DPVK - An Eclipse Plug-in to Detect Design Patterns in Eiffel Systems

Wei Wang, Vassilios Tzerpos[1,2]

Department of Computer Science
York University
Toronto, Canada

Abstract

Design patterns are not only beneficial to the forward engineering process but also help in design recovery and program understanding, typical reverse engineering activities. In this paper, we introduce DPVK, a reverse engineering tool to detect pattern instances in Eiffel systems. In order to get better detection results, we analyze many different patterns and examine Eiffel software in terms of both static structure and dynamic behaviour. DPVK is implemented as an Eclipse plug-in to ensure better compatibility and extensibility.

Keywords: BON, Design Pattern, Eclipse, Eiffel, Grok, Reverse Engineering

1 Introduction

Design patterns have attracted much 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.[2] capture solutions that have developed and evolved over time. Each design pattern indicates a high level abstraction, encompasses expert design knowledge, and represents a so-

---

[1] Email: weiw@cs.yorku.ca
[2] Email: bil@cs.yorku.ca

lution 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 program understanding and design recovery perspective, patterns provide the organization of a system and information about the role of each component of the system. Meanwhile, patterns can also indicate the design rationale behind the system's implementation.

Usage of design patterns improves the understandability of object-oriented designs. Based on this observation, since a design pattern reflects the interconnection between a certain design and the purpose behind the design, finding instances of design patterns in a system can potentially yield the motivation for the system's design. A developer can get a better understanding of a software system by studying the design patterns within it.

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.

Although different tools may have different structures, all of them have to address several basic but critical issues. The three most important issues are: how to define patterns, how to present patterns, and how to store patterns.

Definition is the kernel of these issues since it reflects the traits of the patterns and differentiates one pattern from another. 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 reverse engineering tools to find correct pattern instances without missing any. The more redundancy in a pattern's definition, the more false positives could be reported. A typical way to define a pattern is to simply consider the composition of its UML diagram. However, the trend is to define patterns from both their static structure and dynamic behaviour. This way of defining patterns not only increases completeness but also reduces redundancy.

Pattern presentation refers to the format used to facilitate documentation and communication activities. The patterns can be presented visually in the form of a graph, or textually using some sort of description language.

Pattern storage concerns the form by which the pattern definition and its instances are stored. In recent research, relational databases are commonly used to store detected pattern instances and other recovered design information.

The challenge of fully automatic pattern detection is how to automatically

eliminate most false positives of pattern instances. Many recently developed tools use multiple passes[18], or recursive filtering[10], to improve the detection output. Meanwhile, both static structure and dynamic interaction[5] have been taken into account in order to increase the hit rate and reduce the missing rate of true pattern instances.

In general, design pattern usage in reverse engineering, especially pattern detection, is still in its infancy and has plenty of room to improve. Since this field appears to attract much attention recently, we could expect more effective and robust pattern detection tools to be developed in the future.

The purpose of this paper is twofold: to present a catalogue of design pattern diagrams, and to propose a reverse engineering tool—DPVK(Design Pattern Verification toolKit). The former shows both static structure and dynamic behaviour of design patterns and the latter is used to detect/verify patterns in Eiffel systems. It is developed in Java to enhance its portability to various platforms, and to allow easier integration with Eclipse.

Eclipse, as introduced by Gamma and Beck[3], can be explained as a technology, a development platform, or a group of tools. It is also an ecosystem that enables diversified contributors to interact with each other. Therefore Eclipse is a collection of both plug-ins and access points of plug-ins. To take advantage of the latest development techniques and to enhance DPVK's extensibility, we chose the Eclipse framework as our development platform and decided to implement DPVK as an Eclipse plug-in.

An Eiffel Development Tool for Eclipse is currently under development in York University. Once it is available, DPVK can interact with it and be more efficient and productive in detecting pattern instances.

The paper is organized as follows. In section 2, we conduct a survey of related reverse engineering tools. Most of these tools are capable of detecting design patterns. However, they are designed in a different context and show various levels of availability, extensibility, efficiency, and effectiveness of pattern detection. Section 3 presents some background on design patterns and the Eiffel language. Related development tools are also introduced. In section 4, we present the catalogue of pattern diagrams and explain how we compose it. Section 5 introduces DPVK's design and implementation. Certain parts of this section describe ongoing work. Section 6 concludes the paper and discusses our future work.

2 Design Pattern Detection Tools

Design pattern detection tools and techniques have been catalogued in various ways in the literature. The evaluation of the tools and techniques focuses

on their supported language, analysis approach, by-products (e.g., function reports, call graphs, data flow diagrams)[4] and effectiveness/efficiency. In this paper, we discuss several design pattern detection tools simply in chronological order.

(i) Pat[12]: 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.

(ii) KT[1]: 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 the artifact of design patterns must support both static and dynamic modelling constructs. In terms of diagram, static information includes is-kind-of and has-a relationship 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.

(iii) Seemann-Gudenberg[17]: The authors present some ideas on how to recover design information from Java source code. The pattern recognition approach proceeds step by step and different layers of abstraction are revealed. 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.

(iv) SPOOL[13]: The SPOOL (Spreading Desirable Properties into the Design of Object-Oriented, Large-Scale Software Systems) project is a joint industry/university collaboration. As a part of the project, the SPOOL environment for design pattern engineering supports both forward and reverse engineering of design patterns. The SPOOL reverse engineering

environment has a three tier architecture. From bottom to top, they are object-oriented database, repository schema and end-user tools. The lowest tier provides physical storage of the reverse engineering model and design information. The middle tier contains the object-oriented schema of the reverse engineering model, comprising static structure and dynamic behaviour. The upper-tier consists of end-user tools implementing domain specific functions, such as source code capturing and visualization analysis.

(v) JBOORET[15]: 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.

(vi) Heuzeroth-Holl-Hogstrom-Lowe[5]: 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 convention. 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

protocol conformance of patterns. So far the tool can detect Observer, Composite, Mediator, Chain of Responsibility and Visitor Patterns. According to [5], 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.

3   Design Patterns and Eiffel

Classes in each design pattern interact in a precise manner and each class is expected to show proper behaviour. Eiffel provides a mechanism, namely, Design by Contract[16], to define such mutual obligations and benefits among classes. Design by Contract has a sound theoretical basis, and it provides a guideline for constructing robust designs. It lets developers precisely specify the obligations and explicitly assign the responsibilities of clients and suppliers. In the language level, assertions, precondition and postcondition clauses, are used to define contracts between clients and suppliers. Moreover, class invariants enable the definition of general consistency properties within each class. A failure to comply with the contract indicates a bug. This can help developers discover and deal with such design bugs.

Eiffel is a language with a built-in Design by Contract mechanism as well as a language giving developers a full object-oriented design approach. Moreover, Eiffel offers multiple inheritance, polymorphism, static typing and dynamic binding, genericity and safe exception handling.

3.1   EiffelStudio

There are many Eiffel compilers available in the market, some of them are free. Among those compilers, SmallEiffel (or the GNU Eiffel Compiler, now SmartEiffel)[19] and ISE's EiffelStudio[9] are the most popular.

EiffelStudio is a powerful tool for realizing the full strength of the Eiffel language. EiffelStudio contains an IDE. However, it is more than just an IDE.

System modelling, designing, implementing and debugging can be streamlined and done within EiffelStudio. Developers never need to switch between multiple development environments to fulfill those tasks. Eiffel's built-in Design by Contract mechanism can prevent many of the bugs from ever occurring and the remaining bugs can be easily traced and fixed. Also, developers do not need extra tools to change the system architecture. Built-in testing, metrics and productivity tools enable easy and safe development.

Since EiffelStudio runs on Windows, Unix, Linux, embedded, and even VMS systems, developers can create an application in Eiffel and then migrate

it to any other platform Eiffel compiles to. Moreover, the latest version of EiffelStudio supports Microsoft's .NET framework.

Even though EiffelStudio lacks many of the features provided by Eclipse, such as online compilation, source control, and refactoring, it still provides a number of facilities that make development of Eiffel systems easier and more efficient. When the Eiffel Development Tool for Eclipse is ready, we plan to integrate our work with it.

In this paper, we chose the EiffelStudio 5.3 environment to take advantage of its IDE and built-in support of BON (Business Object Notation). BON [20] is a notation for analysis and design of object-oriented systems, which emphasizes seamlessness, reversibility and software contracting. There are two major reasons we chose BON for our work. First, BON is integrated with EiffelStudio. EiffelCase, an integrated tool of EiffelStudio directly interfaces with the other tools of the ISE environment and supports generation of new system architectures as well as reverse engineering of existing ones. Second, BON has similar function with UML while the former is easier to learn and use. BON is based on concepts similar to those of Eiffel but can be used independently of Eiffel.

3.2 BON CASE tool

The BON CASE tool[14] is designed to create a seamless interface between the stages of planning, coding, and documenting. It is a self contained system which provides facilities for dealing with use case diagrams, class diagrams, and dynamic diagrams. This tool provides a system tree which allows easy access to all diagrams and documentation of an Eiffel project. According to the author's plans, future revisions will allow developers to forward engineer source code into languages such as Java and Eiffel from the diagrams one can build with this tool, or to reverse engineer a class diagram from Eiffel source code. Since EiffelStudio does not provide facilities to draw dynamic BON diagrams, we use the BON CASE tool to draw dynamic diagrams and make accompanied notations in this paper.

4 Pattern diagram catalogue

In order to get a better understanding of GOF patterns and to detect them in Eiffel source code, we analyzed both their static structure and dynamic behaviour. As a by-product, a diagram catalogue of GOF patterns is compiled[22]. In this catalogue, each design pattern is analyzed and presented from both static and dynamic perspective. The static diagram illustrates the pattern structure, especially the inheritance and invocation relationships between com-

ponent classes. On the other hand, the dynamic diagram indicates message sequence at run time.

Jezequel et al.[11] implemented all GOF[2] design patterns in Eiffel. We use these Eiffel implementations, EiffelStudio and the BON CASE tool to analyze and generate a series of diagrams of GOF patterns. The Eiffel implementations were based on the SmallEiffel compiler and they used a different Eiffel base library from the one of EiffelStudio. Therefore, before creating the diagram catalogue, we modified these implementations and made them compatible with the EiffelStudio base library while keeping the original design. After that, aided by the integrated diagram tool of EiffelStudio, we generated static diagrams of each pattern and removed unnecessary component classes from the diagrams to keep them simple and complete.

Contrary to the process of creating static diagrams, creation of dynamic diagrams is done mostly manually. There are three steps to draw a dynamic diagram: dynamic fact extraction, operation elimination and BON diagram drawing. In the first step, we execute the Eiffel system in question, and collect information about every feature call in chronological order (the executable has been created by compiling with the call tracing option turned on). Next, similar to what we did with static diagrams, we need to simplify the output script since some operations are unrelated to the patterns. By reference to the UML collaboration diagrams in [11] we remove unrelated operations and make sure all pattern related operations remain. Finally, we draw the dynamic diagrams manually using the BON CASE tool. To make the diagrams more understandable, we give a detailed description of each operation in the diagrams.

Each dynamic diagram consists of two parts: the first part is a graph expressing the sequence of operations within the pattern and the second part is a notation that provides a detailed description of each operation. We present the diagrams for the Singleton pattern as an example. The following figure is the static diagram for the Singleton Pattern.
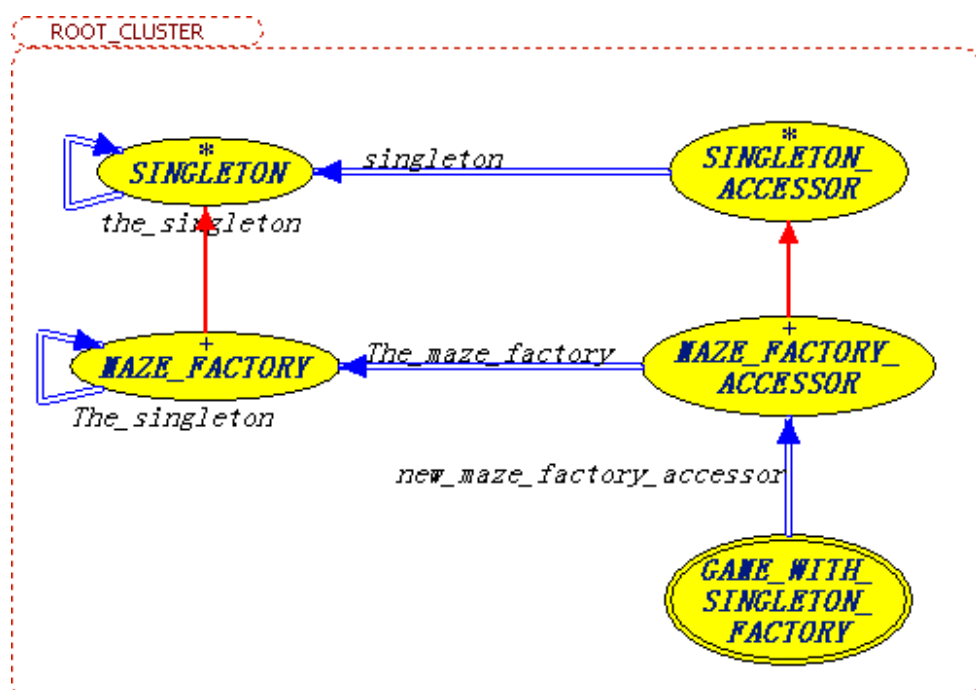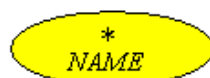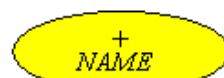
Legend
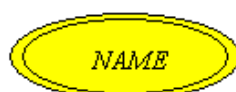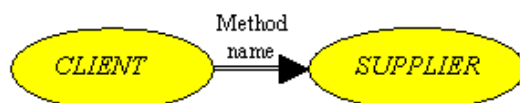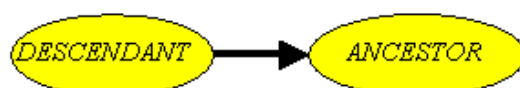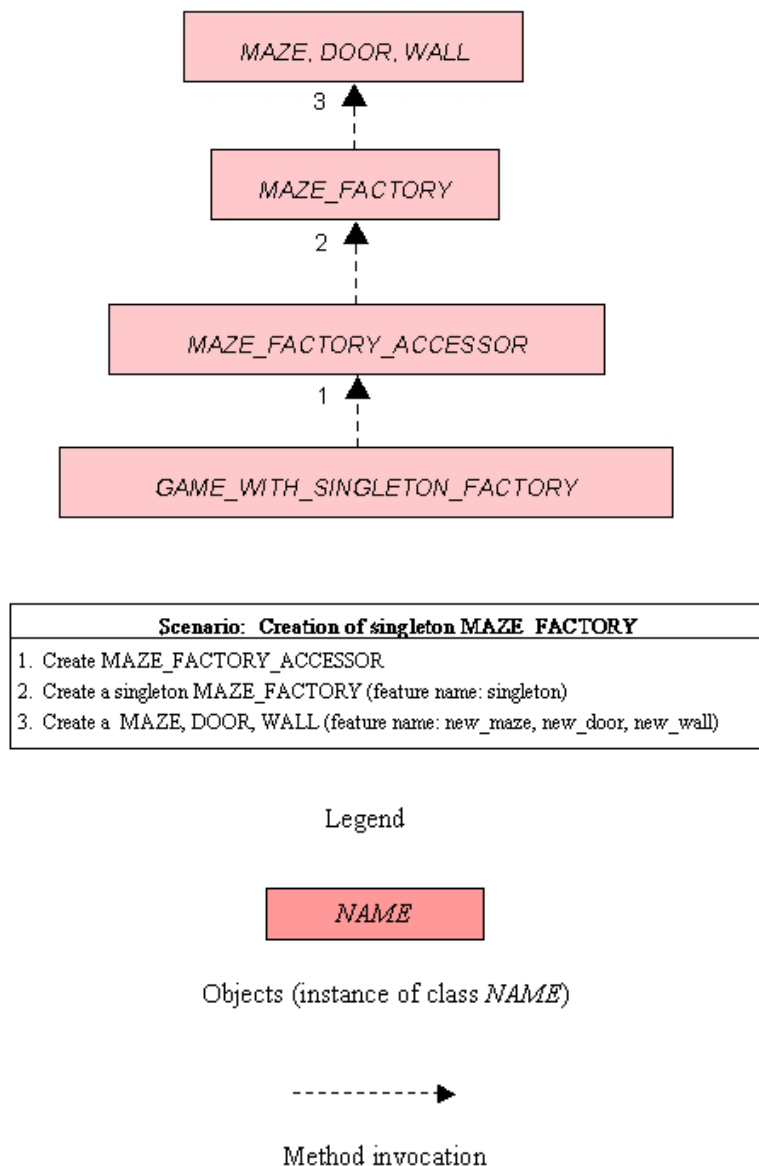

Root class

Deferred class   Effective class

Client/supplier relationship

Inheritance relationship

Following is the dynamic diagram for the Singleton Pattern.

MAZE, DOOR, WALL

3 ▲
⋮

MAZE_FACTORY

2 ▲
⋮

MAZE_FACTORY_ACCESSOR

1 ▲
⋮

GAME_WITH_SINGLETON_FACTORY

Scenario: Creation of singleton MAZE FACTORY
1. Create MAZE_FACTORY_ACCESSOR
2. Create a singleton MAZE_FACTORY (feature name: singleton)
3. Create a MAZE, DOOR, WALL (feature name: new_maze, new_door, new_wall)

Legend

NAME

Objects (instance of class *NAME*)

- - - - - - - - - - ▶

Method invocation

5   DPVK—A toolkit to detect design patterns in Eiffel

Reverse engineering tasks are commonly done by means of text-based search tools such as *grep*, *sed*, or *awk*, or by querying graph representations of source code, such as an ASG (Abstract Syntax Graph). Our implementation, DPVK, is an integrated text-search tool that detects design patterns in Eiffel programs. DPVK is written in Java to gain better compatibility and portability for multiple operating systems and development environments, such as Eclipse.

We begin by presenting Grok, a fact manipulation engine that we use for our purpose.

5.1   Grok

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.

Grok extracts facts from a factbase—a list of relations written in RSF (Rigi Standard Format). The factbase collects all relationships between each entity in a universe (it could be a single file or several files). 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). Grok provides set operators similar to that of a relational database. Those operators mainly support set operations, such as union, intersection, subsection, comparison and so on. The user can create a script and let the Grok interpreter perform a batch of operations upon a factbase. Grok will calculate, filter and output qualified sets of entities which have the relationships defined by the Grok script. Grok has many other powerful features[6].

In DPVK, Grok is an ideal language to manipulate static relationships among classes and objects, since an extension presented in [21] allows for efficient graph pattern matching. Each pattern static structure is defined as several pattern rules, which are expressed as RSF triples. In DPVK, we use the extension to match the design pattern's static definition with static facts extracted from Eiffel implementations. Grok captures the interactions among a set of components and generates the candidate set of pattern instances. In terms of static structure, DPVK takes inheritance and method invocation relationships between classes into account.

5.2 Design

A key aspect of detecting a design pattern is how to model and define a design pattern precisely and how to express that definition. In DPVK, 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 the special structure and behaviour to identify and pinpoint design patterns and differentiate a design pattern from another.

More specifically, DPVK compares these two definitions with the static and dynamic fact files of the target software system respectively. This approach is similar to the one used in [5]. Theoretically, if both static and dynamic definitions are complete and precise, all 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 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 an endless work as software systems evolve. 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 the factbase, the better the chances we will find design patterns and their variants in software systems.

5.3 Implementation

As shown in the following figure, DPVK includes three modules and runs in four stages. These modules/stages are *static fact extraction*, *candidate instance discovery* and *false positive elimination*. The fourth stage is the manual evaluation of the results of our approach. In the following we discuss these four stages in detail.
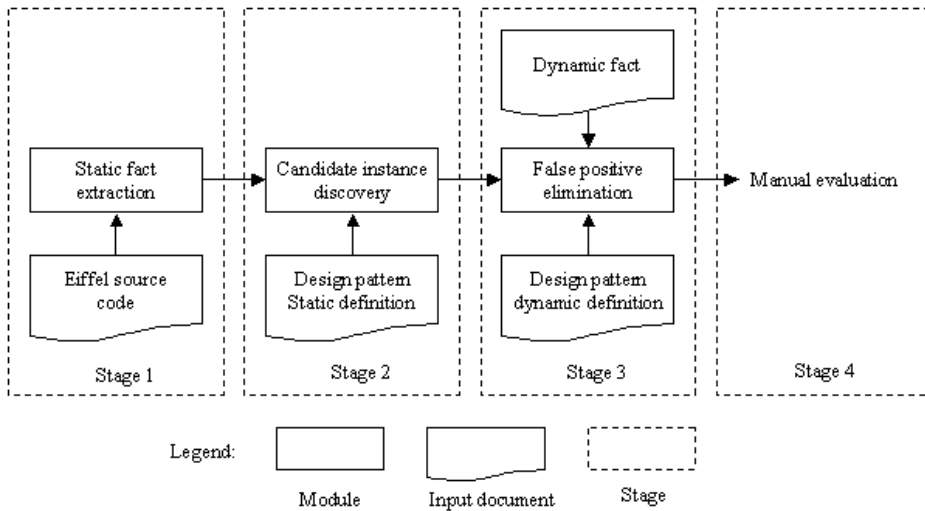
5.3.1   Static fact extraction

In the first step, Eiffel source code is the only input. To simplify user-system interaction, DPVK only requires 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 needed facts from the Eiffel source code.

The static fact extraction navigates around the source file directory, scans and parses all .e files(Eiffel source code files). The output of this stage is static relationships between all classes. All relationships are expressed as a triple of the form (*relationship, entity1, entity2*). The relationships we include are inheritance (*inherit class1 class2*) and client-supplier (*use ClientClass SupplierClass*).

Static relationships are retrieved by using EC (Eiffel Compiler). EC is a command-line Eiffel compiler developed by ISE[9] (it is the same compiler EiffelStudio uses). The extraction module manages EC to traverse all .e files to collect static information, and consolidates the static information into static facts. Finally, it renders the collected static facts into RSF format to interface with Grok in stage two.

5.3.2   Candidate instance discovery

As mentioned in the GOF design pattern book [2], each design pattern has a distinct static structure and dynamic behaviour. On one hand, the static structure depicts the compile-time relationships between classes, which are

necessary for the design pattern, such as class A inherits class B, or class C calls class D by method E. On the other hand, the dynamic behaviour indicates the run-time character of the design pattern. It is usually shown as a chronological sequence of method invocations.

In this stage, we compose the definition of the static structure of each design pattern we would like to detect. This definition is rendered as RSF and it defines the inheritance and invocation relationship among classes participating in the design pattern. Using Grok, we compare the definition with the facts about static relationships derived in stage one.

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 fits 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.

5.3.3   *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.

We start by composing the dynamic definition of each design pattern. The dynamic definition is formatted as TA[7], an extension of RSF, and it is based on the calling sequence of classes in the design pattern. Besides the dynamic definition, we need to obtain dynamic behaviour facts for the target software system. EC provides a "call tracing" function which generates a log file recording all method calls chronologically. We check each set of classes by comparing the dynamic definition with the dynamic fact file. Many false positives are removed in this manner.

DPVK generates a list of pattern instances in XML format. This design aims at better adaptability that will enable other tools, especially other Eclipse plug-ins, to access DPVK output and perform further analysis upon it.

5.4   *Manual evaluation*

This is the final stage of the pattern detection process. Since this stage is not implemented yet, the following describes our immediate implementation plan.

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, we plan to solicit feedback from experienced developers. This stage will let developers

manually verify and justify the candidate instances from the output of stage three. Their knowledge and experience could be helpful in filtering out unwanted output, and in discovering undetected pattern instances.

To let developers easily manipulate the detected pattern instances, we are going to develop a DPVK extension that accepts the XML output from previous stage, renders the output to a BON diagram, and allows developers to comment on the candidate set.

6 Conclusion

In this paper, we introduce a diagram catalogue of design patterns. Each pattern diagram includes both a static structure diagram and a dynamic behaviour diagram. The detailed process of generating these diagrams is also discussed.

We also propose a reverse engineering tool called DPVK, which is used to detect/verify patterns in Eiffel systems. DPVK operates in four stages. The first three stages are *static fact extraction*, *candidate instance discovery* and *false positive elimination*. Finally, the fourth stage lets developers manually evaluate and justify the output of previous stages.

As mentioned in the previous section the implementation of DPVK is still ongoing. We need to implement the final stage of DPVK and integrate it into the existing architecture. To evaluate DPVK's effectiveness and efficiency, we are going to test it upon all Eiffel implementations in [11]. Also, the EiffelBase library in EiffelStudio is an ideal test base.

DPVK can be enhanced in user interaction and visual presentation. For example, DPVK could automatically generate BON diagrams for each pattern candidate and enable users to work directly upon the diagram to change the candidate's composition.

As stated earlier, an Eiffel Development Tool for Eclipse is under development in York University. Once it is ready, DPVK can use it to replace EiffelStudio and EC which are currently used to extract static and dynamic facts about the software system under examination. In the long run, DPVK will be extended to detect design patterns in other popular object-oriented language such as Java, C++ etc.

References

[1] Kyle Brown. *Design reverse-engineering and automated design pattern detection in Smalltalk*, Master's thesis, North Carolina State University, 1996.

[2] Erich Gamma, R. Helm, R. Johnson, J. Vlissides. "Design Patterns — elements of reusable object-oriented software", Addison-Wesley, 1995.

[3] Erich Gamma, Kent Beck. "Contributing to Eclipse: Principles, Patterns, and Plugins", Addison-Wesley, 2003.

[4] Gerald C. Gannod, Betty H. C. Cheng. *A framework for classifying and comparing software reverse engineering and design recovery techniques*, Working Conference on Reverse Engineering, pages 77–88, 1999.

[5] Dirk Heuzeroth, Thomas Holl, Gustav Hogstrom, Welf Lowe. *Automatic Design Pattern Detection*, 11th IEEE International Workshop on Program Comprehension, pages 94–103, 2003.

[6] R.C. Holt. "Introduction to the Grok Programming Language", University of Waterloo, 2002.

[7] R.C. Holt. "An Introduction to TA: The Tuple Attribute Language", Department of Computer Science, University of Waterloo, 1998.

[8] Eclipse.org. URL:http://www.eclipse.org/.

[9] Eiffel Software Inc. URL:http://www.eiffel.com/.

[10] Jens H. Jahnke, Jorg P. Wadsack. *A History Concept for Design Recovery Tools*, Proceedings of 6th European Conference on Software Maintenance and Reengineering, pages 37–46, 2002.

[11] Jean-Marc Jezequel, Michel Train, Christine Mingins. "Design Patterns and Contracts", Addison-Wesley, 1999.

[12] Christian Kramer, Lutz Prechelt. *Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software*, Working Conference on Reverse Engineering,pages 208–215, 1996.

[13] R. Keller, R. Schauer, S. Robitaille, and P. Page. "Pattern-based reverse-engineering of design components", Proceeding of 21st Conference on Software Engineering, Los Angeles, USA, pages 226–235. IEEE, 1999.

[14] BON CASE Tool. URL:http://www.cs.yorku.ca/∼eiffel/bon_case_tool/.

[15] Hong Mei, Tao Xie, Fuqing Yang, Peking University. "JBOORET: an Automated Tool to Recover OO Design and Source Models", 25th Annual International Computer Software and Applications Conference, pages 61–76, 2001.

[16] Bertrand Meyer. "Object-Oriented Software Construction, Second Edition", Prentice Hall, 1997.

[17] J. Seemann and J.W. von Gudenberg. "Pattern-Based Design Recovery of Java Software", ACM SIGSOFT Software Engineering Notes, ACM Press, 1998.

[18] Forrest Shull, Walcelio L. Melo, and Victor R. Basili. *An Inductive Method For Discovering Design Patterns From Object-Oriented Software Systems*, Technical Report, Computer Science Department, University of Maryland, 1996.

[19] SmartEiffel. URL:http://smarteiffel.loria.fr/.

[20] Kim Walden, Jean-Marc Nerson. Seamless object-oriented software architecture: analysis and design of reliable systems, Prentice Hall, 1995.

[21] Jingwei Wu, Ahmed E. Hassan and Richard C. Holt. *Using Graph Patterns to Extract Scenarios*, Proceedings of 10th International Workshop on Program Comprehension, 2002.

[22] URL:http://www.cs.yorku.ca/∼weiw/research/DPDiagrams.html.