Design Pattern Mining from Source Code for Reverse Engineering

N Basu
NIIT Technologies Ltd.
Salt Lake Electronics Complex
Kolkata, India
nilux_b@yahoo.com

S Chatterjee
Dept. of Information Technology
Jadavpur University
Kolkata, India
schatterjee@talk21.com

N Chaki
Dept. of Computer Science
University of Calcutta
Kolkata, India
nccomp@caluniv.ac.in

Abstract

Design patterns are micro architectures that have proved to be reliable, easy-to implement and robust. There is a need in science and industry for recognizing these patterns. This paper aims toward development of a new method for discovering design patterns in the source codes. The method provides a precise specification of how the patterns work by describing basic structural information like inheritance, composition, aggregation and association, and as an indispensable part, by defining call delegation, object creation and operation overriding. We have tried to introduce a new XML–based language, the Extensible Pattern Markup Language XPML, which provides an easy way for the users to modify pattern descriptions to suit their needs, or even to define their own patterns or just classes in certain relations they wish to find. The proposed method is tested on four open-source systems, and is found to be effective in discovering design pattern instances.

Keywords: Design patterns, Unified Modeling Language (UML), XML, Reverse Engineering Engine.

1. Introduction

Design patterns can be a measure of the quality of an object oriented software system. So a software system can be characterized among other things by the number of the design patterns used. Of course, one must fully understand the design patterns who would like to use because if improperly used, they can result in unnecessary huge class structures that can in the worst case even decrease the quality of the code.

The recognition of design patterns is a crucial question in reverse engineering, since they represent a high level of abstraction in Object Oriented Design. As mentioned above, one possible usage might be in measuring the quality of a software system. This can help the project managers to decide whether a code is good enough to be used in the project. Good enough means that the code should be readily understandable, and it should be easy to modify parts of the code without modifying the whole code. So if the design patterns are well documented, it should be much easier to understand the source, and to make appropriate modifications on a well-defined part of it.

Another possible usage is to help in documenting a source without proper comments on patterns for gaining advantages as described above. Well-commented program code is much easier to maintain than the one without comments or with poor comments. We can find pattern instances and this way it helps in inserting comments where it is necessary. Yet another possible usage is in forward engineering, when the system designers inspect the source to see if the coders have implemented the pattern correctly.

Design patterns are described by listing the intents, motivations, applicability, structure (with UML diagrams), participants, collaborations, consequences, implementation details, sample code, known uses and related patterns. All of these except the sample code are written for humans, they do not prescribe how the pattern will be implemented. Even the sample code is only useful for comparing structures, as function names and implementations may differ. So we must find a way to efficiently describe the patterns and then find a way to compare these pattern descriptions to the code. It is obvious, that the direct comparison to the pure code is not going to work. The problem is to find an

intermediate format in which patterns can be described and to which the code can be relatively easily converted, and to find an algorithm that can efficiently find patterns in the transformed code. There have only been a few publications on this topic, most of them search only the *structure* of the patterns. We have tried to develop a new method which tries to solve the part of the problem above and which can be solved based on the information collected from the source code. Our approach tries to provide as much information as possible from the source. First, we have analyzed the C#/Java/C++ source code with the REE system which builds the Semantic Graph. Next, the pattern descriptions are which that are stored in *Extensible* Pattern Markup Language (XPML), a new language based on XML [1] which have been designed especially for this purpose. These pattern descriptions are easy to modify to suit the needs of the users. Finally our algorithm tries to bind the classes found in the source code to pattern classes that are part of the pattern description and checks whether they are related in a way that has been described in the pattern. Here we have used composition, aggregation, association and inheritance relationships for classes, and call delegation, object creation and operation redefinition (overriding) for operations. The results of function-body analysis are what gives us more precision compared to others in detecting design pattern occurrences in the source code.

Our system aims to offers methods to the users to define their patterns in a very precise way. This means that one can define patterns in the sense of functionality. This way only the pattern instances that fulfill these fine-grained requirements will be found. This precise definition is unfortunately not applicable for every design pattern. Some patterns are so general, that they cannot even be described in this way, like the Facade pattern, which defines a higher-level interface to a set of interfaces.

2. Related Work

Only a few works have been published on the topic of recognizing design patterns from C++ source code, and even fewer papers could be identified with concrete results. Kraemer et al. [13] used the Pat system that was developed by Computec GmbH in cooperation with the University of Karlsruhe, Germany. It works on the output of the Paradigm Plus OOCASE tool , which is converted to Prolog facts. It gives the structural analysis of the code based on C++ header files.

The second work [8] describes a method based on a multi-stage reduction strategy using software metrics and structural properties to extract *structural design patterns* from OO design model or source code. Code and design are mapped to an intermediate representation, called Abstract Object Language (AOL).

The authors have found true instances of Adapters. They have also found false instances of Adapters, Bridges and Proxies. The testing included three design patterns: Adapter, Bridge and Proxy. Six public domain code systems were analyzed: LEDA, galib, groff, libg++, mec and socket. The first two stages were executed together (metric based filtering, structural filtering) and then the third stage (delegation filtering). The first stage reduced the input by three to four orders of magnitude. The second stage gave a reduction of one-two orders of magnitude, while the third stage reduced the input two to three times. The precision after the first two steps was about 55%, and an increase of 35% was obtained using the delegation constraint with respect to the use of structural constraints alone. The correctness was 100% because of the conservative approach adopted.

In [6], design pattern detection from C++ source code was accomplished with the integration of two existing tools called Columbus [5] and Maisa. The method combines the extraction capabilities of the Columbus reverse engineering tool with the clause-based pattern mining ability of Maisa. First the C++ code is analyzed to a form understandable for Maisa. This form is a clause-based design notation in Prolog. Afterwards, it is analyzed by Maisa, and instances are searched that match the previously given design pattern descriptions. No real-world projects were analyzed, but the reference implementations of seven different design patterns were identified.

PTIDEJ [10] Pattern Trace Identification, Detection and Enhancement for Java was developed to perform the search using a constraint satisfaction problem (CSP). The authors analyzed the Java AWT and net libraries and found occurrences of Composite and Facade design patterns.

Brown [3] developed a method for detecting design patterns in SmallTalk. He encoded the pattern detection methods for Composite, Decorator, Template Method and Chain of Responsibility into his algorithm. He performed testing on four projects in which he found pattern instances.

3. Reverse Engineering Engine (REE)

The main motivation behind developing the (REE) system was to create a tool that implements a general framework for combining a number of reverse engineering tasks, and to provide a common interface for them. Thus, REE is a framework which supports project handling, data extraction, data representation, data storage and filtering. All these basic tasks of the reverse engineering process are accomplished by using the appropriate modules (*plug-ins*) of the system. Some of these plug-ins are present as basic parts of REE, and the system can be extended to include other reverse engineering functionality as well.

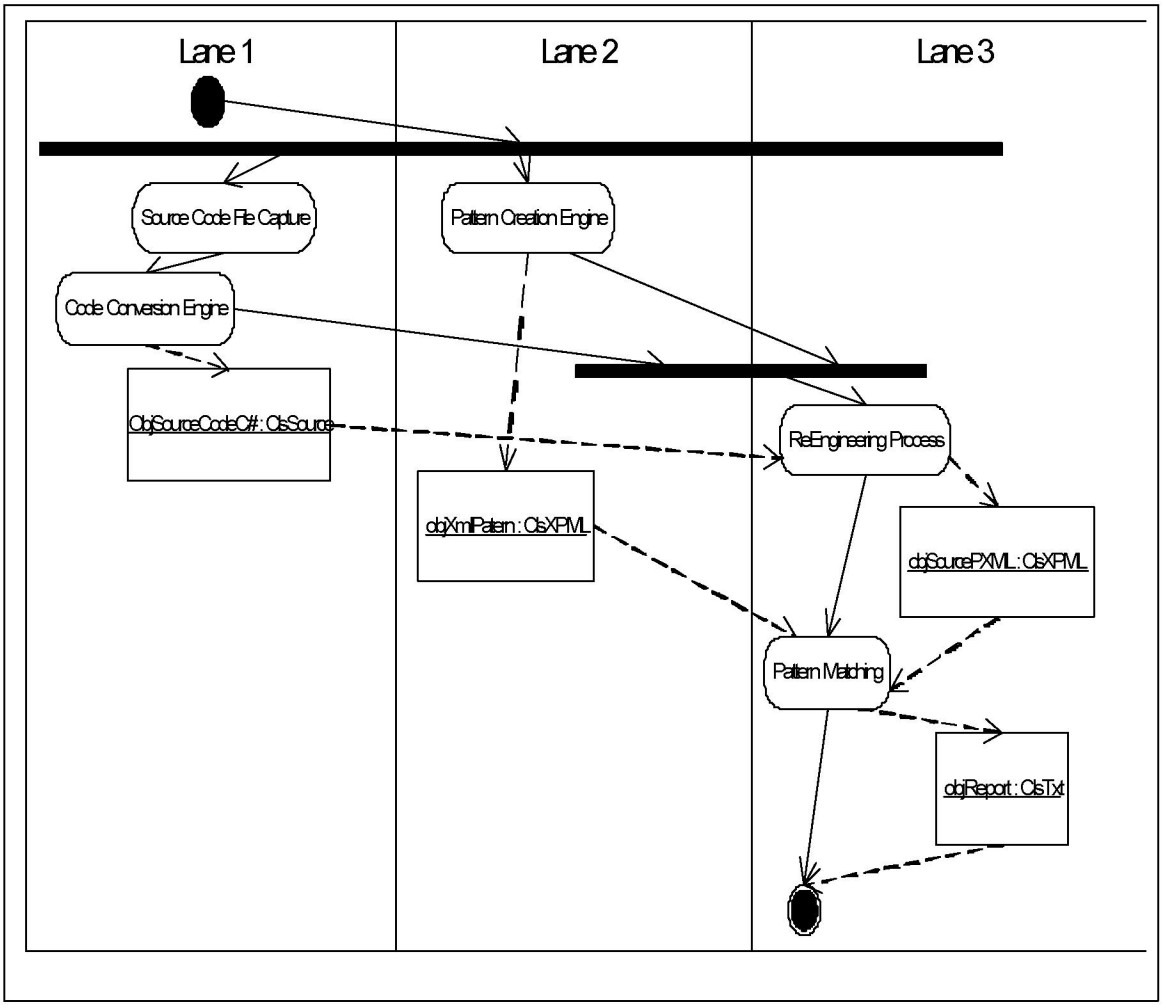

Figure 1: Reverse Engineering Framework

2.1 Mining Design Pattern

Design pattern mining is a process where the structure of a design pattern is searched in the source code. The structure should include the main properties of the design pattern and it should be flexible enough at the same time to describe the slightly distorted occurrences as well. Because of this the structure should be easy to modify and to adapt them to the needs of the user. To meet this commitment, we have tried to use an XML–based language to describe design patterns. The language is easy to understand and the descriptions are easy to modify. Searches for patterns are performed by trying to match source classes to pattern classes. The search is ivided into two stages. The first stage is concerned with filtering the candidates for the pattern classes. In the second stage, source classes are bound to pattern classes, and the constraints are checked to see if they form a pattern.

```
01 <?xml version='1.0'?>
02 <!DOCTYPE DesignPattern SYSTEM 'dpml-1.6.dtd'>
03
04 <DesignPattern name='Proxy'>
05
06 <Class id='id10' name='Subject' isAbstract='true'>
07 <Operation id='id11' name='Request' kind='normal'
08 isVirtual='true' isPureVirtual='true'>
09 <hasTypeRep ref='id50'/>
10 </Operation>
11 </Class>
12
13 <Class id='id20' name='Proxy'>
14 <Base ref='id10'/>
15 <Aggregation ref='id30'/>
16 <Operation id='id21' name='Request' kind='normal'
17 isVirtual='true' isPureVirtual='false'>
18 <defines ref='id11'/>
19 <calls ref='id31'/>
20 <hasTypeRep ref='id50'/>
21 </Operation>
22 <Attribute id='id22' name='realSubject'>
23 <hasTypeRep ref='id52'/>
24 </Attribute>
25 </Class>
26
27 <Class id='id30' name='RealSubject'>
28  <Base ref='id10'/>
29 <Operation id='id31' name='Request' kind='normal'
30 isVirtual='true' isPureVirtual='false'>
31 <defines ref='id11'/>
32 <hasTypeRep ref='id50'/>
33 </Operation>
34 </Class>
35
36 <TypeRep id='id50'>
37 <TypeFormerFunc>
38 <hasReturnTypeRep ref='id51'/>
39 </TypeFormerFunc>
40 </TypeRep>
41
42 <TypeRep id='id51'/>
43
44 <TypeRep id='id52'>
45 <TypeFormerPtr/>
46 <TypeFormerType ref='id30'/>
47 </TypeRep>
48
49 </DesignPattern>
```

Figure 2: Description of Proxy pattern in XMPL

2.2 Extensible Pattern Markup Language – XPML

Our algorithm has used an XML–based language for design pattern description. This is the Extensible Pattern Markup Language –XPML An example in figure 2 illustrates the grammar of XPML.

The root element is the <DesignPattern name='...'> element.Within this element are the classes of the patternand the type representations. Classes are represented by the <Classid='id...' name='...' isAbstract='...' isChangeable='...'> element. The id and name properties are required, while isAbstract and isChangeable are optional. The last property tells that the class can have multiple incarnations in a pattern instance. There is no limit for the number of classes but, unfortunately, the algorithm cost grows exponentially with each class. In our example the root element of the pattern description is the DesignPattern element at line 4, which stores the name of the pattern. The specification of the classes starts with their relations.<Composition ref='id...' accessibility='...' multiplicity='...'/> is the element for the composition relationship.

The accessibility and multiplicity properties can be omitted. The other three relations have the same form: <Aggregation ...>, <Association ...> and <Base...>. When specifying the Base (inheritance relation), it makes no sense to specify the multiplicity as it is always 1. In my example, there are class definitions at lines 6, 13 and 27.The operations of a class can be defined with the <Operation id='id...' name='...' accessibility='...' storageClass='...'kind='...' isVirtual='...' isPureVirtual='...'>element.

The id and name properties are required, but the others can be omitted. It can have a <defines ref='id...'/> child element to specify which operation it needs to redefine (override). The ref property is required. Additional child elements can be <calls ref='id...'> and <creates='id...'> for describing which operations it calls and which objects it creates, respectively. The ref property is required. The operation has a <hasTypeRep ref='id...'/> child element which refers to its type representation (see below). The ref property is required. The operation can have parameters that are specified by the <Parameter id='id...' name='...'> element.

Both properties are required. The parameter element also has a <hasTypeRep ref='id...'/> child element which refers to the type representation of the parameter. Class attributes can be defined with the <Attribute id='id...' name='...' accessibility='...' storageClass='...'> element. The id and name properties are required, but the other two can be omitted. A class attribute – like operations has a <hasTypeRep ref='id...'/> child element which refers to its type representation (see below). In my example the first class (lines 611) called Subject is abstract, and has an operation Request which is pure virtual. Its type (line 9) is represented by TypeRep id50. The second class Proxy (lines 13-25) is derived (line 14) from class Subject–id10 and aggregates the class RealSubject–id30 (line 15). It also has an operation Request (lines 16-21) which is virtual, but not pure virtual. It defines/overrides the inherited operation Request–id11 (line 18) and calls the operation Request–id31 from class RealSubject (line 19). Its type is represented by TypeRep id50 (line 20). This class has an attribute called realSubjectas well (lines 22-24) whose type is represented by TypeRep id52 (line 23). The third class RealSubject (lines 27-34) is derived (line 28) also from class Subject–id10. It also has an operation Request (lines 29-33) which is virtual but not pure virtual. It also defines/overrides the inherited operation Request–id11 (line 31). Its type is represented by TypeRep id50 (line 32).The <TypeRep id='id...'> element represents a type. It can have different child elements which modify the referred type, like <TypeFormerArr/> for arrays, <TypeFormerPtr/> for pointers/references and <TypeFormerFunc> for functions. The reference to the type is stored in a <TypeFormerType ref='id...'/> element. The <TypeFormerFunc> element has a <hasReturnTypeRep ref='id...'/> child element for referring to the return type and can have several <hasParameterTypeRep ref='id...'/> child elements for referring to the types of the parameters.

In the example the first type representation (lines 36-40) shows a function type (line 37) and its return type is described in TypeRep id51 (line 38). The second type representation (line 42) is empty, which means that it can represent any type. The third type representation (lines 44-47) indicates a pointer (line 45) to an object of type id30– RealSubject (line 46).

3. The Algorithm

First, the REE framework analyzes the source code (C#, C++,JAVA) and builds an semantic graph in XML format from it. Secondly, the appropriate XPML pattern description file is loaded into a standard XML DOM [12] tree. The Algorithm then matches the DOM tree to the SG.

This process is similar to graph matching where the vertices are classes and the edges are relations between the classes. Since class diagrams are reconstructed from source code differently by different reverse engineering tools, We will explain here what do we mean by the concept of inheritance, composition, aggregation and association. *Inheritance* means that a class derives from another class, the base class (for design pattern detection, inheritances are stored transitively as well). *Composition* means that a class directly contains another class instance by a data member. *Aggregation* means that a class indirectly contains another class by having a pointer or a reference to it. *Association* means that a class uses another class by getting it as an operation parameter or by returning it as a return type. The class diagram and call graph are used later in the algorithm. The algorithm starts by collecting *source class* (a class found in the source code) candidates for each *pattern class* (a class found in the pattern description). This is accomplished by searching classes with an appropriate number of attributes and operations with desired properties.Only the properties given in the pattern description are checked. The number of different relations of the class is checked as well. If a source class has all the required attributes, operations and relations it is then stored as a candidate for the appropriate pattern class. One source class can become a candidate for multiple pattern classes.

In the second stage, the candidates are filtered. This is done by checking the connections with other pattern class candidates: if two pattern classes are related in some way, then all candidate classes of the first pattern class have to be connected in the same way to at least one candidate class of the second pattern class. If a class does not have the necessary relations, it is removed from the candidates list. This step is repeated iteratively until no further classes are removed.

Next, all combinations of the candidate classes have to be tested to find design pattern instances. The algorithm searches for these combinations recursively to check every possibility (procedure bindSource Class To Pattern Class). If a combination of classes has all the required connections, the attributes and operations of the source classes have to be matched to the attributes and operations of the pattern classes. These attributes and operations are also matched recursively to try out every combination. A source class attribute is bound to a pattern class attribute, if it has all the required properties, checking this time its type as well. For operations, beside the basic properties, the return types and parameters have to be checked too.

After a combination was found, a further check has to be performed to see if the bound functions contain the needed call delegations, object creations and whether they rede-fine/override the appropriate operations. This is performed sequentially, for each class and within them for each function.

If the last check was successful, then the current bound source classes form a design pattern instance. These classes, their path and line information and the role they play in the design pattern instance are displayed together with the bound operations and attributes.

4. Conclusion and Future Work

In this paper, the attempt was to present a method for discovering design pattern instances in JAVA, C#, C++ source code. We have tried out a new approach to the problem of pattern detection, which includes the detection of call delegations, object creations and operation redefinitions. These are the elements that identify pattern occurrences more precisely. The pattern descriptions are stored in an XML–based format, the Extensible Pattern Markup Language (XPML). This gives the user the freedom to modify the patterns, adapt them to his or her own needs, or create new pattern descriptions. we should make even more effort on optimizing our algorithm.

The main advantages of this work are:

1. It provides design pattern instance detection using a user-friendly language for design pattern description. Thus, it will be straightforward to identify pattern instances with this tool
2. It can be helpful in code comprehension, code documentation.
3. Correct pattern implementation testing, which leads to a total quality improvement of the software and in other fields, which require pattern mining.
4. To detect the presence of the anti – patterns in the code, and to eliminate it.

The testing we have performed has revealed several things. Firstly, the more large a project

is, more likely it will contain design patterns. Since the patterns are implemented to solve a specific problem in a specific environment, they rarely follow the strict descriptions[9].The implementations usually violate some rules, so the pattern description must be simplified to be able to recognize them. But the more we simplify the pattern description, the more false instances we will get. Sometimes it is almost impossible to determine whether a pattern instance We have found is a real or false instance. The structure is proper, but the pattern instance does not do what we expect. Thus, while one can find source classes that are connected in the right way, the method cannot check whether they fulfill the intent of the original pattern. This can be checked only manually. Except for the instances of the *Adapter Object* pattern (because of their high number) we have manually checked the found instances.

For future, we need to take care of the standard template containers. Developers rarely use simple arrays, but normally use standard containers, and it is important to find out what kind of data is stored in them. This would allow us to find many more relations, and to find patterns that use lists like Flyweight, Mediator or Observer. We need to write pattern descriptions for simplified pattern instances that do not use every class from the original pattern description. As for example, if there is only one *Implementor* in the *Bridge* pattern, there is no need for an abstract parent class.

References

[1] World Wide Web Consortium (W3C). "Extensible Markup Language (XML)", version 1.0 edition, 2000.

[2] J. Bansiya; " DP++ is a tool for C++ programs." *Dr. Dobb's Journal*, June 1998.

[3] K. Brown. "Design reverse-engineering and automated design pattern detection in Smalltalk." *Master's thesis*. Department of Computer Engineering, North Carolina State University, 1996.

[4] R. Ferenc , A. Besz´edes, "Data Exchange with the Columbus Schema for C++"; Proceedings of the 6th European Conference on Software Maintenance and Reengineering*(CSMR 2002)*, pp. 59–66, Mar. 2002.

[5] R. Ferenc, A. Besz´edes, M.Tarkiainen, T. Gyim´othy Columbus –"Reverse Engineering Tool and Schema for *C++*" Proceedings of the 6th International Conference on Software Maintenance (ICSM 2002), pages 172–181. IEEE Computer Society, Oct. 2002.

[6] R. Ferenc, J. Gustafsson, L. Muller, J. Paakki."Recognizing Design Patterns in C++ programs with the integration of Columbus and Maisa." Acta Cybernetica journal vol. 15, pages 669–682. University of Szeged, 2002.

[7] Rudolf Ferenc, István Siket and Tibor Gyimóthy. "Extracting Facts from Open Source Software." Proceedings of the 20th International Conference on Software Maintenance (ICSM 2004), Chicago Illinois, USA, pages 60-69, September 11-17, 2004.

[8] R. F. G. Antoniol, L. Cristoforetti. "Using Metrics to Identify Design Patterns in Object-Oriented Software." Proceedings of the Fifth International Symposium on Software Metrics (METRICS98), pages 23–34, Nov. 1998.

[9] E. Gamma, R. Helm, R. Johnson, J. Vlissides. "Design Patterns : Elements of Reusable Object-Oriented Software", Addison-Wesley Pub Co, 1995.

[10] Y.-G. Gu´eh´eneuc, N. Jussien. "Using explanations for design patterns identification." Proceedings of IJCAI Workshop on Modelling and Solving Problems with Constraints, pages 57–64, Aug. 2001.

[11] International Standards Organization. *Programming languages C++*, ISO/IEC 14882:1998(E) edition, 1998.

[12] World Wide Web Consortium (W3C). "Document ObjectModel (DOM), 2000"

[13] C. Kraemer and L. Prechelt; "Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software." Proceedings of the 3rd Working Conference on Reverse Engineering, pp. 208–215, Nov. 1996.

[14] J. Paakki, A. Karhinen, J. Gustafsson, L. Nenonen, A. Verkamo. "Software metrics by architectural pattern mining." Proceedings of the International Conference on Software: Theory and Practice (16th IFIP World Computer Congress), pages 325–332, 2000.

[15] K. Mehlhorn, S. Naeher Leda: "A platform for combinatorial and geometric computing." Cambridge University Press, 1997.

[16] Object Management Group Inc. "OMG Unified Modeling Language Specification" version 1.3 edition, 1999.

[17] J. Paakki, A. Karhinen, J. Gustafsson, L. Nenonen, A. Verkamo. "Software metrics by architectural pattern mining." Proceedings of the International Conference on Software: Theory and Practice (16th IFIP World Computer Congress), pages 325–332, 2000.