary relations. Grok operates at the level of a relational database, in that operators generally apply across entire relations, and not just to single entities. The Grok interpreter has been optimized to handle large fact bases (up to several hundred thousands of facts, or tuples). It keeps all of its data structures in memory. Grok is written in the Turing language.

QL is a Java re-implementation of Grok, developed at the University of Waterloo. While serving essentially the same purpose as Grok, QL is not identical to it. While being slower than Grok, QL makes up for it with new operators and built-in commands. QL can also handle non-binary relations, an advantage that was important
for our purposes. Grok and QL can be used interactively or by passing a script as an argument. We provide all scripts with this thesis that are used during the static analysis.

We use all three tools for our static code analysis. First we extract all facts from Java class files with Javex. Then we use Grok to reduce these facts to only represent inheritance and association relationships between classes. After that we have individual QL scripts for all design patterns that we want to detect and run QL to extract possible candidate instances of design patterns. These candidate instances are stored and used as input for the dynamic analysis phase.

2.2.2 Eclipse

We developed our software using Eclipse. Eclipse is an open-source software framework written in Java. In its default form it is a Java Integrated Development Environment (IDE), comprised of the Java Development Toolkit (JDT) and compiler (ECJ). Users can easily extend its capabilities by installing plug-ins written for the Eclipse software framework, such as development toolkits for other programming languages, and can write and contribute their own plug-in modules. Over the years it has become the most popular IDE for developing Java programs. The portability and large number of plug-ins make it the standard development platform for most
developers.

In our case we are using one additional plug-in for Eclipse. We use the plug in from the Test and Performance Tools Platform Project (TPTP) which allows us to build test and performance tools, such as debuggers, profilers and benchmarking applications. Particularly we use Probekit from TPTP to instrument the Java bytecode of the software that we are analyzing. While we run the instrumented Java program we create an output file that includes all method calls and their order in which they appeared during run-time. These are the dynamic facts that we use to verify the found candidate instances from the static analysis.
3 Approach

Design patterns have their own unique intent and are described with roles and responsibilities. In the source code, each role is commonly represented by a class and the responsibilities are coded in the classes with attributes and methods. The patterns also describe the collaboration between objects at runtime. In this thesis, we use the roles, responsibilities and collaboration information to analyze applications, detect design patterns and rank the results by their classification. Understanding the structure and intent of the software system will give the developer a faster overview of the whole system without going into details of the source code.

Figure 3.1 shows an excerpt of an application represented in UML notation. The left side illustrates a UML diagram of a software system. The right side shows a UML diagram of the Adapter design pattern that we want to detect. After the detection process we present all classes that match design pattern instances. In UML notation, classes are represented as boxes with the class name as a title and
the dependency and association relationships shown as arrows going from one class to another. During the detection process we will use this information to detect the design pattern. We will use the Adapter design pattern as a running example in this chapter to present all features of our approach.

The intent of the Adapter design pattern is to convert the interface of a class into another interface that the clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces. The Adapter design pattern consists of four roles:

![Figure 3.1: UML for Adapter pattern detection](image-url)

Figure 3.1: UML for Adapter pattern detection
• A **Target** interface that represents the contract between the Adapter class and the Clients that want to use the Adapter.

• An **Adapter** that holds a reference to the Adaptee and translates Target methods to Adaptee methods.

• An **Adaptee** that has the actual implementation that the Client wants to use.

• One or more **Clients** that want to use a specific functionality.

Our approach in detecting design patterns is divided into static and dynamic analysis. Figure 3.2 presents the different stages of our approach. For the first stage we use the class files of the application that we want to analyze. We also use UML class and sequential diagrams that describe the design pattern we want to detect. The second stage is the static analysis. We use the UML class diagram to create static definitions of the design pattern. Then we use the Java class files and the created static definition to extract static facts from the software. The results include class names, relationships between classes and attributes of the classes. These facts are processed to find possible candidate instances that match the UML notation of a design pattern. In the third stage we run the software and create dynamic facts. We instrument the bytecode of the software, execute the software and record all
method calls of the running instances. For the dynamic analysis it is important to execute as many methods in the software as possible. Good test suites will help to obtain good dynamic facts since they call most of the implemented methods. We also use the UML sequential diagram to create the dynamic design pattern definition. In the last stage we process the facts from the dynamic analysis and the candidate instances from the static analysis together with the dynamic definition. In this stage we verify the candidate instances if they match the dynamic definition. After the detection process we rank the found design pattern instances according to how well they match the static and dynamic definitions. We will provide static and dynamic definitions for the original software design patterns from the GoF book, see Appendix E.
Figure 3.2: Detection process
3.1 Static analysis

Now we go into more detail in presenting the model that is used for the detection process described before. First we describe the static and then the dynamic analysis. For the static analysis we carry on the work that has been done in the master thesis of Wang (Wan04) and Lebon (Leb06). Their work was related to the programming language Eiffel but the principal ideas and methods can be used for Java as well. In their work they extracted static facts from the Eiffel Studio automatically. The static facts are represented in Rigi Standard Format (RSF) showing uses and inherits relations between classes and interfaces. For example, uses classC classA declares that classC has a method call to classA, or in another example, inherits classA classT indicates that classA inherits from classT, see Figure 3.3. Using these facts and the SWAG Tool Kit from the University of Waterloo, Wang was able to find possible candidate instances in the source code that matched the static definitions of the design patterns, see Figure 3.4. The first line represents all roles of the design pattern. The following lines represent possible candidate instances of the Adapter design pattern. In this example, classC represents the Client, classT represents the Target, classA represents the Adapter and classE represents the Adaptee. He used the relational calculator QL to find sets of classes that would match the static definitions and created QL scripts for each pattern.
Most of the scripts are used in this thesis with some changes for detecting patterns in Java source code.

```
uses classC classA
uses classA classE
inherits classA classT
```

Figure 3.3: Facts in Rigi Standard Format

```
// client target adapter adaptee
DP classC classT classA classE
```

Figure 3.4: Output from QL after manipulating the factbase

In order to use this approach for the static analysis with Java source code, we needed to find a way to extract static facts from Java code the same way as in the Eiffel Studio. We decided to use Javex from the SWAG Tool Kit for this task. Javex is a fact extractor for Java that extracts facts about all entities from the Java class files. These entities are directories, source files, classes, interfaces, methods, variables, arrays, fields, constants, local variables and parameter variables. For each entity Javex records all relations that occur in the source code, i.e. extends class, implements interface, instantiates, method invocations, containment (class contains
methods, classes, variables etc.) and many more. Information about Javex and a detailed list of all relations can be found in the Appendix C. Since these facts are fine grained and more detailed than what we need at this stage we use the relational calculator Grok from the SWAG Toolkit to extract only the uses and inherits facts from the Javex output and lift all relations to class level. In our case lifting relations is illustrated in Figure 3.5. In our pseudocode example (see Figure 3.6), methodA calls methodB, ClassA contains methodA and ClassB contains methodB. Lifting these three facts to class level means we will store the information that ClassA calls ClassB in our output file in RSF format. Chapter 4 explains the implementation details of this step in more detail.

Figure 3.5: Lifting facts to class level with Grok.
Listing all facts to class level can be expressed as:

\[ m_1 \text{ calls } m_2 \land c_1 \text{ contains } m_1 \land c_2 \text{ contains } m_2 \implies c_1 \text{ calls } c_2 \]

\( m_1, m_2 : \text{ methods}, c_1, c_2 : \text{ classes} \)

Lifting all relations to class level reduces the number of facts on average by 95%.

As part of the future work for this thesis, one may consider using the detailed facts directly. One example for the use of the detailed facts is used in this thesis for the Singleton pattern. Here, it is important that the constructor of the Singleton class is private. With the facts from Javex and the use of Grok we can select all classes that have a private constructor. We now have an additional restriction in the QL script that is useful in detecting the Singleton pattern. Without this fact it is impossible to find reliable results for the Singleton pattern due to its simple static and dynamic structure.
### 3.2 Dynamic analysis

For the dynamic analysis we are collecting run-time information of the application that we want to analyze. These dynamic facts represent the method calls that were executed during run-time. We explain how these dynamic facts are useful during the design pattern detection. Design patterns are not just described by their UML static class notation as shown in Figure 3.1. Their dynamic behavior is also very important and distinguishes patterns that have an identical static structure, see the State and Strategy pattern in the Appendix E. This can be expressed in terms of specific sequences of messages between a fixed number of objects with UML Sequence diagrams. These diagrams consist of vertical lines which represent the objects during run-time and arrows that represent the messages from one object to another. Figure 3.7 is an example of the Adapter design pattern UML sequence diagram. Looking at the diagram it is easy to deduce the order in which messages are passed. First, the Client calls the Adapter object, and that one calls the Adaptee directly and passes the return value back to the Client. We use UML sequence diagrams to create our dynamic definitions of the *GoF* design patterns. The UML specification for sequence diagrams allows very precise definitions of the sequential interaction of objects. Special types of associations between objects are create calls and return calls which are represented in UML with different arrows.
Create calls construct new objects and return calls return objects back to the sender.

The chronological order of method calls is shown by the order in the diagram from top to bottom. Our dynamic definitions make use of this notation to be more precise in the detection of valid instances of design patterns.

Figure 3.7: UML sequence diagram for Adapter design pattern

The dynamic definitions that we use for detecting design patterns consist of a number of method calls and temporal restrictions that apply for these calls. Temporal restrictions define in which order method calls have to happen. One method call can appear before or after another. In some cases it is useful to define that the second call must take place before the first call returns. Each of these
method calls can be described with various attributes. The idea is that we specify a list of method calls that need to be matched with the dynamic facts of the program that we are analyzing. For the Adapter pattern for example we know that a Client object calls the Adapter object and the Adapter object forwards this call and invokes a method in the Adaptee object. This order of method calls can be represented as follows in our dynamic design pattern definitions.

First method call: The Client calls the Adapter object.

- class name - Adapter (receiver)
- called by class - Client (sender)
- object id - run-time object id of the Adapter object (receiver)

Second method call: The Adapter object calls the Adaptee object.

- class name - Adaptee (receiver)
- called by class - Adapter (sender)
- object id - run-time object id of the Adaptee object (receiver)
- called by object id - run-time object id of the Adapter object (sender)

The restrictions mentioned before are role restrictions. For each method call we specify which roles of the design pattern are involved. In the first method call the Adapter receives the method call from the Client, which sends the method call. Since the roles only represent class names we added object restrictions
to our dynamic model. In the first method call the object id of the Adapter object for the receiver must be the same as the Adapter object id in the second call for the sender. With this restriction we can specify that the object that was called by the Client in the first call is the same object that called the Adaptee object in the second method call.

Our approach for the dynamic definitions contains several restrictions that can be grouped into five different categories as follows:
• Role restriction
  – arguments - role names of arguments passed to the object
  – class name - role name of the object on which the method is called (receiver)
  – called by class - role name of the object (sender) that called the receiver
• Method restriction
  – method name - name of method in object (receiver) that is called
  – called by method - name of method from which the current method was called
• Temporal restriction
  – nextCallInSubtree
    * yes - the next method call must take place before this method call returns
    * no (default) - no restriction specified
  – nextCallInOrder
    * yes - the next method call needs to take place after the current method call
    * no (default) - the next method call can be before or after the current call
• Object restriction
  – this object - object id of this object (receiver) at run-time
  – called by object - object id from the sender object at run-time from which the current method was called
• Ranking
  – quantifier - integer greater than zero that can be used to rank the weights of each method call

The first three attributes are role restrictions. We check if they match the
static facts from the candidate instances that we detected during the static analysis. These are the roles from the design pattern. These roles are exchanged with the actual class names that we get as a result from the static analysis. In the static analysis we get a list of possible candidate instances. For the Adapter pattern this would look like Figure 3.8. According to this Figure classC is the Client, classT is the Target interface, classA the candidate Adapter and classB the candidate Adaptee. If we use the roles client, adaptee, adapter or the role target somewhere in our dynamic definition then we will exchange these roles with the class names of the candidate instances that we find in the static analysis. After this step we iterate through the dynamic facts and try to find matches for all defined method calls.

```
// client target adapter adaptee
DP classC classT classA classE
```

Figure 3.8: Static analysis output shows the candidate instances

The next two attributes define the method restrictions. These attributes can be set to Constructor, a specific method name, or they can be left blank. The method name can be used for different things. If we want to define that an object creates another object then it would call the constructor method. By set-
ting method name=Constructor we can create this restriction on the method call. In case the actual name of the called method in an object is known, then this can be specified by adding that name for the method name. For example: method name=main if the main method was called in the object. Since method names differ in the implementations of design patterns, we only use this attribute to specify if a constructor of a class is called. The same applies for the called by method attribute.

The third category has to do with the temporal restriction in which method calls occur. We can specify that the next method call has to be in the subtree of the current method call or that a method call has to be after another method call. These restrictions are useful to define that a call appears as a result of another call. For the Adapter pattern this is useful since the call to the Adaptee object is a result of the Adapter object being called by the Client. Figure 3.9, Figure 3.10 and Figure 3.11 explain these temporal restrictions in more detail. We represent the method calls during run-time in a tree structure. This shows the dependency of calls (senders) that call other methods (receivers) that might call other methods before they return back to the first call. Each node in the tree represents one specific method call during run-time. In these Figures the box represents the position of a matching call found in this tree of dynamic facts. The box represents the
position where the next method call has to be according to the dynamic definition. Here is an example: if we want to specify that the next method call needs to be in the subtree of the call then the next method call can only appear in the positions shown in Figure 3.9.

![Figure 3.9: Temporal restricting - next call in subtree](image)

We can also specify that a method call has to happen after another call. This concept is shown in Figure 3.10. In this Figure the next method call is represented by and appears somewhere after in the dynamic facts tree.

We can also specify that there is no order between two method calls, see Figure
3.11. In this Figure the next method call is represented by $\boxed{2}$ and can appear before or after $\boxed{1}$ in the dynamic facts tree. By default there is no order specified. Sometimes it is not important if objectA is created before objectB. Therefore having an order is not required but for almost all design patterns some kind of order of method calls applies. Being able to specify the order of method calls produces better results and increases the overall performance and efficiency.

The fourth category in our list is about the object restrictions. During runtime each object has its unique id. If we want to distinguish or compare objects at run-time then we can use the object restrictions in the dynamic facts. These
restrictions are useful to declare that method calls have to be from the same or different objects. It often happens that at run-time we have many objects of the same type i.e. Adapter objects. A problem can occur when we want to make sure that the method calls happens on the correct object. The following example demonstrates this case.

**Example:** We have four objects. Object A is of type Client, objects B and C are of type Adapter and object D is of type Adaptee. In our dynamic facts we have the following facts, represented here with arrows between the objects (see Figure 3.12, Case I). We would like to distinguish Case I and II.
In our dynamic definition we would like to specify that a Client object E calls an Adapter object F and this object calls the Adaptee object G (see Figure 3.12, Case II). Since we do not know which objects are created at run-time before we actually run the software we need to find a way to describe this scenario. By giving the attributes this object and called by object specific names we can use these names in other method calls. This way we make sure that the method calls are executed on the same objects.

In our running example with the Adapter design pattern we specify that in
the first method call the attribute \texttt{this object} is set to \texttt{object1}. For the second method call, we set the attribute \texttt{called by object} again to \texttt{object1} because we want to make sure that this call came from the same object that was created in the first method call. The dynamic definition of the Adapter pattern matches the description of Figure 3.12, Case II.

The last attribute from the dynamic definitions is \texttt{quantifier}. This can be set to any positive integer. This attribute is related to the ranking of the results and the nature of the detection process. If we define five different method calls in the dynamic definition and all but one of these calls are matched in the dynamic facts then this candidate instance is to 80\% a design pattern. We need to consider that different implementations of design patterns might use a slightly different approach than the original \textit{GoF} patterns. Therefore not all of our defined method calls might occur. If we do not allow one method call to be missing then we will not detect a design pattern for that candidate instance. To add flexibility to the approach we use this quantifier to calculate how likely a candidate instance can be declared to be a design pattern. If we set the quantifier to 1 for all method calls and we get four out of five matches, the possibility that the found instance is a pattern is 80\%. We can use this percentage to rank the found instances and use a threshold to reject matches that are too low to be considered a design pattern. Ranking the results is
another big advantage for the software re-engineer. This will help in focusing on
the more likely design patterns first and the less likely patterns at a later stage.

For some patterns it is also interesting to adjust the weight of each method
call according to its importance for the existence of a pattern instance. Weighting
method calls can be done by setting the quantifier attribute to a higher value. In
Figure 3.13 we see the UML sequence diagram for the Command design pattern.
The first method call in this example is a constructor call from the Client to the
Command object. Some implementations might implement this pattern slightly
different by creating the Command object already before and then passing it to the
Client. In cases when we are not concerned with what object constructed another
object, we can set the quantifier to a low value of 2. The constructor method
call is something that happens a lot at run-time. But a method call from one
role to another, is something more unique that can be weighted higher. If we find
such a match in our dynamic facts then this match is more important to us then
the constructor call mentioned before. This is the reason why we introduced the
quantifier attribute in our model. In our dynamic definitions we rank method
calls from 1 to 4 (where 1 is the lowest and 4 is the highest). We have achieved good
results while running tests against the GoF design patterns with these settings.
The following formula shows how we calculate the percentage on how good a candidate instance matches a design pattern:

\[
percentage = \frac{\sum_{i=1}^{n} q_i \times x}{\sum_{i=1}^{n} q_i} \tag{3.1}
\]

\(q_i\) = value of the quantifier at position \(i\) in the dynamic definition (\(1 \leq q_i \leq 4\))

\(n\) = number of method calls in the dynamic definition

\[x = \begin{cases} 
1 & \text{at least one method call matches the dynamic definition at position } i \\
0 & \text{otherwise}
\end{cases}\]

The last method call \texttt{Action()} in the Command pattern is another example for the use of the \textit{quantifier}. In some cases, the last method call to the receiver does not take place. It might not be needed due to \texttt{if-statements} or slightly different implementations other than described in the GoF book. Giving this last method call a low weight of \(1\) will still give us correct results but will include more flexibility in how things are implemented in real world applications.
Figure 3.13: UML sequence diagram for the Command pattern
3.3 Limitations

There is one limitation to this approach that has to be mentioned. Defining design patterns strictly according to the *GoF* pattern description will in some cases miss intended design patterns that have been implemented slightly different by developers. A solution to this problem would be to create multiple definitions for different implementations of design patterns. However, for now we focus only on the definitions used in the *GoF* book. We do this to show the effectiveness of our approach. If we can prove that our approach works with the *GoF* design patterns, than it will be easy to extend our software to other design patterns as well.
4 Tool - Pattern Detection Engine (PDE)

This chapter presents the implementation of our approach that we described in Chapter 3. First we give an overview of the different steps of the detection process. We talk about the static and then the dynamic analysis. We show how our tool works and the steps for detecting design patterns. Then we explain the design of the tool. After that we explain how we implemented the dynamic model and how the user can define their own dynamic definitions to detect other design patterns. At the end, we give an example from our experiments on how to use this tool.

4.1 Overview

The overview is divided into static and dynamic analysis. For the static analysis we use tools that were written at the University of Waterloo. For the dynamic analysis we developed a piece of software called Pattern Detection Engine that combines the static and dynamic facts of a software system and verifies the design pattern candidates that we found during the static analysis. Figure 4.1 shows the
different steps of the detection process. For the first stage we need the class files of
the software that we want to analyze. It is also useful to have UML class diagrams
and UML sequential diagrams of the design patterns that we want to detect. In the
second stage we manually create static definitions for the design pattern using the
UML class diagram. We create static facts using the static definition and the class
files of the software we are analyzing. For that stage we use the tools Javex, Grok
and QL. In the third stage we manually create dynamic definitions using the UML
sequence diagram. We also extract dynamic facts using Probekit from the Test and
Performance Tools Platform (TPTP) in Eclipse. In the fourth and last stage we
use the dynamic definitions, the static facts and the dynamic facts to process all
the information and detect the design pattern. The output ranks all found design
pattern instances according to how good they match the dynamic definition.
4.1.1 Static analysis

The first step in our detection process involves static analysis. For this part we extract static facts from Java classes using the tool Javex. This constructs a fact base that contains information about the class hierarchies, interface hierarchies, attributes and methods of classes. After that we use Grok to parse these facts and gather the following relations. The parenthesis below show the abbreviations that
Javex uses for the relations.

- class extends class (E258)
- class implements interface (E259)
- class overrides method (E256)
- class references class (E183)
- class contains method (contain)
- class contains attribute (contain)
- method calls method (E182, E183, E184, E178, E187)
- method creates class (E263)

Grok allows us to unify the fact base and reduce the facts to **inherits** and **uses** relations on class level. These are the only facts we use for the static analysis. These facts represent the structure of the program in a simplified way that relates to the UML notation of class diagrams. We lift all facts about attributes and methods to class level and then store these facts in a **inherits** or **uses** relation.

The following expression explains the process of lifting the facts to class level:

\[ m_1 \text{ calls } m_2 \land c_1 \text{ contains } m_1 \land c_2 \text{ contains } m_2 \Rightarrow c_1 \text{ calls } c_2 \]

\[ m_1, m_2 : \text{methods}, c_1, c_2 : \text{classes} \]
The following expression shows what facts we store in the uses and inherits relation:

\[
\text{uses} = \text{calls}(\text{class}, \text{class}) \cup \text{references}(\text{class}, \text{class}) \cup \text{creates}(\text{class}, \text{class}) \\
\text{inherits} = \text{extends}(\text{class}, \text{class}) \cup \text{implements}(\text{class}, \text{class}) \cup \text{overrides}(\text{class}, \text{class})
\]

The extracted facts are stored in Rigi Standard Format as shown in Figure 4.2. Each line represents a fact tuple.

<table>
<thead>
<tr>
<th>uses</th>
<th>classC classA</th>
</tr>
</thead>
<tbody>
<tr>
<td>uses</td>
<td>classA classE</td>
</tr>
<tr>
<td>inherits</td>
<td>classA classT</td>
</tr>
</tbody>
</table>

Figure 4.2: Facts in Rigi Standard Format

In the next step, we use QL to filter qualified sets of entities that match the static definition of the design pattern that we are looking for. Statically, design patterns are defined by their inheritance and uses relations on class level. For each role of a design pattern we specify what relations this role has with other roles. All relations are taken from the Adapter UML class diagram. Figure 4.3 shows the QL script for the Adapter design pattern. Appendix C introduces QL in more detail and shows how it is used to manipulate RSF facts.
The Adapter design pattern has four roles: the Client, the Target interface, the Adapter class and the Adaptee class. Statically we can express, that the Client uses the Adapter, that the Adapter class inherits from the target interface and that the Adapter uses the Adaptee. These are all static facts that make up the Adapter pattern. We use these relations and create a set of possible candidate instances that match this definition. We store these results as our static facts and use them later during our dynamic analysis.

4.1.2 Dynamic analysis

Before we can verify the possible candidate instances that we detected during the static analysis we need to create dynamic facts of the software that we are analyzing. By dynamic facts we mean information about method calls during run-time,
the order in which these method calls happen and the objects that are created at run-time. We decided to use Probekit from the Eclipse Test & Performance Tools Platform Project. This tool allows us to instrument the Java bytecode of the software we are analyzing. We collect the runtime data with user-defined probes. Probes are reusable Java code fragments that are written to collect detailed runtime information about objects, instance variables, arguments, and exceptions in a software. Probekit provides a framework on the Eclipse platform to create and use these probes. One advantage of these probes is that we only collect data that we are interested in. We can apply these probes on classes and jar files. Figure 4.4 shows the Probekit Editor. Here we define the data items (method names, arguments, objects) that we want to access during run-time and we specify the Java code that is added to the original class files.

We use Probekit to instrument every method throughout the software with additional code. Every time a method is called and exited it will write a line to a text file with information about the method name, class name, object id at run-time, arguments that were passed and other information. This gives us a very detailed overview of the software at run-time.

All these facts are written to a file. Later we will use these facts and convert them into an XML structure using DOM (W3C Document Object Model). This
In order to create good dynamic facts it is necessary that we call as many methods in a software as possible. In cases when user input is involved it is very hard to cover every part of a software. Therefore it is useful having good test suites for a software system since they cover most method calls.

Figure 4.5: Tip: Use test suites of software to create extensive dynamic facts
will help us parse the file during the detection process. Instrumenting the bytecode rather than the source code directly has many advantages. First of all, we only specify what information we want to collect from the methods and all methods will be instrumented exactly the same. Therefore, we get uniform information from each method call during run-time that look like the results from Figure 4.6:

```xml
<entry args="" callDepth="2" calledByClass="ajp_code/adapter/RunPattern"
calledByMethod="main"
calledByObject="ajp_code/adapter/RunPattern@0"
className="ajp_code/adapter/ContactAdapter"
methodName="Constructor" orderNumber="2"
thisObject="ajp_code/adapter/ContactAdapter@14576877">
<entry args="" callDepth="3" calledByClass="ajp_code/adapter/ContactAdapter"
calledByMethod="Constructor"
calledByObject="ajp_code/adapter/ContactAdapter@14576877"
className="ajp_code/adapter/ChovnatlhImpl"
methodName="Constructor" orderNumber="3"
thisObject="ajp_code/adapter/ChovnatlhImpl@11077203">
```

Figure 4.6: Excerpt from dynamic facts

Touching the source code of a software system is not needed in the pattern detection process. Our approach works with compiled class files and instruments the Java bytecode with additional code. The code will not interfere with the real program. This makes a good case for using Probekit with any kind of software. Appendix D.1 shows how Probekit works and how it is used in detail. For the dynamic analysis it is only important to understand that we work with dynamic
facts that show the method calls that were executed during run-time.

At this point of the detection process we have a list of candidate instances from our static analysis and a large fact base of dynamic facts. We will now combine these facts in our software to verify the candidate instances.
4.2 Software design

Our software consists of different parts. First we process the candidate instances from the static analysis. After that we process the dynamic facts from the facts that were created using Probekit. Then we read the dynamic definitions of the design patterns. After that we iterate over all candidate instances and compare them with the dynamic pattern definitions if we find correct matches in the dynamic facts. At the end, we rank the results and report all pattern instances that were verified.

4.2.1 Static facts

In the first step of PDE we store all candidate instances in a linked list in the CandidateInstancesProcessor class. Later we will iterate through this list and check if a candidate instance matches the dynamic definition of the design pattern.

The input file with the candidate instances has to match the following format:

```plaintext
// client target adapter adaptee
adapter.RunPattern adapter.Contact adapter.ContactAdapter adapter.Chovnatlh
adapter.RunPattern adapter.Contact adapter.ContactAdapter adapter.ChovnatlhImpl
```

Figure 4.7: Adapter candidate instances

The example in Figure 4.7 contains four roles of the Adapter design pattern in
the first line: client role, target role, adapter role and adaptee role. The other two lines contain possible candidate instances. The number of classes in each line has to match the number of roles. In the CandidateInstanceProcessor we create a new CandidateInstance object for each line with the names of the classes and the name of the roles from the first line. This way we know what class represents which role in each CandidateInstance object.

4.2.2 Dynamic facts

In the second step we parse the dynamic facts that we created with Probekit. PDE reads the dynamic facts line by line and transforms the input into XML format. Each line will correspond to a Node in an XML document. The DynamicFactsProcessor class transforms the input file into DOM XML structure and stores all Nodes in an XML document. The next step involves some post processing. We iterate over the XML document and add an order number for each method call as well as facts about the parent method call. This gives us additional facts for each Node (method call) so that we know who called a method and at what point during the program execution.
4.2.3 Dynamic definition

As described in Chapter 3, we created dynamic definitions for the GoF design patterns. These definitions are expressed in XML. The structure has to match the DTD XML definition that is provided with PDE (see Figure 4.8).

```
<!ELEMENT entry (entry*)>
<!ATTLIST entry args CDATA #REQUIRED>
<!ATTLIST entry className CDATA #REQUIRED>
<!ATTLIST entry methodName CDATA #REQUIRED>
<!ATTLIST entry calledByClass CDATA #REQUIRED>
<!ATTLIST entry calledByMethod CDATA #REQUIRED>
<!ATTLIST entry calledByObject CDATA #REQUIRED>
<!ATTLIST entry thisObject CDATA #REQUIRED>
<!ATTLIST entry nextCallInSubtree CDATA #REQUIRED>
<!ATTLIST entry nextCallInOrder CDATA #REQUIRED>
<!ATTLIST entry quantifier CDATA #REQUIRED>
```

Figure 4.8: DTD XML Schema for the dynamic definition

The dynamic definitions have to be specified by the user. For the GoF patterns we provide templates which can be used as examples to detect other design patterns. Figure 4.9 is an example of how dynamic definitions look like. The definition is represented as an XML file with a node for each method call. The structure of the XML file is given by the DTD XML Schema. The Schema is used to validate the XML before processing the dynamic definition. The elements of the XML file, shown with a little icon, represent the method calls. The attributes, icon,
represent the values that have to be matched.

![Figure 4.9: Dynamic definition of the Adapter design pattern](image)

The **DynamicDefinitionConverter** class handles these definitions. Since we use the role names in the definition as place holders for the actual class names we need to replace them by the real class names. The **DynamicDefinitionConverter** class substitutes the role names in the dynamic definition with the class names from the candidate instance. It takes the dynamic definition as input and one candidate instance object. It then exchanges the roles of the dynamic definition with the real class names of the candidate instances. This process is shown in Figure 4.10.
the end of this substitution process we have a dynamic definition in XML format that can be used to find matches in the dynamic facts. The role names can only be used for the following attributes in the dynamic definitions: `arguments`, `className` and `calledByClass`.
Figure 4.10: Dynamic definition conversion
4.2.4 Validation

The **Validator** class does most of the work in our software. It takes the transformed dynamic definitions and finds matching nodes in the dynamic facts. At the initial stage it compares the following attributes from the dynamic definitions and the dynamic facts: `className`, `methodName`, `calledByClass`, `calledByMethod` and `arguments`. If these attributes match the method calls in the dynamic facts then the matching nodes from the dynamic facts are stored in a data structure. The data structure is an array of linked lists. The length of the array corresponds to the length of the dynamic definitions. Figure 4.11 shows an example for a `Node_{x1}` and the values that it can contain, see Figure 4.11 (point 1). If we find a matching node in the dynamic facts, see Figure 4.11 (point 2), then we add this node to the linked list at the first position in the array, see Figure 4.11 (point 3). It is likely that a dynamic definition will match many Nodes from the dynamic facts, therefore we add all of them to their corresponding positions. Figure 4.11 shows this data structure graphically. The array contains a linked list for each dynamic definition. If each linked list contains at least one node then the dynamic facts match the dynamic definitions 100% because we found a match for each node from the dynamic definition. In Figure 4.11 only 4 out of 5 dynamic definitions were matched. For the 4th field in the array we did not find a match. In the next step
of the processing we take the data structure with all nodes and check if the order of the method calls is correct.

We also check if the object ids are correct and match the dynamic definition. If some nodes do not match these definitions than we remove them from this data structure. For our final reporting, we loop through the data structure and check if there is at least one node for each field in the array. Since the stored nodes include all details, we can later print out the exact method calls for further analysis. Therefore, we know the method names, the class names, arguments and many more information that could be useful for the user. We believe that this information can be used in future work to remove even more false positives from the results.

The current implementation does not store the relations between different nodes that are dependent on each other. Here is an example of what we mean. If Node\textsubscript{x2} has to be in the subtree of Node\textsubscript{x1} then we would specify this in the dynamic definition by setting nextCallInSubtree to yes. As a result of the processing we would get Node\textsubscript{1}, Node\textsubscript{2} and Node\textsubscript{4} for the first linked list and Node\textsubscript{3} and Node\textsubscript{5} for the second linked list. Currently this data structure does not contain any additional information if Node\textsubscript{3} was validated because of Node\textsubscript{1}, Node\textsubscript{2} or Node\textsubscript{4}. This is not crucial for our current approach, since we do not use this information but future work might want to improve on this limitation.
Figure 4.11: Matching dynamic definition in dynamic facts
4.3 Model

The dynamic definitions are defined in XML form. The XML schema is shown in Figure 4.8. We defined this schema to match our approach. For future work this can be extended with additional attributes. Figure 4.12 shows the template of one node in graphical form. Each node represents a method call in the dynamic facts.

![Dynamic definition template](image)

Figure 4.12: Dynamic definition template

Our tool uses ten attributes for each node to detect and verify design patterns during the dynamic analysis. The dynamic definitions usually contain a couple of nodes. For most of the GoF design patterns we use three to five nodes to define the design patterns dynamically. As described in the Chapter 3, each attribute is used to specify values of a specific method call in the dynamic facts. Not all attributes need to contain a value. Here is the list of attributes and their usage. Using these attributes allows us to create a very precise hierarchy of method calls that describe
the design pattern.

- **args** - this field specifies the parameters that are passed to a method. We are interested in objects that represent the roles of a design pattern, therefore we insert the role names of a design pattern in this field (i.e. *adapter*, *adaptee* or *client* for the Adapter design pattern).

- **className** - this field specifies the class that contains this method. Here we also use the role names for the design pattern (i.e. *adapter*, *adaptee* or *client* for the Adapter design pattern).

- **calledByClass** - this field specifies the class from which this method was called. Here we also use the role names for the design pattern (i.e. *adapter*, *adaptee* or *client* for the Adapter design pattern).

- **methodName** - here we can specify if we are looking for a Constructor method call or if we know a specific method name, ex. *visit()* for the Visitor design pattern.

- **calledByMethod** - here we can specify the name of the method from which the current method was called.

- **nextCallInSubtree**
  
  - yes: the next method call is in the subtree of this method call.
- no (default): the next method call does not have to be in the subtree of this method call.

- nextCallInOrder

  - yes: the next method call needs to be after the current method call.
  
  - no (default): the next method call can be before or after the current method.

- thisObject - this defines the object id (memory location) of this object. For this field we can define a variable that is used as a place holder for the object id of the objects at run-time. When the same place holder is used more than once in a dynamic definition, PDE will verify that the objects ids at run-time are the same. In the dynamic facts these object ids contain the dynamic value \texttt{object + @ + hash value} of the memory location of the object.

- calledByObject - identical to the attribute \texttt{thisObject}, except that this attribute is used to specify from which object the method call came from.

- quantifier - this attribute contains an integer greater than zero that can be used to rank the weight of the method calls in the dynamic definition. Since there is a number of different method calls, we can value their importance for a specific design pattern. This is useful when we are ranking the candidate
instances according on how good they match the dynamic definitions.

4.4 Example

In this last section we give an example with on how to use the Pattern Detection Engine. Before running PDE we need the static facts from QL, the dynamic facts from Probekit and the dynamic definition for the design pattern that we want to detect/verify. In the Appendix D we have examples on how to extract the static and dynamic facts.

4.4.1 Command pattern

We use the Command design pattern to demonstrate each stage of our detection process. Figure 4.13 shows the UML class diagram of the Command pattern. During the first stage we extract the static definitions from this UML diagram. The Command pattern has 4 roles: Invoker, Command, ConcreteCommand and Receiver. We have to describe the uses and inherits relations of these roles between each other. There are three relations that are interesting:
1. Invoker uses Command
2. ConcreteCommand inherits Command
3. ConcreteCommand uses Receiver

![UML class diagram for the Command pattern](image)

Figure 4.13: UML class diagram for the Command pattern

Then we use these relations and create a QL script that represents these relations, see Figure 4.4.1.

For the next step we use Javex to extract static information from the class files of the software that we analyze, see Appendix C.1 how to run Javex. Then we lift all facts to class level since we are only interested in the relations between classes, see Appendix C.2. In the second stage we run QL with the lifted facts from Grok and the QL script, see Figure 4.4.1. The result is a list of candidate instances. We store these results in a file and call them static facts from now on.

In the third stage we extract dynamic facts. We use Probekit, see Appendix D.1, to extract these facts at run-time of the application that we analyze. We also create the dynamic definition using the UML sequence diagram, see Figure 4.15. The
Figure 4.14: QL script for Command design pattern

first method call creates the ConcreteCommand object. We specify the attribute thisObject and set it to object1 so we can use it for the other method call where we want to make sure that exactly this object is used. In the second method call we pass the ConcreteCommand object as an argument to the Invoker. In the third method call the ConcreteCommand object with the id object1 is called by the Invoker. In the last method call the Receiver is called by the ConcreteCommand object with the id object1. This method call gets a low value for the quantifier since this method call is not always invoked during run-time.
Figure 4.15: UML sequence diagram for the Command pattern
1. First method call
   - className="concreteCommand"
   - methodName="Constructor"
   - thisObject="object1"
   - nextCallInOrder="yes"
   - quantifier="2"

2. Second method call
   - args="concreteCommand"
   - className="invoker"
   - nextCallInOrder="yes"
   - quantifier="2"

3. Third method call
   - className="concreteCommand"
   - calledByClass="invoker"
   - thisObject="object1"
   - nextCallInOrder="yes"
   - quantifier="4"

4. Fourth method call
   - className="receiver"
   - calledByClass="concreteCommand"
   - calledByObject="object1"
   - quantifier="1"

In the fourth stage we use the static facts, the dynamic facts and the dynamic
definition and pass all information to PDE. Our tool analyzes and compares the
dynamic facts with the dynamic definitions for all candidate instances and prints
the results on the command line or in a file. An example of the output can be seen
in Figure 4.16.
4.4.2 Ranking the results

A nice example of how the results are ranked is given in Figure 4.16. This example shows the results from the Command pattern implementation that we used from the *Applied Java Patterns book* (SL01). After the static analysis we found 38 possible candidate instances. The dynamic analysis verified 7 out of these 38 candidates that passed the threshold of 80%. All 7 candidates are ranked by their percentage in decreasing order. The output in Figure 4.16 gives us the role names of the design pattern in line 9 and the class names from the example source code for each candidate that matches these roles. The user can use this output to go through the application, analyze the listed classes and verify manually if the candidates are part of this design pattern or not.
# Code example from: command

# Pattern we want to detect: adapter

Number of positive candidate instances after the dynamic analysis:
7 out of 38 (threshold = 80%)

Here is a ranked list of all candidate instances with the corresponding class names (and pattern roles): (0=target, 1=adapter, 2=adaptee)

0 97% {0=java.awt.event.ActionListener, 
1=ajp_code.command.CommandGui, 
2=ajp_code.command.Appointment}

1 96% {0=java.awt.event.ActionListener, 
1=ajp_code.command.CommandGui, 
2=ajp_code.command.LocationImpl}

2 95% {0=java.awt.event.ActionListener, 
1=ajp_code.command.CommandGui, 
2=ajp_code.command.CommandGui$WindowCloseManager}

3 94% {0=ajp_code.command.UndoableCommand, 
1=ajp_code.command.ChangeLocationCommand, 
2=ajp_code.command.Appointment}

4 92% {0=ajp_code.command.LocationEditor, 
1=ajp_code.command.CommandGui, 
2=ajp_code.command.Appointment}

5 89% {0=ajp_code.command.LocationEditor, 
1=ajp_code.command.CommandGui, 
2=ajp_code.command.LocationImpl}

6 80% {0=ajp_code.command.LocationEditor, 
1=ajp_code.command.CommandGui, 
2=ajp_code.command.CommandGui$WindowCloseManager}

Figure 4.16: Ranking the output of PDE
5 Experiments

We compared our tool, Pattern Detection Engine (PDE), with two other pattern detection tools, PINOT and FUJABA 4.2.0 (with Inference Engine version 2.1). In this chapter we present the results from our experiments. We tested the tools in terms of accuracy against the demo source from Applied Java Patterns (AJP) (SL01). This book contains example code for the 23 GoF design patterns. We also conducted robustness tests with PDE using JHotDraw version 7.0.8. For the robustness test we created 42,000 lines of dynamic facts from the running JHotDraw software, see http://www.jhotdraw.org. After that we ran PDE with the Adapter, Command and Composite pattern. The results are presented at the end of this Chapter.

5.1 Tool comparison

PINOT is a fully automated pattern detection tool. It takes a source package and detects the pattern instances. In the latest version that was available to us all
detection algorithms for the design patterns are hard-coded. Currently PINOT can not be used to detect other design patterns other than the 23 GoF design patterns. They reclassify the patterns into language-provided, structure-driven and behavior-driven patterns. Iterator and Prototype belong to the language-provided category. Bridge, Composite, Adapter, Facade, Proxy, Template Method and Visitor pattern are grouped together in the structure-driven patterns. Singleton, Abstract Factory, Factory Method, Flyweight, Chain of Responsibility, Decorator, Strategy, State, Observer and Mediator belong to the group of behavior-driven patterns. In our experiments, PINOT was able to detect 17 out of 23 design patterns.

FUJABA is a Tool Suite that contains a number of tools for software engineering and reverse engineering. Design patterns are defined as graph transformation rules, with respect to the abstract syntax graph (ASG) of a system’s source code. FUJABA allows the user to define patterns in a visual UML-like format. In order to run FUJABA the user only need to specify the location of the source code. In our experiments, FUJABA did not perform as well as PINOT and our tool. It only detected 14 out of 23 design patterns.

Our tool was able to detect 22 out of 23 design patterns. The only pattern that we can not detect is Facade. We can not define this pattern statically and
dynamically with our approach. Since the Facade pattern is more an architectural pattern rather than a design pattern it makes it hard to detect.

PDE allows to set a threshold that determines how much the dynamic definitions have to match the dynamic facts in order to declare a candidate instance to be a design pattern instance. During our experiments we discovered that a threshold of 80% will return the best results for most design patterns. Since we rank all candidates based on how well they match the definitions, the user can start looking at the more likely design pattern instances and then move to candidates that are less likely to be design patterns.
5.2 Results

The following Figure 5.2 shows the results from our experiments. We ran all three tools with sample implementations of the 23 GoF design patterns. We took the implementations from the Applied Java Patterns (AJP) book (SL01). We ran our experiments on a Pentium 4, 2.2GHz computer with 512 MB RAM.

The first observation is that even though FUJABA has a nice user interface and many features, it was only able to detect 14 out of 23 patterns. The inference engine in FUJABA claims to recognize 16 out of the 23 patterns. Behavioral patterns seem to be particularly hard to detect with FUJABA. During our tests FUJABA was not able to detect the State and Prototype design pattern even though it claims to do so in the documentation.
<table>
<thead>
<tr>
<th></th>
<th>PDE</th>
<th>PINOT</th>
<th>FUJABA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creational</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Abstract Factory</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Builder</td>
<td>√</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Factory Method</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Prototype</td>
<td>√</td>
<td>-</td>
<td>№</td>
</tr>
<tr>
<td>Singleton</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Structural</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adapter</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Bridge</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Composite</td>
<td>√</td>
<td>№</td>
<td>√</td>
</tr>
<tr>
<td>Decorator</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Facade</td>
<td>-</td>
<td>√</td>
<td>-</td>
</tr>
<tr>
<td>Flyweight</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Proxy</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Behavioral</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chain of Resp.</td>
<td>√</td>
<td>√</td>
<td>-</td>
</tr>
<tr>
<td>Command</td>
<td>√</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Interpreter</td>
<td>√</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Iterator</td>
<td>√</td>
<td>-</td>
<td>√</td>
</tr>
<tr>
<td>Mediator</td>
<td>√</td>
<td>√</td>
<td>-</td>
</tr>
<tr>
<td>Memento</td>
<td>√</td>
<td>√</td>
<td>-</td>
</tr>
<tr>
<td>Observer</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>State</td>
<td>√</td>
<td>√</td>
<td>№</td>
</tr>
<tr>
<td>Strategy</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Template Method</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Visitor</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td><strong>Sum</strong></td>
<td>22/23</td>
<td>17/23</td>
<td>14/23</td>
</tr>
</tbody>
</table>

✓  tool is able to correctly identify the pattern in the code example.

№  tool claims to recognize this pattern but fails to identify it in the example.

-  tool excludes recognition for this pattern.
PINOT on the other hand is able to detect 17 out of the 23 patterns. Five patterns are currently not hard coded in PINOT and can not be detected. The Composite pattern was not detected even though the documentation of the tool claims it can. Hard coding the patterns has advantages since each pattern can be approached different. The disadvantage on the other hand is that the tool can not be easily extended to detect other design patterns. The results from PINOT are good and the tool is easy to use. It prints all found design patterns with the roles and classes.

The results from PDE are even better than the aforementioned tools. We are able to detect 22 out of 23 design patterns. Just like PINOT, our tool does not provide a GUI that shows the results graphically. We follow the same approach as PINOT in printing the results on the console. One advantage of our approach is that the results are ranked, meaning more likely design pattern instances appear on top of the results. Another advantage is the extensibility of the design patterns that can be detected. Our tool can detect any kind of design pattern, that can be defined in terms of static and dynamic definitions. As shown in the results above it is possible to detect creational, structural and behavioral patterns with our tool.
5.3 PINOT vs. PDE

Our first experiments indicate that PINOT and PDE produce the best results for detecting design patterns. Therefore we decided to take a closer look at the results from both tools. We ran the two tools again with the example code from AJP. This time we wanted to know how many false positives the tools would produce. For each code example, we checked how many other design pattern instances the tool detects. For example, we used the Adapter code from the *Applied Java Pattern code* and checked if we can detect Builder, Command, Composite and all other patterns in the Adapter pattern example. Since the code examples are very small and only involve a couple of classes, we do not expect to detect many design patterns in each code example.

Figure 5.1 shows the detailed results for PINOT. Each row represents the pattern implementation used for the experiment and the column the design pattern that we detected in the code. The numbers in the table show how often the design pattern was detected by the tool in that code. The diagonal in the table represents experiments where a design pattern was detected against its implementation. Each field in the diagonal should be greater than zero. For example, cell (3,3) contains a 1, which means that PINOT found one instance of the Adapter design pattern.
in the Adapter code from the *Applied Java Pattern book*. On the other hand, cell (5,5) shows a zero. Therefore PINOT was not able to detect the Builder design pattern in the Builder code example. An example for a false positive is cell (7,1). Here PINOT detected 1 Abstract Factory in the code example of the Command design pattern. On the diagonal PINOT is able to detect 17 out of 23 design patterns correctly. We manually verified the results on the diagonal and the pattern instances that were found by PINOT are correct.

The results show a lot of false positives for the Facade, Flyweight, Mediator, Proxy and Strategy design pattern. Manual inspection of the code could not verify so many pattern instances. Even though there are also many false positives in the result set, we believe that the overall results from PINOT are good.
Figure 5.1: Detailed results for PINOT. On the left side are the code examples and the top represents the design patterns that we detected.
Figure 5.2 shows the detailed results from PDE using a threshold of 100%. The results are represented the same as before with PINOT. The diagonal shows all the correct matches. With a threshold of 100%, PDE is not able to detect the Adapter, Decorator, Flyweight, Interpreter, Iterator, Mediator, Observer, State and Strategy pattern. In total PDE detects 122 pattern instances for all 22x22 combinations. 28 of these pattern instances are correctly identified on the diagonal. We manually checked the results where PDE would detect a design pattern in different examples. For the Abstract Factory and Factory Method PDE correctly identified cell(2,1) as a match. The manual inspection of the results is discussed later in this chapter and can be viewed in Figure 5.5. To detect more design patterns we modified the threshold and got better results with a threshold of 80% (see Figure 5.3).
Figure 5.2: Detailed results for PDE (threshold=100%). On the left side are the code examples and the top represents the design patterns that we detected.
Figure 5.3 shows the detailed results from PDE using a threshold of 80%. With this threshold, PDE is able to detect all 22 design patterns correctly. Overall PDE finds more pattern candidates. In total PDE finds 297 pattern candidates for all 22x22 combinations in the table. 44 of these candidates are on the diagonal and match the correct pattern with the correct implementation. There are a lot of candidates for the Abstract Factory, Adapter, Command, Memento, Singleton, State and Strategy pattern. It is important to remember that PDE will rank the results on how well they match the dynamic pattern definitions. Therefore it is recommendable to take a look at the more likely pattern candidates first that are ranked with a higher percentage.
Figure 5.3: Detailed results for PDE (threshold=80%).
Figure 5.4 shows the results of Figure 5.2 and Figure 5.3 combined. All patterns that were correctly detected using a threshold of 100% are added to this table. For the Adapter, Decorator, Flyweight, Interpreter, Iterator, Mediator, Observer, State and Strategy pattern we use the results with a threshold of 80%. If we specify the threshold for each pattern separately we can reduce the number of possible candidates that do not match the dynamic definition exactly. In the last row of Figure 5.4 there is a field for each pattern that specifies which threshold we used to create these results. Overall we reduce the number of candidates from 297 by 30% to 205. The number of candidates on the diagonal is 44, which is 21% of the overall number of candidates.
Figure 5.4: Detailed results for PDE (combined threshold of 80% and 100%).
After analyzing the results from the previous Figures we wanted to know how good the results from PDE are. We wanted to know how many of the candidate instances in the tables are true positives, false positives, true negatives and false negatives. We were especially interested in the results from the diagonal in the previous tables. Therefore we looked at the output from PDE, the AJP documentation and the source code to compare the results. Figure 5.5 shows our manually verified results for each design pattern. On the left side of the table we listed all design pattern implementations. The first column shows the number of candidate instances that were detected during the static analysis. The other columns show the number of true positives, false positives, true negatives and false negatives for the results with a threshold of 80% and 100%. True positives are correctly identified pattern candidates in the source code. True negatives are correctly rejected candidates. False positives are candidates that are identified as a pattern even though they are not correct pattern implementations. False negatives are candidates that are rejected by PDE even though they are correct pattern implementations.

The static analysis found 106 possible candidate instances. Using a threshold of 80% the result of PDE is 96% correct. 102 out of 106 candidates are identified correctly either as true positives or true negatives. Only 4 candidates are not correctly identified. Using a threshold of 100%, PDE is able to identify 89 candidates correct. Therefore 84% of the results with a threshold of 100% are correct. There are 16
candidates that PDE identified wrong as false negatives even though they are true positives. After inspecting these results we found that for most of these candidates not all method calls from the dynamic facts could be matched. Therefore PDE ranked these candidates between 80% to 99%. This supports our approach that we introduced the quantifier and threshold. Ranking the results allows us to define precise dynamic definitions knowing that not all method calls are always executed during run-time.

<table>
<thead>
<tr>
<th>Threshold</th>
<th>Number of Candidate Instances</th>
<th>True Positives</th>
<th>False Positives</th>
<th>True Negatives</th>
<th>False Negatives</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>80%</td>
<td>100%</td>
<td>80%</td>
<td>100%</td>
</tr>
<tr>
<td>Abstract Factory</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Factory Method</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Adapter</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Builder</td>
<td>9</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Chain of Responsibility</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Command</td>
<td>7</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Composite</td>
<td>6</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Decorator</td>
<td>6</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Flyweight</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Interpreter</td>
<td>16</td>
<td>14</td>
<td>14</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Iterator</td>
<td>8</td>
<td>7</td>
<td>7</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Mediator</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Memento</td>
<td>12</td>
<td>1</td>
<td>1</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>Observer</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Prototype</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Proxy</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Singleton</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>State</td>
<td>7</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Strategy</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Template Method</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Visitor</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

|                           | True Positives | False Positives | True Negatives | False Negatives |
|                           | 80% | 100% | 80% | 100% | 80% | 100% | 80% | 100% |
| Sum                       | 106  | 40   | 27  | 4    | 1   | 62  | 62  | 0    | 16   |
| Percentage                | 38% | 25% | 4%  | 1%  | 58% | 58% | 0%  | 15%  |

<table>
<thead>
<tr>
<th>Threshold</th>
<th>80%</th>
<th>100%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correct results</td>
<td>102</td>
<td>89</td>
</tr>
<tr>
<td>False results</td>
<td>4</td>
<td>17</td>
</tr>
<tr>
<td>Percentage</td>
<td>96%</td>
<td>84%</td>
</tr>
</tbody>
</table>

Figure 5.5: Manually checked results from PDE for 22 design patterns.
5.4 PDE model

In this section, we take a closer look at our tool. In particular, we show our improvements removing false positives using dynamic analysis. Figure 5.6 shows several interesting results that we collected while running PDE against the set of design pattern implementations. We used the same approach as in Chapter 5.3. For each AJP code example we tried detecting all 22 design patterns. Each row in Figure 5.6 contains the sum of all patterns that were detected. The first column shows the number of candidate instances we obtained as a result of our static analysis. For the Adapter design pattern we obtained 30 candidate instances. The second column shows the number of instances deemed to be true implementations of the design pattern. For the Adapter pattern, PDE recognized two instances. This shows that the dynamic analysis removes most candidates that were found during the static analysis. Overall, dynamic analysis reduces the number of instances by 89%. This number is rather high and there are several reasons that lead to this result. Our dynamic model defines the method calls that need to occur during run-time as well as the order in which the method calls have to happen. If we are detecting design patterns in software and we do not run all parts of that software, it is difficult to understand every part of that software. It is crucial that we make use of test suites that call all parts of a software so that we can record these method calls
and use these facts during the dynamic analysis. The AJP implementations of the 
GoF design patterns are very small and there is no overhead of classes that are not 
necessary for the implementation of the design pattern. Therefore all methods are 
called during run-time. If there are design patterns used in an application but they 
are never called during run-time then we will not be able to detect them. The user 
can still use the results from the static analysis which will give him a large number 
of candidate instances. Statically, many design patterns are similar and they do 
not differ in their inheritance and uses relations.

Another result we were interested in is the effectiveness of defining an order in 
which method calls need to appear in the dynamic facts. We called this model 
temporal restrictions in the approach Chapter. The second column in Figure 
5.6 shows the results when using an order for the dynamic definitions. The third 
column shows the results when we do not use any type of order. We were able to 
reduce the number of possible candidates by 32% in our tests when we used order. 
We reduced the results from 435 to 297 detected instances in all 22x22 tests. To 
verify these results we randomly picked 20 instances that were removed using the 
temporal restrictions and 19 of these 20 were removed correctly. This supports our 
opinion that many design patterns can only be distinguished if we look at their 
dynamic behavior and try to capture this behavior for our detection. For a begin-
ning we are pleased with the results but see also different ways for improvement.
The dynamic definitions can be improved to be more specific to remove more false positives. Overall, the model we introduced seems to work very good for detecting design patterns.

<table>
<thead>
<tr>
<th></th>
<th>with temporal restrictions</th>
<th>without temporal restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Threshold:</strong> 80%</td>
<td></td>
<td></td>
</tr>
<tr>
<td># of Candidate Instances</td>
<td># of detected design patterns</td>
<td># of detected design patterns</td>
</tr>
<tr>
<td>Abstract Factory</td>
<td>103</td>
<td>12</td>
</tr>
<tr>
<td>Factory Method</td>
<td>51</td>
<td>8</td>
</tr>
<tr>
<td>Adapter</td>
<td>30</td>
<td>2</td>
</tr>
<tr>
<td>Bridge</td>
<td>83</td>
<td>5</td>
</tr>
<tr>
<td>Builder</td>
<td>105</td>
<td>19</td>
</tr>
<tr>
<td>Chain of Responsibility</td>
<td>134</td>
<td>2</td>
</tr>
<tr>
<td>Command</td>
<td>118</td>
<td>20</td>
</tr>
<tr>
<td>Composite</td>
<td>56</td>
<td>11</td>
</tr>
<tr>
<td>Decorator</td>
<td>236</td>
<td>15</td>
</tr>
<tr>
<td>Flyweight</td>
<td>92</td>
<td>10</td>
</tr>
<tr>
<td>Interpreter</td>
<td>503</td>
<td>38</td>
</tr>
<tr>
<td>Iterator</td>
<td>51</td>
<td>1</td>
</tr>
<tr>
<td>Mediator</td>
<td>201</td>
<td>42</td>
</tr>
<tr>
<td>Memento</td>
<td>30</td>
<td>2</td>
</tr>
<tr>
<td>Observer</td>
<td>62</td>
<td>20</td>
</tr>
<tr>
<td>Prototype</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Proxy</td>
<td>61</td>
<td>7</td>
</tr>
<tr>
<td>Singleton</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>State</td>
<td>202</td>
<td>27</td>
</tr>
<tr>
<td>Strategy</td>
<td>98</td>
<td>11</td>
</tr>
<tr>
<td>Template Method</td>
<td>46</td>
<td>7</td>
</tr>
<tr>
<td>Visitor</td>
<td>240</td>
<td>42</td>
</tr>
<tr>
<td><strong>Number of design patterns</strong></td>
<td>2667</td>
<td>297</td>
</tr>
<tr>
<td><strong>% of false positives removed</strong></td>
<td>89%</td>
<td>84%</td>
</tr>
</tbody>
</table>

Figure 5.6: Results from dynamic analysis
5.5 Robustness

In this section we present benchmarks we ran with PDE while extracting the results for Section 5.3. After that we show the results after we ran PDE on a real life software. We chose JHotDraw for these tests.

The following results were created while we ran PDE with all 484 combinations of the design pattern (22x22). The static analysis took 2:40 min. Part of the static analysis is Javex, Grok and QL. Javex produced 34.017 lines of facts from all 22 design patterns. Grok reduced these facts to 2081 lines after lifting all facts to class level. At the end QL produced 2667 design pattern candidates from these facts. The dynamic analysis took 17:40 min until it processed all 484 combinations. Instrumenting the source code and creating the dynamic facts took 10 min, since we needed to run all 22 AJP implementations separately. For the dynamic analysis PDE used 2667 lines of static facts and 7230 lines of dynamic facts. These tests were run on a Pentium 4, 2.2 GHz computer with 512 MB RAM.

In order to see how well our approach works with a real software we conducted additional tests. Figure 5.7 shows our results with a large software system. We chose JHotDraw version 7.0.8 to test our approach and PDE for robustness. For
the static analysis we extracted the candidate instances for the Adapter, Command and Composite design pattern. The static analysis involved running Javex, Grok and QL. In the worst case this took twelve seconds for the Command pattern. For the Adapter and Composite pattern it took 1 second each.

For the dynamic analysis we extracted 11,000 and 42,000 lines of dynamic facts. Instrumenting JHotDraw with Probekit and creating the dynamic facts took us about 5 minutes. Then we measured the times for detecting all three design patterns when running PDE. We used a threshold of 80% while detecting the pattern instances. The dynamic detection lasted about 1 hour for the Adapter and Composite pattern. In both cases we had about 1,000 to 1,600 candidates and 42,000 lines of dynamic facts. The Command pattern needed about 6 hours to process all 12,000 candidates together with 42,000 lines of dynamic facts. The run-time between 11,000 and 42,000 lines of dynamic facts increases linear. Figure 5.7 shows the running time for the four most important parts of PDE. The first involves processing all candidate instances. This took less then a second for all design patterns since PDE only had to process 1,000 to 12,000 candidates. Processing the 42,000 lines of dynamic facts took about 70 seconds. This is identical for all three patterns since they all used the same facts. The next part involves validating the roles in the dynamic definitions. This is the most intensive part of PDE since we have to detect matches of the dynamic definitions in the dynamic facts. The last part does
additional verification that is used for checking temporal restrictions and object restrictions from the dynamic definitions.

We see opportunities for improving the run-time of PDE but for now we leave this for future work.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Adapter</th>
<th>Command</th>
<th>Composite</th>
<th>Adapter</th>
<th>Command</th>
<th>Composite</th>
</tr>
</thead>
<tbody>
<tr>
<td># of lines in dynamic facts</td>
<td>11000</td>
<td>11000</td>
<td>11000</td>
<td>42000</td>
<td>42000</td>
<td>42000</td>
</tr>
<tr>
<td># of possible candidates</td>
<td>1691</td>
<td>12246</td>
<td>1026</td>
<td>1691</td>
<td>12246</td>
<td>1026</td>
</tr>
<tr>
<td>Time for Static Fact Extraction</td>
<td>1 sec</td>
<td>1 sec</td>
<td>1 sec</td>
<td>1 sec</td>
<td>1 sec</td>
<td>1 sec</td>
</tr>
<tr>
<td># of found instances</td>
<td>308</td>
<td>13</td>
<td>1</td>
<td>308</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>Time for Candidate Instance Collection()</td>
<td>0.127</td>
<td>0.310</td>
<td>0.038</td>
<td>0.126</td>
<td>0.444</td>
<td>0.087</td>
</tr>
<tr>
<td>Time for Dynamic Facts Processor()</td>
<td>74</td>
<td>72</td>
<td>71</td>
<td>1333</td>
<td>1311</td>
<td>1215</td>
</tr>
<tr>
<td>Time for Validator validate()</td>
<td>308</td>
<td>3673</td>
<td>204</td>
<td>1642</td>
<td>18318</td>
<td>1401</td>
</tr>
<tr>
<td>Time for Validator validateObjects()</td>
<td>0.002</td>
<td>0.002</td>
<td>0.002</td>
<td>0.002</td>
<td>0.002</td>
<td>0.002</td>
</tr>
</tbody>
</table>

Figure 5.7: JHotDraw results
6 Conclusion

The detection of design patterns during the process of software reverse engineering is difficult. Many approaches have been proposed and implemented but still lack flexibility and effectiveness. Our major contribution to the research area includes a new approach that uses static and dynamic code analysis to detect design patterns, and a reverse engineering tool: PDE.

6.1 Research contribution

In this thesis we present a reverse engineering approach that detects software design patterns in Java applications. Our approach uses static and dynamic program analysis. We introduce several models for the dynamic analysis that we use to remove false positives from our static results. The models include the order in which methods have to be called and the objects that are involved at run-time. We provide a catalogue of static and dynamic definitions for the GoF design patterns. We also propose a reverse engineering tool called PDE, which is used to detect/verify
design patterns in Java applications. PDE uses the results from the static analysis, together with dynamic facts from the Java application and dynamic definitions of the pattern that we want to detect. PDE is flexible since it allows the user to define own pattern definitions. PDE is written in Java using the Eclipse IDE. We tested PDE with design pattern implementations from the *Applied Java Patterns* book (SL01). We are able to detect 22 out of 23 design patterns with PDE. The results prove that our approach is effective and that the dynamic analysis removes many false positives from the static results. We also tested the robustness of PDE with JHotDraw and ran the full detection process for three design patterns. The following section offers suggestions on how to extend the work presented in this thesis.

### 6.2 Future work

For future work we definitely see potential to improve the dynamic definitions. More detailed analysis of the GoF design patterns should allow better static and dynamic definitions. Right now, we do not exploit all possibilities for the dynamic definitions of the GoF design patterns. We also believe that it will be useful to have multiple definitions of design patterns that do not have one common implementation.
The static analysis also leaves space for improvement. Especially the high number of possible candidate instances could be reduced if more information about design patterns are considered. The Singleton pattern is an example on how our approach can be extended to use additional information from the static facts.

This thesis covers a new approach in detecting design patterns and presents a tool that implements this approach. In order to increase the usability of this approach it is important to integrate the static and dynamic analysis so that PDE becomes easier to use. We also think it is important to provide an enhanced graphical user interface (GUI) for PDE to improve usability. For this step it is important to define the user interface in a way that it supports the user during the process of reverse engineering. This includes presenting the results graphically, i.e. as UML diagrams.

We tested PDE with the implementations of the Applied Java Patterns examples and with a large Java application, JHotDraw. However, there is certainly room for more tests. Currently we only tested JHotDraw with three design patterns. This can be extended to all GoF design patterns. Other tests can cover more applications and should cover at least all GoF design patterns. These tests can help to refine the design and increase the effectiveness and efficiency of PDE.
During the experiments and robustness tests we found some parts that can be optimized easily. The current implementation is stable and works fine for large data sets. There is still space for improvement but since the performance was not an important issue in this thesis we did not work on improving the running time of PDE.

Future work might also want to consider to standardize the input and output files that are created to allow integration of other tools and applications. For example: if the results are defined in XML format, this could be interesting for the graphical output.
A  PDE - User Manual

In this Appendix we explain how to run PDE to detect design patterns. In order to run PDE the user needs static facts and dynamic facts from the software that is analyzed as well as the dynamic definition of the design pattern that he wants detect.
PDE usage

Usage: java PatternDetectionEngine -input <FactsFile>
-ci <CandidateInstancesFileName> -df <DynamicFactsFileName>
-dd <DynamicDefinitionFileName> -redirect <output_filename>
[-threshold] [-create_report] [-example] [-testsuite]
[[-print_statistics]/[-ps] [-print_datastructure]/[-pd]
[-print_time] -debug [-h] [-help]

<FactsFile> : facts file with several facts
<CandidateInstancesFileName> : extracted facts with javex, grok and ql
<DynamicFactsFileName> : dynamic facts generated using probekit
<DynamicDefinitionFileName> : dynamic design pattern definition in XML format

This program can be used in two ways:
First: -ci <CandidateInstancesFileName>
     -df <DynamicFactsFileName>
     -dd <DynamicDefinitionFileName>
The user can pass three fact files that are needed for the
design pattern detection/verification process.
Second: -input <FactsFile>
This parameter allows the user to pass a file that contains
a list of files that need to be processed. Each line contains
three file names. The first file name is for the candidate instances
the second for the dynamic facts file and the third for the dynamic
definition.

There are a couple of optional parameters that allow to specify different
output from the program:
-example: If this argument is passed to the program
then example files are loaded and processed.
This can be used to see how PDE works.
-redcet <output_filename> Redirect command line output to text file.
-threshold Value between 0.0 and 1.0 that is used in
ranking the found candidate instances.
-create_report Creates report for test of GoF Patterns.
-print_statistics or -ps Prints statistics with the results of the
candidate instances.
-print_datastructure or -pd Prints datastructure that contains all matching
node that match the dynamic definition.
-print_time Print time in millisecond for each method call.
This can be used for benchmarking PDE.
We created a list of examples and provide all necessary files to run these examples in PDE’s distribution package. The first step to verify that everything is running properly is to pass only one argument to the program.

**Shell command**

```
prompt> java PatternDetectionEngine -example
```

**Figure A.1: Simple PDE example**

This will run PDE with static and dynamic facts that we provide in the examples folder. This simple example is taken from the Abstract Factory implementation of the *Applied Java Patterns* book. The dynamic definitions define the Abstract Factory design pattern (SL01). The results after running PDE will show the output shown in Figure A.2. The output shows which pattern we were looking for as well as all candidate instances from the static facts that match the dynamic definitions with more than 80%. At the end of the output we rank all candidate instances according on how well they match the dynamic definitions.

One way to run PDE is to provide the static facts, dynamic facts and the dynamic definitions separately. This can be done by passing them to PDE as shown in Figure A.3:
# Code example from: example
# Pattern we want to detect: AbstractFactory

Candidate Instance is pattern? true 100% threshold=80%
Candidate Instance is pattern? false 0% threshold=80%
Candidate Instance is pattern? true 100% threshold=80%
Candidate Instance is pattern? false 0% threshold=80%
Number of positive candidate instances after the dynamic analysis:
2 out of 4 ( threshold = 80% )

Here is a ranked list of all candidate instances with the corresponding class names (and pattern roles):
{0=abstractFactory, 1=concreteFactory, 2=product, 3=abstractProduct}

0 100% {0=ajp_code.AbstractFactory.AddressFactory,
1=ajp_code.AbstractFactory.FrenchAddressFactory,
2=ajp_code.AbstractFactory.FrenchPhoneNumber,
3=ajp_code.AbstractFactory.PhoneNumber}

1 100% {0=ajp_code.AbstractFactory.AddressFactory,
1=ajp_code.AbstractFactory.USAddressFactory,
2=ajp_code.AbstractFactory.USPhoneNumber,
3=ajp_code.AbstractFactory.PhoneNumber}

Figure A.2: Simple PDE example output

Shell command

```
prompt> java PatternDetectionEngine -ci <CandidateInstancesFileName>
            -df <DynamicFactsFileName>
            -dd <DynamicDefinitionFileName>
```

Figure A.3: Another PDE example
Since we wanted to know how well PDE works compared to other available tools, we created static facts, static definitions, dynamic facts and dynamic definitions for twenty two GoF design patterns that we wanted to detect. We used code examples from the Applied Java Patterns book (SL01). In order to run all tests for Chapter 4 we created an input file that contains 22x22 combinations of all 22 design patterns that PDE can detect. We compare each design pattern implementation with all twenty two design patterns and record how many positive candidate instances are detected. All input files for PDE are provided with our software and can be run as follows:

```
Shell command

prompt> java PatternDetectionEngine -input input.file.eclipse.txt
-ps -print_results -create_report
```

Figure A.4: Running PDE with the configuration for our experiments

By using the parameters in Figure A.4, all our test results can be verified. The output from the command line can be redirected to a text file called report.txt. A list of all parameters that can be passed to PDE can be printed with -help or -usage.
In this section we present the design of our tool PDE. Figure B.2 shows the different classes of PDE as an UML class diagram. We explain each class and its dependencies. The main class of our tool is the PatternDetectionEngine. This class takes different parameters as input, parses them and calls all other classes. There are two ways to start PDE. For the first one the user provides the static facts (candidate instances), dynamic facts and dynamic definitions in text files for PDE, see Figure A.3. For the second step, the user can provide a FactFile. This file can contain a list of static fact files, dynamic fact files and dynamic definitions, see Figure B.1 shows an example of such a FactFile. The FactFiles class processes the input file, verifies that the files exist and stores each line of the file in an object. Later on PDE iterates through this list for the detection process. In each line of Figure B.1 we detect a different design pattern. In the first line we have the static facts from the Chain of Responsibility design pattern, the dynamic facts from the AJP pattern implementation and the dynamic definitions of the pattern we want
% This fact file contains a list of static fact files, dynamic fact files
% and the dynamic definitions of the design patterns that we want to detect
% in the application.
%
% Static fact file     dynamic fact file     dynamic definition
chain.chain.ql.out.instances chain.RunPattern.txt chain.xml
command.command.ql.out.instances command.RunPattern.txt command.xml
composite.composite.ql.out.instances composite.RunPattern.txt composite.xml
decorator.decorator.ql.out.instances decorator.RunPattern.txt decorator.xml
flyweight.flyweight.ql.out.instances flyweight.RunPattern.txt flyweight.xml

Figure B.1: PDE input FactFile

After reading the parameters, the PatternDetectionEngine processes the static
and dynamic facts. For the static facts we use the CandidateInstance class. This
class represents one candidate instance as an object. All candidate instances are
stored in a linked list in the class CandidateInstancesList and can be retrieved
through a getter method. The dynamic facts are processed in the DynamicFactsProcessor
class. It takes the file name of the dynamic facts file, parses it and stores all Nodes
in an XML Document. In the next step the tool parses the dynamic definition of
the design pattern, and transforms it using the role names from the CandidateIn-
stance class. In this step the role are replaced by the real class names from the
candidate instances. This is done in the DynamicDefinitionConverter. The final
processing and verification is done using the Validator class. This class uses the
transformed dynamic definition and the dynamic facts to detect and verify the de-
sign pattern candidate instances. At the end of the processing we can retrieve a
datastructure that is attached to each `CandidateInstance` object. This datastructure contains all nodes from the dynamic facts that match the dynamic definition. `PatternDetectionEngine` takes this datastructure for each `CandidateInstances` object and prints if the candidate instance is a design pattern or not. All information from the dynamic facts that matched the definition is stored in this datastructure, therefore we can have very detailed results what the method names and class names as well as arguments, object ids and other things.

![Static UML diagram of PDE](image)

Figure B.2: Static UML diagram of PDE


C Static Analysis

Appendix C introduces the tools that were developed by the Software Architecture Group, SWAG, at the University of Waterloo that we used in this thesis. We explain how we use Javex to extract static facts from Java class files. After that we show how we use Grok and QL to detect candidate instances for design patterns in these static facts.

C.1 Javex

Javex is a fact extractor for Java, that extracts facts from the Java class files. It is actively developed by Ian Davis at the University of Waterloo, see http://www.swag.uwaterloo.ca/javex/index.html. We use Javex to extract information about classes, interfaces, methods and attributes. Figure C.1 shows an example shell script how to run Javex to extract static facts from a software. We use JHotDraw to demonstrate this example. If we have the source code of the
software then we need to compile the source first with `javac`. After that we find all class files and pass them as parameters to `Javex`. We use the option `-f` and `-l` to specify that we want to collect information about fields in classes as well as local variables with methods and arguments passed to these methods. Figure C.2 shows all parameters that can be passed to Javex. Please refer to the documentation of Javex for more details.

```bash
#!/bin/csh

# Compile all source files
set javafiles=`find jhotdraw/JHotDraw_7.0.8 -name '*.java' | tr '
' ' '`
javac $javafiles

# Run Javex with the
set classfiles=`find jhotdraw/JHotDraw_7.0.8 -name '*.class' | tr '
' ' '`
set javex_out=jHotDraw.javex.out
javex -f -l $classfiles > $javex_out
```

Figure C.1: Extract static facts from Javex

### Synopsis

```
javex [-j/javac <program>] [-c/classpath <path>] [-i/input <file>] [-d/dynamic]
[-e/edges <number>]* [-f/fields] [-l/locals] [-m/mutators]
[-p/permissions [-p/permissions]] [-z/zero] [-v/verbose] *.class|java
```

Figure C.2: Javex synopsis
C.2 Grok

Grok is a programming language designed for manipulating collections of binary relations. It was developed by Ric Holt at the University of Waterloo and is part of the SWAG toolKit http://swag.uwaterloo.ca/~nsynytskyy/grokdoc/index.html.

We use Grok to reduce the static facts that we get from Javex to get only those facts that represent the uses and inherits relations between classes. We call this step lifting facts to class level. Figure C.3 shows the shell command to run Grok. We use the output from Javex as input and store the results in another file. The script lift_to_classlevel.grok, see Figure C.5, lifts all relations from the Javex output to class level. In this script we first lift all facts to class level by using a Grok procedure written by Ric Holt, see Figure C.4. Then we extract only the uses and inherits relations that we are interested in, see Figure C.5.

```
# Run Grok
grok lift_to_classlevel.grok jHotDraw.javex.out jHotDraw.grok.out
```

Figure C.3: Run Grok to extract uses and inheritance relations.
% This is a Grok procedure (grp).
% Given an in-core data base and set n, this procedure
% lifts all the contents of nodes in n.
% It is assumed that the 'contain' relation defines a tree on all nodes.
% This procedure is executed from a Grok script via:
% exec somepath/link.grk
% where 'somepath' must locate the directory containing this script.
% Author: Ric Holt, Date: 13 Jan 2002

% Rest of script will lift the set n of nodes, ie, the rest of the script
% is schema independent lifter.

reln := relnames % Find these before any temp rels are introduced.
attrs := prefix "@_" reln % Attributes start with "@_"
rels := reln - attrs - {"contain"} % Rel's that aren't attrs or contain

Do := contain* % Inclusive descendants
Ao := inv Do % Inclusive ancestors
D := contain+ % Descendants
A := inv D % Ancestors
root := dom(contain) - rng(contain)

N := id(n)
notInN := id (root . Do - n . Do)

% Delete attributes of nodes that descend from n.
% This deletion is necessary before deleting contents of N (otherwise
% the attributes are moved up to be attached to members of N).
% Beware: fails to delete any attributes of edges.
% Bug: Leaves in @_order attributes on children edges.
% This is OK (I guess) in that @_order is deleted elsewhere

insideN := id(rng(N o D)) % Descendents of n ?? Better: id(N . D) ??
for a in attrs
    $ a := $ a - insideN o $ a % Delete attribute a of descendents of n
end for

% Elide nodes in n
% Raise edges entering or leaving nodes in n
for r in rels % For (name of) every relation
  R := $ r % Get value of relation named r
  Racross := N o Do o R o Ao o N - N % Edges: Both ends in A (part to part)
  Rout := notInN o R o A o N % Edges: Right end in A
  Rin := N o D o R o notInN % Edges: Left end in A
  $ r := R + Rout + Rin + Racross % Add induced edges to rel named r
  delrel Racross
  delrel Rout
  delrel Rin
end for

% Delete nodes in N
Internal := rng (N o D) % All descendent nodes of A ??Better: n . D ??
delset Internal % Delete all descendent nodes with their edges

delrel Do
delrel Ao
delrel D
delrel A
delrel N
delrel notInN
delrel R

Figure C.4: Grok procedure by Ric Holt
% Read TA file and lift all relations to class level
% Uses lift.grp procedure

if $n \neq 2$ then
    put "Usage: grok liftclasslevel.grok inputTa outputRsf"
    put "Author: Marcel Birkner, Date May 2007"
    quit
end if

inputTa := $1$
outputRsf := $2$

% Declare relations
inherits := EMPTYREL
uses := EMPTYREL
relToFile inherits outputRsf
relToFile uses outputRsf

E258 := EMPTYREL
E259 := EMPTYREL
E256 := EMPTYREL
E183 := EMPTYREL
E193 := EMPTYREL
E192 := EMPTYREL
E189 := EMPTYREL
E187 := EMPTYREL
E197 := EMPTYREL
E261 := EMPTYREL
E180 := EMPTYREL
E179 := EMPTYREL
E179 := EMPTYREL
E178 := EMPTYREL
E185 := EMPTYREL
E184 := EMPTYREL
E182 := EMPTYREL
E257 := EMPTYREL
E262 := EMPTYREL
E261 := EMPTYREL
E260 := EMPTYREL
E263 := EMPTYREL

catta inputTa
% collect private methods/classes with private constructors
private_methods := @_access . "private"
constructors := @_label . "<init>"
private_constructors := private_methods ^ constructors
classes_with_private_constructors := contain . private_constructors
private := classes_with_private_constructors . @_label

% Bil
classes := $INSTANCE . $_C
id_classes := id(classes)
inner_class_contain := id_classes o contain o id_classes
contain := contain - inner_class_contain

n := $INSTANCE . {"$_C","$_I"}
exec lift.grp

% Bil
contain := contain + inner_class_contain

% Extends class
if # E258 = 0 then
    inherits := inv @_label o E258 o @_label
    appendRelToFile inherits outputRsf
end if

% Implements interface
if # E259 = 0 then
    inherits := inv @_label o E259 o @_label
    appendRelToFile inherits outputRsf
end if

% Overrides Method
if # E256 = 0 then
    inherits := inv @_label o E256 o @_label
    appendRelToFile inherits outputRsf
end if
%Invoke special
if # E183 ^= 0 then
  uses := inv @_label o E183 o @_label
  appendRelToFile uses outputRsf
end if

% Instantiates
if # E263 ^= 0 then
  uses := inv @_label o E263 o @_label
  appendRelToFile uses outputRsf
end if

% Inner class
if # E260 ^= 0 then
  E260 := inv @_label o E260 o @_label
  appendRelToFile E260 outputRsf
end if

% Throws class
if # E261 ^= 0 then
  E261 := inv @_label o E261 o @_label
  appendRelToFile E261 outputRsf
end if

% Array of
if # E262 ^= 0 then
  E262 := inv @_label o E262 o @_label
  appendRelToFile E262 outputRsf
end if

% Interface method
if # E257 ^= 0 then
  E257 := inv @_label o E257 o @_label
  appendRelToFile E257 outputRsf
end if
% Invoke virtual
if # E182 ~= 0 then
    uses := inv @_label o E182 o @_label
    appendRelToFile uses outputRsf
end if

% Invoke static
if # E184 ~= 0 then
    uses := inv @_label o E184 o @_label
    appendRelToFile uses outputRsf
end if

% Invoke interface
if # E185 ~= 0 then
    uses := inv @_label o E185 o @_label
    appendRelToFile uses outputRsf
end if

% get static
if # E178 ~= 0 then
    uses := inv @_label o E178 o @_label
    appendRelToFile uses outputRsf
end if

% put static
if # E179 ~= 0 then
    E179 := inv @_label o E179 o @_label
    appendRelToFile E179 outputRsf
end if

% get field
if # E180 ~= 0 then
    E180 := inv @_label o E180 o @_label
    appendRelToFile E180 outputRsf
end if
Figure C.5: Lift static facts to class level and extract uses and inherits relations
Figure C.6 shows additional information that can be extracted from the Javex facts. In this script we extract all classes that have a private constructor. We use this information for the static definition of the Singleton design pattern, see Figure E.24. We believe that there are more information in the static facts that can help to reduce the number of false positives in the static facts. If we define design patterns in more detail statically then we can achieve better results at this stage. Improving the static analysis will be part of future work and will not be covered in this thesis.

--- Grok script ---

% Read TA file and find all private constructors
if $n \neq 2$ then
  put "Usage: grok private_constructor.grok inputFile outputFile"
  put "Author: Marcel Birkner, Date June 2007"
  quit
end if

inputTa := $1
outputTa := $2

getta inputTa
private_methods := @_access . "private"
constructors := @_label . "<init>"
private_constructors := private_methods ^ constructors
classes_with_private_constructors := contain . private_constructors
private_classes := classes_with_private_constructors . @_label

putset private_classes outputTa

--- Figure C.6: Find classes with private constructor ---
C.3 QL

QL is a Java re-implementation of Grok, written by Jingwei Wu at the University of Waterloo. While being slower than Grok, QL makes up for it with new operators and built-in commands. We use QL to filter qualified sets of entities that match the static definition of the design pattern. Compared to Grok, QL allows us to define multiple relations. After analyzing each design pattern together with their UML sequence diagrams, we create dynamic definitions for each design pattern. Thesis dynamic definitions are stored in QL format. Figure E.1 shows the QL script for the Abstract Factory design pattern. In Appendix E we provide the dynamic definition for all 22 GoF design pattern that we can detect with PDE. See http://swag.uwaterloo.ca/~nsynytskyy/grokdoc/operators.html for more information about the operators in QL.
D Dynamic Analysis

In this section we introduce Probekit and show how we use this tool to extract dynamic facts from the Java application that we are analyzing.

D.1 Test and Performance Tool Platform, Probekit

Probekit is part of the Eclipse Test and Performance Tools Platform Project. It allows us to write fragments of Java code that can be placed at specified points in the Java class. The injection points can be at method entry, method exit, catch/finally blocks, class loading, etc. This allows us to collect runtime data about the application. We use Probekit to trace the method invocation during runtime. These facts are later used by PDE to verify the candidate instances that were detected during the static analysis. Probekit comes as a plug-in for Eclipse and is easy to use. For our approach we inject additional Java code at method entry and method exit.

- method entry fragments run upon method entry.
- exit fragments run upon method exit: a normal exit, when the method throws an exception, or when a thrown exception propagates out of the method.

There is a list of data types available that we can access from the Java code that we inject in the application. We are only interested in the `className`, `methodName`, `thisObject` and `args` data types. Every time a method is called or exited we write the following list of information to a text file. Later on PDE will pass these runtime facts and create a valid XML document. This XML will then be parsed and handled by PDE for the pattern verification process. Here is a list of the data types that we used from Probekit:

- `className` (String): For method probes, the class name of the probed method, including the package name, in internal format; for callsite probes, the class name of the called method.

- `methodName` (String): For method probes, the method name of the probed method, in internal format; for callsite probes, the method name of the called method. Constructors have the method name `\texttt{\_init\_\_}`, and static class initializers have the method name `\texttt{clinit\_\_}`.

- `thisObject` (object): The this object (for instance methods) that was passed to the probed method. Not valid for staticInitializer fragments. `thisObject` is 124
null for static methods, for entry fragments that are applied to constructors, and for exit fragments applied to constructors when the constructor throws an exception.

- **args (object[]):** An array of Object references representing the arguments to the probed method. There is one element in this array for each argument to the method (not counting the this argument). Arguments that are primitive types are boxed into temporary objects of the appropriate reference type, for example: Integer for int. If the method takes no arguments, the size of the Object[] array is zero. Note that constructors for non-static inner classes have one hidden argument per *inner* level, so the argument array will contain more elements than appear in the source code. Not valid for staticInitializer fragments.

For more detailed information about Probekit please refer to the official website, see [http://www.eclipse.org/tptp/platform/documents/probekit/probekit.html](http://www.eclipse.org/tptp/platform/documents/probekit/probekit.html).

Figure D.1 and Figure D.2 show our code that is injected in every method. The first Figure shows the Java code that is added to the top of each method call. And the second Figure shows the code that is injected at the bottom of each method call. After instrumenting the Java classes with these code fragments we have to run
the application. In order to achieve good code coverage it is beneficial to have a
test suite that covers most of the source code. Therefore most methods are invoked
at some point in time and we can use the method trace to detect patterns in all
parts of the application.
if( methodName.contains("<init>") || methodName.contains("<clinit>"))
    methodName = "Constructor";

callDepth = callDepth + 1;
String arguments = "";

for (int i=0; i<args.length;i++){
    // only take those arguments that are Objects;
    // throw out Strings that might mess up the XML file
    if(args[i] != null && args[i].toString().contains("@") { 
        arguments = arguments + "|" + args[i];
    }
}

String output = "<entry args="" + arguments
    + "\" className="" + className
    + "\" methodName="" + methodName
    + "\" thisObject="" "
    + "\" calledByClass="" "
    + "\" calledByMethod="" "
    + "\" calledByObject="" "
    + "\" callDepth="" + callDepth + "\" >";

String outputFilename = className.replaceAll("/","");
outputFilename = outputFilename + ".txt";

try {
    BufferedWriter out = new BufferedWriter(new FileWriter(outputFilename, true));
    out.write( output );
    out.newLine();
    out.close();
} catch (IOException e) {
    System.out.println("PROBEKIT: Exception in BufferedWriter, entry");
}

Figure D.1: Probekit code: method entry
callDepth = callDepth - 1;

if( methodName.contains("<init>") || methodName.contains("<clinit>") )
    methodName = "Constructor";

String output = "<exit" + " className="" + className
+ " methodName="" + methodName
+ " thisObject="" + className + "@"
+ System.identityHashCode(thisObject)
+ " calledByClass="" calledByMethod=""
+ calledByObject="""
+ " callDepth="" + callDepth + ""></exit></entry>";

try {
    BufferedWriter out =
        new BufferedWriter(new FileWriter(outputFilename,true));
    out.write( output );
    out.newLine();
    out.close();
} catch (IOException e) {
    System.out.println("PROBEKIT: Exception in BufferedWriter, exit");
}

Figure D.2: Probekit code: method exit
E  Design Pattern Definitions

In this section we present the static and dynamic definitions for all 22 GoF design patterns that we are detecting with our tool. This section contains the static and dynamic definitions for all GoF design patterns as well as the UML class diagrams that we used to extract these definitions. We used the UML class diagrams and if available the UML sequence diagrams from the GoF book (GHJV95), as well as the intent to describe each design pattern.

For the Abstract Factory we show a detailed example in Figure E.1. The first line of the static definition shows to which category of design patterns this pattern belongs. The second line shows the pattern name. Line 4-7 explain the abbreviations that are used for the roles of the design pattern. Line 9 reads the factbase that is passed to this QL script. Line 11-13 describes the inherits and uses relations that define the design pattern. Line 15-22 removes the Java API Libraries from the static results. Line 25 at the end of the static definitions writes the re-
sults to file. The static definitions for all design patterns are identical except line 11-13. These lines show the relations that define a design pattern. For all following design patterns we will only show the pattern name, roles and the relations. All dynamic definitions that are listed in this section can be found in XML format on the attached CD.
E.0.1 Abstract Factory

**Intent**: Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

**Static definition:**

```plaintext
// Creational patterns
// Abstract Factory
// absFact -> abstractFactory
// conFact -> concreteFactory
// p       -> product
// absP    -> abstractProduct

def getdb($1)

DP[absFact,conFact,p,absP] = {inherits[conFact,absFact];
  uses[conFact,p];
  inherits[p,absP]}

// Remove Java API Libraries
DP_java = DP [&0 =~ "java.*"]
DP = DP - DP_java
DP_java = DP [&1 =~ "java.*"]
DP = DP - DP_java
DP_java = DP [&2 =~ "java.*"]
DP = DP - DP_java
DP_java = DP [&3 =~ "java.*"]
DP = DP - DP_java
putdb($2,\"DP\")
```

Figure E.1: Abstract Factory static definition
Figure E.2: UML class diagram - Abstract Factory
Dynamic definition:

• First method call
  – className="concreteFactory"
  – methodName="Constructor"
  – nextCallInOrder="yes"
  – quantifier="1"

• Second method call
  – className="product"
  – calledByClass="concreteFactory"
  – quantifier="4"
E.0.2 Adapter

**Intent:** Convert the interface of a class into another interface clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces.

**Static definition:**

```
1 // Structural patterns
2 // Adapter
3 //
4 // c -> client
5 // t -> target
6 // ad -> adapter
7 // ae -> adaptee
8 //
9 DP[c,t,ad,ae] = {uses[c,ad];
10    inherits[ad,t];
11    uses[ad,ae]}
```

Figure E.3: UML class diagram - Adapter
Dynamic definition:

• First method call
  – className=”adapter”
  – calledByClass=”client”
  – thisObject=”object1”
  – nextCallInSubtree=”yes”
  – quantifier=”2”

• Second method call
  – className=”adaptee”
  – calledByClass=”adapter”
  – calledByObject=”object1”
  – thisObject=”object2”
  – quantifier=”3”
Figure E.4: Adapter design pattern UML sequence diagram
E.0.3 Bridge

Intent: Decouple an abstraction from its implementation so that the two can vary independently.

Static definition:

```plaintext
// Structural patterns
// Bridge

// rfAbs -> refinedAbstraction
// abs -> abstraction
// imp -> implementer
// conImp -> concreteImplementer

DP[rfAbs,abs,imp,conImp] = {inherits[rfAbs,abs];
                        uses[abs,imp];
                        inherits[conImp,imp]}
```

Figure E.5: UML class diagram - Bridge
Dynamic definition:

• First method call
  – className="concreteImplementer"
  – methodName="Constructor"
  – quantifier="1"

• Second method call
  – className="abstraction"
  – methodName="Constructor"
  – quantifier="1"

• Third method call
  – className="abstraction"
  – args="concreteImplementer"
  – quantifier="2"

• Fourth method call
  – className="concreteImplementer"
  – calledByClass="abstraction"
  – quantifier="2"
E.0.4 Builder

**Intent:** Separate the construction of a complex object from its representation so that the same construction process can create different representations.

**Static definition:**

```plaintext
// Creational patterns
// Builder
//
// dir  -> director
// bld  -> builder
// conBld -> concreteBuilder
// prod -> product

DP[dir, bld, conBld, prod] = {uses[dir, bld];
    inherits[conBld, bld];
    uses[conBld, prod]}
```

Figure E.6: UML class diagram - Builder
Dynamic definition:

• First method call
  – calledByObject="concreteBuilder"
  – nextCallInOrder="yes"
  – methodName="Constructor"
  – quantifier="1"

• Second method call
  – args="concreteBuilder"
  – className="director"
  – nextCallInSubtree="yes"
  – quantifier="1"

• Third method call
  – className="concreteBuilder"
  – calledByClass="director"
  – nextCallInOrder="yes"
  – quantifier="1"

• Fourth method call
  – className="product"
  – calledByClass="concreteBuilder"
  – quantifier="2"
Figure E.7: Builder design pattern UML sequence diagram
E.0.5 Chain of Responsibility

**Intent:** Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

**Static definition:**

```plaintext
// Behavioral patterns
// Chain of Responsibility
//
// c -> client
// h -> handler
// ch -> concreteHandler

DP[c,h,ch] = {uses[c,h];
              inherits[ch,h];
              uses[ch,h]}
```

![UML class diagram - Chain of Responsibility](image)

Figure E.8: UML class diagram - Chain of Responsibility
Dynamic definition:

• First method call
  – className="concreteHandler"
  – calledByClass="client"
  – nextCallInOrder="yes"
  – quantifier="2"

• Second method call
  – className="concreteHandler"
  – calledByClass="client"
  – nextCallInOrder="yes"
  – quantifier="2"

• Third method call
  – className="concreteHandler"
  – calledByClass="client"
  – nextCallInOrder="yes"
  – quantifier="1"

• Fourth method call
  – className="concreteHandler"
  – calledByClass="concreteHandler"
  – quantifier="1"
E.0.6 Command

**Intent:** Encapsulate a request as an object, thereby letting you parametrize clients with different requests, queue or log requests, and support undoable operations.

**Static definition:**

```
// Behavioral patterns
// Command
//
// ivk  -> invoker
// cmd  -> command
// conCmd -> concreteCommand
// re   -> receiver

DP[ivk, cmd, conCmd, re] = {uses[ivk, cmd];
                           inherits[conCmd, cmd];
                           uses[conCmd, re]}
```

Figure E.9: UML class diagram - Command
Dynamic definition:

• First method call
  – className="concreteCommand"
  – methodName="Constructor"
  – thisObject="object1"
  – nextCallInOrder="yes"
  – quantifier="2"

• Second method call
  – args="concreteCommand"
  – className="invoker"
  – nextCallInOrder="yes"
  – quantifier="2"

• Third method call
  – calledByClass="invoker"
  – className="concreteCommand"
  – thisObject="object1"
  – nextCallInOrder="yes"
  – quantifier="4"

• Fourth method call
  – calledByClass="concreteCommand"
  – className="receiver"
  – calledByObject="object1"
  – quantifier="1"
E.0.7 Composite

**Intent:** Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

**Static definition:**

```plaintext
// Structural patterns
// Composite
//
// l -> leaf
// c -> component
// cp -> composite

DP[l,c,cp] = {inherits[l,c];
              inherits[cp,c];
              uses[cp,c]}
```
Figure E.10: UML class diagram - Composite
Dynamic definition:

- First method call
  - className="leaf"
  - methodName="Constructor"
  - quantifier="1"

- Second method call
  - className="composite"
  - methodName="Constructor"
  - nextCallInOrder="yes"
  - quantifier="1"

- Third method call
  - args="leaf"
  - className="composite"
  - nextCallInOrder="yes"
  - quantifier="4"

- Fourth method call
  - className="leaf"
  - calledByClass="composite"
  - quantifier="4"
E.0.8 Decorator

**Intent:** Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub-classing for extending functionality.

**Static definition:**

```plaintext
Decorator

1 // Structural patterns
2 // Decorator
3 //
4 // cc -> concreteComponent
5 // c  -> component
6 // d  -> decorator
7 // cd -> concreteDecorator
8
9 DP[cc,c,d,cd] = {inherits[cc,c];
10    inherits[d,c];
11    uses[d,c];
12    inherits[cd,d])
```
Figure E.11: UML class diagram - Decorator
Dynamic definition:

• First method call
  – className="concreteComponent"
  – methodName="Constructor"
  – thisObject="object1"
  – quantifier="2"

• Second method call
  – className="concreteComponent"
  – methodName="Constructor"
  – thisObject="object2"
  – quantifier="2"

• Third method call
  – className="concreteDecorator"
  – methodName="Constructor"
  – quantifier="1"

• Fourth method call
  – args="concreteComponent"
  – className="concreteDecorator"
  – quantifier="2"

• Fifth method call
  – calledByClass="concreteDecorator"
  – className="concreteComponent"
  – quantifier="2"
E.0.9 Factory Method

**Intent:** Define an interface for creating an object, but let subclass’s decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

**Static definition:**

```
  // Creational patterns
  // Factory Method
  // c -> creator
  // conC -> concreteCreator
  // conP -> concreteProduct
  // p -> product

  DP[c,conC,conP,p] = {inherits[conC,c];
                      uses[conC,conP];
                      inherits[conP,p]}
```

![UML class diagram - Factory Method](image)

Figure E.12: UML class diagram - Factory Method
Dynamic definition:

- First method call
  - className="concreteCreator"
  - methodName="Constructor"
  - nextCallInOrder="yes"
  - quantifier="1"

- Second method call
  - args="concreteCreator"
  - className="concreteCreator"
  - quantifier="2"

- Third method call
  - className="concreteProduct"
  - calledByClass="concreteCreator"
  - quantifier="2"

- Fourth method call
  - className="concreteProduct"
  - calledByClass="concreteCreator"
  - quantifier="2"

- Fifth method call
  - className="concreteProduct"
  - calledByClass="concreteProduct"
  - quantifier="1"
E.0.10  Flyweight

**Intent:** Use sharing to support large numbers of fine-grained objects efficiently.

**Static definition:**

```plaintext
// Structural patterns
// Flyweight
//
// ff -> flyweightFactory
// f  -> flyweight
// cf -> concreteFlyweight

DP[ff,f,cf] = {inherits[cf,f];
               uses[ff,cf];
               uses[ff,f]}
```

![UML Class Diagram - Flyweight](image-url)

Figure E.13: UML class diagram - Flyweight
Dynamic definition:

• First method call
  - className="caretaker"
  - methodName="Constructor"
  - nextCallInOrder="yes"
  - quantifier="1"

• Second method call
  - className="originator"
  - calledByClass="caretaker"
  - methodName="Constructor"
  - quantifier="2"

• Third method call
  - className="memento"
  - calledByClass="originator"
  - calledByMethod="Constructor"
  - quantifier="2"

• Fourth method call
  - className="memento"
  - calledByClass="originator"
  - quantifier="2"

• Fifth method call
  - className="memento"
  - calledByClass="caretaker"
  - quantifier="1"
E.0.11 Interpreter

**Intent:** Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

**Static definition:**

```plaintext
// Behavioral patterns
// Interpreter
// ee -> expression
// ae -> abstractExpression
// c -> context

DP[ee, ae, c] = {inherits[ae, ee];
    uses[ee, c]}
```

![UML class diagram - Interpreter](image)

Figure E.14: UML class diagram - Interpreter
Dynamic definition:

- First method call
  - className="expression"
  - methodName="Constructor"
  - thisObject="object1"
  - nextCallInOrder="yes"
  - quantifier="1"

- Second method call
  - className="expression"
  - methodName="Constructor"
  - thisObject="object2"
  - quantifier="1"

- Third method call
  - className="context"
  - nextCallInOrder="yes"
  - quantifier="1"

- Fourth method call
  - args="expression"
  - className="context"
  - nextCallInOrder="yes"
  - quantifier="2"

- Fifth method call
  - className="context"
  - calledByClass="expression"
  - quantifier="2"
E.0.12 Iterator

**Intent:** Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

**Static definition:**

```plaintext
// Behavioral patterns
// Iterator
//
// it -> Iterator
// ca -> concreteAggregate
// cit -> concretelIterator

DP[ca,it,cit] = {uses[ca,it];
                inherits[cit,it]}
```

Figure E.15: UML class diagram - Iterator
Dynamic definition:

• First method call
  – className="concreteIterator"
  – methodName="getIterator"
  – calledByClass="concreteAggregate"
  – nextCallInOrder="yes"
  – quantifier="2"

• Second method call
  – className="concreteIterator"
  – calledByClass="concreteAggregate"
  – quantifier="1"
E.0.13 Mediator

**Intent:** Define an object that encapsulate how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

**Static definition:**

```plaintext
// Behavioral patterns
// Mediator
//
// cm -> concreteMediator
// m -> mediator
// c -> colleague
// cc -> concreteColleague

DP[cm,m,cc] = {inherits[cm,m];
               uses[cc,m];
               uses[cm,cc]}
```

Figure E.16: UML class diagram - Mediator
Dynamic definition:

• First method call
  – className="concreteMediator"
  – methodName="Constructor"
  – quantifier="1"

• Second method call
  – className="concreteColleague"
  – methodName="Constructor"
  – quantifier="2"

• Third method call
  – className="concreteMediator"
  – calledByClass="concreteColleague"
  – quantifier="2"

• Fourth method call
  – className="concreteColleague"
  – args="concreteMediator"
  – quantifier="2"

• Fifth method call
  – className="concreteColleague"
  – calledByClass="concreteMediator"
  – quantifier="1"
Figure E.17: Mediator design pattern UML sequence diagram
### E.0.14 Memento

**Intent**: Without violating encapsulation, capture and externalize an object’s internal state so that the object can be restored to its state later.

**Static definition:**

```plaintext
// Behavioral patterns
// Memento
//
// ori -> originator
// m  -> memento
// ct  -> careTaker

DP[ori, m, ct] = {uses[ct, ori];
    uses[ct, m]}
```

![UML class diagram - Memento](image)

Figure E.18: UML class diagram - Memento
Dynamic definition:

- First method call
  - className="caretaker"
  - methodName="Constructor"
  - nextCallInOrder="yes"
  - quantifier="1"

- Second method call
  - className="originator"
  - calledByClass="caretaker"
  - methodName="Constructor"
  - quantifier="2"

- Third method call
  - className="memento"
  - calledByClass="originator"
  - calledByMethod="Constructor"
  - quantifier="2"

- Fourth method call
  - className="memento"
  - calledByClass="originator"
  - quantifier="2"

- Fifth method call
  - className="memento"
  - calledByClass="caretaker"
  - quantifier="1"
Figure E.19: Memento design pattern UML sequence diagram
E.0.15 Observer

**Intent:** Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

**Static definition:**

```plaintext
// Behavioral patterns
// Observer

// cs -> concreteSubject
// s  -> subject
// ob -> observer
// co -> concreteObserver

DP[s,cs,ob,co] = {inherits[co,ob];
                  uses[co,cs];
                  uses[s,ob])
```
Figure E.20: UML class diagram - Observer
Dynamic definition:

- First method call
  - className="concreteSubject"
  - methodName="Constructor"
  - quantifier="1"

- Second method call
  - className="concreteObserver"
  - methodName="Constructor"
  - thisObject="object1"
  - quantifier="3"

- Third method call
  - *this is another concrete observer*
  - className="concreteObserver"
  - methodName="Constructor"
  - thisObject="object2"
  - quantifier="1"

- Fourth method call
  - className="concreteObserver"
  - args="concreteSubject"
  - thisObject="object1"
  - quantifier="4"

- Fifth method call
  - className="concreteSubject"
  - calledByClass="concreteSubject"
  - nextCallInOrder="yes"
  - quantifier="2"

- Sixth method call
- className="concreteObserver"
- calledByClass="concreteSubject"
- thisObject="object1"
- nextCallInOrder="yes"
- quantifier="3"

- Seventh method call
  - className="concreteSubject"
  - calledByClass="concreteObserver"
  - calledByObject="object1"
  - nextCallInOrder="yes"
  - quantifier="3"

- Seventh method call
  - className="concreteObserver"
  - calledByClass="concreteSubject"
  - thisObject="object2"
  - quantifier="1"

- Seventh method call
  - className="concreteSubject"
  - calledByClass="concreteObserver"
  - calledByObject="object2"
  - quantifier="1"
Figure E.21: Observer design pattern UML sequence diagram
E.0.16 Prototype

**Intent:** Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

**Static definition:**

```plaintext
Prototype

// Creational patterns
// Prototype
// pi -> prototypeInterface
// pr -> prototype
// c  -> client

DP[pi,pr,c] = {uses[c,pr];
    inherits[pr,pi]}
```

![Figure E.22: UML class diagram - Prototype](image)

Figure E.22: UML class diagram - Prototype
Dynamic definition:

- First method call
  - className="prototype"
  - calledByClass="client"
  - methodName="Constructor"
  - nextCallInOrder="yes"
  - quantifier="1"

- Second method call
  - className="prototype"
  - calledByClass="client"
  - methodName="copy"
  - quantifier="2"
E.0.17 Proxy

**Intent:** Provide a surrogate or placeholder for another object to control access to it.

**Static definition:**

```plaintext
Proxy

// Structural patterns
// Proxy
// rs -> realSubject
// s  -> subject
// p  -> proxy

DP[rs,s,p] = {inherits[rs,s];
             inherits[p,s];
             uses[p,rs]}
```

![UML class diagram - Proxy](image-url)

Figure E.23: UML class diagram - Proxy
Dynamic definition:

- First method call
  - className="proxy"
  - methodName="Constructor"
  - nextCallInOrder="yes"
  - quantifier="1"

- Second method call
  - className="proxy"
  - nextCallInOrder="yes"
  - quantifier="1"

- Third method call
  - className="realSubject"
  - calledByClass="proxy"
  - methodName="Constructor"
  - quantifier="1"

- Fourth method call
  - className="realSubject"
  - calledByClass="proxy"
  - quantifier="1"
E.0.18 Singleton

**Intent:** Ensure a class only has one instance, and provide a global point of access to it.

**Static definition:**

```plaintext
// Creational patterns
// Singleton
//
// Note: For this pattern we check
// if the singleton class
// has a private constructor.
//
// -> c = client
// -> s = singleton

DP[c,s] = {uses[c,s];
            private[s]}
```

Figure E.24: UML class diagram - Singleton
Dynamic definition:

• First method call
  – className=”singleton”
  – methodName=”Constructor”
  – quantifier=”2”

• Second method call
  – className=”singleton”
  – calledByClass=”client”
  – quantifier=”1”
E.0.19 State

Intent: Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

Static definition:

```java
// Behavioral patterns
// State
//
// c -> context
// s -> state
// cs -> conreteState

DP[c,s,cs] = {uses[c,s];
              inherits[cs,s]}
```

Figure E.25: UML class diagram - State
Dynamic definition:

- First method call
  - className="context"
  - methodName="Constructor"
  - nextCallInOrder="yes"
  - quantifier="1"

- Second method call
  - className="concreteState"
  - calledByClass="context"
  - thisObject="object1"
  - nextCallInOrder="yes"
  - quantifier="2"

- Third method call
  - className="concreteState"
  - calledByClass="context"
  - nextCallInOrder="yes"
  - thisObject="object2"
  - quantifier="2"

- Fourth method call
  - className="context"
  - calledByClass="concreteState"
  - nextCallInOrder="yes"
  - quantifier="1"

- Fifth method call
  - className="context"
  - nextCallInOrder="yes"
  - quantifier="2"
• Sixth method call
  - className="concreteState"
  - calledByClass="context"
  - nextCallInSubtree="yes"
  - quantifier="2"

• Seventh method call
  - className="context"
  - calledByClass="concreteState"
  - quantifier="1"
E.0.20 Strategy

Intent: Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Static definition:

```java
// Behavioral patterns
// Strategy
// c -> context
// s -> strategy
// cs -> concreteStrategy

DP[c,s,cs] = {uses[c,s];
              inherits[cs,s]}
```

Figure E.26: UML class diagram - Strategy
Dynamic definition:

- First method call
  - className="concreteStrategy"
  - thisObject="object1"
  - methodName="Constructor"
  - quantifier="2"

- Second method call
  - className="concreteStrategy"
  - methodName="Constructor"
  - thisObject="object2"
  - quantifier="2"

- Third method call
  - className="context"
  - methodName="Constructor"
  - quantifier="1"

- Fourth method call
  - className="context"
  - nextCallInOrder="yes"
  - quantifier="2"

- Fifth method call
  - args="concreteStrategy"
  - className="context"
  - quantifier="1"

- Sixth method call
  - className="concreteStrategy"
  - calledByClass="context"
  - quantifier="1"
E.0.21 Template Method

**Intent:** Define the skeleton of an algorithm in an operation, deferring some steps to subclass.

**Static definition:**

```plaintext
// Behavioral patterns
// Template Method

// ac = abstractClass
// cc = concreteClass

DP[ac,cc] = {inherits[cc,ac];
    uses[cc,ac]}
```

![UML class diagram - Template Method](image)
Dynamic definition:

- First method call
  - className="concreteClass"
  - methodName="Constructor"
  - thisObject="object1"
  - quantifier="1"

- Second method call
  - className="concreteClass"
  - methodName="Constructor"
  - thisObject="object2"
  - quantifier="1"

- Third method call
  - className="concreteClass"
  - calledByClass="concreteClass"
  - quantifier="2"
E.0.22 Visitor

**Intent:** Represents an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

**Static definition:**

```plaintext
Visitor
data DP[v,cv,e,ce] = {inherits[cv,v];
                      inherits[ce,e];
                      uses[e,v]}
```
Figure E.28: UML class diagram - Visitor
Dynamic definition:

• First method call
  - className="concreteVisitor"
  - methodName="Constructor"
  - quantifier="1"

• Second method call
  - className="concreteVisitor"
  - args="concreteElement"
  - quantifier="2"

• Third method call
  - className="concreteVisitor"
  - calledByClass="concreteElement"
  - quantifier="1"

• Fourth method call
  - className="concreteElement"
  - calledByClass="concreteVisitor"
  - quantifier="1"
Figure E.29: Visitor design pattern UML sequence diagram
Bibliography


