A FINE-GRAINED MODEL FOR DESIGN PATTERN DETECTION IN EIFFEL SYSTEMS

MAURICE LEBON

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

GRADUATE PROGRAM IN SOFTWARE ENGINEERING
YORK UNIVERSITY
TORONTO, ONTARIO
SEPTEMBER 2007
A FINE-GRAINED MODEL FOR DESIGN
PATTERN DETECTION IN EIFFEL
SYSTEMS

by Maurice Lebon

a thesis submitted to the Faculty of Graduate Studies of
York University in partial fulfilment of the requirements
for the degree of

MASTER OF SCIENCE
© 2007

Permission has been granted to: a) YORK UNIVERSITY LIBRARIES to lend or sell copies of this dissertation in paper, microform or electronic formats, and b) LIBRARY AND ARCHIVES CANADA to reproduce, lend, distribute, or sell copies of this thesis anywhere in the world in microform, paper or electronic formats and to authorise or procure the reproduction, loan, distribution or sale of copies of this thesis anywhere in the world in microform, paper or electronic formats.

The author reserves other publication rights, and neither the thesis nor extensive extracts for it may be printed or otherwise reproduced without the author’s written permission.
A FINE-GRAINED MODEL FOR DESIGN PATTERN DETECTION
IN EIFFEL SYSTEMS

by Maurice Lebon

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 York University 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. Vassilios Tzerpos
2. Jonathan Ostroff
3. Wolfgang Stuerzlinger
4. Marvin Mandelbaum
Abstract

Design patterns have been used for many years in order to build software systems whose design has a high level of flexibility and scalability. In the reverse engineering field, detecting design patterns not only simplifies the understanding of the target system implementation but also provides the rationale behind the system’s design, i.e. why was this design used? Therefore, automating the detection of design pattern instances significantly helps the process of reverse engineering large-scale or legacy software systems.

In this thesis, we extend on Wei Wang’s work in detecting design patterns in Eiffel systems [26] by adding a new layer to the detection process based on fine-grained information contained in the corresponding system. As a result, we provide a list of fine-grained rules which best describe the static design of each of the 23 GoF patterns so that only true pattern instances are retained from the detection process (thus filtering a large number of false positives that could have been found without this phase).
We also introduce a new tool, FiG (Fine-Grained Analysis Tool). This tool not only detects design patterns on the static and dynamic levels but also on the fine-grained one. It was also designed in a flexible fashion by creating specific modules for each phase. Consequently, it can be used to detect design patterns in other languages such as Java or C++ since modules can either be made language dependent (e.g. system facts extractors) or independent (e.g. definition matching phases).

Finally, we performed a variety of case studies to validate the tool and determine its efficiency and precision.
Acknowledgements

I would like to thank my family for its constant support as well as my friends in and outside of university and my supervisor Prof. Vassilios Tzerpos who has been bravely guiding me throughout this whole experience with pertinent suggestions and lots of valuable feedbacks.

Special thanks to Wei Wang who first worked on detecting design patterns in Eiffel systems. His work was quite inspiring and allowed me to grasp faster the main points and steps to undertake to make an extended and more efficient design pattern recovery tool based on fine-grained information.

Finally, I would like to thank the York University Computer Science department for its full support and availability as well as everyone who has had his or her share of helping support in a direct or indirect way.
# Table of Contents

Abstract iv

Acknowledgements vi

Table of Contents vii

1 Introduction 1

1.1 Design pattern detection 3

1.1.1 Coarse-grained detection of design patterns 3

1.1.2 Fine-grained detection of design patterns 5

1.2 Research Contributions 6

1.3 Thesis outline 7

2 Background 11

2.1 Coarse-grained design pattern detection 12

2.2 Fine-grained design pattern detection 16
2.2.1 Sub-patterns in FUJABA ................................. 16
2.2.2 Elemental Design Patterns ............................. 18
2.3 Design Patterns in Eiffel Systems ........................ 22
2.4 DPVK: Design Pattern Verification toolKit ............... 23
  2.4.1 Supporting software .................................. 26
2.5 Eiffel language and features ............................... 29

3 Fine-grained rules 31
  3.1 Creational Patterns ...................................... 32
    3.1.1 Factory Method ...................................... 32
    3.1.2 Abstract Factory ..................................... 33
    3.1.3 Builder ............................................... 34
    3.1.4 Prototype ............................................. 35
    3.1.5 Singleton ............................................. 35
  3.2 Structural Patterns ...................................... 38
    3.2.1 Adapter ............................................... 38
    3.2.2 Bridge ............................................... 39
    3.2.3 Composite .......................................... 40
    3.2.4 Decorator .......................................... 41
    3.2.5 Facade .............................................. 42
3.2.6 Flyweight ............................................. 43
3.2.7 Proxy .................................................. 44

3.3 Behavioral Patterns ........................................ 45

3.3.1 Chain of Responsibility .................................. 45
3.3.2 Command .................................................. 47
3.3.3 Interpreter ................................................. 48
3.3.4 Iterator ..................................................... 49
3.3.5 Mediator .................................................... 49
3.3.6 Memento ..................................................... 50
3.3.7 Observer ..................................................... 51
3.3.8 State ......................................................... 52
3.3.9 Strategy ...................................................... 53
3.3.10 Template Method ......................................... 54
3.3.11 Visitor ....................................................... 54

4 The FiG tool ....................................................... 56

4.1 DPVK .......................................................... 56

4.1.1 Stage 1: Static and dynamic fact extraction ............... 59
4.1.2 Stage 2: Candidate instance recovery ......................... 62
4.1.3 Stage 3: False positive elimination ......................... 63
4.2 FiG: An improvement and extension of DPVK

4.2.1 Command line interface and order of phases

4.2.2 Standardization of input and output formats

4.2.3 The fine-grained analysis phase

4.2.4 The post-processing analysis phase

4.2.5 An overview of the contributions in numbers

4.2.6 FiG diagrams

5 Experimental Case Studies

5.1 Case Study 1: Sample implementations

5.1.1 Comparison with PDE

5.2 Case Study 2: Patterns in FiG’s parsing system

5.2.1 Bridge pattern in FiG

5.2.2 Builder pattern in FiG

5.2.3 Summary

5.3 Case Study 3: Student implementations

6 Conclusion

6.1 Research contributions

6.2 Future Work

A Appendix-A
A.1 Catalogues of Design Patterns .............................. 118
A.1.1 Abstract Factory ........................................... 119
A.1.2 Builder ....................................................... 120
A.1.3 Factory Method .............................................. 121
A.1.4 Singleton ..................................................... 122
A.1.5 Adapter ....................................................... 123
A.1.6 Bridge ......................................................... 124
A.1.7 Composite .................................................... 125
A.1.8 Decorator ..................................................... 126
A.1.9 Flyweight ..................................................... 127
A.1.10 Proxy ......................................................... 128
A.1.11 Chain of Responsibility ................................. 129
A.1.12 Command ................................................... 130
A.1.13 Interpreter .................................................. 131
A.1.14 Iterator ....................................................... 132
A.1.15 Mediator ..................................................... 133
A.1.16 Memento ..................................................... 134
A.1.17 Observer ..................................................... 135
A.1.18 State ......................................................... 136
A.1.19 Strategy ..................................................... 137
A.1.20 Template Method ............................................. 138
A.1.21 Visitor ......................................................... 139
A.2 Table of fine-grained rules ...................................... 140

B Appendix-B ...................................................... 144
B.1 FiG: User Manual ............................................... 144

Bibliography .......................................................... 153
1 Introduction

Software developers use design patterns to make better software in terms of reliability, comprehensibility and reusability. Design patterns strive to simplify the software design and decouple concrete implementations from their abstractions. Conversely, comprehending a software design can be greatly simplified by looking for the design patterns used.

In other words, design patterns are solutions to programming problems that automatically implement good design techniques. Common problems solved previously according to a specific design are deposited in a pattern repository and the design solution can be reused for developing new software facing the same type of issues or to understand the design rationale behind a system’s implementation (especially in legacy systems lacking proper documentation).

The origin of design patterns goes all the way back to the late seventies from the work of Christopher Alexander (Alexander et al., 1977) in the domain of architecture, when he suggested the following: "Each pattern describes a problem which
occurs over and over again in our environment, and then describes the core of the solution to this problem in such a way that you can use this solution a million times over, without ever doing it the same way twice.”

In 1995, Design patterns: Elements of Reusable Object-Oriented Software written by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides - now all known as the Gang of Four (GoF) - was published. This book introduced the concept of software design patterns, their uses, and a comprehensive catalogue of 23 different patterns.

Today, design patterns are widely accepted as useful tools for guiding and documenting the design of object-oriented systems. They are even more important in projects that involve software maintenance and/or the addition of new features or extensions. Not only do they allow greater scalability in terms of adding new modules but they let developers keep track of the ”big picture” which simplifies the understanding and extending of the system.
1.1 Design pattern detection

The process of detecting design patterns in existing software may either be manual, automatic or semi-automatic. Considering that most software targeted for reverse-engineering analysis may contain thousands of lines of code, manual detection of patterns might be a tedious or impossible task. It is obviously more efficient to implement tools to handle the process automatically (or at least semi-automatically).

1.1.1 Coarse-grained detection of design patterns

The detection of design patterns can be done on two levels: statically or dynamically. For each level, patterns are thoroughly defined to represent their static structure or expected dynamic behavior. The pattern definitions are sets of rules that abide to the pattern’s characteristics, whether it be statically or dynamically.

At the static level, these rules define the pattern’s structure by considering the composition of its UML diagram. Such rules need to be as strict and accurate as possible to clearly differentiate one pattern from another, to avoid redundancy between definitions and to detect all true positives without missing any. To detect patterns using those definitions, we need facts about the system. Static facts are represented by a set of inheritance and client-supplier relationships amongst classes of the system. Definitions also follow the same pattern of representation. For exam-
ple, one definition could be: \textit{inherits} \texttt{classA classB} and \textit{uses} \texttt{classC classA}.

When all the static facts have been collected, the next step consists of matching them against the definitions in order to return candidate instances that abide to the rules.

At the dynamic level, definitions are illustrations of the expected behavior of the pattern in the target system. The facts are obtained after tracing the running system and are only expressed in terms of message passing (which is also the case for the definitions). For example, if \texttt{classA} calls \texttt{classB} via some method, one of the facts would be: \texttt{calls classA classB}. Once the dynamic facts are obtained, pattern matching occurs once again but at the dynamic level this time. Matching instances are retained as true positives. This phase cannot be used to discover candidate instances because it is not possible to consider all scenarios such that all classes participate at runtime (abstract classes would also be excluded). However, it can be used to remove false positives obtained after the static analysis phase.

It is a well-known fact that, at the static level, it is hard to have a strict enough definition to discard all possible false positives when opting for a fully automatic pattern detection. This is mostly due to the redundancy between definitions as mentioned earlier. That is why false positive elimination phases can tremendously improve the results after the candidate instances discovery phase. Some tools use multiple passes [22], recursive filtering [15], dynamic analysis [26] and/or
fine-grained analysis [23] to address this issue.

1.1.2 Fine-grained detection of design patterns

In this thesis, we expand on Wang’s work [26] by investigating what types of fine-grained information can be used in order to reduce the number of false positives in the list of design pattern instances detected in an Eiffel system.

A small number of tools such as SPQR [23] or FUJABA [20] have been created to detect fine-grained information in a system for specific languages such as Java or C++. SPQR takes into account a list of Elemental Design Patterns (EDPs), suggested by Jason McC.Smith and David Stotts in [23] when they introduced a new fine-grained model to detecting sub-elements of a pattern taken from the source code itself (e.g. method signatures, return type of a method, etc...). FUJABA adopts a similar approach since it also detects sub-patterns to reconstitute normal design patterns at the end. However, in FUJABA’s case, the catalog of sub-patterns has not been released to date.

In addition to the normal detection phases, the new fine-grained analysis phase relies exclusively on source code parsing of the target system to filter out false positives. The requirement for a parser makes this phase language-dependent which can be seen as a possible limitation to this method. However, using fine-grained rules for each pattern to draw the difference between true and false positives in
a list of potential candidate instances can help drastically improve the results of
the detection process. While the parsing is being performed, an ASG (Abstract Syntax Graph) is being shaped at the same time by adding newly read elements to
its leafs such as special keywords, methods signatures, etc... When that phase is
completed, the fine-grained rules are matched with the fine-grained facts contained
in the ASG onto the candidate instances and only matching ones (i.e. true positives)
are retained.

As mentioned earlier about class-level static analysis, some patterns have similar
static definitions and this often results in the detection of many false positives
due to the ambiguity and redundancy between those definitions. The fine-grained
rules almost completely eliminate the existing redundancy between definitions and
contribute to a much more accurate detection of true and false positives. Therefore,
fine-grained analysis can increase the precision and recall of the results.

1.2 Research Contributions

The purpose of this thesis is to present a new more precise approach for detecting
design patterns in Eiffel systems as well as for listing the different patterns involved
and their characteristics. We based ourselves on Wang’s study in detecting design
patterns in Eiffel systems at the coarse-grained level and improved the precision
of the results by adopting a finer grained approach. The DPVK tool (Design
Pattern Verification toolKit) developed by Wang has become the core component of our new tool FiG (Fined-Grained Analysis Tool) which improves the detection of design patterns by using a new fine-grained analysis layer for filtering out false positives. FiG supports language-independent detection process of patterns and standard input and output data formats making it a far more versatile and flexible tool.

1.3 Thesis outline

- Chapter 1 - Introduction

This chapter introduces the motivation behind detecting design patterns as well as the contributions of this thesis. We talk about the usefulness of detecting design patterns in the reverse engineering world and how research has made significant progress in this area in the last decade. We then focus on detecting design patterns in Eiffel systems. We present our research contributions with improvements made to the current detection tool for discovering design pattern in Eiffel systems.

- Chapter 2 - Background

We present a list of reverse engineering tools used to discover design patterns. We review their efficiency and limitations and describe their overall usefulness in detecting design patterns. We also describe fine-grained level detection of design
patterns and the usefulness of such an approach. Wang’s work on design pattern detection in Eiffel Systems [26] is presented and we talk about Eiffel features that should be of interest in helping to detect design patterns in this language.

- **Chapter 3 - Catalogue of Design Patterns**

  All GoF patterns are presented and described in detail. We deduce required and optional fine-grained rules that could help detect these patterns. Many class-level detection limitations are also addressed and a way to improve them using fine-grained information is discussed.

- **Chapter 4 - The FiG tool**

  We describe the overall usage of our tool and how the definitions and facts are used to produce a list of potential candidate instances for the pattern being detected. We also present the changes and improvements brought to the tool for better performing. The implementation of the fine-grained analysis phase as well as its functioning are described in detail.

- **Chapter 5 - Experimental Case Studies**

  Three case studies are presented to showcase FiG’s precision and recall. For each case study, we introduce the background, present the experiments and give conclusions. The three case studies differ in type, complexity and duration.
The first case study consists of running FiG on each pattern’s Eiffel implementation given in [16]. A complete table giving the numbers of true and false positives for each case is provided. Many general yet interesting observations are made.

In the second case study, we apply FiG on a big Eiffel system. FiG’s Eiffel implementation is used as the target system of the detection process. Amongst the patterns found, we present our results for the Builder and Bridge patterns and show how they help understanding the system’s structure.

The last case study is the study of two different set of student implementations that should contain respectively the Decorator and the State patterns. With a total of 62 student implementations each, we try to detect all GoF patterns (excluding Facade and Prototype) in those and we draw a table of results for each case. Analyzing the results leads to many interesting observations.

- **Chapter 6 - Conclusion**

The chapter concludes the thesis and summarizes the improvements and contributions of FiG into the design patterns field and most importantly in Eiffel systems. Different limitations of FiG that we were not able to address at this time are also mentionned as the basis of our future work.

- **Appendix A - Catalogue of definitions**
This second appendix presents a catalogue of all GoF patterns. Each pattern has its static, dynamic and fine-grained definition.

- **Appendix B - User manual of FiG**

  This appendix describes how to use FiG to detect design patterns in Eiffel systems. It contains a collection of steps to undertake in order to handle the detection process.
2 Background

Detecting design patterns in a given target system can help us understand its structure in terms of class relationships as well as dynamic behavior. Throughout the history of programming, software designers have made considerable progress in making more complex and powerful abstractions. New programming techniques such as the object-oriented paradigm allowed far better designed software by aiming for more flexible implementations that would make the updating or re-adaptation of a system an easier and smoother process. Nowadays, it is common that systems and programs are conceived using already existing patterns or specific program structures.

In this chapter, we present several coarse-grained design pattern detection tools and their impact in the reverse engineering area. Then, we describe fine-grained approaches presented in the literature. Finally, we discuss design patterns in Eiffel systems while we introduce a coarse-grained detection tool called DPVK.
2.1 Coarse-grained design pattern detection

There are several contributions in the literature concerning design pattern detection and some promising or already established detection tools. As examples, we cite the following:

1. The Pat System [18] which is a design recovery tool for C++ applications. It extracts design information directly from C++ header files and stores it in a repository. Patterns are defined as PROLOG rules and the design information is translated into facts. The Pat system can only be used to detect structural design patterns. The actual matching work is done by a PROLOG engine.

2. KT [10], a tool that can reverse-engineer design diagrams from Smalltalk code and use this information to detect patterns. The author of KT advocates that any tool designed to detect design patterns must support both static and dynamic modelling constructs. In terms of diagrams, static information includes is-a and has-a relationships and dynamic information includes object interaction or message flow.

3. SPOOL [17] (Spreading Desirable Properties into Design of Object-Oriented, Large-Scale Software Systems) aims at both software comprehension and software design quality assessment. The SPOOL environment for design patterns engineering supports both forward and reverse engineering of design patterns.
This environment comprises functionality for design compositions, change impact analysis, and most importantly, support for the recovery of design patterns. Analyzers can then zoom into design components that resemble patterns and check whether they do match existing pattern descriptions. True and false positives can therefore be assessed more easily this way.

The SPOOL system is comprised of two major phases:

(a) The first phase is source code capturing and system facts extracting from the source code. At the moment, SPOOL can only parse C++ source code and the support for Java is still under development. The output, being independent of the language being parsed, is imported into the SPOOL repository. In order to leverage existing tools, XML and DOM (Document Object Model) are used to describe collected facts about the software system’s structure. The information collected includes file system structure, classifiers, inheritance relationships, attributes, methods, return types, etc.

(b) The second phase consists of retrieving design patterns from the facts collected previously. This can be done in 3 possible means: fully automatic, semi-automatic and fully manual recovery. For the fully automatic design recovery, pattern description is stored in the repository as abstract
components (class-level relationships) and it is used to help SPOOL identify pattern instances. For the manual recovery, the analyzers are free to try and recognize design patterns from their own perspectives.

4. JBOORET [19] (Jabe Bird Object-Oriented Reverse Engineering Tool) has a flexible design allowing it to extract higher-level design information from system artifacts. It adopts a parser-based approach to do the extraction. The JBOORET for C++ consists of three major components: a data extractor, a knowledge manager and an information presenter. First created for C++ systems, it can also easily be adapted to other languages.

5. CrocoPat [8] was created as a response to two major problems found in conventional detection tools: the lack of an easy and flexible specification of the patterns (mainly due to the limitations of the specification language) and poor detection speed performances from existing tools which are not acceptable for large real-world systems. Therefore, CrocoPat aims at satisfying three requirements in the automation of design pattern detection paradigm [8]:

- The analysis is done automatically by the tool (does not require user interaction).

- The properties of a system are specified in an easy and flexible way due to the patterns being described by relational expressions. The user can
freely add new patterns he is interested in or modify existing ones to solve specific problems.

- The tool is able to analyze large object-oriented programs (up to 10,000 classes) in acceptable time.

The tool represents the abstract model of the program using a data structure based on binary decision diagrams, which are proven to allow for an efficient recognition also for large systems comprising several MLOC (Million Lines of Code) source code.

6. DPVK (Design Pattern Verification toolKit) is a basic coarse-grained design pattern detection tool for Eiffel systems. It does both static and dynamic analysis of the target system before inferring a list of potential candidate instances. DPVK will be presented in detail in chapter 4.

The common objectives of all these tools is to detect design patterns based on high-level static and dynamic analysis exclusively. However, studying program source code in more detail to detect design patterns is a more recent approach that seems to bring better results in terms of precision and recall of the detection process.
2.2 Fine-grained design pattern detection

It is important to adopt an effective approach to reverse-engineering systems or portions of systems. The need to abstract the target implementation is necessary to understand, learn or analyze the system. Design Patterns can play an important role in this quest for abstraction.

Detecting design patterns to provide solutions to common implementation challenges has brought the idea of decomposing patterns into smaller and recurring elements they consist of. These elements or sub-components have been recognized in both forward and reverse engineering and they have been named fragments [13], motifs [12], minipatterns [11], sub-patterns [20] and elemental design patterns [23]. Two important approaches and tools are described below.

2.2.1 Sub-patterns in FUJABA

FUJABA (From Uml To Java And Back Again) [20] is a tool which focuses on detecting sub-patterns defined in a design pattern catalog in order to incrementally detect patterns. As implied by the name, design patterns are defined in terms of UML class diagrams which present their structures. The behavioral aspects of design patterns are expressed through story-diagrams, which are a combination of activity and interaction diagrams. Within FUJABA, common parts of design
patterns are defined separately as sub-patterns to define other sub-patterns or patterns.

During the design pattern detection phase, information is extracted from the source code and expressed as an abstract syntax graph (ASG). Additional annotations enrich the graph to notify the presence of patterns or sub-patterns. These annotations are added by using graph transformation rules [21, 27] which define the different sub-patterns (or patterns). Transformation rules work in a hierarchical level system. Rules depending only on information extracted from source code belong to level 1, while rules depending on other rules get a higher level number consistent with the topological order of rules. This order of rules is useful in order to reconstitute patterns by applying rules on the ASG.

One advantage of FUJABA is that it can be applied to other programming languages other than Java, its native and target language, thanks to the use of ASG during the detection process. The definition of patterns and sub-patterns and their behavior through UML diagrams makes it easier for software engineers to understand, use and extend. The introduction of sub-patterns makes the detection process incremental, enabling the provision of first results in a short time with consistent information [6]. However, FUJABA fails to provide a catalog of sub-patterns nor information about how they should be identified.

With the introduction of sub-patterns, there are more levels of abstraction be-
tween the source code and high-level specification of design patterns. Consequently, the differences between two levels of abstraction are significantly reduced. A sub-pattern may have different implementation variants, but at the upper abstraction level(s) it is seen as a unique element.

The existence of intermediate abstraction levels allows the detection algorithm to use both bottom-up and top-down strategies [6]. This helps to provide useful results in a short amount of time.

2.2.2 Elemental Design Patterns

An analysis of the Gang of Four patterns (GoF) reveals many shared structural and behavioral elements, such as the similarities between Composite and Visitor, for instance [23]. Jason McC.Smith and David Stotts [23] conducted several studies on the decomposition and recomposition of patterns. They therefore introduced a lower level approach to the detection of Design Patterns by adding a layer between a system’s abstraction and its implementation: that layer is composed of ”mini-patterns” that can form bigger and more abstract patterns while being quite close to the code itself (e.g. method signatures, attribute types, etc...).

The problem is that the common design patterns definitions used in the detection process mainly refer to class-level relationships between classes of a system. But restricting ourselves to class-level relationships of the different patterns can
lead to a great number of false negatives due to the class-level relationships not being distinctive enough.

Their next step was therefore to examine the GoF text. Instead of a purely structural inspection, Smith chose to identify common elements used in the patterns. From those fine-grained components (such as method signatures and creation calls), a catalogue of Elemental Design Patterns (EDPs) was presented.

This approach showed how to recognize a given pattern by detecting both static and dynamic EDPs that constitute the pattern. They also showed how useful EDPs were in improving the results of the detection process since they act as an additional filter layer for removing false positives.

Here are some concrete examples of EDPs:

1. Figure 2.1 shows a basic class diagram for the RedirectInFamily EDP [23]. FamilyHead defines the interface that contains a method that can be overridden. Redirecter implements that interface through inheritance. The Redirect-InFamily EDP specifies in this case that Redirecter invokes a similar method (similarity based on the signature of the method) to the one currently being executed on an object of type FamilyHead.

2. The ExtendMethod EDP is used to extend, not replace, the functionality of an existing method in a superclass. Figure 2.2 shows the structure of such a
The list of EDPs they came up with can be seen in Figure 2.1.

Based on this new approach, Smith and his colleagues developed a design pattern detection tool called SPQR (System for Pattern Query and Recognition) [24] for C systems.

A parse tree is first retrieved (using gcc) from the source code and stored in a file. An intermediary tool, called gcctree2oml, is then used to translate the
<table>
<thead>
<tr>
<th><strong>Object Elements EDPs</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>CreateObject</td>
<td>An object creation call</td>
</tr>
<tr>
<td>Abstract Interface</td>
<td>The class is defined as abstract or deferred</td>
</tr>
<tr>
<td>Retrieve</td>
<td>Gets an expected particular type of object from a method call.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Type Relation EDPs</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Inheritance</td>
<td>A class is inheriting from another class</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Method Invocation EDPs</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Recursion</td>
<td>A call to the same feature in the class</td>
</tr>
<tr>
<td>Redirect</td>
<td>A method calls a similar method in another object</td>
</tr>
<tr>
<td>ExtendMethod</td>
<td>A method defined in the superclass is extended in the subclass</td>
</tr>
<tr>
<td>RedirectInFamily</td>
<td>A redirection to a similar method but within one’s own inheritance family</td>
</tr>
<tr>
<td>RedirectInLimitedFamily</td>
<td>Like RedirectInFamily but limiting to a subset of the family tree, excluding possibly messaging an object of one’s own type</td>
</tr>
<tr>
<td>RedirectedRecursion</td>
<td>A form of object level iteration</td>
</tr>
<tr>
<td>Delegate</td>
<td>A method delegates part of its behavior to another method in another object</td>
</tr>
<tr>
<td>DelegateInFamily</td>
<td>Gathers related behaviours from the local class structure</td>
</tr>
<tr>
<td>DelegateInLimitedFamily</td>
<td>Limits the behaviours selected to a particular base definition</td>
</tr>
<tr>
<td>Conglomeration</td>
<td>Aggregating behaviour from methods of Self</td>
</tr>
<tr>
<td>DelegateConglomeration</td>
<td>Gathers behaviors from external instances of the current class</td>
</tr>
<tr>
<td>RevertMethod</td>
<td>A subclass wants to not use its own version of a method for some reason</td>
</tr>
</tbody>
</table>

Table 2.1: EDP Catalog
parse tree of the previous stage into an XML description of the features. That Object XML file is then fed to a second tool oml2otter which will create a feature-rule input file to an automated theorem prover (Argonne National Laboratory’s OTTER [1]) which will use the EDP Catalog as well as a derived type of Rho Calculus to deduce pattern instances. Since EDPs are considered small types of design patterns that can form more complex and general design patterns, they can be used to form a mathematical formalization of the pattern descriptions. The use of Rho Calculus, an extension of the Sigma calculus to support reliance operators (relationships between objects, methods and fields), makes expressing the EDPs easier since they can be deduced by a set of low-level facts such as inheritance links, method calls, etc...

2.3 Design Patterns in Eiffel Systems

Detecting design patterns in systems written in a language with a number of unique characteristics, Eiffel, has not attracted considerable attention yet. Wei Wang presented a design pattern detection approach for Eiffel systems that uses class-level relationships, such as inheritance or client-supplier relationships [26]. Even though his approach has shown considerable merit, there are several types of fine-grained information that were not used that could help improve the precision and recall of such a detection process. As presented in the previous section, looking
for fine-grained information can prove to be considerably useful in removing false positives and improving in general the results of the detection process since each pattern’s definition is not limited to the class-level relationships layer but can be further refined with fine-grained information for more precision during the detection process.

### 2.4 DPVK: Design Pattern Verification toolKit

The Design Pattern Verification toolKit (DPVK) was created by Wang to perform pattern matching using static and dynamic facts from a given system and a set of rules for the aimed pattern in that system. DPVK also uses two external tools. First, EiffelStudio is used to retrieve a list of static and dynamic facts by using the software’s Eiffel compiler and tracer. Second, REQL is used to apply the pattern’s rules on these facts only to return the ones matching. These two different tools are described below in Section 2.4.1.

We will now have a closer look at the detection approach presented by Wang. There are two definitions that Wang has created for each pattern: a static definition which contains relationships between classes, and a dynamic definition, which presents rules that describe the behavior of the potential pattern (in terms of message passing).

We will use the Command pattern as a running example. The class-level re-
Figure 2.3: Class-level relationships of the Command pattern - The double arrows correspond to Client-Supplier relationships. The single one represents an inheritance link.

These relationships can also be described with the static definition presented in Figure 2.4.

At runtime, ConcreteCommand objects are first created and associated with a Receiver. When the action corresponding to the command is triggered (a button pressed, an item of a list double-clicked, etc...), then the Invoker calls the associated Command. The ConcreteCommand executes its code on the Receiver. These interactions can be expressed with the dynamic definition presented in Figure 2.5.

To detect the Command pattern, we could therefore use a tool that would read both static and dynamic facts about the target implementation, as well as the two definitions and return design pattern instances that match the definitions.

That was the initiative undertaken by Wang. He presented a Design Pattern Verification toolKit (DPVK) that first extracts from the source code static facts
about the target system. These facts are then manipulated by REQL scripts [28] that compare the static definition for the corresponding pattern and the static facts obtained. Each subset of classes in the system is considered as a candidate instance. If a candidate instance does not comply to the definition, it is discarded. A list of candidate instances that match the pattern’s static definition is produced at the end of this stage.

The next stage of the detection process consists of eliminating false positives using the dynamic facts. These facts are extracted from a running trace log that can be obtained if the system is run under EiffelStudio [2] with the appropriate options turned on. DPVK attempts to match the dynamic facts with the dynamic definition provided for the aimed pattern. This often removes a large number of false positives.

In theory, if both static and dynamic definition are perfectly accurate and con-
tain only necessary conditions for the detection of each pattern, there should not be any false positives or negatives at the end of the detection process. However, in practice, most of the patterns can be implemented in many different ways. It is therefore practically impossible to detect all possible variants of a pattern in a given system since it would be very time-consuming. Also, some patterns have similar class-level structures and for some variants of these patterns, it is impossible to distinguish between the two by using solely class-level relationships as a detection measure. This is why, it is still possible to obtain many false positives in the results of the detection. The work presented in this thesis attempts to improve the precision of DPVK’s detection process by utilizing fine-grained information.

2.4.1 Supporting software

1. EiffelStudio

ISE’s EiffelStudio [2] is one of the most popular Eiffel compilers available to the public. It has recently been released as an open-source system [2]. EiffelStudio provides a practical and efficient IDE and serves very well the best potential of the Eiffel language by offering a large variety of facilities. Indeed, system modeling, designing, implementing and debugging can be streamlined and done within EiffelStudio. Most of the important tools needed by developers are present in this powerful IDE. Debugging is also much simplified with
the Design by Contract mechanism since it can prevent most of the bugs from occurring.

The other advantage of EiffelStudio is its portability. It can indeed run on both Windows or Unix systems and even VMS systems. It is therefore easy for developers to migrate an application from platform to platform. Microsoft’s .NET framework is also supported as of the latest version of EiffelStudio.

2. **REQL**

REQL (Reverse Engineering Query Language) is an extended implementation of Grok written in Java. Grok is a relational calculator that supports a scripting language. It was initially created by Ric Holt in 1995 in order to manipulate binary relations with the purpose of understanding large-scale software systems. It includes an interpreter that can be treated as a relational processor.

REQL extracts facts from a factbase that collects all relationships between each entity in a universe. This factbase may be a single file or several files in Rigi Standard Format (RSF). In RSF, each fact is represented as a triple of the form (R,x,y), which indicates relation R contains the ordered entity pair (x,y). REQL provides set operators, such as union, intersection, substraction, comparison and so on. The user can create scripts and let the REQL inter-
preter perform a batch of operations upon a factbase. REQL will calculate, filter and output qualified sets of entities which have the relationships defined by the REQL script. REQL has many other powerful features [28].

REQL is used in the detection process to manipulate the static relationships among classes and objects since both inheritance and client-supplier relationships can be represented as a triple in RSF: \texttt{inherits} \texttt{x y} or \texttt{uses} \texttt{x y}. By feeding both the static fact file and the static definition of the pattern to REQL, it will automatically do the matching and output the potential candidate instances.

3. **Eiffel parser**

In order to statically analyze each class that could potentially form the pattern we’re looking for, we need to parse it. The tool we used for this work is the ES-Parser tool developed at York University in a project which was first initiated by Eric Kerfoot. That parser is based on Gobo [4] (a portable Eiffel class library covering data structures and algorithms) library’s tools \texttt{geyacc} [5] and \texttt{gelex} [3]. The ES-Parser Eiffel library was therefore imported in our tool and the different features provided by the parser were used when needed. The first step in most cases is to point to the source cluster or ACE file of the target system. ES-Parser would parse all the necessary source files automatically.
Although that can prove to be memory consuming depending on the size of the system analyzed, it has shown to be quite fast and efficient with relatively small (to medium) sized software systems (e.g. systems with about 100,000 lines of codes or more). Moreover, it is possible to target a subcluster of the original cluster to accommodate the resources used (CPU usage or memory consumption) or to focus on a particular part of the system. The parser is quite powerful since it can also generate a complete tree structure of the system.

The most important features that our tool used were the ones returning class names, feature names, attributes and return types as well as export restrictions and any keyword that was judged useful to detect for some patterns (such as the "once" feature for the Singleton pattern).

2.5 Eiffel language and features

1. **Design by Contract.** Perhaps the feature that Eiffel is most famous for is Design by Contract, i.e. the fact that method pre- and post-conditions, as well as class invariants can be explicitly stated in the system source and asserted at runtime. Despite the main benefits of Design by Contract, it is not a feature one can utilize for pattern detection, since contracts are not necessary elements of design pattern implementation.
However, since many design patterns contain interesting contracts, an interesting piece of future work would be to examine the contracts in detected design pattern instances and possibly suggest additional contracts to the developers.

2. **Restrictive Access**. For some patterns, it is important to restrict access to certain methods to specific classes only. Methods in Eiffel can be exported to either all classes, or only a specific set of them. This allows for tighter control of system dependencies than the public/protected/private mechanism of C++/Java.

3. **Covariance**. Let us consider an example that illustrates the concept of covariance. First, let us define the following: Class FRAME has a feature called `contains(widget: WIDGET): BOOLEAN`. A PANEL is a FRAME. A BUTTON is a WIDGET. However, PANEL may be able to contain only certain types of WIDGETs like BUTTONs but not others like SCROLLBAR which must be contained first in another type of FRAME like a VIEWAREA. Thus, the feature `contains` will be ”redefined” in the PANEL subclass to reflect the right argument type (i.e. `contains(button: BUTTON): BOOLEAN`). This is called a covariant redefinition.
3 Fine-grained rules

Although a few other fine-grained tools also exist, they are mainly aimed at Java and C++ systems. The fact that little work in design pattern detection has been oriented towards Eiffel systems was our motivation to research this area and explore it with the addition of a fine-grained level model. As is commonly known, design patterns are mostly used in object-oriented software since it provides greater flexibility in terms of defining good design. Eiffel is known to be a very precise and tidy object-oriented language. We were interested to determine how the unique features of Eiffel would affect the detection process and results.

In this chapter, we present all of the GoF patterns [14] with fine-grained detection rules that will be used to remove false positives. Each pattern has at most 4 different rules. However, most of them only have 1 or 2 rules. Some rules are labeled as being "Optional”. This means that in some variant of the pattern, it is possible to observe such a rule in the corresponding implementations of the pattern but it is not mandatory for all variants of the pattern’s implementation. Such rules can
be useful when a large number of pattern instances remains even after fine-grained analysis. They can indicate instances that are more likely to be true positives. A catalogue of the patterns with all the definitions (static, dynamic and fine-grained definitions) is presented in an appendix at the end of the thesis.

3.1 Creational Patterns

3.1.1 Factory Method

"The Factory Method lets a class defer instantiation to subclasses" as defined by Gamma et al. It establishes an interface for creating an object which at creation time can let its subclasses decide which class to instantiate.

Each time the factory method is called, a new object is created. Obviously, a factory is not needed to make an object. A simple call to the constructor is sufficient. However, the use of factories gives the programmer the opportunity to define an interface for creating an object while subclasses can decide which class to instantiate at runtime. This promotes loose coupling by eliminating the need to bind specific classes into the code.

Rule 1: We require the presence of the `create` keyword in a factory method in the ConcreteCreator class.

Since the Factory Method pattern defines an interface, the objects that will be created in the subclasses may be of a type that is a descendant of the type defined...
in the interface. This is called **covariance**, and it was explained in Section 2.5.

| Rule 2 (Optional): The return type of the factory method in the ConcreteCreator may be different than the one in the Creator. |

### 3.1.2 Abstract Factory

An Abstract Factory pattern instance can be seen as a composition of multiple Factory Method pattern instances, where the client can get several types of final products. One drawback of Wang's approach can be observed. Using class level relationships prevents us from differentiating an Abstract Factory from a Factory Method. The static and dynamic definitions used in detecting both patterns are indeed the same as shown in Figure 3.1. The difference between the two structures relies on the concept behind them: the Factory Method is used for creating a single object type while the Abstract Factory allows the creation of multiple, related objects. For both, the instantiation is deferred to subclasses that implement the corresponding generic interface.

We can use fine-grained information to see if a candidate Abstract Factory is indeed a collection of Factory Methods.

| Rule 1: We require the presence of at least 2 or more factory methods in the AbstractFactory class. |

In many applications (or frameworks), the designer may want to make sure that the design remains extensible. For example, it might be useful to restrict the
Figure 3.1: Static and Dynamic definitions for the Abstract Factory and Factory Method patterns

construction process of certain types of concrete products to factory classes only. Such restrictions will greatly simplify any changes affected to a product family. Restricting the access is therefore a good design practice to constrain the creation of concrete products by means of factory classes only. However, this rule is not a requirement.

Rule 2 (Optional): The constructor of the ConcreteProduct may be exported only to the AbstractFactory (e.g. `creation {ABSTRACT_FACTORY}`).

3.1.3 Builder

The main role of the Builder pattern is to decouple the construction of an object from its representation allowing a greater flexibility in the building phase which
can create various other representations using the same construction process.

The ConcreteBuilder needs to return an object accessible by the client. As opposed to the Abstract Factory, the object is returned as a separate final step. Therefore, the Builder class needs to contain at least an attribute or feature that returns an object of that product. Otherwise, we are definitely not in the presence of the Builder pattern.

| Rule 1: We require that there is at least one public attribute or feature that returns an object of type Product. |

### 3.1.4 Prototype

A prototype is a class that has facilities to clone. Therefore, every class in Eiffel is a prototype since the features `clone` and `deep_clone` can be inherited from ANY, which is the ancestor of all other classes.

### 3.1.5 Singleton

The goal of this pattern is to prevent having more than one instance of a particular class during runtime. Although it is not possible to rigorously implement the Singleton pattern in Eiffel [7], we will attempt to detect the most common variant.

To understand the variant we will observe, we provide sample code of the SINGLETON and SINGLETON_ACCESSOR classes which are both represented in Figures 3.2 and 3.3 respectively.
class SINGLETON
create {SINGLETON_ACCESSOR}
    make
feature
    make is
        do
            .
            .
            .
        end
end -- SINGLETON

Figure 3.2: SINGLETON Class

class SINGLETON_ACCESSOR
feature {ANY}
    make: SINGLETON is
        once
            Result := singleton.make
        end
end -- SINGLETON_ACCESSOR

Figure 3.3: SINGLETON_ACCESSOR Class
The class we want to instantiate a unique object from is the SINGLETON class. The product SINGLETON is returned by the class SINGLETON_ACCESSOR. We can base ourselves on that structure to determine a set of rules to detect the pattern.

The restrictive access in the SINGLETON_ACCESSOR class is also important to note since the singleton object shouldn’t be accessible to any class by default but only to classes inheriting from the accessor exclusively. If the access was not restricted, the whole purpose of the Singleton pattern would be defeated since other classes would be able to create instances of the singleton.

Rule 1: We require the presence of restrictive access for the constructor of SINGLETON.

The whole purpose of the Singleton pattern is to have a unique SINGLETON object that is created during the lifetime of the running software. It is therefore necessary to recognize a unique once feature that returns the SINGLETON object.

Rule 2: We require the presence of the once keyword in the SINGLETON_ACCESSOR.

As mentioned in Rule 2, detecting creation calls is a crucial step in fine-tuning the detection of the pattern. But there is an additional check that is required to make sure we indeed found the right pattern: we need to ensure that the return type of the created object (inside the once block) is of type SINGLETON because it is the class from which we want to instantiate only one object.

Rule 3: We require the return type of the once method to be of type SINGLETON.
Moreover, it is important to ensure that the SINGLETON object returned by the `once` method was not imported in any way but simply instantiated following a creation call.

**Rule 4:** We require the presence of a creation call within the `once` block that returns an object of type SINGLETON.

Dynamically, we can check if at most one instance of the candidate SINGLETON class is created during runtime. That would ensure that there are no false positives.

### 3.2 Structural Patterns

#### 3.2.1 Adapter

The Adapter pattern is used when the Client cannot communicate directly with a fixed interface (called the Adaptee). Another class called the Adapter is then introduced to carry out the request from the Client to the Adaptee. Therefore, the Adapter provides the interface that the Client expects, and uses the interface that the Adaptee provides.

Note that it is possible to have many Adaptees used by an Adapter. However, the detection process will most likely return a different instance of the Adapter pattern for each of the Adaptees used by the same Adapter.

The Adapter pattern can be represented in 2 different forms but with the exact same intent: Class Adapter and Object Adapter. For the former possibility, the
Adapter will inherit both Client and Adaptee’s interfaces. But for the latter, an instance of the Adaptee is used to delegate the actual work that needs to be done.

Since the Adapter class is meant to provide the functionality required by the Client by using the Adaptee class, we can check whether most features of Adapter call Adaptee features.

| Rule 1: We require that most features (at least 50%) of Adapter call features of the Adaptee. |

### 3.2.2 Bridge

The goal of the bridge pattern is to decouple an abstraction from its implementation so that the two can vary independently.

The Abstraction class must use an attribute of type Implementor in order to forward requests to it.

| Rule 1: We require the presence of an attribute of type Implementor in the Abstraction class. |

Since the main objective is to decouple an abstraction from its implementation, most of the features would not be implemented at the abstract level but instead in the ConcreteImplementor. The same applies to the Abstraction class.

| Rule 2: We require that both the Abstraction and Implementor classes are deferred. |
3.2.3 Composite

The design of the Composite pattern is based on a tree structure where a request issued by the client is propagated from non-leaf nodes to non-leaf nodes until a leaf node is reached. During this phase of propagating the request down the tree, other additional operations can also be performed before and/or after forwarding the request. The advantage of the Composite pattern is that the Client doesn’t need to know about the child nodes since both non-leaf and leaf nodes have the same interface.

Since the Composite class propagates a request from nodes to child nodes, there must be a feature that calls the same feature in the child nodes because all nodes have the same interface. Such an example can be seen in Figure 3.4.

Rule 1: We require that in at least one feature of the Composite class, there is a call to another feature of the same name within the feature’s block.

Another thing to note is that the Composite class should inherit from the Component class, and also take that class as a generic parameter. An example can be seen in figure 3.5. However, it is possible to define a Composite pattern without using a list or array of elements. This can be the case when there are only two children per node (i.e. the tree structure is binary). For the first possibility, we need to detect the Component class as a generic parameter. For the second possibility, we could simply look for at least two fields of type Component in the class. The
value: INTEGER is
  local
  i: INTEGER
  do
    from i := children.lower until i > children.upper
    loop
      Result := Result + children.item(i).value
      i := i + 1
    end
  end
end

Figure 3.4: Typical propagation of requests in the Composite pattern - value being the feature called in the child nodes

class COMPOSITE_EQUIPMENT
 inherit
   EQUIPMENT
 feature
   children: ARRAY [EQUIPMENT]

Figure 3.5: Typical structure of the Composite class

following rule covers both of these cases.

Rule 2: We require that the Composite class either contains the Component class as a generic parameter or, contains at least two attributes of type Component.

3.2.4 Decorator

The Decorator can be seen as a "degenerate" case of the Composite pattern but with only one child per node. The main difference lies on the additional responsibilities added to the concrete decorators. It is therefore not intended for object aggregation.

The first rule observed for the Composite pattern can therefore be applied as
draw is
   do
     component.draw
     draw_decoration
   end

Figure 3.6: The same feature `draw` is called in the component being decorated well, meaning that there must be a feature in the ConcreteDecorator that will call the same feature in the Component. An example is shown in Figure 3.6.

Rule 1: We require that there is at least in one feature of the ConcreteDecorator a call to the same feature in the Component class.

As mentioned in the pattern description, the Decorator can be seen as a degenerate Composite with only one component. As a result, there must be exactly one reference to a Component object in the Decorator class that will be used to forward additional requests for extended functionality.

Rule 2: We require that there is exactly one attribute of type Component in the Decorator class.

3.2.5 Facade

The main objective of the Facade pattern is to hide information from the client. As a result, the client communicates with the subsystem by sending requests to the Facade, which forwards them to the appropriate subsystem object(s).

It might be possible to improve Wang’s approach by adopting a new one based
on graph theory (all paths going through one node). However, in the scope of our work, the detection can mostly be done at the class-level. No fine-grained information can help in achieving better results in the detection process.

### 3.2.6 Flyweight

Flyweight is used to improve the general performance of software that uses a great number of similar objects. When the Flyweight pattern is used, we avoid creating again and again the same object. Some existing objects can be reused with only some of their parameters changed when called.

There must be some data structure (e.g. an array) that stores the existing objects. If an object is requested the array is checked first. If it does not exist, the requested object is created and entered in the array. Such information can be used to improve the detection results by removing false positives.

Since the FlyweightFactory class needs to keep track of the objects it creates, these objects will be stored in a collection of Flyweight objects.

**Rule 1:** We require the presence of a collection of objects of type Flyweight in the FlyweightFactory class.

The Flyweight objects are created exclusively by the FlyweightFactory class. Such information can be used in the detection process.

**Rule 2:** We require the presence of creation calls in the FlyweightFactory class that leads to the construction of the new concrete Flyweight products.
Like for the Factory Method pattern, we should be able to get back a new product of type Flyweight from the FactoryFlyweight class.

**Rule 3:** We require that some feature of the FlyweightFactory class returns an object of type Flyweight.

When creating a new Flyweight object, it is often important to make sure that only one such instance exists at runtime so that every class using the same type of object refer to that same unique one. This will ensure that no two identical objects are created since the goal of the Flyweight pattern is to reuse already created objects. However, this rule is not required to implement the Flyweight pattern and thus becomes optional.

**Rule 4 (Optional):** The Flyweight class may be a Singleton.

### 3.2.7 Proxy

A proxy is generally used when a Client needs more convenient ways of accessing a single Subject. The Proxy forwards requests to the RealSubject when it is needed or appropriate.

We note that the same methods are used and called in the Proxy as in the RealSubject. For example, if the RealSubject class has a feature `draw`, there will be the same method in the Proxy. Such relevant detail can come in handy during the final phase of the detection process when removing false positives.
Also, the Proxy acting as a substitute to the RealSubject when requests are issued, it is possible that it does not have the proper reply to a certain request. In such cases, it will call features of the RealSubject to retrieve information that is needed to answer the request. We can set a certain threshold (for example 50%) of the RealSubject’s methods that are called by the Proxy class.

**Rule 1:** We require that at least 50% of the RealSubject’s methods are called by the Proxy class.

As an extension to the previous rule, the Proxy might have to create the RealSubject first before using it in cases where it does not have a direct answer to a certain request. However, this rule does not always apply depending on the implementation of the pattern since the RealSubject might already have been instantiated at startup. Thus, some Proxy implementations would simply focus on feature calls to the RealSubject and not contain creation methods to instantiate objects of RealSubject.

**Rule 2 (Optional):** The Proxy may create the RealSubject.

### 3.3 Behavioral Patterns

#### 3.3.1 Chain of Responsibility

The Chain of Responsibility pattern is helpful when propagating a request along a list of successors and giving each one of them the possibility to handle the request.
This pattern can be compared to the Composite pattern where the child of a specific component could act as its successor. The difference lies in the fact that the request does not necessary reach the leaf nodes since it could be handled prematurely by any element in the chain.

As in the Composite pattern, the Component class has some method that is also present in its subclasses to handle the request and process it or pass it to the successor (commonly called handle - Figure 3.7). That information can be useful to detect the pattern more accurately.

The Handler needs access to all the handlers for any request. Therefore, there needs to be one field of type Handler in the Handler class.
Rule 1: We require the presence of exactly one attribute of type Handler in the Handler class.

If a particular request cannot be handled by a Handler, it is forwarded to the next Handler in the list. There should be in the Handler class a feature that calls another feature of the same name in the successor class.

Rule 2: We require that there is in the Handler class a feature that calls another feature of the same name in the successor.

3.3.2 Command

This pattern is commonly used in software that requires access to the history of operations during its utilization. Requests to "Undo" or "Redo" commands can easily be manipulated through the Command pattern. The Command pattern encapsulates operations as objects.

Since the ConcreteCommand object encapsulates an operation on a Receiver object, that Receiver needs to be specified in the class as an attribute.

Rule 1: We require that the ConcreteCommand class contains an attribute of type Receiver.

The Receiver needs to be passed as an argument to the ConcreteCommand so that the operations can be made on it when the ConcreteCommand is invoked.

Rule 2: We require that at least one feature in the ConcreteCommand gets as argument an object of type Receiver (the constructor or simply a setReceiver feature).
### 3.3.3 Interpreter

It should be noted that the use of this pattern is rare and deeply depends on specific software intents.

Given a language, this pattern defines a representation for its grammar, along with an interpreter that uses the representation to interpret sentences in the language. The Client first builds (or is given) the sentence as an abstract syntax tree of NonterminalExpression and TerminalExpression instances. Then the Client initializes the Context and invokes the `interpret` operation. Each NonTerminalExpression node defines `interpret` in terms of `interpret` on each subexpression. The `interpret` operation of each TerminalExpression defines the base case in the recursion. The `interpret` operations at each node use the Context to store and access the state of the Interpreter.

Any potential NonTerminal class needs to contain a specific feature that calls a feature of the same name on a different object.

**Rule 1:** We require that a feature in the Nonterminal class calls a feature of the same name on another object.

Since the Context contains all the necessary information about the Interpreter's state, it is to be passed as an argument in features of the Nonterminal and Terminal classes when they call other features of the AbstractExpression class.

**Rule 2:** We require that one of the arguments passed when calling features of the AbstractExpression are of type Context.
3.3.4 Iterator

The Iterator is in charge of remembering the current index of an iteration through a collection of objects. It is therefore capable of giving out the succeeding or preceding object in the traversal through specific methods.

Once again, generic parameters are used (the same type of object parameterizes the two classes Iterator and Collection). While the Iterator keeps track of the current object in the aggregate, it needs to be generic so that the Client can browse that type of object through the Iterator. Relying on such information could help in detecting the pattern.

**Rule 1:** We require that the ConcreteIterator be generic.

A collection of objects is used for this pattern. It also needs to have a generic parameter since, as mentioned before, the Iterator class has the same generic parameter than the Collection it is using.

**Rule 2:** We require that the ConcreteIterator contains a Collection with a generic parameter of the same type as the one in the ConcreteIterator.

3.3.5 Mediator

The Mediator’s role is to reduce dependencies between classes. For example, two colleague classes may use the Mediator but not use each other. The mediator will then pass the requests from one colleague to another.
The Mediator acts as a link between colleagues that don’t need to know about each other. Therefore, we can make sure that there are no relationships amongst the colleagues within a certain threshold (depending on the system, a few connections might still be acceptable).

Rule 1: We require that there are no relationships between colleagues within a certain threshold.

Since Colleagues normally pass themselves as argument to the mediator, we could detect such calls to the Mediator with the keyword `Current` (similar to `this` in Java) in most cases.

Rule 2 (Optional): The Colleagues may pass themselves as argument to the Mediator using the keyword `Current`.

### 3.3.6 Memento

A Memento is used to remember a specific state of an object. A Caretaker first asks an Originator to create a Memento, which will record the current state of the Originator. If the Originator needs to revert to an old state, the Caretaker will request from the Originator to restore the object’s state saved in the Memento. The Originator will then retrieve the corresponding state from the Memento.

The methods of the Memento class need only be exported to the Originator. Also, there are two methods that must exist in a potential Originator class to point out: one that creates a Memento object, and another one that takes a Memento
as argument (when setting the memento). Both could be detected to improve our results.

Since the Caretaker decides when to restore a certain Memento, it must call a feature from the Originator that takes as argument the corresponding Memento.

| Rule 1: We require that at least one feature of the Originator class takes a Memento object as argument. |

The Originator is the one that will create the Memento when ordered by the Caretaker.

| Rule 2: We require that an object of type Memento is created within one of the Originator’s features. |

Although it is not a required feature, it is often standard practice to restrict access to the Memento to the Originator class only. This ensures that the handling of Mementos is done solely by the Originator.

| Rule 3 (Optional): There is restrictive access to the Originator class in the Memento class. |

3.3.7 Observer

Whenever a change occurs to the ConcreteSubject, it notifies its Observers, that may then retrieve new information from the subject and process it accordingly.

Since the Subject needs to keep track of the observers attached to it, there should be a collection of Observers in the Subject class.
Rule 1: We require the presence of an attribute being a collection of Observers in the ConcreteSubject (e.g. observers: COLLECTION[OBSERVER]).

The "attachment" of the Observers to the Subject is a required step in the proper functioning of the Observer pattern. Therefore, there should be at least one method in the Subject that receives as an argument an object of type Observer.

Rule 2: We require that at least one feature in the Subject receives an argument of type Observer.

3.3.8 State

The State pattern is used to alter the behavior of a Context object based on changes that occur during its lifetime. Implementations of state machines typically use the State pattern.

The State object is created within the Context class since the Context is the primary interface for clients. The Context is then initially used to create State objects and deal with them.

Rule 1 (Optional): The State object may be created in the potential Context class.

In some variants, the Context would pass itself as an argument (using the Current keyword) to the ConcreteState so that the Context can be accessed if necessary.

Rule 2 (Optional): At least one feature in the ConcreteState receives an argument of type Context.
Features of the State class should only be exported to the Context, since no other class should have an interest in them.

**Rule 3 (Optional):** Features in the State class are exported only to the Context class.

Also, in most variants, the ConcreteStates are partly responsible on what should be the next state. As such, there should be at least one feature that returns a State object in the State class. This indicates the next state of the implemented state machine.

**Rule 4 (Optional):** The return type of at least one feature in the State class is of type State.

### 3.3.9 Strategy

Some common and yet famous uses of the Strategy pattern can be found in the Java language with the different possible layouts one can specify for a GUI application (Flow Layout, Border Layout, Box Layout, etc..). The Strategy pattern is very similar to the State pattern structure-wise. They both contain a Context class and they both have a main State/Strategy class with deferred features and many Concrete State/Strategies classes inheriting from it.

A major difference between the State and Strategy patterns is the location of the creation of the State or Strategy objects. The Strategy object will most likely be created by the Client, whereas the State object is created by the Context itself.

53
Rule 1 (Optional): The Strategy object may not be created in the potential Context class.

3.3.10 Template Method

This pattern defines the skeleton of an algorithm in an operation, deferring some steps to subclasses to be redefined without changing the general structure of the algorithm which is defined in the AbstractClass.

A feature that calls another deferred feature from the same class follows the Template Method pattern. For example, if feature A calls another feature B in the same class, and if that feature B is actually deferred, that is sufficient to deduce the Template Method pattern.

Rule 1: We require that at least one non-deferred feature calls at least one of the deferred features in the same class.

3.3.11 Visitor

This pattern has a very specific application context. We use it when we have a fixed number of types of elements to work with; usually organized in a Composite structure. The Client must first create a ConcreteVisitor object, then traverse the object structure visiting each element with the visitor. When an element is visited, it calls the Visitor operation that corresponds to its class. The element passes itself as an argument to the Visitor which can then freely access it if needed.
As such, the ConcreteElement will receive the ConcreteVisitor as argument in its "accept" method or a differently named but similar type of method.

**Rule 1:** We require that the implementation of `accept` must call exactly one method from the Visitor interface passing `Current` as the argument.

Statically, we know that there is a one-to-one mapping between function calls and element types in such a way that the ConcreteElement calls its corresponding "visit" feature (e.g. ElementA calls `visit(ElementA)`). If there are more calls from one class (defeating the one-to-one mapping rule), we can assume that it is not a Visitor pattern and thus remove false positives.

Therefore, it is necessary to make sure that the one-to-one mapping principle is respected. For each ConcreteElement class, there needs to be exactly one method in the Visitor interface that takes this ConcreteElement as an argument.

**Rule 2:** We require that for each of the ConcreteElements there is one and exactly one method in the ConcreteVisitor that takes this ConcreteElement as an argument.
4 The FiG tool

The tool presented in this chapter is an extension of the Design Pattern Verification toolKit (DPVK) tool made by Wang [26], and works as an integrated text-search tool that detects design patterns in Eiffel systems. We begin by describing DPVK. The remainder of the chapter presents the improvements and extensions that constitute the FiG tool, such as pattern detection using fine-grained information and greater flexibility at the user interface level.

4.1 DPVK

The detection of Design Patterns can be done based on two different types of information: static and dynamic. In the first case, the detection is based on basic relationships between classes, such as inheritance or client-supplier relationships. For the latter, the detection looks precisely at the dynamic behaviour of the system such as method calls. Therefore, each pattern needs to have two rigorous definitions that will be used to accurately identify them. That is the main objective of DPVK.
as conceived by Wang. It compares the two definitions with static and dynamic facts about the target software system, and outputs potential candidate instances.

DPVK runs on three stages which are respectively the *static and dynamic fact extraction*, *candidate instance recovery* and *false positive elimination* [26]. A global overview of DPVK’s structure can be observed in Figure 4.1. The final structure of FiG is shown at the end of this section (in Figures 4.22, 4.23 and 4.24).

Let us take the detection of the Builder pattern as a running example in a sample system taken from [16] that implements the pattern.
The Builder pattern decouples the construction of an object from its representation so that the same construction process can create various representations. Figure 4.2 represents the UML diagram of the Builder pattern as described in GoF [14].

The mapping between the classes defined in GoF for the Builder pattern and the classes in the implementation is shown in Table 4.1.
4.1.1 Stage 1: Static and dynamic fact extraction

For the first stage, DPVK only requires two inputs for simplicity:

1. The description files for all classes in the target system (which can be obtained by running the Eiffel Compiler with the proper parameters). These are normally in a subfolder of the root directory of the Eiffel implementation called "Documentation".

2. The ACE file (Assembly of Classes in Eiffel) defining the Eiffel system to be built\(^1\). This file contains all the information required to analyze the system, such as the root cluster’s location, or the different options and libraries required at compile time.

DPVK retrieves all the .e files (Eiffel source code files) that are used by the system and reads the Documentation folder to output a list of static information in RSF format such as uses client supplier and inherits descendant ancestor.

The final output of this stage is represented in Figure 4.3.

To obtain dynamic facts, the system needs to run in a way that exercises all of the pattern’s features. To get and log the different calls between classes at runtime, it is possible to specify in the Ace file to turn on the tracing capability provided by EiffelStudio (with \texttt{trace(on)}). Then, running the compiled program

\(^1\)Eiffel version 5.6 was used for the work in this thesis
Figure 4.3: Static facts of the sample implementation

under EiffelStudio will automatically generate all the dynamic facts related to the runtime behavior of the program. Once this log is obtained, DPVK reads it, and converts it to a better and more useful formatting structure as presented in Figure 4.4.

It is important to note that the fact extraction modules are language-dependent in the sense that the parsers are expecting as input language specific documentation files or dynamic traces, as is the case for Eiffel.
calls game_with_builder game_with_builder
calls game_with_builder standard_maze_builder
calls standard_maze_builder maze
calls game_with_builder standard_maze_builder
calls standard_maze_builder maze
calls standard_maze_builder maze
calls standard_maze_builder room
calls standard_maze_builder maze
calls game_with_builder standard_maze_builder
calls standard_maze_builder maze
calls standard_maze_builder maze
calls standard_maze_builder room
calls standard_maze_builder maze
calls game_with_builder standard_maze_builder
calls standard_maze_builder maze
calls standard_maze_builder maze
calls standard_maze_builder room
calls standard_maze_builder maze
calls game_with_builder standard_maze_builder
calls standard_maze_builder maze
calls standard_maze_builder maze
calls standard_maze_builder room
calls standard_maze_builder maze
calls game_with_builder standard_maze_builder
calls standard_maze_builder maze
calls standard_maze_builder room
calls standard_maze_builder maze
calls game_with_builder standard_maze_builder
calls standard_maze_builder maze
calls standard_maze_builder maze
calls standard_maze_builder room
calls standard_maze_builder maze
calls game_with_builder standard_maze_builder
calls standard_maze_builder maze
calls standard_maze_builder maze

calls maze maze
calls maze room
calls maze wall
calls maze room
calls maze wall
calls maze door

calls game_with_builder game_with_builder
calls game_with_builder standard_maze_builder
calls standard_maze_builder maze
calls game_with_builder standard_maze_builder
calls standard_maze_builder maze
calls game_with_builder standard_maze_builder
calls standard_maze_builder maze
calls standard_maze_builder maze
calls standard_maze_builder room
calls standard_maze_builder maze
calls game_with_builder standard_maze_builder
calls standard_maze_builder maze
calls standard_maze_builder maze
calls standard_maze_builder room
calls standard_maze_builder maze
calls game_with_builder standard_maze_builder
calls standard_maze_builder maze
calls standard_maze_builder maze
calls standard_maze_builder room
calls standard_maze_builder maze
calls game_with_builder standard_maze_builder
calls standard_maze_builder maze
calls standard_maze_builder maze
calls standard_maze_builder room
calls standard_maze_builder maze
calls game_with_builder standard_maze_builder
calls standard_maze_builder maze
calls standard_maze_builder maze
calls standard_maze_builder room
calls standard_maze_builder maze

calls maze maze
calls maze door
calls maze room
calls game_with_builder maze
calls maze room
calls maze wall
calls maze door

Figure 4.4: Dynamic facts of the sample implementation
4.1.2 Stage 2: Candidate instance recovery

Now that all static facts about the system were retrieved, the next step will be to try and match those facts with the pattern’s definition. The definition is written into an REQL script and defines the inheritance and client-supplier relationships among classes participating in the design pattern. Figure 4.5 shows the static definition of the Builder pattern. The commented lines (starting with 
//”) give a global overview of the different roles involved in the definition. The rule itself uses the roles define in DP with concrete relationships such as uses and ancestor which is the inverse of inherits as shows the line ancestor=inv inherits. The rules to remember are then: director uses builder, concreteBuilder inherits builder and concreteBuilder uses product.

REQL then compares the static definition with the static facts deduced earlier.
in stage 1. At the end of this stage, we obtain the following output in Figure 4.6.

This instance discovery phase is completely language-independent since static facts are represented in a generic RSF format and pattern matching is done using the generic static definition and the list of different possible combinations of classes.

After this stage, we can already observe some false positives such as line 3, 5 or 6 which do not apply to the definition of the Builder pattern as shown in Table 4.1. The next stage will therefore help remove false positives from the output of this stage.

4.1.3 Stage 3: False positive elimination

Figure 4.7 shows the dynamic definition of the Builder pattern.

From the dynamic facts obtained previously in Stage 1 (Figure 4.4) and the output file from Stage 2, DPVK will do the matching once again using the dynamic definition provided for the Builder pattern. The result from this stage is given in

```
roles number=4
director builder concreteBuilder product
game_with_builder maze_builder standard_maze_builder door
game_with_builder maze_builder standard_maze_builder maze
game_with_builder maze_builder standard_maze_builder room
game_with_builder maze_builder standard_maze_builder wall
```
Figure 4.7: Dynamic definition of the Builder Pattern

<table>
<thead>
<tr>
<th>roles#</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-</td>
<td>director</td>
</tr>
<tr>
<td>2-</td>
<td>concreteBuilder</td>
</tr>
<tr>
<td>3-</td>
<td>product</td>
</tr>
<tr>
<td>rules#</td>
<td>3</td>
</tr>
<tr>
<td>1-</td>
<td>* calls director=⇒</td>
</tr>
<tr>
<td>2-</td>
<td>director calls concreteBuilder</td>
</tr>
<tr>
<td>3-</td>
<td>concreteBuilder calls product</td>
</tr>
</tbody>
</table>

Figure 4.8: Final output of Stage 3

| 2 Design Pattern instances are found. |
| director builder concreteBuilder product |
| game_with_builder maze_builder standard_maze_builder maze |
| game_with_builder maze_builder standard_maze_builder room |

Similarly to the previous phase, line 2 enumerates the different roles involved in the pattern detection. As can be seen, 2 of the false positives we mentioned earlier were removed at this stage because door and wall were not called by the ConcreteBuilder standard_maze_builder as it should have been the case, according to the definition, if they were real Products.

Since some patterns have rather complex structures or can have many different variants, there is a small possibility for the tool to miss some true positives or detect false positives. This is the reason why a developer’s eye is helpful at the end to
analyze the precision of the results. From our knowledge of the system, line 4 from Figure 4.8 is not a true instance of the Builder pattern since the ROOM class is not the real Product in this case. MAZE is. ROOM is a sub-element of MAZE which can be constructed to define the concrete maze.

Just like in stage 2, this stage is also fully language-independent since facts and definitions once again follow a generic format.

4.2 FiG: An improvement and extension of DPVK

FiG stands for Fine-Grained Analysis Tool. Extending DPVK, the interface has been completely converted to a flexible command line system. There is no GUI anymore (DPVK was a GUI-only tool), since it usually imposes many restrictions to the users such as not being able to run it on any SSH shells for example or not being able to run FiG within a bash script for automated batch detection of patterns in several systems.

4.2.1 Command line interface and order of phases

The command line system implies many default parameters such as the default output directory, and can run with at least 2 inputs making it much more practical and faster. Many new options can now be used to refine the detection process according to the user’s needs. The most important feature is the ability to tell FiG
in which order to execute the phases. FiG’s command line’s options are thoroughly
described in Appendix B.

To avoid rewriting the same list of commands for particular testings on a given
system, FiG can read what would be called an ”information file” which contains
a list of parameters and their respective values to consider for the detection pro-
cess. FiG only needs this information file as well as the pattern being detected
(mandatory) to run. Other command line options provided as extra will override
the corresponding parameters in the information file. Figure 4.9 shows one example
of information file and how to run FiG using it.

4.2.2 Standardization of input and output formats

One major improvement brought by FiG is the standardization of the input and
output formats (definitions, facts and results files). As we could see in the above
Figure 4.10: Static definition of the Command Pattern

presentation of DPVK, each type of files has a unique header or format. Indeed, the static definition was a low-level REQL script [28]. Figure 4.10 shows the REQL static definition of the Command pattern.

From a reader’s point of view, such a script can prove to be hard to understand at first sight due to the unfriendly syntax barrier. The same could be said about the dynamic definition files as shown in Figure 4.11. For example, they would contain the number of roles as the first line and then the enumeration of all those roles on the following lines. The fact that the number of roles may not match in both static and dynamic definitions (as shown in Figure 4.10 and 4.11) makes DPVK’s rules parsing system even more complex. The outputs of both phases also had unique characteristics such as the dynamic phase output containing a one-line summary of the detection process. Figure 4.12 and 4.13 show typical outputs of both static
and dynamic analyses from DPVK. The second line in both outputs enumerate the different roles involved in the pattern matching.

Such definitions and formattings can be hard to understand for the user. Also, due to the uniqueness of each type of files, the different tool modules are expecting specific types of inputs. As such, it is impossible to reorder the phases of execution for that matter (i.e. The dynamic phase cannot be done before the class-level static analysis phase since this module is not expecting outputs similar to those given after
Figure 4.13: Typical output of the dynamic analysis phase

In FiG, both definition and output files have been standardized and simplified to allow one to be able to grasp the meaning of the rules and results easily. Standardizing outputs was a required step to grant better flexibility to the tool such as being able to feed any output of a phase to another phase following a standard format convention and thus standardizing the processing of the input files (i.e. no unique cases to consider or exceptions to handle). Such flexibility allows FiG to do phases in any order provided by the user (as long as the required files are also provided as input, such as candidate instances or dynamic facts, etc...). Figure 4.14 and 4.15 show respectively FiG’s static and dynamic input formats. The usage of simple keywords, that specifies each of the rules, improves their understanding. Figure 4.16 and 4.17 display the outputs of both static and dynamic phases from FiG.

The different possible orders of phases make FiG a flexible tool which can even be used to detect design patterns in other systems such as Java or C/C++/C#.
invoker uses command
concreteCommand inherits command
concreteCommand uses receiver

Figure 4.14: Static definition of the Command pattern in FiG

invoker calls concreteCommand
concreteCommand calls receiver

Figure 4.15: Dynamic definition of the Command pattern in FiG

(although some definitions might need to be changed for an accurate replication of the pattern’s implementation in the given language).

As an illustration of this possibility, FiG was used by David H Kelk at York University to detect design patterns in C# systems. He first extracted static and dynamic facts using specific tools and IDEs and then run FiG to detect most of the GoF patterns. The results were quite satisfying as he was able to retrieve all the true positives he was able to locate manually. The target system was indeed small enough to be able to try and detect patterns without detection tools.

4.2.3 The fine-grained analysis phase

The fine-grained analysis phase aims at identifying true positives while discarding false ones. As such, if no candidate instances are provided as input, the class-level
invoker command concreteCommand receiver
----------------------------------------
button command macro_command command
button command quit_command text_editor
button command quit_command text_editor
macro_command command macro_command command
macro_command command quit_command text_editor
macro_command command save_command text_editor

Figure 4.16: Typical output of the static analysis phase in FiG

invoker command concreteCommand receiver
----------------------------------------
button command quit_command text_editor
button command save_command text_editor
macro_command command quit_command text_editor
macro_command command quit_command text_editor

Figure 4.17: Typical output of the dynamic analysis phase in FiG
static analysis phase is mandatory: the dynamic or fine-grained analysis phases are both used to remove false positives from a defined set of candidate instances. They cannot be used as of now to discover potential candidate instances. It is the role of the class-level static analysis phase to detect those if need be. However, if a set of candidate instances is provided as input, the user may run the fine-grained analysis phase first if he/she wishes to.

Since the fine-grained analysis phase requires deep interaction with the target system’s source code, an Eiffel parser had to be set up then used during that phase (presented in Section 2.4.1). This parser is part of an external Eiffel executable module that FiG will use to parse the target Eiffel source files and determine which of the candidate instances fit the fine-grained rules presented in the previous chapter.

Since the core of our tool is written in Java, it will have to call the compiled Eiffel "fine-grained" part of the tool with specific parameters which are:

- The source directory - This contains all the files of the target system being analyzed.

- The output file - This is the file where the results from the fine-grained analysis phase are outputted.

- The input file - This is the file containing the candidate instances that will
Figure 4.18: The fine-grained definition of the Builder pattern

be matched against the fine-grained definition.

• The definition file - The definition file contains the different fine-grained rules for the pattern being detected. There is no limit to the number of rules. They will be processed in the order of appearance in the definition file.

4.2.3.1 Parsing and detection process

The first step is to parse all target system files, and store their contents in a data structure. Afterwards, the different roles and rules stated in the definition files are read. In the case of the Builder pattern, we would have the file in Figure 4.18.

The first line enumerates in an ordered manner the different involved roles in the definition. The next line is the only fine-grained rule for this pattern. The rules are independent and the first keyword used to represent a rule tries to be as intuitive as possible. The other parameters following that first keyword are roles that participate in the rule. In some cases, it is also possible to have an integer parameter that defines a threshold for example.

Next, the input file is read line by line and the different rules retrieved earlier
are processed for each candidate instance as it is being read. A special ”result” data structure holds for each of the candidate instances being read a boolean value of true (if all the rules applied perfectly) or false (if the candidate failed to comply to at least one of the rules).

After the whole detection process, only the candidate instances that were flagged as true in the previous step are outputed to the result file. The ones flagged as false are considered false positives and are discarded.

4.2.3.2 How does the rules matching work?

Each rule is represented by its specific keyword in the definition file as shown previously in Figure 4.18. They all inherit the same deferred class RULE and have their custom implementation of the enforce feature (deferred in RULE) that is called during the detection process when ”enforcing” each of the rules for the aimed pattern on the candidate instances.

These implementations of enforce make use of the eParser library to retrieve fine-grained information such as method signatures or return type of features. If the corresponding roles contain in their code the expected static information that constitute the rules then the method will return true, otherwise, it will be false.

In the case of the Builder pattern that was being analyzed previously as an illustration to the detection process, Figure 4.19 shows the corresponding output
file for that phase.

```
director builder concreteBuilder product
-----------------------------

game_with_builder maze_builder standard_maze_builder maze
```

Figure 4.19: The output of the fine-grained analysis phase for the Builder pattern

The fine-grained rule for the Builder pattern applies to the Builder and Product classes. According to the definition, we have a Builder pattern if in the Builder class, there is at least one feature (or attribute) that returns an object of type Product. Amongst the two remaining instances obtained after the dynamic analysis phase, only the one with `maze` as Product complied to the rule. The first one (with `room` as Product) did not meet the requirements of the rule and was thus discarded at this stage. This can be explained by `room` being one sub-element of the real final Product `maze`. As such, the Builder class only needs to return the `maze` Product at the end of the building process.

The fine-grained phase functions in several steps which include parsing the source code and processing the rules in a specific manner. Therefore, this stage is entirely language-dependent since both parsing of the code and processing of the rules works for the Eiffel language only in this case.
4.2.4 The post-processing analysis phase

For some (very few) patterns, it is important to make sure that the instances found are coherent with each other. For instance, as part of the definition of the Mediator pattern (Section 3.3.5), it is necessary that ConcreteColleagues do not communicate between themselves but forward all of their requests through the Mediator instead. As such, we need to compare each potential ConcreteColleagues found at the end of all of the other phases and make sure they do not share a relationship (such as client-supplier relationship) from the static facts. All the phases before this one would only consider one candidate instance at a time. The post-processing phase is the only one to take into account several candidate instances at the same time to filter out false positives according to a post-processing definition. However, most patterns will not have such a definition and as a result, this phase will be bypassed automatically for them. This happens whenever the post-processing definition file for a pattern is empty.

To illustrate the importance of the post-processing phase for the Mediator pattern, we consider the sample implementation given in [16]. Figure 4.20 presents the candidate instances obtained after the static, dynamic and fine-grained analysis respectively. As we can see, the last instance `main dialog_director font_dialog_director` does not seem to resemble the other instances in their structure. Basically, to have
a Mediator pattern we need at least two ConcreteColleagues. Otherwise, the potential Mediator and Colleague classes would simply share a client-supplier relationship. Since we only have here one instance with main as the ConcreteMediator, it becomes inherent that the last candidate should not be part of the Mediator pattern.

```plaintext
concreteMediator colleague concreteColleague
--------------------------------------------
font_dialog_director widget entry_field
font_dialog_director widget list_box
font_dialog_director widget window
main dialog_director font_dialog_director
```

Figure 4.20: Results obtained for the Mediator pattern after the fine-grained analysis phase before doing the post-processing phase.

As for the remaining three candidates, we need to ensure that no relationships (e.g. client-supplier relationships) exist between potential ConcreteColleagues. Therefore, using static facts from the system, we should be able to deduce whether or not entry_field, list_box and window interact at least once amongst each other. If no relationships exist between two candidates, they fulfill the necessary requirements to forming a Mediator pattern.

The post-processing phase for the Mediator pattern thus applies those two rules and remove false positives (failing to comply to at least one of the rules). Figure 4.21 shows the results obtained after that post-processing phase. Since the 3 re-
remaining candidate instances mentioned earlier are still present, it is clear that no relationships were found between `entry_field`, `list_box` and/or `window`.

```
concreteMediator colleague concreteColleague
-----------------------------
font_dialog_director widget entry_field
font_dialog_director widget list_box
font_dialog_director widget window
```

Figure 4.21: Results obtained for the Mediator pattern after the fine-grained analysis phase after doing the post-processing phase.

As described above, this phase deals with static facts and candidate instances in their generic forms. As such, this phase is language-independent.

### 4.2.5 An overview of the contributions in numbers

The Java part of the tool is composed of 235 classes in total among which 17 are the core part of FiG and the other 218 are from the REQL package used by FiG. The 17 classes have in total 3140 lines of code and the REQL extension contains 28768 lines.

The library of pattern definitions contains 64 files containing 219 lines of code.

The Eiffel part of the tool contains 35 classes dedicated to the fine-grained module of FiG while the ES Parser cluster uses 23 classes and Gobo 205 classes. The 35 classes contain in total 4194 lines of code, the ES Parser cluster 19271 lines and the Gobo Library 28971 lines.
All the Java classes altogether represent a grand total of 31908 lines of code and for the Eiffel part we have for all classes 52436 lines of code. As such, if we consider both Java and Eiffel sides of the tool, FiG uses in total 498 classes which is the equivalent of 84344 lines of code. Excluding the packages and clusters used, there are in total 7334 lines of code made from scratch with about 1/5 of them modified from the base implementation of DPVK.

4.2.6 FiG diagrams

Figures 4.22, 4.23 and 4.24 presents diagrams of the different phases of FiG while Figure 4.25 shows the different possible order of phases that FiG can handle provided that the necessary requirements are met.
Figure 4.22: Static Phase
Figure 4.23: Dynamic Phase
Figure 4.24: Fine-Grained Phase
Figure 4.25: Possible order of phases

Legend:
- S: Static Analysis Phase
- D: Dynamic Analysis Phase
- F: Fine-grained Analysis Phase
- : Input Document
5 Experimental Case Studies

5.1 Case Study 1: Sample implementations

In [16], all GoF patterns have been implemented in Eiffel. We tested our tool on all of these implementations and tried to detect in each one of them the different patterns. Figure 5.1 compares the results obtained for this experiment after doing the class-level static and dynamic analysis (i.e. DPVK’s results) with those obtained after also performing the fine-grained analysis phase (i.e. FiG’s results). In total, 966 tests were conducted.

In Figure 5.1, if no pattern instances are found, the corresponding cell is left empty. If the pattern is found in the target implementation, the number of true and/or false positives is shown in the cell. Instances that are found to be false positives are displayed in the cell on a dark gray background while those that are found to be true positives are represented on a clear background. If both true and false positives are found, the corresponding cell is split and the left side shows the

2The Prototype and Facade patterns were not tested for the reasons explained in Chapter 3.
Figure 5.1: Table of results for Case Study 1 with both results from DPVK (top) and FiG (bottom). Dark cells represents the number of false positives found, normal cells with numbers the number of true positives and gray cells the results for the pattern being tested on its corresponding implementation.
number of true positives (on a clear background) and the right side the number of false positives (on a dark gray background).

The cells with light gray background represent experiments where the pattern is tested on its corresponding implementation. Therefore, true positives are most likely to appear for such cases.

The duration for each of these tests did not exceed 10 seconds due to the small size of the implementations. Also, manual verification of the results in most cases took 10 minutes maximum depending on the complexity of the target implementation and the pattern being detected.

From the detection results, we made the following observations:

1. Although some false positives were detected, there were no false negatives. This shows that our approach did not miss any true instances of the patterns in the detection process.

2. For some of the example implementations (such as Abstract Factory or Proxy), many false positives were observed at the class-level analysis. This is due to the complexity of these implementations. However, the fine-grained part of the detection process eliminated most of those false positives. The results are therefore much less affected by the complexity of the targeted system when the detection process includes the fine-grained analysis phase.
3. We can observe that we usually have the same number of instances for the Abstract Factory and Factory Method patterns (first and third lines in Table 5.1). Since an Abstract Factory provides an abstract class with at least two or more factory methods, the tool will detect each factory method implementing that interface separately. This is due to the way the tool processes the data. It will process candidate by candidate but will not be counting the number of candidates that should be merged into one unique pattern instance for example. This limitation might be addressed as post-processing future work for our research which will aim at detecting design patterns by considering cluster of candidates instead of single candidates. Note that this is different from the post-processing phase we introduced in Chapter 4 which is only used to remove false positives, not deduce a pattern instance from a number of candidates instances.

4. The State and Strategy pattern lead to the most amount of false positives at the fine-grained level. They both have the same class-level static structure and their dynamical behaviors are very similar. The main difference is that concrete states are created in the Context class itself while it is not the case for the concrete strategies. The fine-grained definitions of both patterns therefore play on this major difference. That explains the complementarity in the results for the two patterns. If 5 candidate instances are found after the
static analysis, then if two of them are potential State instances, the other 3 are automatically possible Strategy instances since the detection is based on the presence of creational features in the potential Context class.

From a list of 3302924 possible random class combinations for the total number of implementations, only 0.01816% were selected by the static analysis phase (candidate instance recovery phase) and suggested as possible pattern instances. This represents a number of 3302 instances in total. Afterwards, the dynamic and fine-grained analysis phases acting as false positive filtering layers were executed to refine these results.

The dynamic phase deemed 544 instances from the 3302 instances discovered by the static phase as correct. From those 544 instances, 475 were false positives. Therefore, the dynamic analysis phase was able to remove 83.52% of the false positives from the results of the static analysis phase (2758 instances on 3302 were removed).

The fine-grained static analysis phase showed great improvements to the normal class-level static and dynamic analysis. A great number of false positives were removed. For 7 of the patterns out of a total of 475 false positives found following the static and dynamic analysis phases, 449 were removed by the fine-grained analysis phase. This represents a filtering rate of 94.53%.

Both dynamic and fine-grained analysis removed in total 97.12% of the false
positives from the initial set of 3302 instances discovered.

5.1.1 Comparison with PDE

Since this case study is based on detecting GoF patterns on standard implementations of those same GoF patterns, it would be interesting to compare the overall effectiveness of FiG with other fine-grained tools following the same experiment even if the target language for those tools was not Eiffel. Unfortunately, there are no case studies available that represent the efficiency of SPQR or FUJABA on detecting each of the GoF pattern on each of their implementations.

It is also not easy to compare with other tools because:

- Not many tools implement a fine-grained detection level phase.

- Some tools work exclusively at the language level: the detection method for each pattern is hard-coded. This makes comparing definitions and variants used difficult.

- It is also possible for some pattern definitions to vary depending on its implementation language (often due to a language’s unique features).

However, a recently published thesis [9] presents a tool called PDE - which stands for Pattern Detection Engine. PDE is able to work at the fine-grained level but following a dynamic approach only. The Dynamic Analysis phase was refined
so that it can deal with more runtime information to fine-tune the detection results. Therefore, we could consider both approaches similar in the sense that both of our tools contain a phase that work with fine-grained information to filter out false positives.

A similar case study to the one we did in Case Study 1 (by using standard implementations of the GoF patterns as given in the Applied Java Pattern book [25]) was performed and compared to two other tools: PINOT and FUJABA. PDE gave better results in general than FUJABA. As such, we can compare the results obtained by FiG with that of PDE instead of FUJABA. Figure 5.2 shows the table of results obtained with PDE for this Case Study.

The following observations can be made from the comparison of the results:

- PDE can detect Prototype patterns. The reason behind this is due to the pattern being implemented differently in Java.

- We can see that PDE removed many more false positives than FiG after the static and dynamic analysis. However, the fine-grained results of FiG still led to better results with less false positives.
Figure 5.2: Table of results obtained with PDE for Case Study 1. Dark cells represent the number of false positives found. Gray cells showcase the results for the pattern being tested on its corresponding implementation.
5.2 Case Study 2: Patterns in FiG’s parsing system

For this case study, we wanted to try and detect design patterns in a big Eiffel system to test the overall speed and efficiency of FiG. Since FiG is partly using the eParser library to do the parsing of the Eiffel source files, our tool itself (the Eiffel side of FiG) was a perfect target system for our analysis. FiG, embedding the eParser cluster which itself contains other important libraries such as gobo, has a grand total of 240 classes.

To verify the integrity of our tool’s detection system, we first analysed manually the system. We noticed two aspects of the system that we could try to discover using FiG.

The first one can mostly be seen in the gobo cluster. Gobo seems to encompass many different classes which seem to have the same role but on different operating systems. Indeed, the parsing of files on Windows or Unix system for example was done in two different classes respectively depending on the operating system the program was run in. As such, we should be able to identify many bridges between abstractions and concrete implementations of the cluster which selects the right operating system tree. Hence, we will try to see whether FiG is able to detect Bridge patterns within this system.

For the second one, we based ourself on the way the eParser works. Basically
it creates a product (i.e. a parse tree here) that would contain more and more information (such as class names, features names, etc...) while the parsing is being performed. This final product is then returned at the end to the client (the main class). This way of proceeding seem to indicate the usage of the Builder pattern in order to reach this result.

Let us first observe the results found when detecting the Bridge pattern.

5.2.1 Bridge pattern in FiG

Figure 5.3 and Figure 5.4 show respectively the static structure and the static definition of the Bridge pattern.

After the class-level static analysis phase, 836 candidate instances were discovered. The dynamic analysis phase was skipped due to the lack of a comprehensive test suite. This means that possible true positives might be discarded from that
Figure 5.4: Static definition of the Bridge pattern

phase. Thus, skipping this phase allowed us to see if FiG could still detect all of those possible instances using the fine-grained analysis phase. The number of candidate instances discovered earlier was greatly reduced by applying the Bridge fine-grained definition.

Figure 5.5 shows the fine-grained definition of the Bridge pattern. The line containing the keyword ObjectReturnedFromFeatureInClass represents the fact that in a potential abstraction class, a feature (also includes attributes) must return an object of type implementor. Given a candidate instance, FiG will flag this instance as a true positive if that definition is respected. Otherwise, it will return false and the testing of the next definition is skipped. The next two definitions, which contain the keyword DeferredClass this time specify that the candidate instance needs to have the potential abstraction and implementor to be deferred.

FiG found 102 different instances after this stage. All of those 102 instances contain classes from the gobo library as seen in Figure 5.6 which shows the static structure of the cluster (other classes of the cluster that are not displayed in the
Figure 5.5: Fine-grained definition of the Bridge pattern

results generated by FiG are omitted). We can see two bridges on two different levels. One where `KL_SHARED_FILE_SYSTEM` uses `KL_FILESYSTEM`, and the other one where `KL_SHARED_OPERATING_SYSTEM` uses `KL_OPERATING_SYSTEM`. All classes inheriting from `KL_SHARED_FILE_SYSTEM` will at least be represented once in the list of candidates results.

This confirms that the gobo library was indeed based upon the Bridge pattern in order to decouple the actual implementation of the system parsing according to the corresponding operating system the library was used from.

5.2.2 Builder pattern in FiG

The Builder pattern is mostly used by the eParser itself when generating the final parse tree holding in each of its nodes an element representative of the information parsed from the system. The static structure and definition of the Builder pattern can be seen in Figure 5.7 and 5.8 respectively.

The static analysis phase led to the discovery of 1106 candidate instances. Fig-
Figure 5.6: Case Study 2. eParser’s gobo library static structure involving the Bridge pattern.
Figure 5.7: Static structure of the Builder pattern.

```plaintext
//director builder concreteBuilder product
director uses builder
concreteBuilder inherits builder
concreteBuilder uses product
```

Figure 5.8: Static definition of the Builder pattern.

Figure 5.9 shows the fine-grained definition of the Builder pattern. The keyword `ObjectReturnedFromFeatureInClass` was described previously and is applied once again to the Builder pattern this time. In this case, the potential `builder` class needs to contain at least one feature that returns an object of type `product`.

47 instances were found after the fine-grained phase and most were involving the

```plaintext
//director builder concreteBuilder product
ObjectReturnedFromFeatureInClass builder product
```

Figure 5.9: Fine-grained definition of the Builder pattern.
Figure 5.10: Case Study 2. eParser static structure involving the Builder pattern.

client (i.e. the MAIN class which is not shown in the diagram for clarity) initiating
the construction of several objects (could not actually be considered as Builder in-
stances) and others involving classes from the eParser cluster. Figure 5.10 shows the
static structure of the cluster omiting classes not displayed in the results generated
by FiG.

In this particular case, the class ES\_NODE\_TYPE would be the director using the
builder ES\_EIFFEL\_TOKENS. The concrete builder(s) would be all classes inheriting
from ES\_BASE\_INFO in this case (only ES\_FEATURE\_INFO and ES\_CLASS\_INFO were
represented due to lack of space but there exists many more similar concrete builders such as ES\_IDENTIFIER\_INFO). The object tree\_node of type ES\_PARSE\_NODE would be the final product after the building phase is over.

This case study certainly proved that the eParser cluster indeed used the Builder pattern to build its parse tree which is the final product. The usage of the Builder pattern helps create more easily elements of the parse tree dynamically while the system’s code is read: concrete builders are called when needed and help to build the final product.

5.2.3 Summary

FiG was successfully able to retrieve some patterns from the target system. The case studies confirmed our first observations about the target system concerning the presence of the Bridge and Builder patterns in the system. Two (global) instances of the Bridge pattern were found in the gobo library used by the eParser cluster while one instance of the Builder pattern was found within the parsing classes of eParser. Only for the Builder pattern did we find many false positives. However, they are considered false positives simply because the actual builder class was the client itself which was creating new global objects to be used by the system.

The tests were conducted on a Pentium IV 3.4GHz/1Gb RAM computer running Windows XP. In average, they took no longer than 15 minutes to finish. Most of the
time is spent on the fine-grained analysis phase which involved parsing the source code of all classes in the system.

5.3 Case Study 3: Student implementations

This case study contains many pattern implementations developed by students in the Computer Science department of York University and it was also carried out by Wang. These implementations are based on the Decorator and State patterns.

The students were given the source code of an existing system, which transfers a piece of music score from a plain text format to the PostScript format. Students were asked to refactor the provided implementation so that it uses the Decorator pattern. This should result in an implementation with the same external behavior as the original system. Students were given complete freedom in implementing the Decorator pattern. Afterwards, they had to re-implement the system using the State pattern this time. For simplification purpose, we will call the implementation of the Decorator Task D and that of the State pattern Task S.

After removing some implementations that generated compile time errors or provided wrong output, we had 62 implementations that should contain at least one instance of the Decorator pattern for Task D and 54 such implementations for Task S.

The input file we used for the dynamic fact collection was designed to invoke each
component in the patterns at run time. Since there are various ways to implement the pattern, this input file helps but does not guarantee the run-time-invocation of all pattern components.

Let us first consider experiments done on Task D. In order to investigate the effectiveness of our approach in terms of detecting pattern instances, as well as eliminating false positives, we attempted to detect the Decorator pattern, as well as 20 additional patterns in the 62 implementations available to us that are supposed to be using the Decorator pattern (the Prototype and Facade patterns were excluded for the same reason stated previously in Case Study 1).

The results of the experiments are shown in Tables 5.1 and 5.2.

We can see that the class-level static detection of the patterns gave many false positives. The fine-grained analysis phase led to much better results since many false positives were removed. However, we observe that we also find fewer Decorator instances after the fine-grained detection phase than expected. After looking at the implementations individually, we noticed that some of them ended up being true negatives due to wrong implementation of the pattern by the students. To be more precise, Rule 1 for the detection of the pattern (Table A.2) was in most cases not respected.

We also found many Strategy instances as well as a few State instances. The Strategy pattern has a static structure similar to that of the State pattern. The
<table>
<thead>
<tr>
<th>Pattern name</th>
<th>SC</th>
<th>SC→FG</th>
<th>SC→DC</th>
<th>SC→DC→FG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract Factory</td>
<td>293</td>
<td>18</td>
<td>185</td>
<td>9</td>
</tr>
<tr>
<td>Builder</td>
<td>3242</td>
<td>566</td>
<td>64</td>
<td>22</td>
</tr>
<tr>
<td>Factory Method</td>
<td>293</td>
<td>18</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Singleton</td>
<td>1114</td>
<td>54</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Adapter</td>
<td>1114</td>
<td>122</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Bridge</td>
<td>1069</td>
<td>0</td>
<td>200</td>
<td>0</td>
</tr>
<tr>
<td>Composite</td>
<td>1033</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Decorator</td>
<td>1000</td>
<td>922</td>
<td>174</td>
<td>164</td>
</tr>
<tr>
<td>Flyweight</td>
<td>666</td>
<td>0</td>
<td>480</td>
<td>0</td>
</tr>
<tr>
<td>Proxy</td>
<td>8</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Chain of Responsibility</td>
<td>674</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Command</td>
<td>3242</td>
<td>207</td>
<td>301</td>
<td>1</td>
</tr>
<tr>
<td>Interpreter</td>
<td>1114</td>
<td>152</td>
<td>305</td>
<td>2</td>
</tr>
<tr>
<td>Iterator</td>
<td>4502</td>
<td>0</td>
<td>409</td>
<td>0</td>
</tr>
<tr>
<td>Mediator</td>
<td>666</td>
<td>0</td>
<td>480</td>
<td>0</td>
</tr>
<tr>
<td>Memento</td>
<td>250</td>
<td>0</td>
<td>73</td>
<td>0</td>
</tr>
<tr>
<td>Observer</td>
<td>293</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>State</td>
<td>1614</td>
<td>192</td>
<td>135</td>
<td>25</td>
</tr>
<tr>
<td>Strategy</td>
<td>1614</td>
<td>1422</td>
<td>135</td>
<td>110</td>
</tr>
<tr>
<td>Template Method</td>
<td>674</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Visitor</td>
<td>2314</td>
<td>493</td>
<td>71</td>
<td>23</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>26789</td>
<td>4168</td>
<td>3017</td>
<td>360</td>
</tr>
</tbody>
</table>

Table 5.1: Task D results: The total number of instances detected for each pattern across all 62 implementations is shown. SC, DC and FG are respectively, the Class-level Static analysis, the Class-level Dynamic analysis and the Fine-Grained static analysis. The symbol $\rightarrow$ denotes the order of the phases with each next phase using as input the results of the previous phase.
<table>
<thead>
<tr>
<th>Pattern name</th>
<th>SC</th>
<th>SC→FG</th>
<th>SC→DC</th>
<th>SC→DC→FG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract Factory</td>
<td>54</td>
<td>7</td>
<td>54</td>
<td>6</td>
</tr>
<tr>
<td>Builder</td>
<td>61</td>
<td>26</td>
<td>23</td>
<td>15</td>
</tr>
<tr>
<td>Factory Method</td>
<td>54</td>
<td>7</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Singleton</td>
<td>61</td>
<td>54</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Adapter</td>
<td>61</td>
<td>61</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Bridge</td>
<td>45</td>
<td>0</td>
<td>40</td>
<td>0</td>
</tr>
<tr>
<td>Composite</td>
<td>42</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Decorator</td>
<td>40</td>
<td>38</td>
<td>37</td>
<td>35</td>
</tr>
<tr>
<td>Flyweight</td>
<td>62</td>
<td>0</td>
<td>62</td>
<td>0</td>
</tr>
<tr>
<td>Proxy</td>
<td>5</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Chain Of Responsibility</td>
<td>62</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Command</td>
<td>61</td>
<td>41</td>
<td>61</td>
<td>1</td>
</tr>
<tr>
<td>Interpreter</td>
<td>61</td>
<td>44</td>
<td>61</td>
<td>1</td>
</tr>
<tr>
<td>Iterator</td>
<td>62</td>
<td>0</td>
<td>60</td>
<td>0</td>
</tr>
<tr>
<td>Mediator</td>
<td>62</td>
<td>0</td>
<td>62</td>
<td>0</td>
</tr>
<tr>
<td>Memento</td>
<td>54</td>
<td>0</td>
<td>54</td>
<td>0</td>
</tr>
<tr>
<td>Observer</td>
<td>54</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>State</td>
<td>62</td>
<td>62</td>
<td>51</td>
<td>25</td>
</tr>
<tr>
<td>Strategy</td>
<td>62</td>
<td>53</td>
<td>51</td>
<td>48</td>
</tr>
<tr>
<td>Template Method</td>
<td>62</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Visitor</td>
<td>53</td>
<td>35</td>
<td>20</td>
<td>4</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>1140</td>
<td>426</td>
<td>639</td>
<td>137</td>
</tr>
</tbody>
</table>

Table 5.2: Task D results: The number of implementations (out of 62) containing pattern instances is shown. SC, DC and FG are respectively, the Class-level Static analysis, the Class-level Dynamic analysis and the Fine-Grained static analysis. The symbol → denotes the order of the phases with each next phase using as input the results of the previous phase.
main difference between both however is that the Context in the State pattern will
be responsible for creating a ConcreteState object. This is not the case for the
Strategy pattern where the context does not create any kind of ConcreteStrategy.
Unfortunately, this definition is not enough to filter out all possible false positives
since nothing prevents two independent classes from being part of a class structure
similar to that of the Strategy pattern which itself represents a simple application of
polymorphism usage. That explains the false positives we found for both the Strat-
ey and State patterns that seems to have a too "general" fine-grained definition.
Unfortunately, there are no other definitions to add to the list. Since the two defini-
tions are complementary (whether the concrete State or Strategy class needs to be
created from the Context or not), we can indeed observe complementary results for
both patterns. Since they have the same class-level static and dynamic definitions,
they both lead to the same candidate instances after each of those phases. With
135 different instances resulting from the static and dynamic analysis phase, 25
State pattern instances were discovered and the rest (110) was all Strategy pattern
instances.

A few instances from the Builder and Visitor patterns were also discovered and
most were false positives besides a few of them for the Builder pattern where some
students actually introduced a new class to deal with the building of objects. The
few other instances found for the other patterns are all false positives that were
caught due to the odd class structure of their implementations in most cases.

The same kind of experiment was conducted on the other 54 implementations of Task S. The same 21 design patterns were being detected in these implementations as well.

The results of the experiments are shown in Tables 5.3 and 5.4.

For this experiment, the results are even more obvious. The State pattern were found 106 times in total followed by the Strategy pattern with 18 found instances. These results are again complementary.
<table>
<thead>
<tr>
<th>Pattern name</th>
<th>SC</th>
<th>SC→FG</th>
<th>SC→DC</th>
<th>SC→DC→FG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract Factory</td>
<td>180</td>
<td>0</td>
<td>135</td>
<td>0</td>
</tr>
<tr>
<td>Builder</td>
<td>382</td>
<td>108</td>
<td>41</td>
<td>7</td>
</tr>
<tr>
<td>Factory Method</td>
<td>180</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Singleton</td>
<td>451</td>
<td>65</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Adapter</td>
<td>451</td>
<td>103</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Bridge</td>
<td>80</td>
<td>0</td>
<td>43</td>
<td>0</td>
</tr>
<tr>
<td>Composite</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Decorator</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Flyweight</td>
<td>858</td>
<td>0</td>
<td>601</td>
<td>0</td>
</tr>
<tr>
<td>Proxy</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Chain of Responsibility</td>
<td>488</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Command</td>
<td>382</td>
<td>16</td>
<td>165</td>
<td>3</td>
</tr>
<tr>
<td>Interpreter</td>
<td>451</td>
<td>11</td>
<td>237</td>
<td>1</td>
</tr>
<tr>
<td>Iterator</td>
<td>3202</td>
<td>0</td>
<td>692</td>
<td>0</td>
</tr>
<tr>
<td>Mediator</td>
<td>858</td>
<td>0</td>
<td>601</td>
<td>0</td>
</tr>
<tr>
<td>Memento</td>
<td>559</td>
<td>0</td>
<td>189</td>
<td>0</td>
</tr>
<tr>
<td>Observer</td>
<td>180</td>
<td>0</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>State</td>
<td>527</td>
<td>364</td>
<td>124</td>
<td>106</td>
</tr>
<tr>
<td>Strategy</td>
<td>527</td>
<td>163</td>
<td>124</td>
<td>18</td>
</tr>
<tr>
<td>Template Method</td>
<td>488</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Visitor</td>
<td>147</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>10399</strong></td>
<td><strong>830</strong></td>
<td><strong>2982</strong></td>
<td><strong>135</strong></td>
</tr>
</tbody>
</table>

Table 5.3: Task S results: The total number of instances detected for each pattern across all 54 implementations is shown. SC, DC and FG are respectively, the Class-level Static analysis, the Class-level Dynamic analysis and the Fine-Grained static analysis. The symbol → denotes the order of the phases with each next phase using as input the results of the previous phase.
Table 5.4: Task S results: The number of implementations (out of 54) containing pattern instances is shown. SC, DC and FG are respectively, the Class-level Static analysis, the Class-level Dynamic analysis and the Fine-Grained static analysis. The symbol → denotes the order of the phases with each next phase using as input the results of the previous phase.
//client iterator concreteIterator concreteAggregate
client uses concreteAggregate
client uses iterator
concreteIterator inherits iterator

Figure 5.11: Static definition of the Iterator pattern

The detection of the Iterator pattern in the different implementations led to the greatest number of false positives at the class-level static analysis phase. The reason behind those numbers is that the static definition of this pattern is quite generic as shown in Figure 5.11. The simple fact that the concreteAggregate is isolated allows several possible candidates at the static level: if any potential client uses a potential iterator and a possible concreteIterator inherits that iterator then any other classes the client might use will be part of a new candidate instance while being flagged as a concreteIterator. The effect is more noticeable in Task S where there exists many client-supplier relationships.

An important aspect of the results is that the false positives removed using dynamic analysis are mostly different from the ones removed using fine-grained static analysis. This shows the complementarity of the two approaches, since in the case of the Command pattern in Table 5.1, 6.36% of the total number of false positives were removed by the dynamic phase uniquely, 9.26% by the fine-grained phase uniquely and 84.38% were the same false positives removed by both phases.
Figure 5.12: Complementarity of the results obtained for the Command pattern. DC and FG stand respectively for Class-level dynamic phase and Fine-Grained level phase.

After running both phases, only one candidate instance remained. It was shown to be a true positive but the implementation of the Command pattern by the student was completely unrelated to the implementation of the Decorator (Task D). A diagram illustrating their complementarity can be seen in Figure 5.12.

Even though both dynamic analysis and fine-grained static analysis are necessary in order to obtain a detection method of high precision, it is interesting to note that fine-grained analysis appears to be the more effective of the two. In the experiments presented here, fine-grained analysis removed in average 81% of the false positives, while dynamic analysis removed only 77%. The fine-grained analysis
is particularly efficient for removing false positives in detecting structural and behavioral patterns. For the creational patterns, the dynamic analysis phase filtered out much more false positives in general. This can be explained by the nature of creational patterns that often involves client-supplier relationships between specific classes and thus particular dynamic behaviors. It is important to note however that doing dynamic analysis can lead to the removal of true positives if those instances do not get executed at runtime. Fortunately, such a risk does not exist while doing fine-grained analysis making this phase more accurate in detecting true positives. If a test suite exists for the target system, the precision of the dynamic and fine-grained analysis phases in detecting true positives should be about the same.

Finally, it should be noted that the order of applying the two approaches for false positive removal does not affect the final results since the two approaches are independent.

Figures 5.13 and 5.14 show the total number of false positives removed by the different phases (as well as with all phases combined) after the static analysis phase. Although the dynamic analysis phase gave slightly better filtering results than the fine-grained analysis phase in Task D, the latter (phase) led to much better results in Task S. Both dynamic and fine-grained analyses combined gave very good false positives filtering ratios.
Figure 5.13: Total number of false positives removed per phases for Task D.

Figure 5.14: Total number of false positives removed per phases for Task S.
In this case study (and for both experiments), the total number of possible class combinations for all implementations is 138344320. Out of those random combinations, only 0.01936% (or 26789) instances were shown to potentially be real pattern instances after the static analysis phase. From those 26789, the dynamic analysis removed in total 20790 false positives (77.61% filtering rate) and the fine-grained one 21791 (81.34%). The overall filtering rate (for both phases) is 98.15% (with 26294 removed in total).
6 Conclusion

Although design pattern detection can be a tedious task, we have shown that significant progress has been made in the domain. Not only is it possible to detect design patterns at the coarse-grained level, we also showed that using a finer grained approach to the problem led to improved results.

Our major contributions to pattern detection include a catalogue of fine-grained rules for design patterns in Eiffel systems as well as a flexible, modular and powerful language-independent reverse engineering tool: FiG.

6.1 Research contributions

1. In this thesis, we provided a list of fine-grained rules for each pattern that can be used to filter out false positives from a list of candidate instances. A list of those rules can be observed in Appendix A. We were able to pinpoint specific features of most of the patterns so that the resulting fine-grained rules would highlight them and thus considerably reduce possible redundancy with other
2. We created a new tool, FiG, based on the DPVK tool developed by Wang, which encompasses a new false positive elimination layer consisting of the matching of fine-grained rules with fine-grained facts of the target Eiffel system. The fine-grained facts are obtained from the system’s source code and allow the detection of true positives on a better precision scale. FiG can process as many as 5 different detection and filtering phases in any order given by the user (provided that each phase’s inputs are present): candidate instance discovery, static analysis, dynamic analysis, fine-grained analysis and post-processing analysis (which only applies to a few patterns).

3. FiG consists of modules instead of interconnecting and dependent classes as was the case in DPVK. Using modules promotes loose coupling between classes and significantly improves the scalability and flexibility of our tool by allowing wider possibilities such as language independent detection processes (not at the fine-grained level however which would require another module on its own for the target language since it involves source code parsing). As such, it becomes possible to detect design patterns in Java or C++ for example.
6.2 Future Work

- Additional tests

FiG was tested thoroughly on the case studies. But most of the case studies implementations were rather small, not big enough to deduce the impact of the detection process on large-scale systems. It is very likely that the tool would experience performance problems with very large systems at the runtime level or on the memory side. In the fine-grained level analysis, parsing the code can become CPU-intensive or memory-demanding as the system size grows. Knowing the limitations of our tool could help refine the tool to provide better data handling as well as increased efficiency.

- More refined and flexible fine-grained definitions

The different filtering modules, whether they be for static, dynamic or fine-grained analysis, all have a common drawback: they can only process one candidate instance at a time. This can be an issue for some patterns consisting of clusters of instances. For example, the Abstract Factory is a collection of Factory Methods. Thus, the output of this pattern’s detection should include ”clusters” of instances only (hence resulting in a unique pattern instance for each cluster), and not individual candidate instances found for each Factory Method instances as it is the case currently in FiG for each of those 3 phases.
Consequently, there should be a new module in charge of this clustering task as well as corresponding rules to perform the cumulation of the right instances for the right patterns.

- Enhanced flexibility

The tool should be able to allow easy additions of new modules for language specific detection (such as parsing tools) which mostly concerns the fine-grained or fact extraction (static or dynamic) phases. Its implementation should become more abstract such that the main class becomes completely language-independent.

Secondly, it would be quite useful to add a new XML writer and parser module to convert FiG’s input and output formats to XML format and vice-versa. This would allow better interaction with external applications that can parse the popular XML format. This could enhance one step further the standardization, compatibility and extendability of FiG.

- Adding static parsing and dynamic tracing modules

In our current work, we had to use certain features integrated in EiffelStudio to retrieve static and dynamic facts of an Eiffel system. The static facts would be obtained after parsing the Documentation files generated by EiffelStudio when requested. Dynamic facts are retrieved after parsing and cleaning the
raw dynamic tracing output of the running system which can also be produced by EiffelStudio on demand. As such, one important and useful future work could lie on implementing our own parsing and tracing modules to automatically generate static and dynamic facts. This would avoid the need to interact with EiffelStudio beforehand and would therefore grant complete software autonomy to FiG.
A Appendix-A

A.1 Catalogues of Design Patterns

In this Appendix, we show for each of the GoF patterns, the static, dynamic and fine-grained definitions. We also present a table summarizing the fine-grained rules given in Chapter 3 in plain English sentences.
A.1.1 Abstract Factory

```
//abstractFactory concreteFactory product abstractProduct
concreteFactory inherits abstractFactory
concreteFactory uses product
product inherits abstractProduct
```

Figure A.1: Static definition of the Abstract Factory pattern

```
//abstractFactory concreteFactory product abstractProduct
* calls concreteFactory
concreteFactory calls product
```

Figure A.2: Dynamic definition of the Abstract Factory pattern

```
//abstractFactory concreteFactory product abstractProduct
SameFeatureInTwoClassesWith2CreateMinimum concreteFactory abstractFactory
//(Optional) AccessRestrictionOnCreate product abstractFactory
```

Figure A.3: Fine-grained definition of the Abstract Factory pattern
A.1.2 Builder

```java
//director builder concreteBuilder product
director uses builder
concreteBuilder inherits builder
concreteBuilder uses product
```

Figure A.4: Static definition of the Builder pattern

```java
//director builder concreteBuilder product
* calls director
director calls concreteBuilder
concreteBuilder calls product
```

Figure A.5: Dynamic definition of the Builder pattern

```java
//director builder concreteBuilder product
ObjectReturnedFromFeatureInClass builder product
```

Figure A.6: Fine-grained definition of the Builder pattern
A.1.3 Factory Method

//creator concreteCreator concreteProduct product
congcreteCreator inherits creator
congcreteCreator uses concreteProduct
congcreteProduct inherits product

Figure A.7: Static definition of the Factory Method pattern

//creator concreteCreator concreteProduct product
* calls concreteCreator==>
congcreteCreator calls concreteProduct

Figure A.8: Dynamic definition of the Factory Method pattern

//creator concreteCreator concreteProduct product
SameFeatureInTwoClassesWithCreate creator concreteCreator

Figure A.9: Fine-grained definition of the Factory Method pattern
A.1.4 Singleton

```plaintext
//accessorClient singletonAccessor singleton
accessorClient inherits singletonAccessor
accessorClient uses singleton
```

Figure A.10: Static definition of the Singleton pattern

```plaintext
//accessorClient singletonAccessor singleton
*
   //Basically no dynamic definition for this pattern
```

Figure A.11: Dynamic definition of the Singleton pattern

```plaintext
//accessorClient singletonAccessor singleton
OnceInFeatureAndReturnTypeOf accessorClient singleton
```

Figure A.12: Fine-grained definition of the Singleton pattern
A.1.5 Adapter

//target adapter adaptee
adapter inherits target
adapter uses adaptee

Figure A.13: Static definition of the Adapter pattern

//target adapter adaptee
* calls adapter==>
adapter calls adaptee

Figure A.14: Dynamic definition of the Adapter pattern

//target adapter adaptee
CallsFromClassInClassWithMinThreshold adapter adaptee 3

Figure A.15: Fine-grained definition of the Adapter pattern
A.1.6 Bridge

//refinedAbstraction abstraction implementor concreteImplementor
refinedAbstraction inherits abstraction
abstraction uses implementor
concreteImplementor inherits implementor

Figure A.16: Static definition of the Bridge pattern

//refinedAbstraction abstraction implementor concreteImplementor
refinedAbstraction calls concreteImplementor

Figure A.17: Dynamic definition of the Bridge pattern

//refinedAbstraction abstraction implementor concreteImplementor
ObjectReturnedFromFeatureInClass abstraction implementor
DeferredClass abstraction
DeferredClass implementor

Figure A.18: Fine-grained definition of the Bridge pattern
A.1.7 Composite

```
//leaf component composite
leaf inherits component
composite inherits component
composite uses component
```

Figure A.19: Static definition of the Composite pattern

```
//leaf component composite
* calls composite
composite calls component
```

Figure A.20: Dynamic definition of the Composite pattern

```
//leaf component composite
SameFeatureToChild composite
InheritsFromSameGenericParameter composite
```

Figure A.21: Fine-grained definition of the Composite pattern
A.1.8 Decorator

//concreteComponent component decorator concreteDecorator
concreteComponent inherits component
decorator inherits component
decorator uses component
congreteDecorator inherits decorator

Figure A.22: Static definition of the Decorator pattern

//concreteComponent component decorator concreteDecorator
concreteDecorator calls concreteComponent

Figure A.23: Dynamic definition of the Decorator pattern

//concreteComponent component decorator concreteDecorator
SameFeatureToChild decorator

Figure A.24: Fine-grained definition of the BuDecorator pattern
A.1.9 Flyweight

//flyweightFactory flyweight concreteFlyweight
congreteFlyweight inherits flyweight
flyweightFactory uses concreteFlyweight

Figure A.25: Static definition of the Flyweight pattern

//flyweightFactory flyweight concreteFlyweight
flyweightFactory calls concreteFlyweight

Figure A.26: Dynamic definition of the Flyweight pattern

//flyweightFactory flyweight concreteFlyweight
CollectionOfAttributeOfType flyweightFactory concreteFlyweight
CreateInFeatureAndReturnTypeOf flyweightFactory concreteFlyweight
OnceInFeatureAndReturnTypeOf flyweight flyweightFactory

Figure A.27: Fine-grained definition of the Flyweight pattern
A.1.10 Proxy

//realSubject subject proxy
realSubject inherits subject
proxy inherits subject
proxy uses realSubject

Figure A.28: Static definition of the Proxy pattern

//realSubject subject proxy
* calls proxy==> 
proxy calls realSubject

Figure A.29: Dynamic definition of the Proxy pattern

//realSubject subject proxy
SamePublicFeatures proxy realSubject 
CallsFromClassInClassWithMinThreshold proxy realSubject 50%

Figure A.30: Fine-grained definition of the Proxy pattern
A.1.11 Chain of Responsibility

```java
// handler concreteHandler
concreteHandler inherits handler
```

Figure A.31: Static definition of the Chain of Responsibility pattern

```java
// handler concreteHandler
*    // Basically no dynamic definition for this pattern
```

Figure A.32: Dynamic definition of the Chain of Responsibility pattern

```java
// handler concreteHandler
ObjectReturnedFromFeatureInClass handler handler
SameFeatureToChild handler
```

Figure A.33: Fine-grained definition of the Chain of Responsibility pattern
A.1.12 Command

```plaintext
//invoker command concreteCommand receiver
invoker uses command
concreteCommand inherits command
concreteCommand uses receiver
```

Figure A.34: Static definition of the Command pattern

```plaintext
//invoker command concreteCommand receiver
invoker calls concreteCommand
concreteCommand calls receiver
```

Figure A.35: Dynamic definition of the Command pattern

```plaintext
//invoker command concreteCommand receiver
AttributeInClassOfType concreteCommand receiver
FeatureWithArgument concreteCommand receiver
```

Figure A.36: Fine-grained definition of the Command pattern
A.1.13 Interpreter

//expression abstractExpression context
expression inherits abstractExpression
expression uses context

Figure A.37: Static definition of the Interpreter pattern

//expression abstractExpression context
* calls expression
expression calls context

Figure A.38: Dynamic definition of the Interpreter pattern

//expression abstractExpression context
FeatureWithArgument expression context
SameFeatureToChild expression

Figure A.39: Fine-grained definition of the Interpreter pattern
A.1.14 Iterator

//client iterator concreteIterator concreteAggregate
client uses concreteAggregate
client uses iterator
concreteIterator inherits iterator

Figure A.40: Static definition of the Iterator pattern

//client iterator concreteIterator concreteAggregate
* calls concreteIterator
client calls concreteAggregate

Figure A.41: Dynamic definition of the Iterator pattern

//client iterator concreteIterator concreteAggregate
HasGenericParameter concreteIterator
FeatureWithGenericParameter concreteIterator
FeatureWithGenericParameterOfType client concreteAggregate

Figure A.42: Fine-grained definition of the Iterator pattern
A.1.15 Mediator

```
//concreteMediator colleague concreteColleague
concreteColleague inherits colleague
concreteMediator uses concreteColleague
```

Figure A.43: Static definition of the Mediator pattern

```
//concreteMediator colleague concreteColleague
concreteMediator calls concreteColleague
```

Figure A.44: Dynamic definition of the Mediator pattern

```
//concreteMediator colleague concreteColleague
CallsFromClassInClassWithArgument colleague concreteMediator
```

Figure A.45: Fine-grained definition of the Mediator pattern
A.1.16 Memento

//originator memento caretaker
originator uses memento
caretaker uses memento
caretaker uses originator

Figure A.46: Static definition of the Memento pattern

//originator memento caretaker
caretaker calls originator==>
originator calls memento

Figure A.47: Dynamic definition of the Memento pattern

//originator memento caretaker
FeatureWithArgument originator memento
CreateInFeatureAndReturnTypeOf originator memento

Figure A.48: Fine-grained definition of the Memento pattern
A.1.17 Observer

```plaintext
//concreteSubject subject observer concreteObserver
congcreteSubject inherits subject
concreteObserver inherits observer
concreteObserver uses concreteSubject
```

Figure A.49: Static definition of the Observer pattern

```plaintext
//concreteSubject subject observer concreteObserver
congcreteObserver calls concreteSubject
concreteSubject calls concreteSubject
concreteSubject calls concreteObserver
```

Figure A.50: Dynamic definition of the Observer pattern

```plaintext
//concreteSubject subject observer concreteObserver
CollectionOfAttributeOfType subject observer
FeatureWithArgument subject observer
```

Figure A.51: Fine-grained definition of the Observer pattern
A.1.18 State

```
//context state concreteState
context uses state
congreteState inherits state
```

Figure A.52: Static definition of the State pattern

```
//context state concreteState
* calls context //Creation call
context calls concreteState
```

Figure A.53: Dynamic definition of the State pattern

```
//context state concreteState
CreateInClassAnObjectOfType context concreteState
```

Figure A.54: Fine-grained definition of the State pattern
### A.1.19 Strategy

```plaintext
//context strategy concreteStrategy
context uses strategy
concreteStrategy inherits strategy
```

Figure A.55: Static definition of the Strategy pattern

```plaintext
//context strategy concreteStrategy
* calls context //Creation call
  * calls context==>
  context calls concreteStrategy
```

Figure A.56: Dynamic definition of the Strategy pattern

```plaintext
//context strategy concreteStrategy
NotCreateInClassAnObjectOfType context concreteStrategy
```

Figure A.57: Fine-grained definition of the Strategy pattern
A.1.20 Template Method

//abstractClass concreteClass
congreteClass inherits abstractClass

Figure A.58: Static definition of the Template Method pattern

//abstractClass concreteClass
* //Basically no dynamic definition for this pattern

Figure A.59: Dynamic definition of the Template Method pattern

//abstractClass concreteClass
FeatureCallToDeferredFeaturesInClass abstractClass

Figure A.60: Fine-grained definition of the Template Method pattern
A.1.21 Visitor

//concreteVisitor visitor element concreteElement
concreteVisitor inherits visitor
concreteElement uses visitor
concreteElement inherits element

Figure A.61: Static definition of the Visitor pattern

//concreteVisitor visitor element concreteElement
concreteElement calls concreteVisitor==>
concreteVisitor calls concreteElement

Figure A.62: Dynamic definition of the Visitor pattern

//concreteVisitor visitor element concreteElement
FeatureWithArgument concreteElement visitor
CallsFromClassInClassWithMaxThreshold concreteElement concreteVisitor 1

Figure A.63: Fine-grained definition of the Visitor pattern
### A.2 Table of fine-grained rules

**Summary.** Tables A.1, A.2, A.3 and A.4 show the detection rules we found for each pattern.

<table>
<thead>
<tr>
<th>Creational Patterns</th>
<th>Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Factory Method</td>
<td>- We require the presence of the keyword <code>create</code> in a factory method in the ConcreteCreator class.</td>
</tr>
<tr>
<td></td>
<td>- (Optional) The return type of the factory method in the ConcreteCreator may be different than the one in the Creator.</td>
</tr>
<tr>
<td>Abstract Factory</td>
<td>- We require the presence of at least 2 or more Factory Methods in the AbstractFactory class.</td>
</tr>
<tr>
<td></td>
<td>- (Optional) The constructor of the ConcreteProduct may be exported only to the AbstractFactory (e.g. <code>creation {ABSTRACT_FACTORY}</code>).</td>
</tr>
<tr>
<td>Builder</td>
<td>- We require that there is at least one public attribute or feature that returns an object of type Product.</td>
</tr>
<tr>
<td>Prototype</td>
<td>-</td>
</tr>
<tr>
<td>Singleton</td>
<td>- We require the presence of restrictive access for the constructor of SINGLETON.</td>
</tr>
<tr>
<td></td>
<td>- We require the presence of the <code>once</code> keyword in the SINGLETON_ACCESSOR.</td>
</tr>
<tr>
<td></td>
<td>- We require the return type of the <code>once</code> method to be of type SINGLETON.</td>
</tr>
<tr>
<td></td>
<td>- We require the presence of a creation call within the <code>once</code> block that returns an object of type SINGLETON.</td>
</tr>
</tbody>
</table>

Table A.1: List of detection rules for Creational Patterns
<table>
<thead>
<tr>
<th>Structural Patterns</th>
<th>Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adapter</td>
<td>- Rule 1: We require that most features (at least 50%) of Adapter call features of the Adaptee.</td>
</tr>
</tbody>
</table>
| Bridge             | - We require the presence of an attribute of type Implementor in the Abstraction class.  
                      - We require that both the Abstraction and Implementor classes are deferred. |
| Composite          | - We require that in at least one feature of the Composite class, there is a call to another feature of the same name within the feature’s block.  
                      - We require that the Composite class either contains the Component class as a generic parameter or, contains at least two attributes of type Component. |
| Decorator          | - We require that there is at least in one feature of the ConcreteDecorator a call to the same feature in the Component class.  
                      - We require that there is exactly one attribute of type Component in the Decorator class. |
| Facade             | - |
| Flyweight          | - We require the presence of a collection of objects of type Flyweight in the FlyweightFactory class.  
                      - We require the presence of creation calls in the FlyweightFactory class that leads to the construction of the new concrete Flyweight products.  
                      - We require that some feature of the Flyweight Factory class returns an object of type Flyweight.  
                      - (Optional) The Flyweight class may be a Singleton. |
| Proxy              | - We require that at least 50% of the RealSubject’s methods are called by the Proxy class.  
                      - (Optional) The Proxy may create the RealSubject. |

Table A.2: List of detection rules for Structural Patterns
<table>
<thead>
<tr>
<th>Behavioral Patterns</th>
<th>Rules</th>
</tr>
</thead>
</table>
| Chain of Responsibility| - We require the presence of exactly one attribute of type Handler in the Handler class.  
- We require that there is in the Handler class a feature that calls another feature of the same name in the successor. |
| Command                | - We require that the ConcreteCommand class contains an attribute of type Receiver.  
- We require that at least one feature in the ConcreteCommand gets as argument an object of type Receiver (the constructor or simply a `setReceiver` feature). |
| Interpreter            | - We require that a features in the Nonterminal class calls a feature of the same name on another object.  
- We require that one of the arguments passed when calling features of the AbstractExpression are of type Context. |
| Iterator               | - We require that the ConcreteIterator be generic.  
- We require that the ConcreteIterator contains a Collection with a generic parameter of the same type as the one in the ConcreteIterator. |
| Mediator               | - We require that there are no relationships between colleagues within a certain threshold.  
- (Optional) The Colleagues may pass themselves as argument to the Mediator using the keyword `Current`. |
| Memento                | - We require that at least one feature of the Originator class takes a Memento object as argument.  
- We require that an object of type Memento is created within one of the Originator’s features.  
- (Optional) There is restrictive access to the Originator class in the Memento class. |
| Observer               | - We require the presence of an attribute being a collection of Observers in the ConcreteSubject (e.g. `observers: COLLECTION[OBSERVER]`).  
- We require that at least one feature in the Subject receives an argument of type Observer. |

Table A.3: List of detection rules for Behavioral Patterns - Part 1
<table>
<thead>
<tr>
<th>Behavioral Patterns</th>
<th>Rules</th>
</tr>
</thead>
</table>
| State               | - (Optional) The State object may be created in the potential Context class.  
- (Optional) At least one feature in the ConcreteState receives an argument of type Context.  
- (Optional) Features in the State class are exported only to the Context class.  
- (Optional) The return type of at least one feature in the State class is of type State. |
| Strategy            | - (Optional) The Strategy object may not be created in the potential Context class. |
| Template Method     | - We require that at least one other non-deferred feature calls at least one of the deferred features in the same class. |
| Visitor             | - We require that the implementation of `accept` must call exactly one method from the Visitor interface passing `Current` as the argument.  
- We require that for each of the ConcreteElements there is one and exactly one method in the ConcreteVisitor that takes this ConcreteElement as an argument. |

Table A.4: List of detection rules for Behavioral Patterns - Part 2
B Appendix-B

B.1 FiG: User Manual

FiG is a command line tool exclusively. Doing the command `java Fig -h` will bring a list of optional or required arguments to be provided when running FiG. Figure B.1 displays the output of that command.

The tool requires at least 2 arguments: the pattern to detect and the location of a configuration file which contains all parameters used in running the detection process properly. The configuration file is another improvement made over DPVK’s interface. It allows the user to predefine specific detection tasks by providing all the required information within a file. The user could then take advantage of the command line feature and make shell scripts to run FiG with different configuration files for example. This makes FiG much more flexible in the automation area. The parameters that can be defined in the configuration file or given at the command line are:

Here is the detailed description and use of each parameter:
Usage: java FiG <pattern_name> [-i | -f | -o <directory>] [-c | -s | -d | -e <file>]
[-p <phases>] [-b] [-h] <information_file>

<pattern_name> : Name or ID of the pattern to detect in the system

<information_file> : Path and name of the information file that contains
details of the detection

-i : Implementation directory
-f : Documentation directory
-e : Eiffel dynamic facts file
-c : Candidate instances file
-s : Static facts file
-d : Dynamic facts file
-o : Output directory
-p : Order of the detection process. It is "sdf" by default if omitted:
   - s: Static Analysis
   - d: Dynamic Analysis
   - f: Fine-Grained Analysis
-z : Ignore Post-Processing Phase
-h : Displays help
-b : Displays debug messages (off by default)

* The 's' phase will be done first in any case if no candidate instances file is
  provided since the static phase will discover candidate instances in this case.
* If not specified, the output directory is by default the implementation directory.
* If the documentation directory is not specified, FiG will look by default in the
  implementation directory in 'Documentation/root_cluster'.

Example 1: java FiG builder info.txt
Example 2: java FiG builder -o output/results -p d info.txt
Example 3: java FiG 2 -p fd -i implementation/files -e files/dynamic_facts.txt

The valid pattern names are:
---------------------------------------------
[4] prototype (Prototype)       [16] iterator  (Iterator)
[5] singleton (Singleton)       [17] mediator  (Mediator)
[6] adapter  (Adapter)          [18] memento   (Memento)
[7] bridge   (Bridge)           [19] observer  (Observer)
[8] composite (Composite)       [20] state     (State)
[9] decorator (Decorator)       [21] strategy  (Strategy)
[10] facade  (Facade)           [22] template  (Template Method)
[12] proxy   (Proxy)

Figure B.1: The command-line ”help” message of the tool

145
Table B.1: Corresponding command line and information file parameters.

- **-i or ImplementationDirectory.** This specifies where the target system is located.

- **-f or Documentation.** This shows the path to the Documentation folder of the system implementation which is automatically generated by EiffelStudio when requested. This will be used to extract static facts from the system.

- **-e or EiffelDynamicFactsFile.** This points to the raw output of the dynamic facts obtained after using the tracing tool of EiffelStudio. This file can be located in at least two different locations when generated. It depends on whether the system was launched from EiffelStudio or from its executable version. This output will then be refined by FiG to retrieve usable dynamic facts of the system.
- **-o or OutputDir.** This can be left empty and the output directory will be the current directory by default. Otherwise, it will be the one given by this argument.

- **-c or CandidateInstances.** This represent candidate instances provided by the user. If none are provided, candidate instances will be discovered automatically by FiG during the class-level static analysis phase using the REQL engine.

- **-s or StaticFactsFile.** This points to the static facts file. If the user provides this argument as input, FiG will not be extracting the static facts from the system (using the Documentation argument - see above) but will use this input file instead.

- **-d or DynamicFactsFile.** This points to the dynamic facts file. If this argument is provided as input, FiG will not be extracting the dynamic facts file from the EiffelStudio dynamic tracing output (using EiffelDynamicFactsFile).

- **-p or PhasesOrder.** This defines the order of the detection process as mentioned in the Subsection 4.2.3. The different phases are represented by "s" for class-level static analysis, "d" for dynamic analysis and "f" for fine-grained analysis. A combination of those letters represent the phases to execute while respecting the ordered position of each letters in the string.
If candidate instances are provided, the user may choose any order for the
phases (such as ”dsf” for example). If no candidate instances are given, the
class-level static analysis will have to be done first no matter what in order
to discover candidate instances. In this case, if the phase ”s” is not expressly
mentioned, FiG will do that phase first nonetheless. It is good to note that
once a phase is done, the same phase cannot be done again during the same
detection task. For instance, if the string ”sdsf” is provided as input for the
phases order, the second class-level static analysis will not be executed since
it was already done at the beginning.

• -z or NoPostProcess. Removes the need for post-processing the results.
The post-processing phase is only useful for a very limited number of GoF
patterns (such as the Mediator pattern). It is not necessary to use this ar-
gument for patterns which do not use the post-processing phase since their
post-processing definition files are empty and therefore, that phase will be
automatically skipped by FiG.

• -b or Debug. If this flag is turned on, FiG will display debug messages during
the detection process. Otherwise, it will only output the results obtained at
each phase.

• -h. When adding this command line flag at runtime, the help message about
Figure B.2: A typical Information File

```java
java FiG proxy -i /home/implementation/proxy info.txt
```

Figure B.3: Example of running FiG with some parameters. `info.txt` is the information file while `/home/implementation/proxy` points to the implementation directory.

the tool’s usage will be displayed.

One concrete example of an information file can be seen in Figure B.2.

The other possible parameters `-i`, `-f`, `-e`, `-o`, `-c`, `-s`, `-d`, `-p`, `-z`, `-b` and `-h` will all override the ones provided in the information file. Also, if at least the `-i` parameter is provided (for implementation directory), the information file becomes an optional argument and is not required to launch the detection phase.

As an example, Figure B.3 shows how to launch FiG to detect the proxy pattern in the system located at `/home/implementation/proxy` and using the parameters provided in `info.txt`.

Many intermediary files automatically generated by FiG at runtime can be found
in the output directory. Suppose the target pattern name is Bridge. The resulting
files would be:

1. **st_fact_Bridge.rsf**
   - This file is in RSF format. It contains the static facts extracted from
     static information of all classes in the Eiffel system.
   - The prefix *st_fact* means static fact.
   - This file might not exist if the Eiffel static facts extraction is skipped
     (if the static phase is not done or if static facts are already provided as
     input).

2. **dy_fact_Bridge.txt**
   - This file contains the dynamic facts extracted from the dynamic inform-
     mation generated by EiffelStudio’s tracing tool.
   - The prefix *dy_fact* means dynamic fact.
   - This file might not exist if the Eiffel dynamic facts extraction is skipped
     (if the dynamic phase is not done or if dynamic facts are already provided
     as input).

3. **req1_def_Bridge.txt**
• This file contains the REQL script of the pattern which is automatically
generated by FiG at runtime.

• The prefix `reql_def` means REQL script definition.

• This file might not exist if the candidate discovery phase is skipped (if
the static phase is not done or if candidate instances are already provided
as input).

4. `reql_result_Bridge.txt`

• This file contains the candidate instances discovered by REQL at the
class level (statically) in a raw format.

• The prefix `reql_result` means REQL results (or output) file.

5. `st_result_Bridge.txt`

• This file contains the output of the class-level static analysis phase (which
can either be a refined and standardized output of the REQL results
mentioned previously or the output of the static filtering phase in case
candidate instances were already provided as input at launch time).

• The prefix `st_result` means static results (or output) file.

6. `dy_result_Bridge.txt`

• This file contains the output of the dynamic analysis phase.
• The prefix `dy_result` means dynamic results (or output) file.

7. `fg_result_Bridge.txt`

• This file contains the output of the fine-grained analysis phase.

• The prefix `fg_result` means fine-grained results (or output) file.

8. `pp_result_Bridge.txt`

• This file contains the output of the post-processing phase.

• The prefix `pp_result` means post-processing results (or output) file.

• Note: Only a very small number of patterns have post-processing definition rules. And thus, for most patterns, this phase will be skipped automatically.
Bibliography


