Automatic Design Pattern Recovery

Yonggang Zhang

A Thesis

in

The Department

of

Computer Science

Submitted in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

August 2003

© Yonggang Zhang, 2003

National Library of Canada
Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services
Acquisisitons et services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-612-83925-7
Our file Notre référence
ISBN: 0-612-83925-7

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou aturement reproduits sans son autorisation.

Canada

ABSTRACT

Automatic Design Pattern Recovery

Yonggang Zhang

An approach for recovering design patterns from source code is presented. The approach addresses software comprehension issue in reverse engineering domain, by providing a design pattern based representation of the system to facilitate software understanding.

Design patterns are formalized by a simplified version of the LePUS language, which eliminates some higher-order concepts to reduce the complexity and at the same time, incorporates some extensions on entity and relation. Source code is semantically parsed and is further interpreted by a language analysis framework. Entities and relations are extracted and are used to generate a high level abstraction of the program – program model. A multi-stage searching algorithm is adopted to match the design pattern specifications with the program model to identify design patterns implemented in the source code.

The approach has been implemented and can be used to identify several design patterns listed in GoF book. Three initial experiments are conducted on some open source software to demonstrate its suitability for recovering the specified design patterns. The results are discussed in terms of the performance and the precision of pattern recovery.

For Wei

TABLE OF CONTENTS

List of Figures vii

List of Tables viii

Chapter 1 Introduction 1

Chapter 2 Background 4

2.1 Design patterns 4

2.2 Formal specifications of design pattern 6

2.2.1 LePUS 6

2.2.1 Disco 9

2.3 Pattern-based reverse engineering tools 11

2.3.1 Pat 12

2.3.2 AOL 13

2.3.3 Columbus and Maisa 15

Chapter 3 An Approach to Design Pattern Recovery 17

3.1 Objective setting 17

3.2 Research method 19

3.3 Simplified LePUS 22

3.3.1 Ground entities 22

3.3.2 Properties of ground entities 23

3.3.3 Ground relations 24

3.3.4 Higher order entities 26

3.3.5 Formula 27

3.3.6 Formalizing design pattern 29

3.4 Source code analysis 31

3.4.1 Architecture view 32

3.4.2 Parsing 33

3.4.3 Normalizing 35

3.4.4 Program model generating 37

3.5 Design pattern recovery 43

Chapter 4 Experimental Results 46

4.1 Implementation 46

4.2 Experiments setting 48

4.3 Results analysis 51
4.3.1 Memory usage 51
4.3.2 Execution time 52
4.3.3 Precision 53
4.3.4 Consistency of design pattern document 58
Chapter 5 Conclusions and Future Works 61

References 66
Appendix Specifications of Design patterns 69
A.1 Abstract Factory 69
A.2 Factory Method 70
A.3 Singleton 72
A.4 Adapter 73
A.5 Bridge 76
A.6 Composite 78
A.7 Decorator 80
A.8 Template Method 82

List of Figures

Figure 2.2-1 Abstract Factory Pattern in LePUS 7

Figure 2.3-1 Observer Pattern in DisCo 10

Figure 3.2-1 Research Method 22

Figure 3.3-1 Factory Method Pattern [GHJV94] 30

Figure 3.3-2 Specification of Factory Method 31

Figure 3.4-1 Architecture view of source code parsing 32

Figure 3.4-2 Structure of parser 34

Figure 3.4-3 Structure of source code analysis framework 36

Figure 3.4-4 Structure of program model 42

Figure 4.1-1 Output of an identified Composite pattern 47

List of Tables

Table 3.3-1 Properties of ground entities 24

Table 3.3-2 Ground relations 25

Table 4.3-1 Memory usage of pattern recovering process 52

Table 4.3-2 Execution time of pattern recovering process 52

Table 4.3-3 Precision of recovering Singleton pattern 53

Table 4.3-4 Precision of recovering Class Adapter pattern 54

Table 4.3-5 Recovery of the Object Adapter pattern 55

Table 4.3-6 Recovery of the Bridge pattern 55

Table 4.3-7 Precision of recovering Composite pattern 56

Table 4.3-8 Recovery of the Decorator pattern 57

Table 4.3-9 Precision of recovering Template Method pattern 58

Table 4.3-10 Keywords related to patterns 59

Table 4.3-11 Document consistency of java.util 59

Table 4.3-12 Document consistency of jEdit 59

I. Introduction

Software maintenance of existing legacy systems is the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment [IEEE1219-98]. Referring to some publications, maintenance of existing systems is estimated to consume between 50% and 80% of the resources in the total software budget [Bo81, Mc92]. And within the maintenance process, software comprehension requires 47% and 62% of the total time for Improvement and correction tasks, respectively [FH83]. Therefore, software comprehension, including the process and methodology, has significant effect on software maintenance, and the whole software development.

One of the common ways of understanding software is to develop a global picture on the system, sub-system, e.g. UML class diagram, ER diagram, which reflects the physical and logical structure of the components and communications. However, this is still insufficient for a developer to fully comprehend the *purpose* of a given piece [Bi89] of code. One of the possible reasons, as Beck and Johnson said, is that "existing design notations focus on communication the *what* of the designs, but almost completely ignore the *why*". [BJ94] [KSRP99]

The design pattern, as a high-level design element, describes a

commonly-recurring structure of communication components that solves a general design problem within a particular context [GHJV94]. The descriptions of patterns contain not only the knowledge about the components and the inter-relationships, but also the alternatives and design decisions behind the design and implementation. Knowing the patterns existing in the legacy system will give a lot of extra information to the reader, and thus considerably improve the efficiency of software understanding. From this point of view, we consider that recovering design patterns in legacy system is an important step in the software comprehension process, which is, partially answering the question – "why is the software designed like that". The benefits of identifying design patterns can be highlighted as the following:

- Comprehension – recovering design patterns in legacy systems, and building a pattern-level representation of the system to facilitate the software understanding. Furthermore, incorporating with domain knowledge and other software artifacts to provide guidance in rediscovering the design and understanding the design decisions and rationales beyond the raw source code.

- Documentation – using a pattern language [GHJV94] to document the design of the legacy system will significantly improve the precision and

concision of the documents

- Refactoring[FBBOR99] – both identified design patterns in an existing system and their relationships may improve the quality of source code, by allowing for refactoring of parts of the source that were not well designed.

- Validation – the recovery of the implemented patterns can provide some additional insights of the problems the software is trying to address or solve. We can use pattern recovering tools to validate whether the current design/solution fully satisfies the requirement of the pattern implementation by verifying if a certain pattern can be identified.

In this thesis, we introduce an approach that can identify design patterns from Java language source code. The presented approach is designed to extract information from the source code to build a language independent program model. This model is then used to match the specifications of design pattern and identify existing design patterns implemented in the program. Since the program model is designed to represent the program in a high level abstraction on a language independent layer, we consider that the presented approach is not limited to the Java language, but can be easily extended to C++, C#, and most of the other object oriented languages.

II. Background

Many research directions have been proposed within the pattern community, which range from investigating new patterns in specific domains, pattern classifications, to pattern-based methodologies and tools. However, in the context of reverse engineering, most of the ongoing researches on design patterns focus on formal specification of design patterns and pattern-based reverse engineering tools, e.g. [Ed01], [Mi98], [KP96], [AFC98], and [FGMP01].

2.1 Design Patterns

Design patterns are high-level design elements that address recurring problems in object oriented design. A design pattern describes "the problem, the solution, when to apply the solution, and its consequence" [GHJV94]. To a concrete problem, software designers analyze it and its context by referring to some existing design patterns to decide how to apply the solution provided by the pattern; thus, like algorithm or data structure, design pattern also plays a role as design element in software design.

While the usefulness of design patterns in forward engineering is well proven, the usages of design patterns in reverse engineering area is also an important

application area. A design pattern not only provides a proven solution to a recurring problem, but also conveys the rationales behind the solution, i.e., not only "what", but also "why" [BJ94]. Therefore, design patterns may help us understand the design decisions, which are usually not plainly described in the software documents or source code. For that reason, we consider that design pattern plays an important role in reverse engineering, and using design patterns identified from source code can help us comprehend, maintain, and even refactory software in the large.

A few researches (mainly described in section 2.3) have addressed the issue of using design patterns in reverse engineering, especially in software comprehension. Our research addresses the same direction, but we focus our efforts on: firstly, to provide a precise definition to each pattern, and then secondly to provide a language independent framework for a pattern-based reverse engineering tools that may facilitate the understanding of existing software.

While in [BMRSS96], the five authors provided a more general perspective for patterns used in software: Architectural Pattern, Design Pattern, and Language Idioms, in this thesis, we adopt the notion that a "pattern" or "design pattern" refers to design pattern, which can be defined as the following:

A *design pattern* provides a scheme for refining the subsystems or components of a software system, or the relationships between them. It describes a commonly recurring structure of communicating components that solves a general design problem within a particular context. [GHJV94].

2.2 Formal specifications of design pattern

The success of automatic design pattern recovery absolutely depends on how well we understand design patterns. Formal specification is the most precise description of our knowledge, in our case, design patterns. Thus as part of this research we regard that providing a formal specification of design patterns is vital to our research. We have studied several approaches in formalizing design patterns. The following is a literature review of formal design pattern representations.

2.2.1 LePUS

LanguagE for Pattern's Uniform Specification, in short, LePUS [Ed01], is a formal notation of OO design and architecture. As the author concluded, LePUS is defined as "a visual language for the specification of OO software architecture" [Ed02], and it defines "a symbolic equivalent for each well-defined diagram" [Ed02] as well.

In order to formalize the participants of a design pattern, LePUS defines *Class* and *Method* as its ground entities, and further, *Uniform Sets* and higher order sets as a set of participants like Creators, Products, etc. *Clan* and *Hierarchies* are also introduced in order to represent the dynamic binding and inheritance mechanisms of OO programming respectively.

In order to formalize the collaborations of participants, LePUS firstly identifies a small set of relations between ground entities, like *DefinedIn:* $\mathbb{F}$ *x* $\mathbb{C}$ (where $\mathbb{F}$ is the domain of methods and $\mathbb{C}$ is the domain of classes), as *ground relations*. *Bijection* relation, which is mainly used to formalize the one-to-one correspondences between two uniform sets, is also defined.

Based on the above "building blocks" [Ed02], a pattern is specified as its participants and the collaborations of them. For example, Abstract Factory pattern [GHJV94], in LePUS, can be specified as the following:

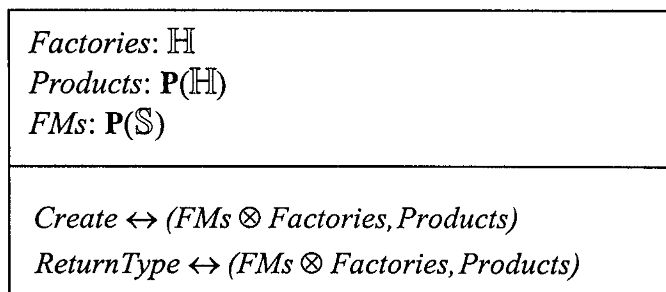


Figure 2.2-1 Abstract Factory Pattern in LePUS

The upper section represents the participants of the pattern, where $\mathbb{H}$ is the

domain of hierarchies, and $\mathbb{S}$ is the domain of signatures. $\mathbf{P}(\mathbb{H})$ and $\mathbf{P}(\mathbb{S})$ indicate the power sets of $\mathbb{H}$ and $\mathbb{S}$ respectively. Then, *Factories* is a hierarchy, *Products* is a set of hierarchies, and *FMs* (Factory Methods) is a set of signatures. The next section represents the collaborations of these entities, where selection operator $\otimes$ in $FMs \otimes Factories$ will yield all methods defined in *Factories* whose signature is in *FMs*, namely, the set of factor-method clans, and $\leftrightarrow$ indicates a bijection of two sets. [Ed02]

"LePUS is a highly concise language" [Ed02], which implies that it stands on a very high level of abstraction. We consider LePUS is ideal for architectural level specification, but it lacks some details in class design level specification. For instance, the implementation of the Singleton pattern, which "ensures a class only has one instance" [GHJV94], requires a private construction method and a static instance of itself. The specification of the Singleton pattern is impossible in LePUS because LePUS does not provide facilities for describing the properties of class attributes. Another example is that some patterns like the Composite [GHJV94] and the Iterator [GHJV94] require some participants to access their attributes to maintain a list of elements, which also cannot be described by LePUS. In addition, because of its high abstraction, from a reverse engineering perspective, mapping source code to LePUS expressions is very difficult.

2.2.2 Disco

DisCo (Distributed Co-operation) [Mi98] is a formal specification method for reactive systems. The formal basis of DisCo is temporal logic of actions [La94], which is logic for specifying and reasoning about concurrent and reactive systems. Consequently, DisCo specification of Design Pattern focuses on the interactions among participant objects, and as the authors said, DisCo emphasizes on collective behavior, rather than on behavior of individual objects.

A DisCo specification of a software system includes three parts: *classes*, *relations*, and *actions*. In [Mi98], the author demonstrates DisCo on the Observer pattern [GHJV94] as the following:

class *Subject* = { *Data* },
class *Observer* = { *Data* },
relation (0..1) · *Attached* · (*): *Subject* x *Observer,*
relation (0..1) · *Updated* · (*): *Subject* x *Observer.*

The first two formulas represent that there two classes *Subject* and *Observer* that involved in this pattern, and the next two formulas indicate that there are one-to-many relations, called *Attached* and *Updated*, between *Subject* and *Observer*.

In order to formalize the interactions of *Subject* and *Observer*, the following actions are used:

$$Attach(s : Subject; o : Observer):$$
$$\neg s \cdot Attached \cdot o$$
$$\rightarrow s \cdot Attached' \cdot o,$$

$$Detach(s : Subject; o : Observer):$$
$$s \cdot Attached \cdot o$$
$$\rightarrow \neg s \cdot Attached' \cdot o$$
$$\wedge \neg s \cdot Updated' \cdot o$$

$$Notify(s : Subject, d):$$
$$\rightarrow \neg s \cdot Update' \cdot \mathbf{class}\ Observer$$
$$\wedge s.Data' = d$$

$$Update(s : Subject; o^{*} : Observer; d):$$
$$s \cdot Attached \cdot o$$
$$\wedge \neg s \cdot Updated \cdot o$$
$$\wedge d = s.Data$$
$$\rightarrow s \cdot Updated' \cdot o$$
$$\wedge o.Data' = d$$

Figure 2.3-1 Observer Pattern in DisCo

Above formulas represent the four interactions: Attach, Detach, Notify, and Update respectively. Each action consists of a list of participants and parameters incorporated with pre- and post-conditions of the execution of the action.

DisCo also provides facilities for class inheritance and pattern combination, and thus for specifying more complex patterns.

Since DisCo mainly addresses the dynamic aspects of design patterns, it is very

difficult to apply DisCo in static source code analysis.

2.3 Pattern-based reverse engineering tools

Reverse engineering focuses on creating "representations of a system in another form at a higher level of abstraction" [CC90]. Design patterns, as a kind of design elements, capture "the rationale behind recurring proven design solutions and illuminate the trade-offs that are inherent in almost any solution to a non-trivial design problem" [KSRP99]. Introducing design patterns to facilitate reverse engineering tools will significantly improve the expressiveness of the created representations, and thus help people to understand the design decisions behind the source code.

Several pattern-based reverse engineering tools have been introduced to facilitate software comprehension within the last few years. Most of these tools are based on pattern detecting techniques. However, only a very few of these tools have a solid theoretical basis. The following sections give a brief introduction and review of each of the tool and discuss the approach used, the system structure, and the results and experiences from applying these tools.

2.3.1 Pat

The Pat [KP96] system is a Reverse Engineering tool that searches for design pattern instances in existing software. Within Pat, design information is "extracted directly from C++ header files and stored in a repository" [KP96]; "the patterns are expressed as PROLOG rules and the design information is then used to search for all patterns" [KP96]. Since the Pat system is designed to collect information just from C++ header files without semantic analysis in the method body, the authors limit the considered patterns to five structural patterns introduced in GoF book [GHJV94]: Adapter, Bridge, Composite, Decorator, and Proxy.

The approach of the Pat system is simple: representing both patterns and designs in PROLOG and let PROLOG engine do the matching job. The recovering process starts from representing each pattern as a static OMT diagram, and converting the diagram to a PROLOG representation, which means one rule for each pattern. Another reverse engineering tool, ooCASE, is used to analyze C++ header files and store the information in the repository in OMT form, which is then translated into to a PROLOG representation. Finally, a PROLOG query will detect all instances of design patterns from the two repositories. [KP96]

In order to evaluate the quality of the design pattern recognition of the Pat system,

the author used *precision* and *recall* [FB97], which is the most commonly used measure of retrieval effectiveness. Precision can be defined in this context as: the ratio of the number of true patterns found by Pat over the number of all candidates found by Pat. Recall is the ratio of the number of true patterns found by Pat over the number of all true patterns that exist in the software. The authors of the Pat system state that in the 4 benchmark applications, the precision ranges from 14 to 50 percent, and will be "much higher if Pat could also check for correct method call delegations" [KP96].

2.3.2 AOL

In [AFC98], Antoniol et al present a conservative approach to recover design patterns from design and source code, which is mainly based on "a multi-stage reduction strategy using software metrics and structural properties to extract structural patterns for OO design or code" [AFC98]. The reason why the authors call this approach conservative is because any patterns in the code will be surely reported in the result. Similar to the Pat system, this approach also focuses on the same five structural patterns mentioned in [KP96]. However, in addition to Pat, method delegation, which means a method delegates its responsibilities by calling another method of associated class, is used as a design pattern constraint to reduce the reported false candidates.

The process of this approach consists of four activities. The first is AOL (Abstract Object Language [Pe97], a general purpose design description language) representation extraction, which represents the C++ source code or design in AOL, and the second step is to analyze the AOL representation to generate an AOL abstract syntax tree. In the next step, relevant class metrics are extracted. Finally, pattern constraints like metrics constraints, structural constraints, and delegation constraint are applied to the AOL abstract syntax tree and class metrics obtained in the previous steps and a set of pattern candidates are conducted. The authors claim that the constraints are all necessary conditions, and thus the output does not contain false negatives but it may contain false positives, which must be manually inspected by the user. [AFC98]

By recovering design patterns from more than 14 codes of public and industrial systems, the authors provide a comprehensive experimental result analysis. In addition to measuring the results in terms of *recall* and *precision*, the reduction of candidates through the stage filters is also provided, with respect to Initial stage, metrics based filter stage, structural filter stage, and delegation filter stage. The authors conclude that, in public domain code case studies, the average precision is 55 %, and there is “an increase of about 35% using also the delegation constraint with respect to the use of structural constraints alone” [AFC98].

2.3.3 Columbus and Maisa

Columbus [FMBMTG00] is a versatile reverse engineering system that transforms C++ programs into a number of abstract representations, including UML class diagrams. Maisa [NGPV00] is a metrics tool that analyzes the quality of a software architecture given as a set of UML diagrams. In [FGMP01], Ferenc et al present a technique for automatically recognizing design patterns form object-oriented (C++) source code by integrating Columbus and Maisa. The authors state that Columbus and Maisa pairs can be used to document and analyze software implemented in C++, and to verify the architectural design decisions during the software implementation phase as well.

The tools are applied by combing both the Columbus and the Maisa. The process of this tools can be described as two major steps: The Columbus system firstly transforms the C++ source code into UML class diagrams; and these diagrams are then traversed and matched against a set of predefined design patterns by the Maisa, which consists of a set of CSP (Constraint Satisfaction Problem) formulas to describe each UML class diagram incorporated with additional information obtained from different types of UML diagrams. The authors are "studying the possibility of using dynamic information (such as sequence diagrams) for defining patterns more accurately" [FGMP01].

At last, the authors conclude that some patterns, like Iterator and Observer, cannot be recognized with the current method because the definitions of these patterns contain generated facts, i.e., "structural facts that are dynamically pushed to the input by Maisa when it recognizes a particular kind of pattern or a special kind of a common class relation". As well as this limitation, they also report that the performance degrades with large software systems [FGMP01].

III. An Approach to Design Pattern Recovery

3.1 Objective setting

In the recent decade, design patterns have been extensively investigated in different domains. However, the 23 patterns introduced in [GHJV94] are still the most common design patterns implemented in current object-oriented software. Our approach therefore focuses on these patterns, that is, trying to recovery the 23 GoF patterns listed in [GHJV94] from source code. However, since the theoretical background of our pattern formalization is based on LePUS, which is declared as a formal specification language dedicated to the general OO design and architecture [Ed02], our approach is not restricted only to the GoF patterns. Theoretically, any Object-Oriented pattern that can be formalized in LePUS can also be recovered by our approach.

In order to reduce the complexity, the recovering work is performed on Java [GJS96] source code rather than C++. One of the major reasons why we choose the Java language is that we consider the Java language, in most cases, is simpler and cleaner than C++, but still, contains most of the Object-Oriented programming language concepts.

For the pattern recovery process to be usable for large software system, the recovery work performed on the source code should be fully automatic, which means there will be no human interaction required during the identification of the patterns. We are aware and have also stated in the conclusions that without humans participating with their domain knowledge, it will be impossible to achieve a 100% precision in identifying all design patterns correctly. Our presented work will help to illustrate the possibilities of automatic design pattern recovery.

In addition, in order to analyze the source code written in Java, it is inevitable that one has to study in detail the syntax and semantics of the Java language. The CONCEPT(Comprehension Of Net-CEntered Programs and Techniques) [http://www.cs.concordia.ca/CONCEPT/] project, which addresses current and future challenges in the comprehension of large and distributed systems by providing programmers with novel comprehension techniques, also focuses at present on the analysis and comprehension of Java programs. One of our main objectives is to integrate the pattern recovery techniques into CONCEPT, as well as to provide a reference implementation for a Java source code analysis framework that can be reused by other developers within the CONCEPT project. Such a code analysis framework has two major functions: 1. it uses source code files as input, analyzes them, and stores the result (normalized abstract syntax tree) into repositories; 2. it provides an interface to developers to access the

repository, extracts and manipulates various source elements of the Java language.

3.2 Research method

Many researchers have presented their efforts on automatic design pattern recovery, e.g. [KSRP99], [AFC98], [KP96], [FGMP01], etc. Reviewing the approaches they have used, we conclude that the quality of recovering work relies on how well we answer the following two research questions:

What are we looking for?

Design patterns are always specified by natural languages, which contain a lot of ambiguity and informality in their descriptiveness. In order to recover design patterns automatically, a more formal specification of design patterns is absolutely necessary to remove the ambiguity of the natural language.

Formal specification of design patterns focuses on giving a precise and unambiguous description of design patterns. This is achieved by only representing the essence of the pattern without any irrelevant details and variations in the implementation. Formal specification languages used to describe patterns should be based on higher-level abstraction of general object oriented

paradigms that not only can be used to describe that existing patterns, but also can describe any patterns that have not been investigated. Besides, a good pattern specification language should give us some reasoning facilities to analyze the relationships among the patterns. For instance, whether a pattern is a refinement of another pattern, or whether a pattern is a combination of several other patterns.

In this thesis, we use a simplified version of LePUS to formalize design patterns, which eliminates some high-order concepts to reduce the complexity of the formalism. At the same time we introduce additional relations to extend the expressiveness of the language. As a result, each pattern is formalized in several formulas that can be easily implemented in programming languages.

What is the recovery work performed on?

Obviously, to perform the recovery work, source code should be analyzed and represented/translated into its semantic level, which is usually represented as abstract syntax tree (AST). This allows the system to identify and analyze the entities and corresponding relations of the program. However, since the generalization of the formal languages as well as the variations of pattern implementations, matching the pattern specification on the AST level is extremely

difficult. A higher-level of abstraction is required to reduce the complexity of the programming language, which ideally, should be a language independent abstraction.

In terms of pattern identification, the specification of a design pattern not only offers knowledge about what represents a pattern, but also implies important information on what should be extracted from the source code for better efficiency of matching the abstract formulas to the concrete source code. In our approach, following the ideas of pattern specification languages, we represent the source code in a high level abstraction – the program model, and perform the recovery work on this intermediate representation.

As the following diagram shows, the fundamental idea of our research is representing both the design patterns and source code in the same level of abstraction, and performing the matching work on this level to recover design patterns.

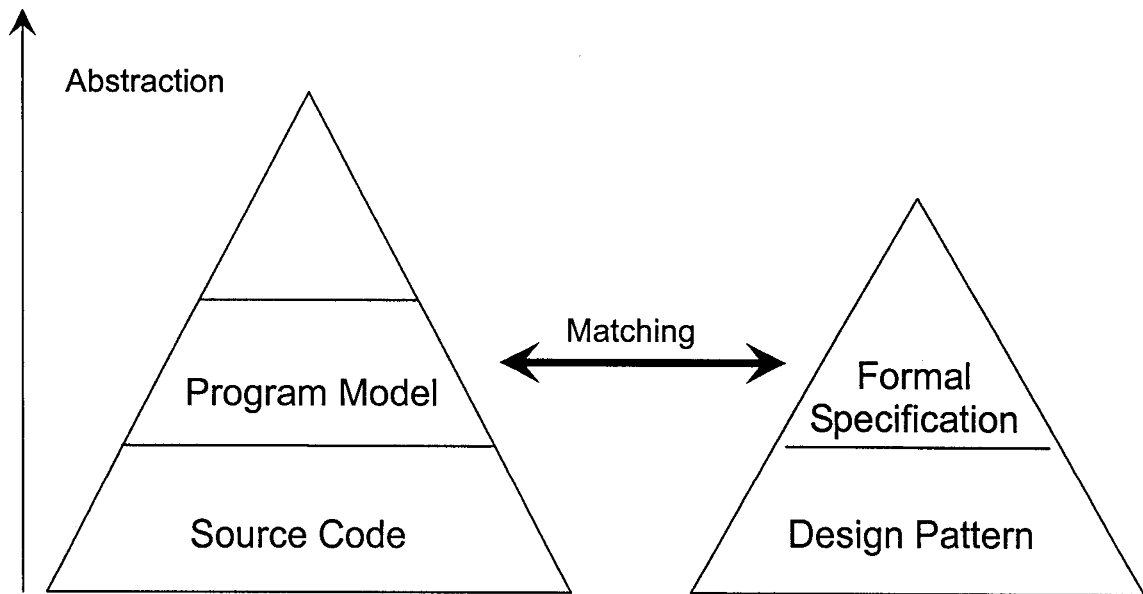


Figure 3.2-1 Research Method

3.3 Simplified LePUS

In this section, we present the theoretical basis of our approach – a simplified version of LePUS[Ed01]. The simplified version eliminates some higher-order concepts to reduce the complexity of the formalism. At the same time it incorporates some extensions on entity, property of entity, and relations.

3.3.1 Ground Entities

Class, method, and attribute are three major language elements in Object-Oriented languages. All programs in OOP are defined by a set of classes,

and all classes are defined by a set of methods and attributes. The behavior of a program is defined by the operation of the three kinds of entities, e.g. object creation and method invocation. For compatibility issues, some languages (e.g. C++) may define instance variables outside classes. In LePUS, those language elements are ignored. In our approach, we consider class, attribute, and method to be **ground entities** of a program, which are represented as $c_1, c_2, \ldots, a_1, a_2, \ldots,$ or $m_1, m_2, \ldots,$ respectively, or $e_1, e_2, \ldots,$ in general.

Let $\boldsymbol{C}$ be the set of all classes in a program, $\boldsymbol{A}$ be the set of all attributes, and $\boldsymbol{M}$ be the set of all methods, we have the set of all ground entities $\boldsymbol{E} = \boldsymbol{C} \cup \boldsymbol{A} \cup \boldsymbol{M}$.

3.3.2 Properties of ground entities

A ground entity has its own properties. For examples, a class is abstract, or a method is declared as protected. The **properties of a ground entity** are represented as $p(c)$, $p(a)$ or $p(m)$, where, c, a, or m is the ground entity and p is the name of the property. In our approach, we introduce access modifiers to LePUS as the properties of ground entities to formalize the encapsulate mechanism of Object Oriented Programming. For example, $Private(m_1)$ represents method m_1 is a *private* method. We summarize the properties of ground entities that are currently used in our approach in table 3.3-1.

Table 3.3-1 Properties of ground entities

Name	Apply to	Description
Abstract	*C, M*	A class or method is abstract. An Interface in Java is considered as an abstract class.
Public	*C, A, M*	A class, method, or attribute is declared as public.
Protected	*C, A, M*	A class, method, or attribute is declared as protected.
Private	*C, A, M*	A class, method, or attribute is declared as private.
Static	*A, M*	A class or attribute is declared as static

3.3.3 Ground relations

In our approach, we identify a set of ground relations to specify the relationship between entities, which are mainly based on LePUS, but introduced some new relations to specify the attribute, and to enhance expressiveness.

Definition I: **ground relation** – the ground relation is the relation whose domain and range are ground entities. Ground relation is represented as $\boldsymbol{r}:\boldsymbol{D}\times\boldsymbol{R}$, where, $\boldsymbol{D}$ and $\boldsymbol{R}$ are the set of ground entities or a subset of ground entities, and $\boldsymbol{r}$ is the name of relation.

For example, $\boldsymbol{DefinedIn}:\boldsymbol{M}\times\boldsymbol{C}$ is a relation whose domain is the method set and range is class set, which represents that a method is defined in a class; $\boldsymbol{Invoke}:\boldsymbol{M}\times\boldsymbol{M}$ is a relation whose domain and range are method set, which

represents that a method may invoke another method, independently whether the invoking statement is executed by the given input.

The relations currently used in our approach are summarized in the following table. More relations can be adopted when necessary.

Table 3.3-2 Ground relations

Name	Apply to	Description
DefineAttribute	$A \times C$	Attribute a is defined in class c
DefineMethod	$M \times C$	Method m is defined in class c
HasAttribute	$A \times C$	Class c has an attribute a, which maybe defined in class c, or inherited from the superclass of c
HasMethod	$M \times C$	Class c has an method m, which maybe defined in class c, or inherited from the superclass of c
Inherit	$C \times C$	Class c_1 is inherited from class c_2
Reference	$C \times C$	Class c_1 defines an attribute whose type is c_2
ReturnType	$M \times C$	The declared return type of method m is class c
AttributeType	$A \times C$	The declared type of attribute a is class c
Argument	$M \times C$	Method m has a formal argument whose type is class c
SameSignature	$M \times M$	Method m_1 and m_2 has same signature. i.e. same method name and same formal arguments
Construct	$M \times C$	Method m is a constructor of class c
DeclareLocal	$M \times C$	In the definition of method m, a local variable is declared whose type is class c
Create	$M \times C$	In the definition of method m, an instance of c is created
InvokeOwn	$M \times M$	Method m_1 may invoke method m_2, and m_2 is defined in the same class defined m_1

InvokeSuper	$M \times M$	Method m_1 may invoke method m_2, and m_2 is defined in the super class of class defined m_1
InvokeOther	$M \times M$	Method m_1 may invoke method m_2, and *InvokeOwn*(m_1, m_2) and *InvokeSuper*(m_1, m_2) don't hold.
Forward	$M \times M$	Method m_1 may invoke method m_2, and transfer all its argument to m_2
ReadOwn	$M \times A$	Method m may read the attribute a, and a is defined in the same class defined m
ReadSuper	$M \times A$	Method m may read the attribute a, and a is defined in the super class of class defined m
ReadOther	$M \times A$	Method m may read the attribute a, and *ReadOwn*(m, a) and *ReadSuper*(m, a) don't hold
WriteOwn	$M \times A$	Method m may write the attribute a, and a is defined in the same class defined m
WriteSuper	$M \times A$	Method m may write the attribute a, and a is defined in the super class of class defined m
WriteOther	$M \times A$	Method m may write the attribute a, and *WriteOwn*(m, a) and *WriteSuper*(m, a) don't hold
Return	$M \times C$	Method m may return a object whose type is class c

3.3.4 Higher order entities

We observed that in [GHJV97], the structure of a pattern is usually described by the participants and their collaborations. In order to map the entities to participants more precisely and concisely, higher order entities are introduced into LePUS. In our approach, only parts of these higher-order concepts are adopted to reduce the complexity.

Definition II: **Hierarchy** – a hierarchy h is a set of classes, which contains a class c such that any other class that belongs to h is inherited from c.

For example, in Abstract Factory pattern, Factory is a hierarchy, where class AbstractFactory is the root class of this hierarchy and any other members of the hierarchy are ConcreteFactory.

Definition III: **Clan** – a clan f^h is a set of methods on a hierarchy h, where all the methods share the same signature, and each method is defined in a class belongs to h, respectively.

For example, in Bridge pattern, the set of OperationImp methods is a clan defined on hierarchy Implementor.

3.3.5 Formula

A formula in LePUS language is a statement intended to express the fundamental truth of a program. Each formula is a conjunction on sequence of clauses, which consists of a combination of variables, predicates, and operators [Ed01].

Variables

In our approach, variables range over entities, including classes, attributes, methods, hierarchies, and clans.

Predicates

Predicates include connectives, quantifiers, and functions. In the simplified version of LePUS, most of the connectives and quantifiers used in predicate calculus can be applied in the formulas, e.g. $\neg, \wedge, \vee, \exists, \forall$. Two important functions, used to define the properties of relations, are employed: total function and isomorphic function

Total function is a function whose domain is equal to the set from which the first element of its pairs is taken. In our approach, total function is denoted as $\boldsymbol{r}:\boldsymbol{D} \rightarrow \boldsymbol{R}$. For example, $\boldsymbol{Create}:\boldsymbol{M}_1 \rightarrow \boldsymbol{C}_1$ indicates that each method in $\boldsymbol{M}_1$ will create an instance of class in $\boldsymbol{C}_1$.

Isomorphic function is a bijective function. In our approach isomorphic function is denoted as $\boldsymbol{r}:\boldsymbol{D} \leftrightarrow \boldsymbol{R}$. For example, $\boldsymbol{Return}:\boldsymbol{M}_1 \leftrightarrow \boldsymbol{C}_1$ indicates that each method in $\boldsymbol{M}_1$ may return an object whose type is a class in $\boldsymbol{C}_1$, and each class in $\boldsymbol{C}_1$ may be returned by a method in $\boldsymbol{M}_1$.

Operators

Operators used in our approach include unary operators, binary operators, and relational operators.

Only one unary operator is employed: *root*, which yields the root class of a hierarchy, or the root method of a clan. For example *root(h), root(f^h).*

Select operator $\otimes$, as a binary operator, yields a set of methods from a given set of classes. For example, $m \otimes h$ will yield all methods defined in a class that belongs to hierarchy h and the signatures of these methods are same as m.

Relational operators are used to yield a set of relations from the given sets. For example, given a hierarchy h_1, $Inherit^+ : h_1 \times h_1$ will yield all the inherit relations over h_1, direct and indirect.

3.3.6 Formalizing design pattern

Definition IV: Design pattern is a formula $\varphi(x_1, x_2, ..., x_n)$, where $x_1, x_2, ..., x_n$ are free variables in ϕ. We say $x_1, x_2, ..., x_n$ are the participants, and the relations in ϕ are the collaborations. [Ed02]

As the effort of the creator of LePUS, variables in LePUS formula map to entities very well; the ground relations described in section 3.3.3 so far can specify the collaboration sufficiently. In our approach, we formalize the structure of a design pattern in ***Z*** language schema [In88] style, in which that signature part specifies the participants and the predicate part specifies the collaborations of participants.

For example, the Factory Method [GHJV94] pattern describes a solution for deferring the instantiation of a set of products to subclasses of an abstract interface. The structure of this pattern is shown as the following:

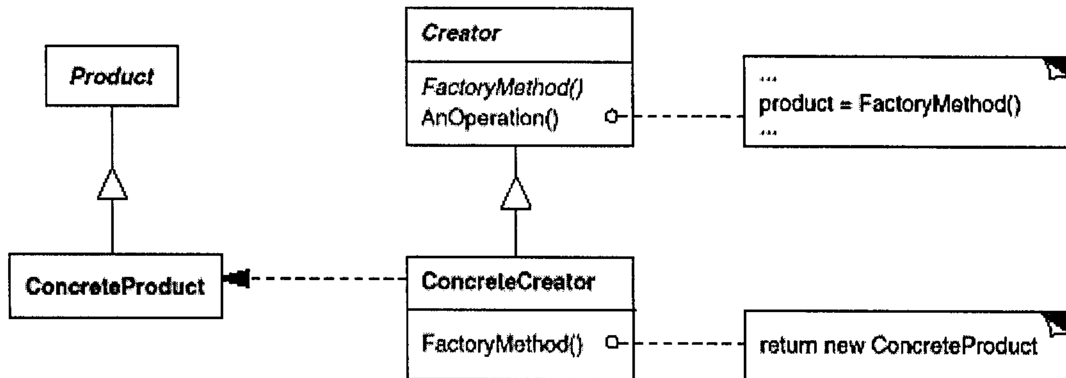


Figure 3.3-1 Factory Method Pattern [GHJV94]

The participants of the Factory Method pattern include:

Product – defines the interface of objects the factory method creates.
ConcreteProduct – implements the Product interface
Creator – declares the factory method, which returns an object of type Product.

The collaborations are: Creator relies on its subclasses to define the factory method so that it returns an instance of the appropriate ConcreteProduct [GHJV94].

We use the following formula to formalize the Factory Method pattern:

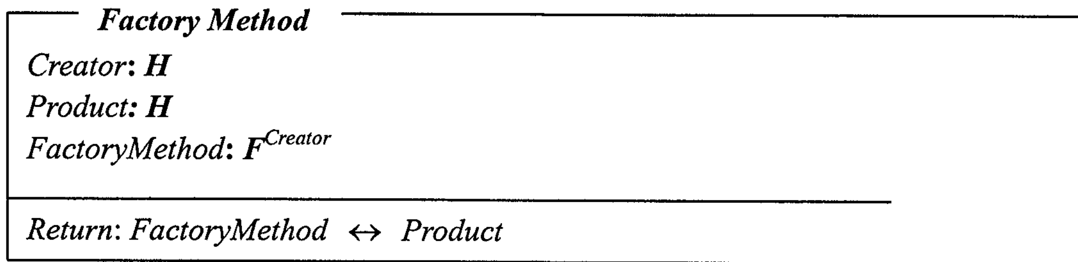


Figure 3.3-2 Specification of Factory Method

The participants include: *Creator* – a hierarchy range over $\boldsymbol{H}$ (the set of all hierarchies); *Product* – a hierarchy as well; *FactoryMethod* – a clan on *Creator* ($F^{Creator}$ refers to all the clans on *Creator*). The collaborations described in the lower part means that a one to one *Return* relation on *FactoryMethod* and *Product* must hold.

Please refer to Appendix I for more formal specifications of GoF patterns.

3.4 Source code analysis

As mentioned earlier, the objective of source code analysis is to build a high level representation that: 1. represents the semantic information of a program; 2. contains the entities and relations that may map to our specifications of design patterns straightforwardly. In our approach, we use "program model" to provide this level of abstraction.

Definition V: Program model – given a program $\boldsymbol{p}$, the program model $\mathbb{M}(\boldsymbol{p})$ is $\{\boldsymbol{E}, \boldsymbol{R}\}$, where $\boldsymbol{E}$ is the set of all ground entities, and $\boldsymbol{R}$ is the set all ground relations.

The program model contains all the ground entities and ground relations in the program, and thus represents the program in a high level abstraction. Since the ground entities and ground relations are the basic elements of our design pattern formalization, we can match the program model to LePUS formulas directly.

3.4.1 Architecture view

The following figure shows the general procedure of the source code analysis:

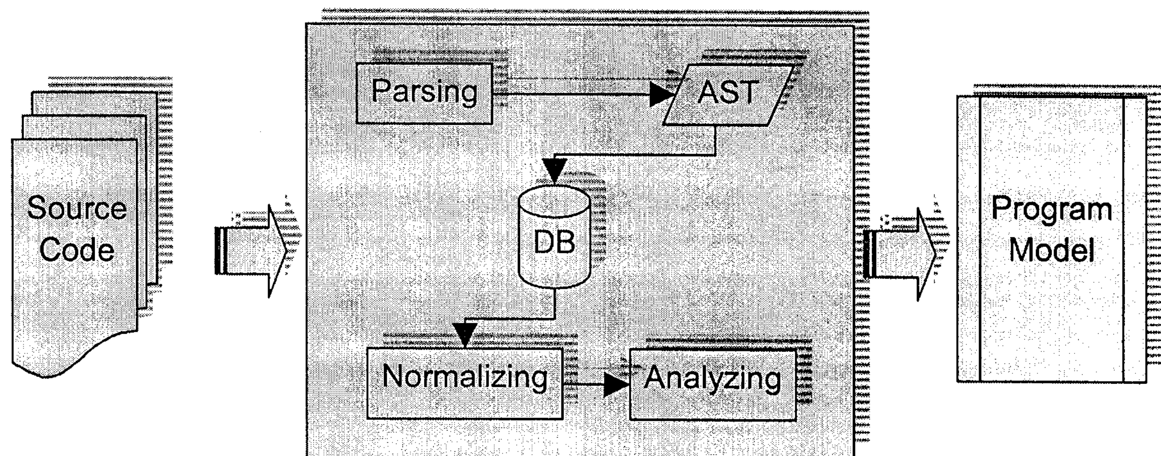


Figure 3.4-1 Architecture view of source code parsing

A semantic level AST structure has been defined. It contains all the information of semantic analysis, like identifier references, expression types, etc. The source

code is parsed into the semantic level AST and stored into database; later, the AST will be analyzed and a normalized memory representation (please refer to section 3.4.3) will be created; next we identify the ground relations based on this memory structure and generate the program model. In the next three sections, we describe the three major steps to create the program model – parsing, normalization, and model generating.

3.4.2 Parsing

A typical compiler includes lexical analysis, syntactic analysis, semantic analysis, optimization, and code generation etc. In order to identify the relations of entities, the source code must be parsed on the semantic level, i.e. all identifiers must be solved and all implicit information must be extracted. Due to the limited time of this research, we had to adopt an existing compiler for this purpose – in our approach, javac was chosen.

There exist many free Java language parsers and parser generators, however, most of them only provide syntactic level analysis, which is not sufficient to identify the ground relations. We consider that javac, which is a fully implemented java language compiler included in JDK, has the ability to perform semantic analysis, and most importantly, the source code of javac can be obtained from

SUN Micro's website. For these reasons, we selected javac as the parser of our choice and made some modifications to adjust it to the particular need of our research.

The structure of the source code parsing part is shown in the following figure.

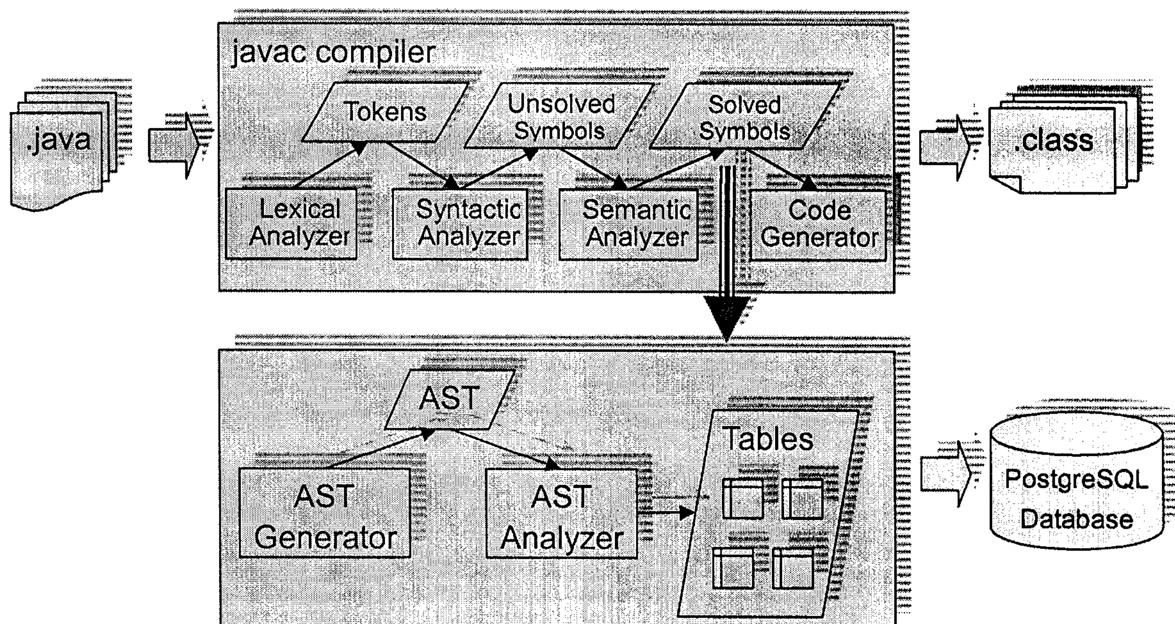


Figure 3.4-2 Structure of parser

In the context of our research, the javac acts like a normal compiler that accepts java source code files as input and conducts some routine processes to generate class files. After the compiler resolves the symbol table, our program will be invoked and accepts the symbol table as input to generate predefined AST structures for all the compiled classes. In the meantime, the ASTs will be partially

normalized and these partially normalized ASTs will be stored into the corresponding tables of the database.

We carefully designed the structure of the AST, so that it will be very compact, but still it contains necessary information from the semantic analysis. Each identifier node in the AST is associated with the type of that identifier and each reference is associated the exact place the referred variable, field, of method is defined.

3.4.3 Normalizing

The semantic level ASTs contains sufficient information about a program. However, from a program manipulation point, it is difficult to perform an analysis on such a structure, because any analysis will require a large number of operations to traverse the tree and to search the reference. Additionally, may recursive procedure calls will be required to provide an adequate navigation and search through the AST. In our approach, we further parse the AST and represent it into an OO structure that significantly reduces the complexity in extracting information. In addition, this normalization will also serve as a framework for other users within the CONCEPT project to access the semantic analysis results.

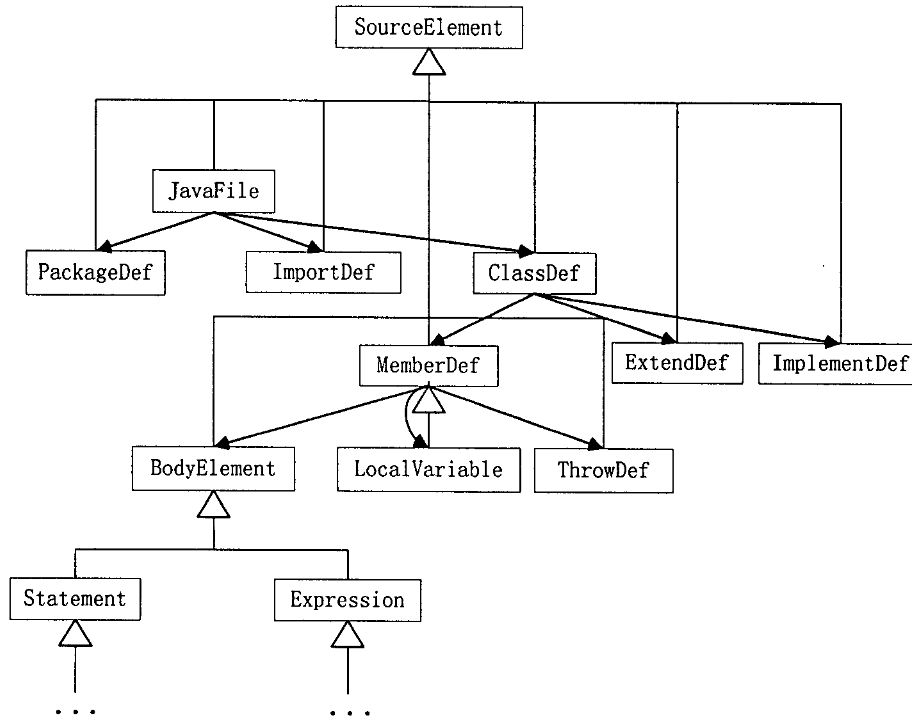


Figure 3.4-3 Structure of Source code analysis framework

As figure 3.4-3 shows, each element of Java language is represented as a type of SourceElement class or its subclass. JavaFile represents the .java file, which contains its PackageDef, ImportDef, and ClassDef. A ClassDef represents the class defined in source code, and contains its ExtendDef, ImplementDef, and MemberDef. The MemberDef refers to the attribute, constructor, method, or inner class of a class, in which statements and expressions may be defined.

The semantics of statements and expressions is the most important part of our framework. Most of the relations are identified at this level. As to statements, we use 21 subclasses of Statement to represent all of the Java statements, e.g.

IfStatement, DoStatement, WhileStatement, ForStatement, DeclarationStatement, SwitchStatement, TryStatement, CatchStatement, ReturnStatement, etc. 83 types of expressions are used to represent the expressions in Java language, including UnaryExpression, BinaryExpression, ConstantExpression, etc. For detail information please refer to the specification of concept.java and concept.java.tree package.

As a result of this step, an OO representation of AST is created; users may enquire any information related to the source code by the provided interface.

3.4.4 Program model generating

Based on the result of source code analysis, we identify the entities and relations in this step to create the program model.

Entity

Naturally, all ClassDef and MemberDef objects directly map to LePUS entities. Higher order entities are created during the matching process when necessary. Primary types in Java language, e.g. int, boolean etc. are ignored; array type is ignored as well, but the type of element of a array type is considered.

Relations

In the following, we discuss the consideration and limitation of the relation identification.

DefineAttribute	If an attribute is explicitly defined in a class, then a *DefineAttribute* relation holds.
DefineMethod	If a method/constructor is explicitly defined in a class, then a *DefineMethod* relation holds.
HasAttribute	If an attribute is explicitly defined in a class, or inherited from a superclass of the class, then a *HasAttribute* relation holds.
HasMethod	If a method/constructor is explicitly defined in a class, or a method is inherited from a superclass of the class, then a *HasMethod* relation holds. Note: In Java language, constructors cannot be inherited.
Inherit	If a class extends a class, or implements an interface, then the *inherit* relation holds. Note: We consider an interface as an abstract class.
Reference	If a *HasAttribute* relation holds, then a *Reference* relation holds between the type of the attribute and the class in *HasAttribute* relation.
ReturnType	The *ReturnType* relation holds between a method and its return

type, if the return type is an entity.

AttributeType The *AttributeType* relation holds between an attribute and its declared type, if the type is an entity.

Argument If the type of a formal argument of a method is an entity, then an *Argument* relation holds

SameSignature In the current implementation, two methods have *SameSignature* relation, if and only if they have same name and same formal arguments, besides, the two methods should override a same method declared in the superclass.

Construct If a method is a constructor, then a *Construct* relation holds between the method and the class that this method defined.

DeclareLocal In the definition of a method, if a local variable is declared whose type is an entity, then a *DeclareLocal* relation holds.

Create If in the definition of a method, there is a new instance expression, in which an entity is created, and then a *Create* relation holds.

InvokeOwn If in the definition of a method m_1, there is a method invocation expression that invokes a method m_2, and m_2 is defined in the same class as method m_1 defined in, then the two methods, m_1 and m_2, have *InvokeOwn* relation.

InvokeSuper If in the definition of a method m_1, there is a method invocation expression that invokes a method m_2, and m_2 is defined in the superclass of the class in which method m_1 is defined, then the

two methods, m_1 and m_2 have *InvokeSuper* relation.

InvokeOther If in the definition of a method m_1, there is a method invocation expression that invokes a method m_2, and m_2 is defined in neither the class in which method m_1 is defined, nor the superclass of the class in which method m_1 is defined, then the two methods, m_1 and m_2, have *InvokeOther* relation.

Forward If in the definition of a method, there is a method invocation expression, method m_1 invokes method m_2, and method m_1 sends all its arguments to method m_2 as parameters, then a *Forward* relation holds.

In the current implementation, only a part of the arguments is considered, i.e. if method m_1 sends some of its arguments, we still create *Forward* relation. In addition, the arguments of method m_1 may be modified in the definition of m_1, we ignore such modification.

ReadOwn In the current implementation, if in the definition of a method, there is an identifier that refers to an attribute, and the identifier appears in the right hand side (RHS) of an expression, and the attribute is defined in the same class as the method defined in, then a *ReadOwn* relation holds.

ReadSuper In the current implementation, if in the definition of a method, there is an identifier that refers to an attribute, and the identifier appears in the right hand side (RHS) of an expression, and the attribute is defined in the super class of the class in which the

method is defined, then a *ReadSuper* relation holds.

ReadOther In the current implementation, if in the definition of a method, there is an identifier that refers to an attribute, and the identifier appears in the right hand side (RHS) of an expression, and the attribute is defined in neither the class in which the method is defined, nor the super class of the class in which the method is defined, then a *ReadOther* relation holds.

WriteOwn In the current implementation, if in the definition of a method, there is an identifier that refers to an attribute, and the identifier is the left hand side (LHS) of an assignment expression or increment/decrement expression, and the attribute is defined in the same class as the method defined in, then a *WriteOwn* relation holds.

WriteSuper In the current implementation, if in the definition of a method, there is an identifier that refers to an attribute, and the identifier is the left hand side (LHS) of an assignment expression or increment/ decrement expression, and the attribute is defined the super class of the class in which the method is defined, then a *WriteSuper* relation holds.

WriteOther In the current implementation, if in the definition of a method, there is an identifier that referes to an attribute, and the identifier is the left hand side (LHS) of an assignment expression or increment/decrement expression, and the attribute is defined in neither the class in which the method is defined, nor the super class of the class in which the method is defined, the a

WriteOther relation holds

Return If, in the definition of a method, there is a return statement, and the type of that returned expression is an entity, then a *Return* relation holds.

The *Return* relation is different with *ReturnType* relation, because a method may return a sub-class instance of the return type it declared.

All the entities, properties of entities, and relations we identified will be stored into the database and will be restored into memory to construct the program model. The following diagram shows the classes that we use to represent the program model:

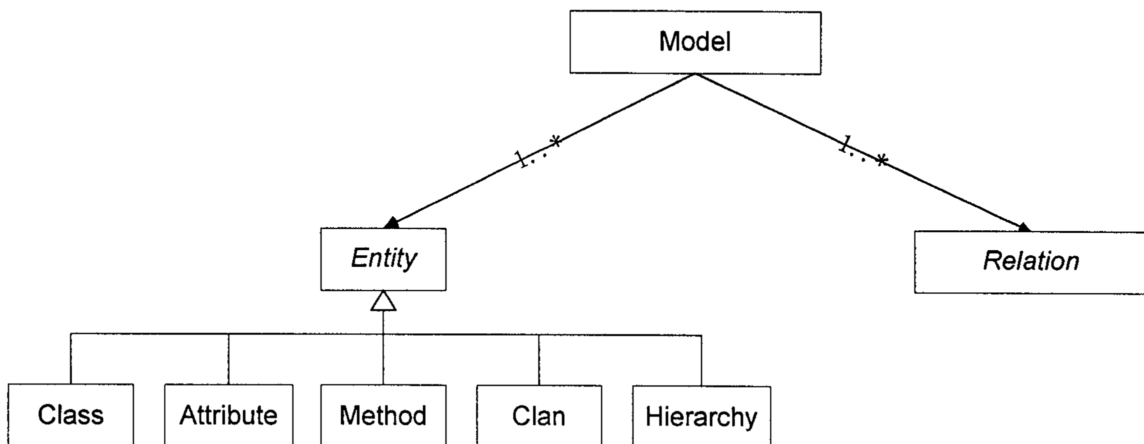


Figure 3.4-4 Structure of Program model

The Model class contains references to a list of Entity objects and a list of Relation

objects. In addition, it also provides a large number of methods to access the entities and their relations. Since the Model class carries too much functionality compared to other classes, we consider a further refinement of the implementation of the program model.

3.5 Design pattern recovery

In the proceeding sections, we presented our effort in formalizing design patterns and analyzing the source code. In the following step, the formal specification of design patterns and the program model of the source will be matched to recover instances of design patterns in the program.

Definition VI: Instance of pattern – let $\boldsymbol{p}$ be a program, $\mathbb{M}(\boldsymbol{p})$ be the program model of $\boldsymbol{p}$, and $\phi(x_1,...x_m)$ be a pattern, if there is a assignment $\{a_1, \ldots, a_m\}$ that satisfies all the constraints in ϕ, and $a_1, \ldots, a_m$ are all entities taken from $\mathbb{M}(\boldsymbol{p})$, we say, there is a instance of pattern ϕ implemented in program $\boldsymbol{p}$.[Ed02]

Following this definition, the matching work is reduced to give assignments to the formulas in the pattern specification. If the result is true, then a pattern is identified. For example, let $\phi(x_1,...x_m)$ be the pattern we want to identify, let $\{a_1, \ldots, a_n\}$ be the set of all entities in the program, then there will be n^m kinds of assignments to the pattern, i.e. the complexity of this algorithm is $O(n^m)$.

To reduce the complexity of the matching process, we adopt a multi-stage filtering algorithm. First, we simplify the pattern specification by weakening the constraints, and apply the simplified specification to the program model to conduct a set of candidate patterns. Next, some weakened constraints are applied to these potential candidates to improve the matching precision. Finally, after all constrains are applied, the specific instances of patterns are identified.

For a given assignment, we need to confirm whether it satisfies all the constraints. Since the knowledge about the pattern is well defined within the LePUS formula, a rule-based algorithm is ideal in this situation. A rule-based system accepts a set of rules, in our case the LePUS formulas, and facts, in our case the program model, as input, and performs identification and comparison of the facts by applying these rules (usually a bunch of IF… THEN statements) to compute the results. The user of such a rule-based system may create or modify rules without having to modify the algorithm.

It should be noted that because of the time and complexity constraints, the LePUS formulas and the matching algorithm are hard coded in the current implementation, which makes our system difficult to maintain.

At present, seven patterns listed in [GHJV94] are implemented, including one

creational pattern – Singleton, five structural patterns – Class Adapter, Object Adapter, Bridge, Composite, and Decorator, and one behavioral pattern – Template Method. We have identified these patterns from several software systems and present the experimental results in the next chapter.

IV. Experimental Results

4.1 Implementation

We have implemented the approach introduced in chapter 3 in Java language version 1.4. This system can automatically analyze Java source code to identify six design patterns. The user may control the system through a console based interface using some predefined commands, including parse source code, create program model, and perform design pattern searching. When a pattern is found in the program model, the system will print out all relative classes and methods participating in this pattern, which provides the user with some guidance in comprehending that particular design structure from a design pattern based perspective.

As an example, the Statement hierarchy of our system implemented a Composite pattern [GHJV94]. The Statement class provides an abstract interface and each concrete subclass that inherits from Statement class maps to each Java language statement, including If statement, For statement, etc. The output of the pattern recovery contains the following information:

Composite Pattern
Component: concept.java.tree.Statement
Leaves:

concept.java.tree.CaseStatement
concept.java.tree.ExpressionStatement
concept.java.tree.ReturnStatement
concept.java.tree.VarDeclarationStatement
concept.java.tree.ThrowStatement
concept.java.tree.LabelStatement
concept.java.tree.DeclarationStatement
concept.java.tree.DefaultStatement

Composites:

concept.java.tree.CatchStatement
concept.java.tree.SwitchStatement
concept.java.tree.BreakStatement
concept.java.tree.ForStatement
concept.java.tree.WhileStatement
concept.java.tree.TryStatement
concept.java.tree.DoStatement
concept.java.tree.ContinueStatement
concept.java.tree.IfStatement
concept.java.tree.CompoundStatement
concept.java.tree.SynchronizedStatement
concept.java.tree.FinallyStatement

Figure 4.1-1 Output of an identified Composite pattern

concept.java.tree.Statement class is the root class of this Composite pattern. The leaf nodes include the statements that don't contain statements, for example, the return statement only has an expression, Declaration statement only has variable declarations, etc. The composites refer to the statements that may contains other statements, for example, If statement contains true case and false case, Do statement has a set of statements as the loop body, and etc.

If the user of our design pattern recovery tools is familiar with the Composite pattern, he/she may easily understand the design of that piece of code, and may redocument the design in pattern language, in which the corresponding relationships of these classes are clearly conveyed. Furthermore, during the maintenance, having the pattern in mind, the discussions about the implementation of Composite pattern introduced in [GHJV94] will give great help to the designers; at least they will not arbitrarily modify one of these classes without considering the others. In general, the recovered design patterns will provide the maintainer with additional insights with respect to reusability, extendibility, and maintainability of the current system design.

4.2 Experiment setting

In order to evaluate the quality of our pattern recovery process, we perform some initial experiments to study issues like memory usage, execution performance, and the precision [FB97] of the pattern recovery. For the experimental setting, we used three different systems: one system was the implementation of our design pattern recovery environment itself, and the other two systems were open source software.

The experiments are performed on a Dell workstation with Pentium IV 2.4G CPU,

1G memory, and 100G Ultra IDE hard disk. The operation system is Windows 2000 SP3. The database system, PostgreSQL 7.3, is installed on another same level computer running Redhat Linux 9 operation system. The two computers are connected by 100M Ethernet switcher.

We monitored the memory usage through the windows task manager, including the memory usage before and after loading the program model.

The execution time for the pattern identification process is provided by an internal system time recorder. We embed a time consumption recorder in our system that displays the starting time and ending time for most of the major processes involved in the design pattern recovery.

Since the specifications of design patterns only describe the structural part of the patterns, the language dependent features and many implementation variations of design patterns may not be considered thoroughly in our implementation. Therefore, some positive false pattern recognition will occur, referring to a situation when a source code artifact looks similar to the structure of a pattern. However, a human with the necessary design pattern domain knowledge might be able to recognize that this particular design was not intended to be a design pattern in that particular context. Same as the method mentioned in [AFC98], we

use the factor *precision* to measure the accuracy of our pattern recovery processes. The ratio precision is the number of true cases retrieved patterns over the number of all patterns in our system's output.

One of the advantages of Java is that javadoc, which is a part of Java language, provides a standard documentation format based on the source code, and it can usually be obtained. In addition to recovering design patterns in the source code, we also searched some frequent keywords related to patterns in existing javadocs to measure how well the documentation is written in terms of documenting design patterns and how consistent the document is with the corresponding source code.

Programs used in the experimental setting.

- **concept.java package 1.0**

  The *concept.java* package is part of CONCEPT project implementing the design pattern recovering. The *concept.java* package includes several sub packages, the language parser, storage, language analysis, program model representation, matching, and a Java de-compiler. This package contains 196 files, 14,768 lines of code (LOC), and 11,658 lines of code without comments.

- **java.util package 1.4.1**

  The *java.util* package is one of the core packages of Java Development Kit. It contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes. This package contains 120 files, 51,993 lines of code (LOC), and 23,567 lines of code without comments.

- **jEdit 4.1**

  The *jEdit* is a text editor oriented to programmers. The source code of *jEdit* 4.1 contains 300 files, 92,266 lines of code (LOC), and 65,872 lines of code without comment.

4.3 Results analysis

4.3.1 Memory usage

The following table shows the statistical information concerning memory usage of the pattern recovering process.

Table 4.3-1 Memory usage of pattern recovering process

	concept.java	java.util	jEdit
Lines Of Code (LOC)	14,768	51,993	92,266
LOC without comment	11,658	23,567	65,872
Number of Classes	201	299	686
Number of Methods	2,621	3,831	6,555
Number of Attributes	384	329	783
Number of Relations	12,480	19,554	40,777
Memory Usage	3495K	6715K	9253K

Note: memory usage is the average of three times program model loading.

From the table it is evident that there exist a direct relationship between program size and the memory consumption of the program model. In particular the relationship between LOC and the memory consumptions are directly related.

4.3.2 Execution time

The following table shows the execution time of the pattern recovering process.

Table 4.3-2 Execution time of pattern recovering process

	concept.java	java.util	jEdit
Lines Of Code (LOC)	14,768	51,993	92,266
Loading program model	2.526sec	2.963sec	4.000sec
Identifying Singleton	0.609sec	1.797sec	8.062sec
Identifying Class Adapter	0.688sec	0.859sec	2.718sec
Identifying Object Adapter	4.688sec	6.750sec	50.344sec
Identifying Bridge	23.469sec	9.313sec	148.218sec
Identifying Composite	0.688sec	1.063sec	2.782sec
Identifying Decorator	2.594sec	5.500sec	7.687sec
Identifying Template Method	0.859sec	2.797sec	4.125sec

The time used to load program model from database depends on the size of the

program, and in most of cases, as the size of the program increased, the time consumption for searching and identifying design pattern increases accordingly.

From the table it can be observed that the recovery processes for some of the patterns takes more time than for other patterns. This observation is not surprising, because some patterns are more complex than others in terms of their structure and therefore requires additional analysis.

The execution time of searching design pattern for programs between 10K and 100K LOC ranges from 0.6sec to 149sec. The observed time complexity is deemed as acceptable, in particular because users usually only analyze partial or sub-systems.

4.3.3 Precision

The following seven tables show the precision of the pattern recovering process.

Table 4.3-3 Precision of recovering Singleton pattern

	concept.java	java.util	jEdit
Number of identified patterns	3	5	3
Number of Positive false cases	0	0	0
Precision	100%	100%	100%

The Singleton pattern, which is a single class pattern, with a specific initiation of its constructors and attributes, is a much simpler pattern than most of the other patterns. The specification of Singleton pattern is well defined and as a result, the precision of recovering singleton pattern is general very high. In our experimental study we were able to recovery, for all three sample programs, 100% of Singleton patterns.

It should be noted that some of the identified pattern are variations of the original Singleton pattern. For example, a class may maintain a list of self instances rather than only one instance. The client may access these instances by the provided parameterized method(s) but cannot create instances of such a class. In these cases, we still consider them as Singleton patterns in term of their intentions.

Table 4.3-4 Precision of recovering Class Adapter pattern

	concept.java	java.util	jEdit
Number of identified patterns	0	1	0
Number of Positive false case	0	0	0
Precision	N/A	100%	N/A

The Class Adapter pattern uses multiple-inheritance as its design principle. Since multiple-inheritance is not supported very well in Java (we consider interface implementation in Java is a type of multiple-inheritance), Class Adapter patterns

are not widely used in Java. The only case we found in the java.util package is structured like the Class Adapter, and was correctly identified as class adapter after verification through a domain expert.

Table 4.3-5 Recovery of the Object Adapter pattern

	concept.java	java.util	jEdit
Number of identified patterns	17	107	71

Table 4.3-6 Recovery of the Bridge pattern

	concept.java	java.util	jEdit
Number of identified patterns	51	119	265

The Object Adapter and Bridge patterns both use some very general object oriented design mechanism as their basic structure. In particular, both of them rely on object composition, which is a very common structure of object oriented design. Therefore, we consider, even incorporate with the knowledge of a domain expert, the identification and recovery of Object Adapter and Bridge pattern is very difficult.

Although we identified a number of cases of Object Adapter and Bridge patterns, and all of them conform to the structures of the two patterns, we are not able to determine the number of positive false cases. This would require very detailed

domain knowledge of the application domain, the design requirements and of the implementation itself.

Based on the experiences of analyzing concept.java package, which is familiar to us, we found most of the identified patterns are positive false. Thus the precision of identifying Object Adapter and Bridge patterns is relatively low.

Table 4.3-7 Precision of recovering Composite pattern

	concept.java	java.util	jEdit
Number of identified patterns	5	17	6
Number of Positive false case	1	14	0
Precision	80%	18%	100%

Composite patterns are usually used to represented part-whole structures in the system. The composite class in this pattern should hold a list of child objects. However, the ways of storing such a list of objects varies from implementation to implementation. Some implementations use arrays to achieve this goal, while some others may use user defined container classes. In our current implementation, only arrays or single object references are considered. As to the user defined container class, more complex techniques are required to identify this kind of object composition relation. As a result, Composite pattern recovery is in a "it depends" situation, i.e. it depends on the implementation of the design – whether programmer used an array or a user defined container class.

In addition, Composite pattern relies on recursive composition to organize an open-ended number of objects [GHJV94]. The *java.util* package, as a utility package, provides a reference implementation for a variety of data structures. Accordingly, there are many self references in this package, e.g. a class may hold an instance of its super class to carry out its responsibility. This kind of relation is a *use* relation rather than object composition, and the distinction cannot be easily determined by computers. That is the reason why so many positive false cases are identified within the *java.util* package. Therefore, we consider the precision of Composite pattern recovery also depends on the application domain.

Table 4.3-8 Recovery of the Decorator pattern

	concept.java	java.util	jEdit
Number of identified patterns	2	55	30

The Decorator pattern has a similar structure to the Composite pattern. A decorator class inherits from a component class to provide a consistent interface; at same time, it also holds a reference to the component to add additional responsibilities. For the Decorator pattern recovery, we cannot provide the precision data, because the structure of a Decorator pattern uses object composition, which usually depends on the intents of the designer and the context of the problem domain. The number of positive false case is not determinable even by a human domain expert.

Table 4.3-9 Precision of recovering Template Method pattern

	concept.java	java.util	jEdit
Number of identified patterns	0	9	6
Number of Positive false case	0	1	2
Precision	N/A	89%	67%

The Template Method pattern is relatively simple in its structure. The template method in this pattern implements a skeleton of an algorithm, in which the implementations of some primitive operations are deferred into the subclasses. All the identified patterns confirm this kind of structure. The commonality of the positive false cases is that all the template methods in these cases are so simple that they cannot be considered as algorithms. Therefore, the precision of Template Method pattern recovery can be improved by applying constrains in measuring the complexity of the template method.

4.3.4 Consistency of design pattern document

For this particular experimental setting, we searched the source code (including the source code itself and the embedded javadocs) of these three systems for keywords related to the patterns. The motivation was twofold: firstly we wanted to see if design patterns are documented in the source; the second motivation was to identify possible inconsistencies between documentation and the source code implementation. The following table provides a summary of the experimental

results.

Table 4.3-10 Keywords related to patterns

Patterns	Keywords
Singleton	Singleton, one instance, sole, solely
Class Adapter	Adapter, Class Adapter, Adaptee, wrapper
Object Adapter	Adapter, Object Adapter, Adaptee, wrapper
Bridge	Bridge, Body, Imp, Implementor
Composite	Composite, part whole, composition, hierarchy, child, parent, leaf,
Decorator	Decorator, wrapper, attach, dynamically
Template Method	Template Method, template, skeleton

Table 4.3-11 Document consistency of java.util

	Number of Pattern Found	Number of Keyword Found	No. of Keyword/ No. of Patterns
Singleton	5	33	6.6
Class Adapter	1	19	19
Object Adapter	107	19	0.18
Bridge	119	74	0.62
Composite	17	588	34.59
Decorator	55	23	0.42
Template Method	9	1	0.11

Table 4.3-12 Document consistency of jEdit

	Number of Pattern Found	Number of Keyword Found	No. of Keyword/ No. of Patterns
Singleton	3	6	2
Class Adapter	0	94	N/A
Object Adapter	71	94	1.32
Bridge	265	49	0.18
Composite	6	900	150
Decorator	30	58	1.93
Template Method	6	1	0.17

The keywords were carefully chosen to avoid terms that are too general. However, some keywords, e.g. "child" and "parent" that correspond to terms used in the Composite pattern, are not only used to describe these types of patterns, but also used in some other situations. In these cases, the noise created by these terms, when they are used in another context, makes the result of our findings meaningless. Nonetheless, we still consider these keywords may give many clues to the readers of the source code in terms of searching design pattern, and the frequency of using these keywords may also indicate how well the patterns are documented.

These keywords were searched from the javadocs and the identifier names of the source code. For a more meaningful analysis of the documentation quality, a more thorough experimental analysis would have to be conducted. In particular all occurrences of the keywords not related to any pattern documentation would have to be filtered out, therefore reducing the noise associated with the non-relevant occurrences. A more detailed and thorough study has to be conducted to validate the possible application of the design pattern recovery techniques in combination with documentation quality assessment. One of the observations, made during this initial experimental setting, was that the precision of pattern recovery may be improved by incorporating some of these keywords in the pattern recovery process.

V. Conclusions and Future Works

In this thesis an approach for the automatic identification of design patterns from the source code was presented. The presented approach translates both design patterns and source code to the same level of abstraction. In a next step both the design pattern specification is matched with the abstract program model that was created from the source code to recover instances of design pattern.

In this research we formalize design patterns using a simplified version of the LePUS language. This version of LePUS eliminates some higher-order concepts to reduce the complexity of the formalism. At the same time it incorporates some extensions on entity, property of entity, and relations. As a result of the pattern formalization, each pattern is specified by a well defined formula that contains entities and the corresponding relations.

The source code is semantically parsed and is further interpreted by either a language analysis framework. Entities and relations are extracted from this level of language analysis and used to create a program model, which contains information concerning pattern recovery, as well as general structural and semantic information about the source code.

The language analysis framework also served as a core part of the CONCEPT project, which addresses more general application purposes within the reverse engineering domain.

The approach presented in the thesis has been implemented in the Java language, and can be applied to identify and recover several design patterns listed in [GHJV94] from Java source code. Three experiments were conducted to demonstrate the performance and precision of the current implementation.

As to the major objective of our research – the recovery of the 23 design patterns described in [GHJV94], we come to the following conclusions based on the development of the approach and the conducted experiments.

- ***Most of the creational patterns and structural patterns can be identified***.

  Currently, we have implemented the Singleton, Adapter, Bridge, Composite, Decorator patterns. Our experiments on three mid-size programs (LOC 10K to 100K) showed that the precision of pattern recovery ranges from 18% to 100%.

  Some patterns, e.g. Singleton, can be identified with a very high precision,

because the structures of these patterns are relatively simple and thus they can be well specified by our formal representation.

Some patterns, e.g. Composite, may have various implementations. The accuracy of recovery of these patterns depends mostly on implementation issues. We also observed that the precision of Composite pattern recovery even depends on the application domain.

Some patterns, e.g. Object Adapter, Bridge, and Decorator, use very general object oriented mechanism in their basic structure, for example, object composition, thus many positive false cases are identified. All the identified patterns confirm the structure of these patterns, but need to be further examined in terms of the intents of the designer.

We are confident that some of the creational and behavioral patterns in the GoF book, e.g. the Abstract Factory, Factory Method, Flyweight, and Proxy pattern, can be identified in a similar fashion. On the other hand some of the other patterns, e.g. the Builder, Prototype, and Façade pattern, are very difficult to recover without domain knowledge. In particular, the Builder pattern separates the construction of a complex object into smaller operations. However, the design of the separation always depends on the

particular problem domain. The implementation of a Prototype pattern needs a clone method to clone an object according to the structure of the object. The Façade pattern provides a general idea about the design of a subsystem, but the structure of the subsystem also depends on the particular problem or application domain.

- ***Behavioral pattern recovery is feasible.***

Although behavioral patterns focus on the behaviors of objects, they still have clear structural descriptions specified in OMT diagrams. Despite the precision, we consider our approach can identify most of them in terms of static structure. Within the CONCEPT project, since some dynamic information can be obtained, e.g. execution trace and dynamic slicing, future work will address to incorporate the dynamic information with the static structure to improve the accuracy of our approach for the recovery of behavioral patterns.

- ***Design patterns widely exist in current software systems.***

We observed that many patterns are implemented in current software systems, according to the results of our initial experiments. By studying the documentation and the naming convention of the program, we also

observed that not all these instances of design patterns are indented to be, i.e. the designers are not aware they used some kinds of patterns, but indeed, their design, whose structure is similar to some patterns, gains much flexibilities and reusability. We consider this conclusion confirms the motifs of design pattern – capturing the proven solutions to recurring problems [GHJV94] and conveying the experiences of software design community.

Future work will address the following issues:

- Extending the expressiveness of the formal specification language.
- Incorporate some dynamic information to improve the precision of pattern recovery, and to identify behavioral patterns.
- Implement a rule based pattern recovery algorithm.
- Implement more language parsers to analyze programs written in some other programming languages, e.g. C++.
- Study the visualized representation of design patterns to facilitate software comprehension.

References

[AFC98] G. Antoniol, R. Fiutem and L. Cristoforetti "*Design Pattern Recovery in Object-Oriented Software*", In Proceedings of the 6th Workshop on Program Comprehension (WPC), pages 153-160, Ischia, Italy, June, 1998.

[Bi89] Biggerstaff, T. J. *"Design recovery for maintenance and reuse".* IEEE Computer 22, 7, 36-49, 1989

[BJ94] K. Beck, R. Johnson, "*Patterns generate architectures*", In proceedings of the 13th European Conference on Object-Oriented Programming, Lecture Notes in Computer Science Nr. 821. 1994

[BMRSS96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, "*Pattern-Oriented Software Architecture: A System of Pattern*", John Wiley & Sons, 1996

[Bo81] B. W. Boehm "*Software Engineering Economics*", Prentice-Hall, Englewood Cliffs, NH, 1981

[CC90] Chikofsky, E. J. and Cross II, J. H., "*Reverse engineering and design recovery: A taxonomy*", IEEE Software 7, 1 (Jan. 1990), 13-17

[Ed01] A. H. Eden, "*Formal Specification of Object Oriented Design*", International Conference on Multidisciplinary Design in Engineering CSME-MDE 2001, November 21—22, 2001

[Ed02] A. H. Eden, "*A Theory of Object-Oriented Design*", Information System Frontiers (Journal of), Vol. 4, No. 4 (November – December 2002). Kluwer Academic Publishers, 2002.

[FB97] W. B. Frakes and R. Baeza-Yates, "*Information Retrieval: Data*

Structures and Algorithms", Prentice-Hall, Englewood Cliffs, NJ, 1997.

[FBBOR99] Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts, *"Refactoring: Improving the Design of Existing Code"*, Addison-Wesley, 1999

[FGMP01] R. Ferenc, J. Gustafsson, L. Muller, J. Paakki, "*Recognizing Design Patterns in C++ programs with the integration of Columbus and Maisa*", In Proceedings of the 7th *Symposium on Programming Languages and Software Tools* (SPLST 2001), Szeged, Hungary, pages 58-70, June 15-16, 2001.

[FH83] R. K. Fjeldstad, W. T. Hamlen, "*Application Program Maintenance Study: Report to Our Respondents*", Proceedings GUIDE 48, Philadelphia, PA, April 1983

[FMBMTG00] R. Ferenc, F. Magyar, A. Beszedes, G. Marton, M. Tarkiainen, T. Gyimothy, "*Columbus 2.0 – Tool for Reverse Engineering Large Object-Oriented Software Systems*", Technical Report TR-2000-002, University of Szeged, 2000.

[GHJV94] E. Gamma, R. Helm, R. Johnson, J. Vlissides "*Design Patterns – Elements of Reusable Object-Oriented Software*", Addison-Wesley, 1994

[GJS96] J. Gosling, B. Joy, and G. Steele. *"The Java Language Specification".* Addison-Wesley, Reading, MA, 1996

[IEEE1219-98] "*IEEE Standard for Software Maintenance*", IEEE std 1219-1998, 1998

[In88] D. C. Ince, *"An introduction to Discrete Mathematics and Formal System Specification"*, Clarendon Press, Oxford, 1988

[KP96] C. Kramer and L. Prechelt. "*Design recovery by automated*

search for structural design patterns in object oriented software", in Third *Working Conference on Reverse Engineering*, pages 208-215, Amsterdam, The Nederlands, March 1996.

[KSRP99] R. K. Keller, R. Schauer, S. Robitaille, P. Page. "*Pattern-based Reverse Engineering of Design Components*", In Proceedings of the Twenty-First *International Conference on Software Engineering*, pages 226-235, Los Angeles, CA, May 1999. IEEE.

[La94] L. Lamport, "*The Temporal Logic of Actions*" ACM Transactions on Programming Languages and Systems, Vol. 16, No. 3, May 1994, pp. 872-923, 1994

[Mc92] C. McClure, "*The Three Rs of Software Automation*", Prentical Hall, Englewood Cliffs, NJ, 1992

[Mi98] T. Mikkonen, "*Formalizing Design Patterns*", Proceedings of the *International Conference on Software Engineering*, April 19-25, 1998, pp. 115-124, Kyoto, Japan, 1998.

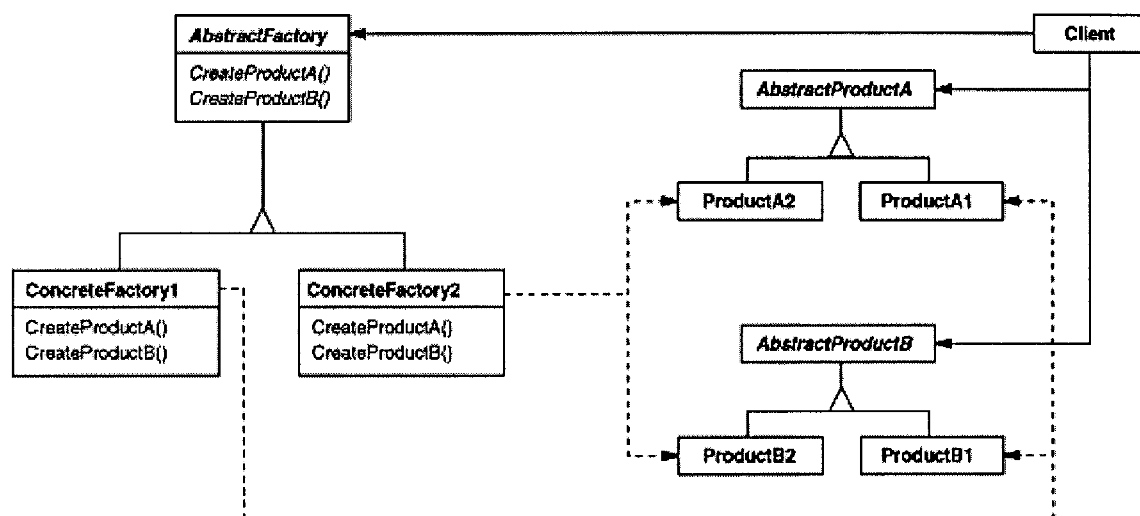
[NGPV00] L. Nenonen, J. Gustafsson, J. Paakki, A.I. Verkamo, "*Measuring object-oriented software architectures from UML diagrams*", In Proceedings of the 4th International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering, 87-100, Sophia Antipolis, France, 2000

[Pe97] A. Petroni, "*Rappresentazione di design □rientate aglioggetti ed analisi basata su metriche*", Tesi di Diploma, University of Trento, Trento, Italy, March 1997.

Appendix Specifications of Design Patterns

1. Abstract Factory [GHJV94, p87]

Abstract Factory pattern intends to provide an interface for creating families of related or dependent objects without specifying their concrete classes.



Participants

- **AbstractFactory**
  - declares an interface for operations that create abstract product objects.
- **ConcreteFactory**
  - implements the operations to create concrete product objects.
- **AbstractProduct**
  - declares an interface for a type of product object.
- **ConcreteProduct**

- defines a product object to be created by the corresponding concrete factory.
  - implements the AbstractProduct interface.
- **Client**
  - uses only interfaces declared by AbstractFactory and AbstractProduct classes.

Collaborations

- Normally a single instance of a ConcreteFactory class is created at run-time. This concrete factory creates product objects having a particular implementation. To create different product objects, clients should use a different concrete factory.
- AbstractFactory defers creation of product objects to its ConcreteFactory subclass.

Specification

Abstract Factory

$AbstractFactory: \mathbf{H}$

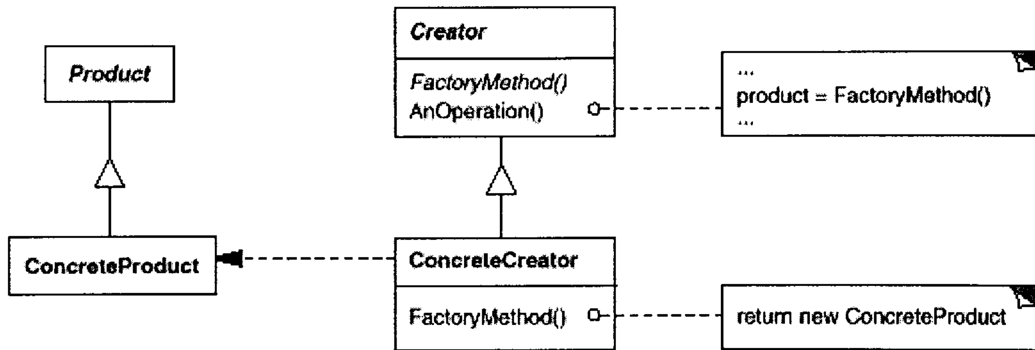
$Products: \mathbf{P}\,(\mathbf{H})$

$FactoryMethods: \mathbf{P}\,(F^{CAbstractFactory})$

$Return: FactoryMethod \leftrightarrow Products$

2. Factory Method [GHJV94, p107]

Factory Method pattern intends to define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.



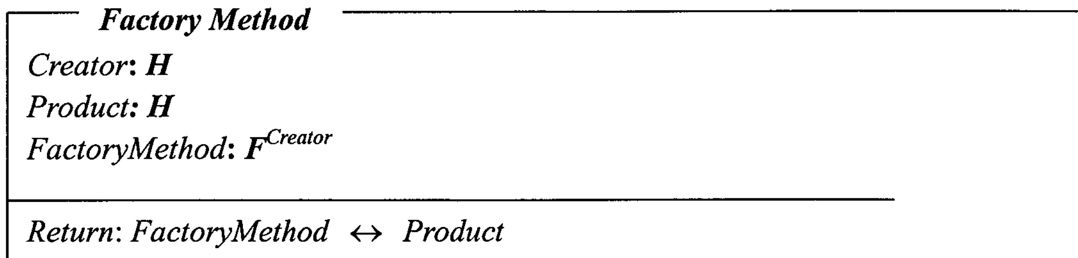
Participants

- **Product**
  - defines the interface of objects the factory method creates.
- **ConcreteProduct**
  - implements the Product interface.
- **Creator**
  - declares the factory method, which returns an object of type Product. Creator may also define a default implementation of the factory method that returns a default ConcreteProduct object.
  - may call the factory method to create a Product object.
- **ConcreteCreator**
  - overrides the factory method to return an instance of a ConcreteProduct.

Collaborations

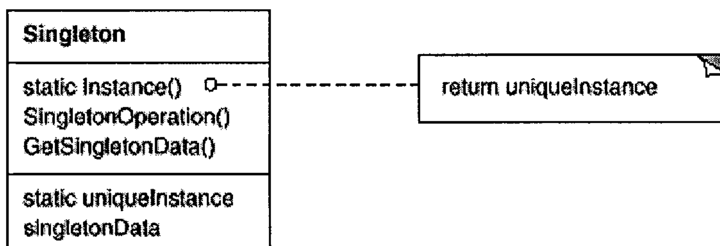
- Creator relies on its subclasses to define the factory method so that it returns an instance of the appropriate ConcreteProduct.

Specification



3. Singleton [GHJV94, p127]

Singleton pattern intends to ensure a class only has one instance, and provide a global point of access to it.



Participants

- **Singleton**
  - defines an Instance operation that lets clients access its unique instance. Instance is a class operation (that is, a class method in Smalltalk and a static member function in C++).
  - may be responsible for creating its own unique instance.

Collaborations

- Clients access a Singleton instance solely through Singleton's Instance operation.

Specification

Singleton

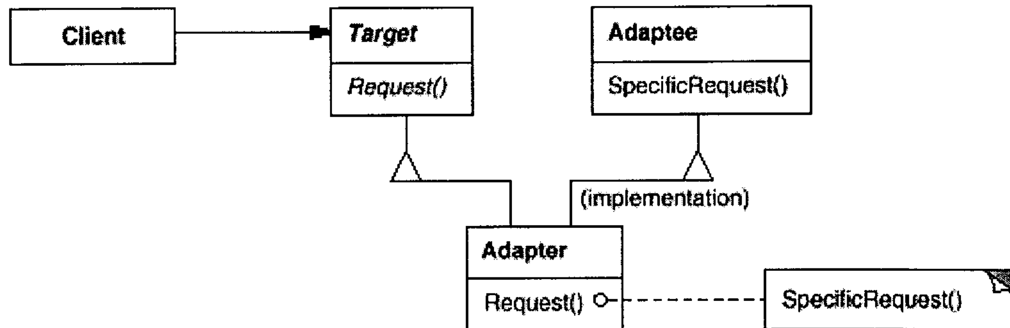
Singleton: ***C***
getInstance: ***M***
constructors: **P** ***(M)***

DefineMethod (getInstance, Singleton)
ReturnType (getInstance, Singleton)
! Private (getInstance)
Static (getInstance)
Construct (constructors, Singleton)
Private (constructors)

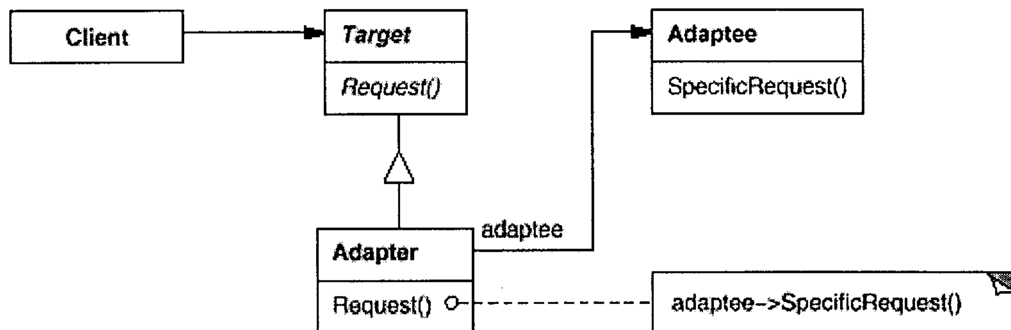
4. Adapter [GHJV94, p139]

Adapter pattern intends to convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Class Adapter



Object Adapter



Participants

- **Target**
  - defines the domain-specific interface that Client uses.
- **Client**
  - collaborates with objects conforming to the Target interface.
- **Adaptee**
  - defines an existing interface that needs adapting.
- **Adapter**
  - adapts the interface of Adaptee to the Target interface.

Collaborations

- Clients call operations on an Adapter instance. In turn, the adapter calls Adaptee operations that carry out the request.

Specification

Class Adapter

Target: **C**
Adaptee: **C**
Adapter: **C**
Request: **M**
RequestImp: **M**
SpecificRequest: **M**

DefineMethod (Request, Target)
DefineMethod (RequestImp, Adapter)
DefineMethod (SpecificRequest, Adaptee)
SameSignature(Request, RequestImp)
Inherit (Adapter, Target)
Inherit (Adapter, Adaptee)
InvokeSuper (RequestImp, SpecificatRequest)

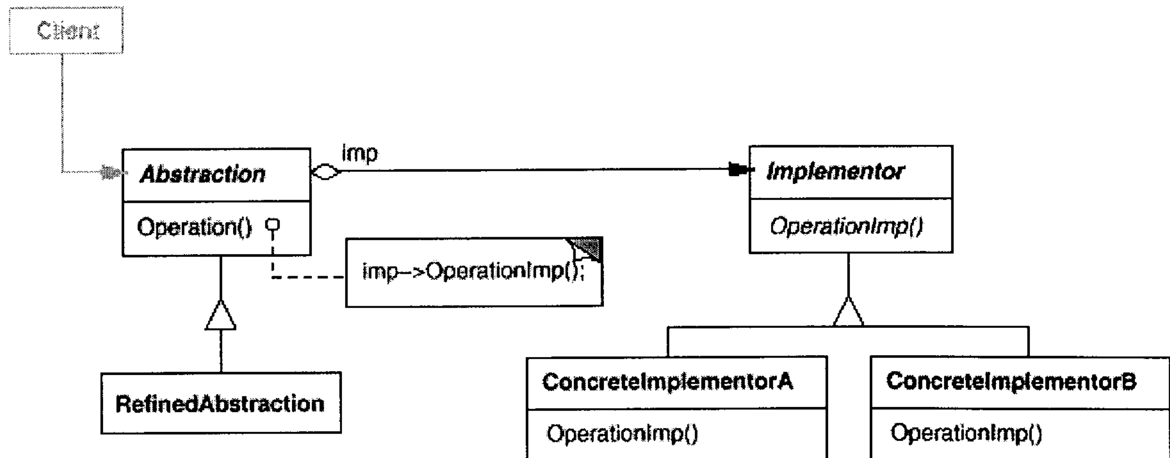
Object Adapter

Target: ***C***
Adaptee: ***C***
Adapter: ***C***
Request: ***M***
RequestImp: ***M***
SpecificRequest: ***M***
adaptee : ***A***

DefineMethod (Request, Target)
DefineMethod (RequestImp, Adapter)
DefineMethod (SpecificRequest, Adaptee)
DefineAttribute (adaptee, Adapter)
SameSignature(Request, RequestImp)
Inherit(Adapter, Target)
AttributeType (adaptee, Adaptee)
InvokeOther(RequestImp, SpecificatRequest)
ReadOwn (RequestImp, adaptee)

5. Bridge [GHJV94, p151]

Bridge pattern intends to decouple an abstraction from its implementation so that the two can vary independently



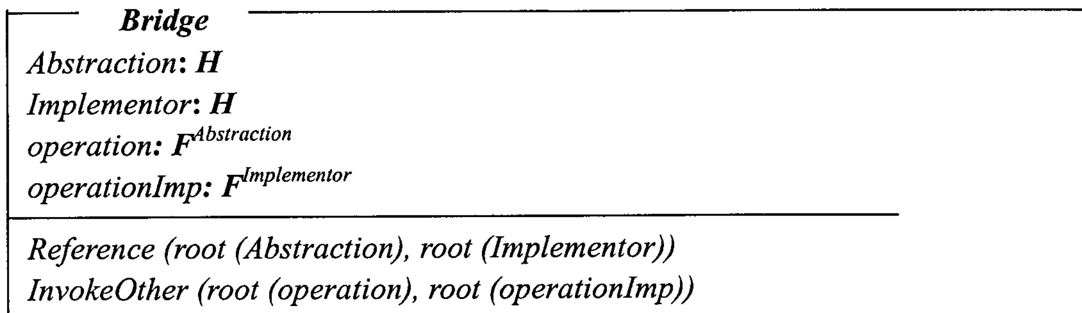
Participants

- **Abstraction**
    - defines the abstraction's interface.
    - maintains a reference to an object of type Implementor.
- **RefinedAbstraction**
    - Extends the interface defined by Abstraction.
- **Implementor**
    - defines the interface for implementation classes. This interface doesn't have to correspond exactly to Abstraction's interface; in fact the two interfaces can be quite different. Typically the Implementor interface provides only primitive operations, and Abstraction defines higher-level operations based on these primitives.
- **ConcreteImplementor**
    - implements the Implementor interface and defines its concrete implementation.

Collaborations

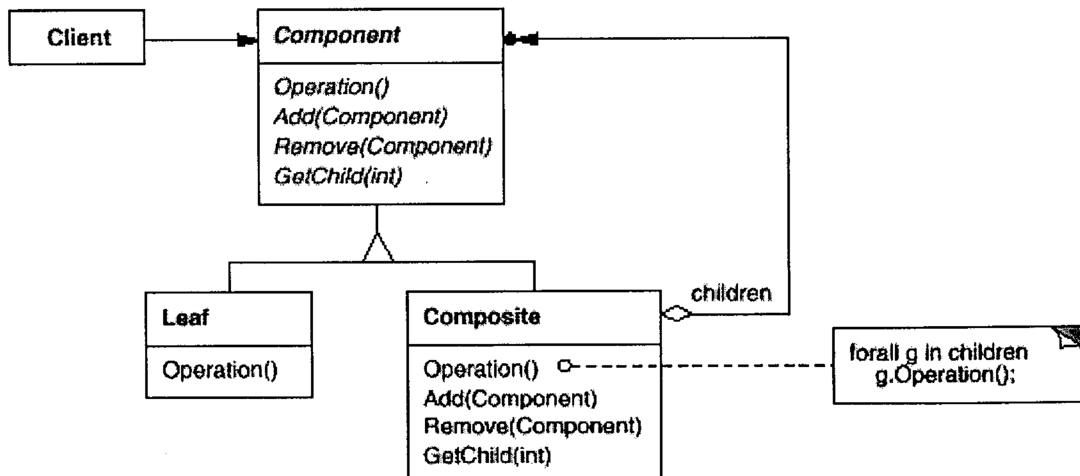
- Abstraction forwards client requests to its Implementor object.

Specification



6. Composite [GHJV94, p163]

Composite pattern intends to compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.



Participants

- **Component**
  - declares the interface for objects in the composition.
  - implements default behavior for the interface common to all classes, as appropriate.
  - declares an interface for accessing and managing its child components.
  - (optional) defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate.
- **Leaf**
  - represents leaf objects in the composition. A leaf has no children.
  - defines behavior for primitive objects in the composition.
- **Composite**
  - defines behavior for components having children.
  - stores child components.
  - implements child-related operations in the Component interface.
- **Client**
  - manipulates objects in the composition through the Component interface.

Collaborations

- Clients use the Component class interface to interact with objects in the composite structure. If the recipient is a Leaf, then the request is handled directly. If the recipient is a Composite, then it usually forwards requests to its child components, possibly performing additional operations before and/or after forwarding.

Specification

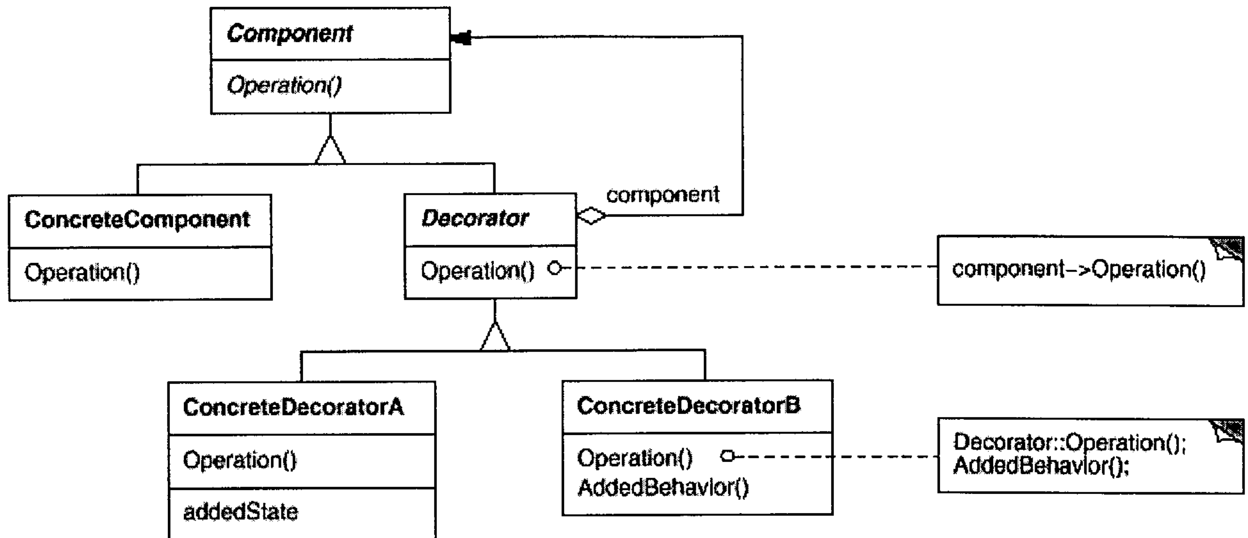
Composite

Component: ***H***
operation: $\boldsymbol{F}^{Component}$
Composite: ***C***
children: ***A***
compositeOperation: ***M***

Inherit (Composite, root (Component))
DefineAttribute (children, Composite)
AttributeType (children, root (Component))
DefineMethod (compositeOperation, Composite)
SameSingature (compositeOperation, root (operation))
InvokeSuper(compositeOperation, root(operation))
ReadOwn(compositeOperation, children)

7. Decorator [GHJV94, p175]

Decorator pattern intends to attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.



Participants

- **Component**
  - defines the interface for objects that can have responsibilities added to them dynamically.
- **ConcreteComponent**
  - defines an object to which additional responsibilities can be attached.
- **Decorator**
  - maintains a reference to a Component object and defines an interface that conforms to Component's interface.
- **ConcreteDecorator**
  - adds responsibilities to the component.

Collaborations

- Decorator forwards requests to its Component object. It may optionally perform additional operations before and after forwarding the request.

Specification

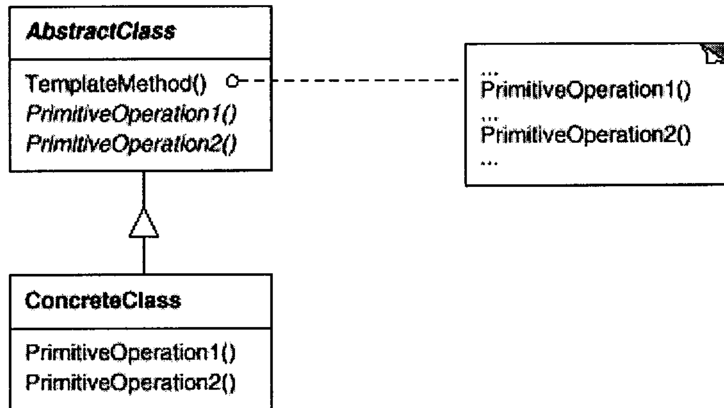
Decorator

Component: ***H***
Decorator: ***H***
operation: $F^{Component}$
component: ***A***
decoratorOperation: ***M***

Inherit (root (Decorator), root (Component))
DefineAttribute (component, root(Decorator))
DefineMethod (decoratorOperation, root(Decorator))
AttributeType (component, root (Component))
SameSignature (decoratorOperation, root (operation))
InvokeSuper (decoratorOperation, root (operation))
ReadOwn (decoratorOperation, component)

8. Template Method [GHJV94, p325]

Template Method pattern intends to define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.



Participants

- **AbstractClass**
  - defines abstract **primitive operations** that concrete subclasses define to implement steps of an algorithm.
  - implements a template method defining the skeleton of an algorithm. The template method calls primitive operations as well as operations defined in AbstractClass or those of other objects.
- **ConcreteClass**
  - implements the primitive operations to carry out subclass-specific steps of the algorithm.

Collaborations

- ConcreteClass relies on AbstractClass to implement the invariant steps of the algorithm.

Specification

Template Method

AbstractClass: ***C***
ConcreteClass: ***C***
templateMethod: ***M***
abstractPrimitiveOperation: ***M***
concretePrimitiveOperation: ***M***

Inherit (ConcreteClass, AbstractClass)
Abstract (AbstractClass)
Abstract (abstractPrimitiveOperation)
DefineMethod(templateMethod, AbstractClass)
DefineMethod(abstractPrimitiveOperation, AbstractClass)
DefineMethod(concretePrimitiveOperation, ConcreteClass)
! Abstract (templateMethod)
SameSingature (abstractPrimitiveOperation, concretePrimitiveOperation)
InvokeOwn(templateMethod, abstractPrimitiveOperation)