# Design Pattern Detection in Eiffel Systems

Wei Wang and Vassilios Tzerpos
York University
Toronto, Ontario, Canada
{weiw,bil}@cs.yorku.ca

## Abstract

*The use of design patterns in a software system can provide strong indications about the rationale behind the system's design. As a result, automating the detection of design pattern instances could be of significant help to the process of reverse engineering large software systems.*

*In this paper, we introduce DPVK (Design Pattern Verification toolKit), the first reverse engineering tool to detect pattern instances in Eiffel systems. DPVK is able to detect several different design patterns by examining both the static structure and the dynamic behaviour of a system written in Eiffel. We present three case studies that were performed to assess DPVK's effectiveness.*

## 1 Introduction

Design patterns have attracted significant attention in software engineering in the last decade. An important reason behind this trend is that design patterns are potentially useful in both development of new, and comprehension of existing object-oriented designs, especially for large legacy systems without sufficient documentation.

The design patterns introduced by Gamma et al.[3] capture solutions that have developed and evolved over time. Each design pattern indicates a high level abstraction, encompasses expert design knowledge, and represents a solution to a common design problem. A pattern can be reused as a building block for better software construction and designer communication.

Design patterns are not only beneficial to the forward engineering process. From a design recovery perspective, patterns can provide information about the organization of a system and the role of each component. Moreover, patterns can also indicate the design rationale behind the system's implementation. Since design patterns have well-defined contexts in which they are applied, as well as well-defined consequences, finding instances of design patterns in a system can potentially yield the motivation for the system's design. As a result, several approaches to design pattern detection have been presented in the literature (surveyed in Section 2).

A typical system structure for pattern detection tools includes three parts: a parser, a detector, and a database. The parser extracts facts from the implementation. Then, the detector retrieves pattern definitions from the database, compares these definitions with the facts, and outputs the detection result. The database can also be used to store detected pattern instances for further analysis.

One of the core issues in design pattern detection is pattern definition. It must accurately reflect the traits of each pattern, as well as differentiate it from other patterns. If we treat such a definition as a set of conditions, then the set must contain all necessary conditions without any unnecessary ones. The strictness of definition is a prerequisite for detection tools to find correct pattern instances without missing any. The more redundancy in a pattern's definition, the more false positives will be reported.

This brings us to the second challenge that design pattern detection techniques face: eliminating false positives. Many recently developed tools use multiple passes [11], or recursive filtering [5], to improve the detection output. In the meantime, it is becoming clear that both static structure and dynamic behaviour need to be taken into account in order to increase the precision and recall of existing techniques.

In this paper, we present the first design pattern detection tool for systems written in Eiffel. Eiffel is a pure object-oriented language with many distinct features that require a different approach to pattern detection, such as genericity and executable contracts. Our approach includes rigid pattern definitions based on REQL [12] for static structure (inheritance and client-supplier relationships), and "ordered RSF" for dynamic behaviour (message passing). The results of three case studies indicate that such a minimal approach can be quite effective in detecting design patterns and eliminating a large number of false positives.

The remainder of this paper is organized as follows: Section 2 presents related work on design pattern detection. The DVPK tool is introduced in Section 3 together with our approach for defining design patterns. The results of applying DVPK in three different case studies are presented in Section 4. Finally, Section 5 concludes our work and presents opportunities for further research.

## 2 Background

Although research on design pattern detection is still at an early stage, there are several interesting contributions in the literature. Design pattern detection tools and techniques can be catalogued in various ways, depending on the supported language, analysis approach, and effectiveness/efficiency. In the following, we discuss several design pattern detection tools in chronological order.

**Pat**[8]: The Pat system 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. Both patterns and examined designs are represented in PROLOG format. The actual matching work is done by a PROLOG engine. In contrast to other reverse engineering tools, Pat does not intend to do very detailed program understanding or design recovery. Its limitation is that it has difficulty in dealing with behavioural patterns, since too much semantic information is required.

**KT**[2]: KT is 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. Three GoF design patterns are analyzed: Composite, Decorator and Chain of Responsibility. The methods used for the detection of these patterns are coded directly into the KT source code.

**Seemann-Gudenberg**[10]: The authors present some ideas on how to recover design information from Java source code. The pattern recognition approach proceeds in three steps that reveal different layers of abstraction. First, a compiler collects information about inheritance, method call and particular naming conventions and generates a graph as its output. Next, this graph is transformed by graph grammar productions. The transformation is done at different levels, from method call to class association and delegation. Finally, pattern detection is conducted by predefined criteria. Although there is only one pass through the source code, each subsequent transformation phase relies on the result of the previous phase.

**SPOOL**[7]: The SPOOL (Spreading Desirable Properties into the Design of Object-Oriented, Large-Scale Software Systems) project is a joint industry/university collaboration between the software quality assessment team of Bell Canada and the GELO (GEnie LOgiciel = software engineering) group at the University of Montreal. This project aims at both software comprehension and software design quality assessment.

As a part of the project, the SPOOL environment for design pattern engineering supports both forward and reverse engineering of design patterns. This environment comprises functionality for design composition, change impact analysis, and most importantly, support for the recovery of design patterns. Using this environment, human analyzers can zoom into design components that resemble patterns, extract them as diagrams, compare them with pattern descriptions, or, in the case of a false positive, dismiss the finding of the identified pattern instance.

**JBOORET**[9]: JBOORET (Jade Bird Object-Oriented Reverse Engineering Tool) adopts a parser-based approach to extract the higher-level design information and conceptual model from system artifacts. The JBOORET for C++ consists of three major components: a data extractor, a knowledge manager and an information presenter. Separating data extraction from information representation, this architecture avoids repeating analysis process for each higher-level model extraction. As the front-end, the data extractor is the only part that processes language dependent data. This design enables JBOORET to be easily adapted into reverse engineering a system written in a language other than C++. The knowledge manager contains conceptual models of the developing language. The information presenter component organizes and generates design information according to the user's preference.

**Heuzeroth-Holl-Hogstrom-Lowe**[4]: The authors present a way to automatically detect patterns by combining both static and dynamic analysis. The former restricts the code construction and the latter the runtime behaviour. This analysis does not depend on coding or naming conventions. A pattern instance is defined by a tuple of program elements such as classes, methods or attributes. These elements must conform to the rules of a certain design pattern.

First, source code is processed by static analysis which transforms the implementation into tuples of AST(Abstract Syntax Tree) nodes. Then, static analysis computes predefined pattern relationships on the AST nodes and generates the candidate set of pattern instances. The dynamic analysis takes this set as its input. It monitors the execution of the nodes of every tuple. It also tracks the output of the executed nodes to check whether the candidate complies with dynamic pattern rules. The candidate is eliminated if the rule is violated. Finally, the remaining candidate set contains the detected pattern instances.

The authors argue that neither static nor dynamic analysis by themselves provide an adequate approach to finding patterns in software systems. The number of false positives is tremendously reduced by checking run-time interaction within patterns. So far, the tool can detect Observer, Composite, Mediator, Chain of Responsibility and Visitor pat-

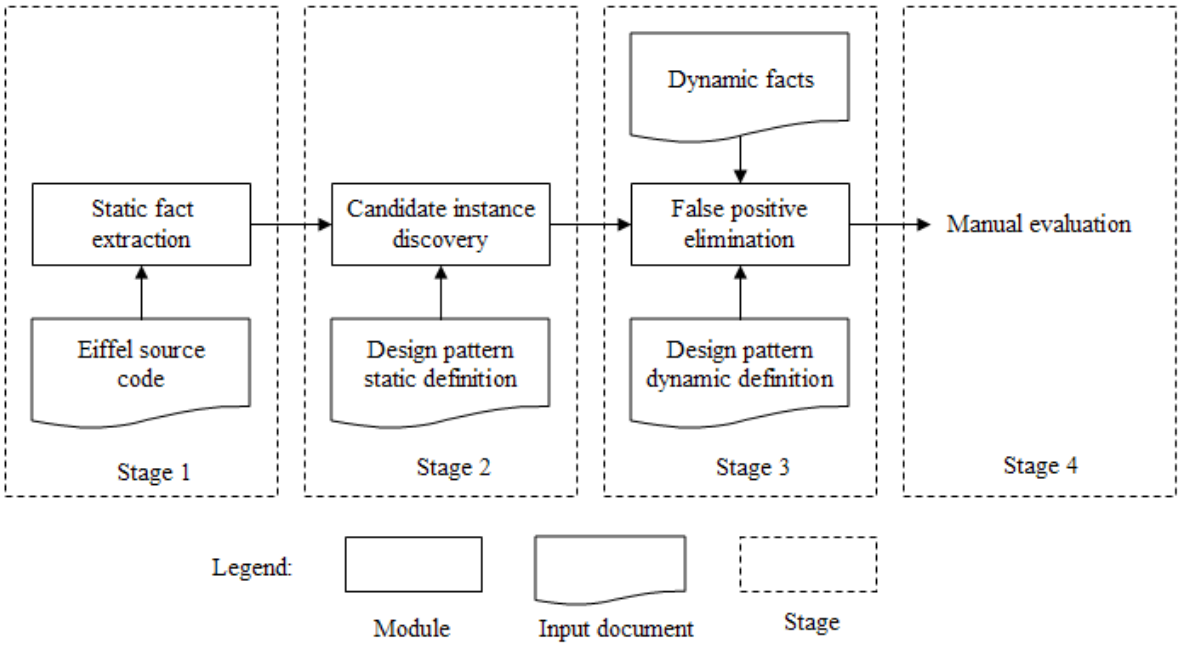


**Figure 1. The structure of our design pattern detection approach**

tern instances. According to [4], when the tool is applied on Java SwingSet Example and the tool itself, the number of false positives is small, in most experiments even zero.

In the next section, we present a tool called DPVK that detects design pattern instances in systems written in Eiffel by matching the system's static structure and dynamic behaviour to that of the design pattern.

## 3 DPVK: Design Pattern Verification toolKit

A key aspect of detecting design patterns is how to model and define a design pattern precisely and how to express that definition. In our approach, each design pattern has two definitions: one definition is based on the static structure of the pattern and the other is based on its dynamic behaviour. DPVK uses these two definitions to identify instances of design patterns, as well as to differentiate between different design patterns.

More specifically, DPVK compares these two definitions with the static and dynamic fact files of the target software system respectively. In theory, if both static and dynamic definitions are complete and precise, all instances of design patterns will be found and no false positives will be output. If we treat each definition as a set of conditions, then complete and precise definition means that no redundant conditions are included and no necessary conditions are missing.

In the real world, a particular design pattern may be implemented as a number of different variants in a given system. Although each variant can still be illustrated and analyzed statically and dynamically in a similar way, to collect all possible variants of each design pattern would be a time-consuming, and possibly never-ending task. Our solution is to create a design pattern definition repository which stores as many variant definitions as possible. It is clear that, the more complete this repository is, the better the chances we will detect design patterns and their variants in software systems.

As shown in Figure 1, our approach includes four stages: *static fact extraction*, *candidate instance discovery*, *false positive elimination*, and *manual evaluation*. The first three stages directly correspond to modules in DPVK's implementation. All four stages are presented in detail in the following subsections.

To help describe the process that DPVK uses in order to detect design patterns, we will present the detection of the Command pattern as a running example. The target software system will be the implementation of the Command pattern presented in [6]. We will refer to this implementation as the sample implementation.

The Command pattern wraps one or more methods in an object which can be passed to other methods or objects as a parameter and tell them to perform this particular operation in the process of fulfilling the request. Command is an entity that carries behaviour, rather than data. Figure 2 is the UML diagram published in [3]. Table 1 presents the mapping between the classes in Figure 2 and the classes in the sample implementation.

### 3.1 Stage 1: Static fact extraction

To simplify user-system interaction, DPVK only requires two inputs for its first stage.

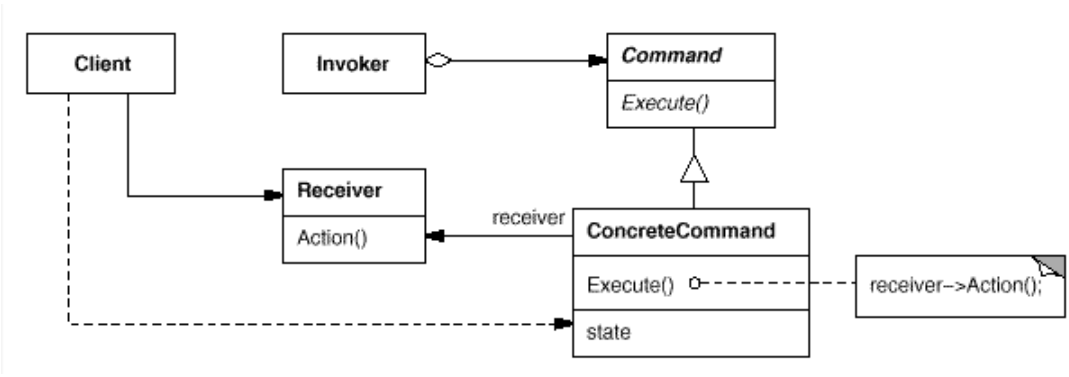


**Figure 2. Diagram of the Command pattern in GoF**

| Class name in GoF | Class name in the implementation |
|---|---|
| Client | MAIN |
| Invoker | BUTTON |
| Command | COMMAND |
| ConcreteCommand | QUIT_COMMAND |
| ConcreteCommand | SAVE_COMMAND |
| Receiver | EDITOR |

**Table 1. Class list of Command pattern**

The first input is description files for all classes in the target system. These files are created by running `ec` (Eiffel Compiler) with appropriate options. `ec` is a command-line Eiffel compiler developed by Eiffel Software [1].

The second input of DPVK is an ACE (Assembly of Classes in Eiffel) file. The ACE file contains important information about an Eiffel project, such as the location of all Eiffel source files and the root class name. This information is required to extract all static facts from the target implementation. Figure 3 presents the relevant part of the ACE file for the sample implementation.

From the sample implementation ACE file, DPVK extracts the following information about this implementation: the root class name is MAIN, and the directory of the source code is `~/behavior/command`. It is important to note that the trace option is set to yes. This setting will generate dynamic facts about this implementation when the system is run. These facts will be used in the stage of false positive elimination explained later.

DPVK looks in the directory specified above and finds out the names of all classes by scanning all .e files (Eiffel source code files). After that, DPVK parses the class description files generated by `ec` and outputs the static information of all classes. The static information is expressed in RSF format as a triple `relationship entity1 entity2` to interface with REQL in stage 2. Figure 4 gives the final output of this

```
root
    MAIN: "make"

default
    ...
    trace (yes)
    ...

cluster
    root_cluster: "~/behavior/command"
    ...
...
end
```

**Figure 3. The ACE file for the sample implementation (Partial contents)**

stage. Note that DPVK utilizes only inheritance and client-supplier relationships (`inherits` and `uses` respectively).

### 3.2 Stage 2: Candidate instance discovery

This stage requires the definition of the static structure of each design pattern we would like to detect. This definition is rendered as an REQL script and it defines the inheritance and client-supplier relationships among classes participating in the design pattern. Using REQL, we compare the definition with the facts about static relationships derived in stage 1. Figure 5 gives the static definition of the Command pattern.

Lines 1 to 5 are commented out, since this part is only useful to DPVK and REQL will bypass it. The first line gives the number of participants in the pattern. For the Command pattern, the number is four. The full names of all participants as well as short names to be used in this script are given in lines 2 to 5. The short names can be arbitrarily chosen. However, the full names must be the same as the names used in the dynamic definition (discussed in the next

```
inherits BUTTON WIDGET
uses BUTTON COMMAND
inherits MACRO_COMMAND COMMAND
inherits MACRO_COMMAND COMPOSITE
uses MACRO_COMMAND COMMAND
uses MAIN BUTTON
uses MAIN MACRO_COMMAND
uses MAIN QUIT_COMMAND
uses MAIN SAVE_COMMAND
uses MAIN EDITOR
inherits QUIT_COMMAND COMMAND
uses QUIT_COMMAND EDITOR
inherits SAVE_COMMAND COMMAND
uses SAVE_COMMAND EDITOR
```

**Figure 4. Static facts of the sample implementation**

```
1  //#roles=4
2  //ivk->invoker
3  //cmd->command
4  //conCmd->concreteCommand
5  //re->receiver
6  getdb($1)
7  DP[ivk,cmd,conCmd,re]=
8    {uses[ivk,cmd];
9     inherits[conCmd,cmd];
10    uses[conCmd,re]}
11 putdb($2,{"DP"})
```

**Figure 5. The static definition of the Command pattern in REQL script**

stage). Line 6 tells REQL to retrieve the factbase from the input file. Lines 7 to 10 are an REQL query that matches all sets of four classes that are connected in the same manner as the Command patterm structure. Line 11 defines the output file.

Once a set of classes that fits the static definition is found, we record it as a candidate instance. Eventually, the output lists all combinations of classes that fit the design pattern static structure. The output may contain false positives, since the static structure of many design patterns contains a small number of nodes and edges in a combination that is likely to be found in software systems that do not contain the particular design pattern.

Figure 6 presents the output of this stage. Line 1 indicates that four participants are involved in the Command pattern. Line 2 lists the names of these participants: `Invoker`, `Command`, `ConcreteCommand` and `Receiver`. Lines 3 to 8 give six pattern instances found by DPVK in this stage. The participants in each line are listed in the order defined in line 2.

```
1 roles number=4
2 Invoker Command ConcreteCommand Receiver
3 BUTTON COMMAND MACRO_COMMAND COMMAND
4 BUTTON COMMAND QUIT_COMMAND EDITOR
5 BUTTON COMMAND SAVE_COMMAND EDITOR
6 MACRO_COMMAND COMMAND MACRO_COMMAND COMMAND
7 MACRO_COMMAND COMMAND QUIT_COMMAND EDITOR
8 MACRO_COMMAND COMMAND SAVE_COMMAND EDITOR
```

**Figure 6. Result of stage 2**

```
1 roles#=3
2 1->Invoker
3 2->ConcreteCommand
4 3->Receiver
5 rules#=2
6 1->Invoker calls ConcreteCommand
7 2->ConcreteCommand calls Receiver
```

**Figure 7. Dynamic definition of the Command pattern**

### 3.3 Stage 3: False positive elimination

This step deals with the elimination of false positives in the output of the previous stage. For a large system, this step is quite important, since the number of false positives removed in this stage may be large.

This stage requires the dynamic definition of the design pattern being detected. The dynamic definition is based on the sequence of messages between classes expected during execution. Figure 7 gives the dynamic definition of the Command pattern.

The first line indicates that three participants are involved. Participant names are given in lines 2 to 4. Although the numbers are not mandatory, DPVK requires that the -> symbol must appear immediately before each participant's name. Line 5 gives the number of rules. Similarly, the -> symbol starts each line of rules. All rules are expressed as an ordered tuple, i.e. `caller calls callee`.

Besides the dynamic definition, we need to obtain dynamic behaviour facts for the target software system. When the call tracing option is turned on during compilation, a log file recording all method calls chronologically is generated during execution. DPVK manipulates the log file by removing calls to library classes that are irrelevant to design pattern detection, and transforming it into an ordered RSF for-

```
1  calls MAIN SAVE_COMMAND
2  calls MAIN QUIT_COMMAND
3  calls MAIN MACRO_COMMAND
4  calls MAIN MACRO_COMMAND
5  calls MACRO_COMMAND MACRO_COMMAND
6  calls MAIN MACRO_COMMAND
7  calls MACRO_COMMAND MACRO_COMMAND
8  calls MAIN BUTTON
9  calls MAIN BUTTON
10 calls MAIN BUTTON
11 calls MAIN MAIN
12 calls MAIN MAIN
13 calls MAIN BUTTON
14 calls BUTTON SAVE_COMMAND
15 calls SAVE_COMMAND EDITOR
16 calls EDITOR EDITOR
17 calls MAIN MAIN
18 calls MAIN BUTTON
19 calls BUTTON QUIT_COMMAND
20 calls QUIT_COMMAND EDITOR
21 calls EDITOR EDITOR
22 calls MAIN MAIN
23 calls MAIN BUTTON
24 calls BUTTON MACRO_COMMAND
25 calls MACRO_COMMAND SAVE_COMMAND
26 calls SAVE_COMMAND EDITOR
27 calls EDITOR EDITOR
28 calls MACRO_COMMAND QUIT_COMMAND
29 calls QUIT_COMMAND EDITOR
30 calls EDITOR EDITOR
```

**Figure 8. Dynamic facts extracted from the sample Command pattern implementation**

mat (i.e. `calls caller callee`) to facilitate instance searching and definition comparison. Figure 8 gives the dynamic facts extracted for the sample implementation.

DPVK checks each set of classes discovered in stage 2 by comparing the dynamic definition with the translated dynamic facts. Many false positives are removed in this process. We applied the dynamic definition of the Command pattern to the candidate set presented in Figure 6. We will take a closer look at the first two instances (lines 3 and 4 in Figure 6).

1. Candidate instance 1: By substituting the classes involved in this instance in the rules of the dynamic definition, we determine that the following two interactions have to take place if this is a true Command pattern instance.

   ```
   rule 1: BUTTON calls MACRO_COMMAND
   rule 2: MACRO_COMMAND calls COMMAND
   ```

   DPVK checks the dynamic facts in Figure 8 and finds rule 1 in line 24 but does not find rule 2. As a result, this candidate instance is deemed a false positive, and is removed from the candidate set since it does not comply to both rules.

2. Candidate instance 2: The rules of the dynamic definition in this case become:

   ```
   rule 1: BUTTON calls QUIT_COMMAND
   rule 2: QUIT_COMMAND calls EDITOR
   ```

   The facts complying to both rules can be found in line 19 and 29 respectively. Therefore, this candidate instance is kept in the candidate set since it satisfies the dynamic definition of the Command pattern.

```
1 4 Design Pattern instances are found.
2 Invoker Command ConcreteCommand Receiver
3 BUTTON COMMAND QUIT_COMMAND EDITOR
4 BUTTON COMMAND SAVE_COMMAND EDITOR
5 MACRO_COMMAND COMMAND QUIT_COMMAND EDITOR
6 MACRO_COMMAND COMMAND SAVE_COMMAND EDITOR
```

**Figure 9. Result of stage 3**

The final output of DPVK can be found in Figure 9. This output indicates that a total of four pattern instances are found. Comparing with the output from stage 2, which performs only static fact checking, we can see that two false positives are eliminated by checking the candidates' dynamic behaviour in this stage.

### 3.4 Stage 4: Manual evaluation

This is the final stage of the pattern detection process. Design patterns can be implemented in many different ways. Also, the same pattern can occur in many parts of the source code, but each time with a different intent. To increase DPVK's successful detection rate, this stage solicits feedback from experienced developers. This stage lets developers manually verify and justify the candidate instances from the output of stage 3. Their knowledge and experience could be helpful in filtering out unwanted output, and in discovering undetected pattern instances.

In the sample implementation, observation of the output from stage 3 indicates that two of the reported instances are not true Command pattern instances. These two instances correspond to lines 5 and 6 in figure 9.

They comply to both static and dynamic definitions. However, they are not Command pattern instances since clearly `MACRO_COMMAND` is not an `Invoker`.

Interestingly, when we take a closer look, we find that these instances are Composite Commands, i.e. the combination of the Composite pattern and the Command pattern. Therefore, if our dynamic definition is expanded to include

this particular variant of the Command pattern, they can also be identified as correct Command pattern instances.

Based on this observation, we conclude that checking both static structure and dynamic behaviour does help us to find pattern instances and reduce the number of false positives. However, we still can not eliminate all false positives and find all true negatives. This is due to the following aspects:

1. Each pattern can be implemented in different forms, also known as variants, depending on the problem context. In this case, it is possible to have true negatives.

2. Some false positives may have exactly the same structure and behaviour as a true pattern instance, as evidenced in the detection of the Command pattern in the sample implementation. In this case, false positives are still included in the final output.

3. Currently, DPVK does not take certain factors into consideration, such as the signature of methods, the class type (concrete/abstract), language keywords etc. For example, the `ONCE` keyword in Eiffel helps developers implement the Singleton pattern. Therefore, its identification is essential to the detection of the Singleton pattern. However, DPVK does not recognize the `ONCE` keyword yet. In order to effectively eliminate false positives DPVK requires more detailed constraints, such as information about in which class the `ONCE` keyword is used.

## 4 Case studies

In this section, we apply DPVK to three different sets of implementations that contain known instances of design patterns.

### 4.1 Case study 1: Sample implementations

All GoF patterns have been implemented in Eiffel in [6]. We applied DPVK to 18 of these implementations. Figure 10 presents the results of our experiments. In total, 414 tests were conducted. Five GoF patterns were not tested: Prototype, Singleton, Facade, Chain of responsibility, and Template Method. The reasons are:

1. Prototype. This pattern is implemented in class ANY (all Eiffel classes inherit from ANY). It does not make much sense to detect this pattern in an Eiffel system.

2. Singleton, Chain of Responsibility and Template Method. These three patterns have simple static structure and dynamic behaviour. Many false positives will be generated if we only consider the definitions based on their static structure and dynamic behaviour. More detailed information is required for their successful detection.

3. Facade. This pattern is hard to define due to the complexity of its static structure and dynamic behaviour. The Facade pattern provides a unified interface to a set of interfaces in a subsystem. There are simply too many combinations to test in a non-trivial system.

In Figure 10, if no pattern instances are found, the corresponding cell is left empty. Some cells contain three numbers and they are listed as A/B/C. A is the number of true positives, i.e. the number of detected pattern instances that were correctly identified. B is the number of false positives, i.e. the number detected incorrectly. C is the number of true negatives in the result, i.e. pattern instances that DPVK missed.

The cells in the diagonal of the table in Figure 10 are shaded. These shaded cells present the result of tests that detect a pattern in its corresponding implementation. The diagonal appears broken due to the fact that we did not attempt to detect 5 patterns, as discussed above.

Since these implementations are rather small, it took no more than ten seconds in an AMD3200/1G RAM computer running Windows XP for DPVK to perform pattern detection in each of them. For the same reason, the manual evaluation stage required only up to ten minutes depending on the complexity of the target implementation and the pattern being detected.

Based on the collected test results, we can make the following observations:

1. Although DPVK reported false positives in some implementations, there were no true negatives. This fact indicates that DPVK did not miss any pattern instances in the detection process.

2. In implementations containing instances of different patterns, DPVK was able to find all these patterns. For example, implementations of Facade, Flyweight, Command and Visitor include the Composite pattern. According to Figure 10, DPVK can recognize the Composite pattern instances in these implementations.

3. Since the implementation of the Visitor pattern is rather complicated, the results indicate a larger number of false positives in this implementation than that in other implementations. Also, since the Interpreter pattern's definitions are simple, many false positives of Interpreter pattern instances were found among the implementations of various patterns. Therefore, the complexity of the targeted system and the pattern itself both have impact on the detection results of DPVK.

4. The recall and precision are 100% and 19.9% respectively. Since creational patterns have comparatively simple static structures and less dynamic interactions, their false positives are more common in our test. On

| Targeted patterns | Tested implementations (Part I) | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Abstract factory | Builder | Factory method | Prototype | Singleton | Adapter | Bridge | Compo-site | Deco-rator | Facade | Flywei-ght | Proxy |
| Abstract Factory | 4/0/0 | | | 4/2/0 | 2/0/0 | | | | | | | 0/1/0 |
| Builder | | 3/3/0 | | | | | 0/1/0 | | | | 0/2/0 | 0/3/0 |
| Factory Method | 0/4/0 | | 2/2/0 | 0/3/0 | | | | | | | | |
| Adapter | | | | 0/6/0 | | 1/0/0 | | | | | 0/3/0 | 0/2/0 |
| Bridge | 0/1/0 | | | 0/2/0 | | | 4/0/0 | | 0/1/0 | | | |
| Composite | | | | | | | | 2/0/0 | 0/1/0 | 0/1/0 | 1/1/0 | |
| Decorator | | | | | | | | 0/2/0 | 1/0/0* | | | |
| Flywreight | | | | | | | | | | | 1/1/0 | |
| Proxy | | | | | | | | | | | | 1/0/0 |
| Command | | 0/3/0 | | | | | | | | | | 0/2/0 |
| Interpretor | 0/5/0 | 0/6/0 | | 0/6/0 | | 0/4/0 | | | | 0/1/0 | 0/4/0 | 0/3/0 |
| Iterator | 0/2/0 | | | 0/4/0 | | | | | | | 0/1/0 | 0/2/0 |
| Mediator | | | | | | | | | | | | |
| Memento | | | | | | | | | | | | |
| Observer | | | | | | | | | | | | |
| State | 0/1/0 | 0/4/0 | | 0/1/0 | | | | | | 0/1/0 | | |
| Strategy | | | | | | | | | | | 0/1/0 | 0/2/0 |
| Visitor | | | | | | | | | 0/1/0 | | | |

| Targeted patterns | Tested implementations (Part II) | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Command | Interpretor | Iterator | Mediator | Memento | Observer | Chain of reponsibility | State | Strategy | Template method | Visitor |
| Abstract Factory | | | | 0/3/0 | 0/1/0 | 0/2/0 | | | | | 0/8/0 |
| Builder | 0/2/0 | 0/1/0 | | | | | 0/1/0 | 0/2/0 | | | 0/3/0 |
| Factory Method | | | | | | | | | | | |
| Adapter | | | | | 0/2/0 | 0/2/0 | | 0/2/0 | | | |
| Bridge | | | | 0/2/0 | | | | | | | 0/4/0 |
| Composite | 1/0/0 | 0/2/0 | | | | | 0/2/0 | | | | 5/3/0 |
| Decorator | | | | | | | | | | | 0/2/0 |
| Flywreight | | | | | | | | | | | |
| Proxy | | | | | | | | | | | |
| Command | 2/2/0 | 0/1/0 | | | | | 0/1/0 | 0/1/0 | | | 0/4/0 |
| Interpretor | | 1/0/0 | | 0/5/0 | 0/6/0 | | | 0/2/0 | | | 0/8/0 |
| Iterator | | | 2/0/0 | | | | | | | | |
| Mediator | | | | 4/0/0 | | | | | | | 0/2/0 |
| Memento | | | | | 0/1/0 | | | | | | |
| Observer | | | | | | 2/0/0 | | | | | |
| State | | | | | | | | 3/0/0 | | | |
| Strategy | | 0/1/0 | | | | | | 0/2/0 | 2/0/0 | | |
| Visitor | | | | | | | | | | | 8/0/0 |

Legend: A/B/C A-#Ture positives, B-#False positives, C-#True negatives
* Notes: The dynamic definition of the Decorator pattern only works for one level of decoration. In order to find other instances, the definition have to be modified

**Figure 10. Results for case study 1**

the other hand, most structural and behavioural patterns contain the special traits of their static structure or dynamic interactions respectively, which helps DPVK filter out false positives.

## 4.2 Case study 2: Bridge pattern in EiffelVision

In order to get a better understanding of the effectiveness and efficiency of DPVK, we had to test it with a larger system. The Vision library in EiffelStudio V5.4 is an ideal system, since it is quite large and the documentation indicates that the Bridge pattern has been used extensively. We chose an example implementation shipped with EiffelStudio v5.4 called Accelerator.

Figure 11 shows the static definition of the Bridge pattern. After applying the Bridge static definition, DPVK found 3941 candidate instances. Since we detect the Bridge pattern in EiffelStudio's Vision library, the number of candidate instances is rather large compared to the first case study.

Figure 12 shows the dynamic definition of the Bridge pattern. $\star$ is a wild card used to represent any class in the system other than the callee class(`refinedAbstraction` in the first rule). ==> adds a restriction that the next rule must happen as a sub-message of the current rule. This restriction confirms the call sequence and avoids the following case: Suppose the two calls did happen in the chronological order as defined in these rules. However, the second call was not triggered by the first call, and thus these two calls are not relevant. Without this restriction, the instances complying to the rules will be

```
//#roles=4
//rfAbs->refinedAbstraction
//abs->abstraction
//imp->implementer
//conImp->concreteImplementer
getdb($1)
ancestor=inv inherits
DP[rfAbs,abs,imp,conImp]=
  {inherits[rfAbs,abs];i
   uses[abs,imp];
   ancestor[imp,conImp]}
putdb($2,{"DP"})
```

**Figure 11. The static definition of the Bridge pattern**

```
roles#=2
1->refinedAbstraction
2->concreteImplementer
rules#=2
1->* calls refinedAbstraction==>
2->refinedAbstraction calls
             concreteImplementer
```

**Figure 12. The dynamic definition of the Bridge pattern**

reported as a pattern instance. However, this would likely be a false positive in this context.

After the false positive elimination stage, only two instances were correctly reported as instances of the Bridge pattern. Furthermore, detailed analysis indicated that the extra restriction described above did remove two instances that would have otherwise be reported incorrectly as Bridge pattern instances.

### 4.3 Case study 3: Student implementations

We collected many pattern implementations developed by students in the Computer Science department of York University. As part of an assignment for a Software Desing course, students had to implement three patterns on an existing system: Decorator, State and Visitor. We will call the implementation of the Decorator pattern Task D, that of the State pattern Task S, and that of Visitor Task V. All three tasks were supposed to result in an implementation with the same external behaviour as the original system. The difference between them is which pattern is implemented.

The students had a variety of choices to make on how to implement the patterns in the three tasks. For example, they could modify the existing classes and create new classes or other supporting resource files, as long as they encompass the three patterns in their implementations.

| Pattern | Task D | Task S | Task V |
|---|---|---|---|
| Decorator | 41 | - | - |
| State | - | 42 | - |
| Visitor | - | - | 45 |
| Total implementations | 58 | 54 | 62 |

**Table 2. Detection results for the three patterns**

After removing some implementations that generated compile time errors or provided wrong output, we had 58, 54, and 62 valid implementations for the Decorator, State, and Visitor patterns respectively.

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

We started with three separate experiments: Detection of the Decorator in task D, detection of the State in task S, and detection of the Visitor in task V. The results are shown in Table 2. Moreover, besides these three patterns, we detected the remaining 15 patterns, that were also tested in case study 1, in the three tasks. Table 3 shows the results. The second column is the percentage of experiments that resulted in false positives. To assist in the comparison between the results of case study 3 and case study 1, the third column gives the false positive percentage for case study 1.

Based on these results, we can make the following conclusions:

DPVK successfully conducted the detection of the three patterns. It detected the appropriate pattern in 73.6% of the student implementations. DPVK effectively found these three patterns while maintaining low false positives. However, we should note that the test implementations were not very large in size. The amount of classes in each implementation is between 20 and 30. The low system complexity helps to reduce the rate of false positives and increase the hit rate of true pattern instances.

As indicated in Table 2, not all pattern instances were detected. There are two possible reasons: Either, the students did not implement the patterns correctly, or they implemented pattern variants that DPVK does not detect. As mentioned earlier, students were allowed to choose different ways to implement the same patterns in this project. In order to detect different variants, DPVK requires additional static/dynamic definitions that correspond to the static structure and dynamic behaviour of the variants. In all experiments, we provided DPVK with the standard definitions, which are based on one of the most common variants. Therefore, some variants will not be found.

| Pattern name | FP% - CS3 | FP% - CS1 |
|---|---|---|
| Abstract Factory | 36.8% | 34.8% |
| Builder | 39.7% | 34.8% |
| Factory Method | 41.4% | 13.0% |
| Adapter | 11.5% | 26.1% |
| Bridge | 20.1% | 21.7% |
| Composite | 24.1% | 21.7% |
| Decorator | 4.3% | 8.7% |
| Flyweight | 1.7% | 0% |
| Proxy | 0% | 0% |
| Command | 54.6% | 30.4% |
| Interpreter | 50.6% | 47.8% |
| Iterator | 55.2% | 13% |
| Mediator | 4.0% | 4.3% |
| Memento | 8.6% | 0% |
| Observer | 0% | 0% |
| State | 20.8% | 17.4% |
| Strategy | 18.7% | 17.4% |
| Visitor | 0% | 4.3% |

**Table 3. Detection results for all patterns**

In the second part of this case study, we found the false positive rate in case studies 1 and 3 to be congruent except for few patterns. Among the three types of patterns, the structural and behavioral patterns are more "detectable", since the false positive rates in both case studies 1 and 3 are lower than that of the creational patterns. However, the Command and Iterator patterns have quite high false positive rates in this case study. The rate is even higher than that of case study 1. The major reason is that the implementations in case study 1 were much simpler. This fact refers not only to the number of classes but also to the number of interaction dependencies.

## 5 Conclusion

We presented an approach to detecting design pattern instances in Eiffel systems that takes both static structure and dynamic behaviour into account. We also presented a reverse engineering tool called DPVK that implements this approach and has been applied to real Eiffel systems. The results of three different case studies indicate that DPVK is quite effective in detecting design pattern instances while eliminating false positives.

Several possibilities for further research present themselves:

We tested DPVK with the Vision library shipped with EiffelStudio, a rather large system. However, there is definitely room for more tests. For example, we would like to test DPVK with an even larger system which contains different kinds of design patterns. The result would be helpful for us to further refine the design and increase the effectiveness and efficiency of DPVK in pattern detection.

Currently, DPVK does not take method signatures and class types into account. This additional information would help us detect pattern instances in a more precise manner. In terms of dynamic behaviour, we would like to be able to distinguish creation calls from method calls with other purposes.

Finally, the current user interface for DPVK is textual. A graphical user interface that can depict discovered instances in UML or BON notation would be quite helpful, especially for the manual evaluation phase.

## References


[1] `http://www.eiffel.com`.

[2] K. Brown. Design reverse-engineering and automated design pattern detection in smalltalk. 1996.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, 1995.

[4] D. Heuzeroth, T. Holl, G. Hogstrom, and W. Lowe. Automatic design pattern detection. In *Proceedings of the Eleventh International Workshop on Program Comprehension*, pages 94–103. IEEE Computer Society Press, 2003.

[5] J. H. Jahnke and J. P. Wadsack. A history concept for design recovery tools. In *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering*, pages 37–46. IEEE Computer Society Press, 2002.

[6] J.-M. Jezequel, M. Train, and C. Mingins. *Design Patterns and Contracts*. Addison-Wesley, 1999.

[7] R. Keller, R. Schauer, S. Robitaille, and P. Page. Pattern-based reverse-engineering of design components. In *Proceedings of the 21st International Conference on Software Engineering*, pages 226–235. IEEE Computer Society Press, 1999.

[8] C. Kramer and L. Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In *Proceedings of the Tenth Working Conference on Reverse Engineering*, pages 208–215. IEEE Computer Society Press, 1996.

[9] H. Mei, T. Xie, and F. Yang. Jbooret: an automated tool to recover oo design and source models. In *Proceedings of the 25th Annual International Computer Software and Applications Conference*, pages 71–76. IEEE Computer Society Press, 2001.

[10] J. Seemann and J. W. von Gudenberg. Pattern-based design recovery of java software. In *Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 10–16. ACM Press, 1998.

[11] F. Shull, W. Melo, and V. Basili. An inductive method for discovering design patterns from object-oriented software systems. *Technical Report*, 1996.

[12] J. Wu, A. E. Hassan, and R. C. Holt. Using graph patterns to extract scenarios. In *Proceedings of the Tenth International Workshop on Program Comprehension*, pages 239–250. ACM Press, 2002.