Visualization of Graphical Modeling Framework as Recovery Process for Reverse Engineering in Object Oriented Design

Kishor R. Kolhe
Research Scholar,
JJTU, Jhunjhunu (Rajasthan), India

Akhilesh R. Upadhyay, PhD.
Dept.of EC Engg., Sagar Institute of Research and Technology – Bhopal, 462041(M.P.), India

ABSTRACT

For measuring software quality, majority of approaches focus on metric calculation based on code, which comes very late in the software development life cycle. The proposed approach presents a forward as well as reverse engineering approach that will detect software design patterns in UML model for forward engineering and from Java source code as a part of reverse engineering. Our approach uses structural, behavioral and semantic analysis. We introduce behavioral and semantic analysis that removes false positives from our structural analysis results. We are interested in assessing the quality of the software design by checking whether it conforms to design pattern and calculating package software metrics. Based on these two parameters the quality of the software system can be analyzed. We provide a tool that implements our approach. An XML schema of design pattern(s) which further facilitates to automate the process of design pattern identification given a class diagram with the help of a tool base. Design patterns are a proven way to build high-quality software.

Keywords: Design Patterns, Semantic Analysis, XML Schema.

1. INTRODUCTION

To avoid unnecessary complexity, and promoting code reuse, maintainability and extensibility Design patterns act as recurring solutions and offer significant benefits. There are several code based design pattern detection approaches which currently exist. However, these approaches have the drawback that the design pattern analysis is performed at implementation level and not at design level. This is at much later stage in the software development life-cycle, because of which good design decisions cannot be incorporated at early stage of software development. That is why design based pattern recognition addressed as a part of our approach is more suitable for forward engineering activities.

As a part of our approach, we propose, the quality of software architecture can be analyzed on the basis of conformance to the design patterns. Patterns are often used in software architecture designs to implement simple concept and have proven their value over time. Design pattern symbolizes good arrangement of data and control as a part of software structure. Therefore it is useful to check the presence of a pattern in software architecture to assess the quality of the design. The quality can be quantified by the degree of conformance software architecture has with a set of already defined design patterns. In our two pronged approach, this is the first factor which we have considered for quality analysis.

For measuring software quality, majority of approaches focus on metric calculation based on code, which comes very late in the software development life cycle. If the calculated Object Oriented metrics are poor at this stage lot of efforts are needed to correct them. Therefore it is better to calculate the Pattern Software Metrics at Design Phase to assure Software Quality well before the Implementation starts. This paper describes how Design Patterns can be identified from UML class diagrams and their quality also can be assessed at the same time i.e. at Design phase.

2. LITERATURE REVIEW

Maplesden [13] proposes the Design Pattern Modelling Language (DPML), a notation supporting the specification of design pattern solutions and their instantiation into UML design models. DPML supports incorporation of patterns at design-time, rather than program coding, assuming that if patterns can be effectively incorporated into a UML class model then conversion to code is straightforward. Though it is just a specification method no one has tested its effectiveness.

Costagliola [6] proposed a dual approach for pattern detection including code and design levels. At the code level, the input is OO source code, which is pre-processed by the Source Code Extractor to obtain an intermediate representation. At the design level, the Class Diagram Abstractor is able to import this representation to generate the corresponding graph structure. The SVG translator adds layout information to the graph, and the corresponding UML class diagram is translated in SVG format. This approach does not require a pre-processing phase to reduce the search space complexity and the cardinality of the set of the retrieved pattern candidates, as carried out by an earlier discussed method by Antoniol [8]. This tool lacks in scalability. Arcelli [9] suggests a process of design pattern detection which is based on micro architectures recognition. Instead of analyzing source code directly, they summarizes it into a set of structures, called subcomponents or micro architectures, which are not ambiguous and involve a limited, number of types. This resolves the scalability issue. The results obtained are then processed by other two modules, one called Joiner that identifies sets of classes which could be "good" candidates to be design pattern instances and the other, called Neural Network which evaluates if the candidates identified by the Joiner are really "good" or not. Results obtained are not completely satisfactory but can be improved by building a larger and more balanced dataset, in order to have more reliable results. All these design based approaches cover structural aspects of design pattern. Later paragraph talk about the design based pattern recognition approaches which cover both structural and behavioral aspects of design patterns.

At the design level, the Class Diagram Abstractor is able to import this representation to generate the corresponding graph structure. The SVG translator adds layout information to the graph, and the corresponding UML class diagram is translated in SVG format. This approach does not require a pre-processing phase to reduce the search space complexity and the cardinality of the set of the retrieved pattern candidates, as carried out by an earlier discussed method by Antoniol [8]. This tool lacks in scalability.

3. SYSTEM PROPOSED

3.1 Design Patterns

With the increasing complexity and size of the software systems, understanding and changing of these systems become difficult tasks, particularly when the architecture and design documentations are incomplete, missing over time, and inconsistent with the source code. Recovering the original design decisions and tradeoffs may help developers to understand large systems and make change more easily. Design patterns generally document the design decisions and tradeoffs as well as possible ways for future evolutions. Thus, recovering the design patterns applied in a software system can assist to cope with the complexity of large systems. Such recovery processes are typically not done from scratch but take advantages of some existing reverse engineering tools to extract the important information from source code into some intermediate representations, such as UML diagrams. When a design pattern is applied in a design, on the other hand, the role information about its participants is generally lost. Recovering such information from the UML diagrams can help the designers to understand the design and communicate with other designers. While the UML diagrams are normally stored in some proprietary format, it is hard to directly manipulate them. To solve this problem, we use the XMI standard to serialize UML into XML file and use it as the intermediate representation. XMI is an XML-based standard proposed by the Object Management Group that maps UML to XML. We use XMI as the intermediate representation based on the following reasons. First, XMI is an interchange format for metadata in terms of the Meta Object Facility. While UML models are generally persisted in some proprietary format of certain tool platforms, XMI specifies how UML models are mapped into a platform-independent XML file. By representing a UML model in XML, the UML model can be searched for patterns.

Thus, the structural analysis can be reduced to the matching of the design pattern matrix with the system matrix as well as the weights of the design pattern classes with the weights of the system classes. We call it a match as long as there exists a sub matrix of the system matrix such that all cells of the sub matrix are the integral multiples of the corresponding cells in the design pattern matrix and that the weights of the classes Fig.3-1. Overall architecture of the approach in the sub matrix are the integral multiples of the weights of the corresponding design pattern classes. We relax the criteria of structural analysis to reduce the false negative cases. As a consequence, however, the number of false positive cases may increase. The false positive cases in the structural analysis results can be eliminated in the later analysis processes, i.e., the behaviour and semantic analysis. Our behavioural analysis checks whether a desired method invocation exists in a class with the right signatures and polymorphic definitions. Different design patterns may require different behavioural analyses which can be determined by the pattern behavioural characteristics described in the XML file of pattern definition. Some design patterns, such as Bridge and Strategy, are similar in their structures and behaviours. They may only differ from their intents and motivations.

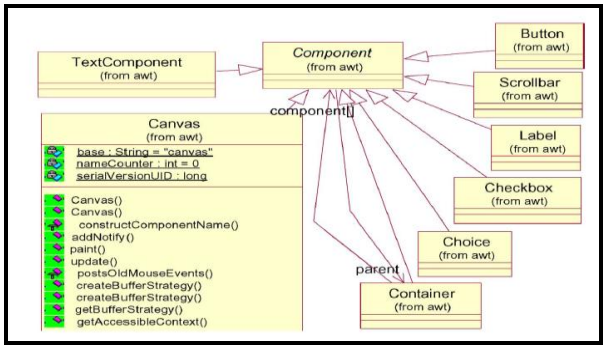

Fig. 3-1. Class diagram — from Java AWT

Although source code generally retains no such semantic information from system design, the naming convention of classes may actually provide some trace of the original design intents and motivations. For example, several class names in the Java.awt contain "Strategy," which is a good indication of the original intents. Our semantic analysis checks the naming conventions when the distinctions are needed. Although naming conventions are not always observed by developers, they actually help to distinguish patterns in many cases as shown in our experiments. The semantic characteristics that need to check for each design pattern are also provided by the pattern definition file in XML.

The behavioural and semantic analyses may require checking the source code directly, in addition to the intermediate representations. However, such checks are based on the results from structural analysis so that only particular classes and methods, instead of the entire source code, are checked.

Table 3.1 – Matrix for class diagram in Fig. 3-1

	Button	Canvas	Checkbox	Choice	Component	Container	Label	Scrollbar	TextComponent
Button	1	1	1	1	7	1	1	1	1
Canvas	1	1	1	1	7	1	1	1	1
Checkbox	1	1	1	1	7	1	1	1	1
Choice	1	1	1	1	7	1	1	1	1
Component	1	1	1	1	1	5	1	1	1
Container	1	1	1	1	35	1	1	1	1
Label	1	1	1	1	7	1	1	1	1
Scrollbar	1	1	1	1	7	1	1	1	1
TextComponent	1	1	1	1	7	1	1	1	1

3.2 Formal Specification of our approach

3.2.1 Structural Analysis

In the previous section, we introduce the main ideas of our approach. To be more precise, clear, and unambiguous, we formally specify our pattern recovery approach in this section. We use three patterns, namely, the Adapter, Strategy and Composite patterns, as examples to illustrate our approach. Structural Analysis as discussed in the previous section, the structural analysis concentrates on the classes in a software system and their attributes, operations, and relationships with other classes. More specifically, we define set ELM for attributes and operations as well as set REL for relations, such as association, generalization, dependence, aggregation, and realization as follows:

ELM ={attr, oper}

REL ={assoc, gener, depd, aggr, realz}.

Class represents a set of classes. PN is a set of prime numbers.

For example

PN = {2, 3, 5, 7, 11, 13}.

Definition 3.1 (Class-to-Class Relation): The relation between two classes is a function

r : Class × Class → 2REL.

Example 3.1 (Class-to-Class Relation):

r(A,B) ={assoc, gener, depd}

r(A,B) ={aggr}

r(A,B) ={realz}

represent that classes A and B have the association, generalization, and dependence relationships, aggregation relationship, or realization relationship, respectively.

Definition 3.2 (Encoding Function): The elements of sets ELM and REL are assigned with a unique prime number by a function

ρ : ELM ∪ REL → PN.

Example 3.2 (Encoding Function): We use the following encodings in this paper:

ρ(attr) =2

ρ(oper) =3

ρ(assoc) =5

ρ(gener) =7

ρ(depd) =11

ρ(aggr) =13.

Definition 3.3 (Cell Value Function): All relationships between two classes are mapped into n integer by a function

γ : Class × Class → N

such that ∀A, B ∈ Class

γ(A,B) = i

ρ(Ri) ∀Ri ∈ r(A,B) if r(A,B) = i

γ(A,B) =1, if r(A,B) = i.

Definition 3.4 (System Matrix): The relationships between the classes of a system are defined as a square matrix

Am = (aij)m

Where aij = γ(Ci, Cj), Ci, Cj ∈ Class, 1 ≤ i, j ≤ m, and

|Class| = m.

Definition 3.5 (Weight): The weight of each class is defined as a function

ω : Class → N

such that ∀A ∈ Class

ω(A) = ρm(attr) × ρn(oper) ×i

γ(A,Ci)

where 1 ≤ i ≤ l, Ci ∈ Class, miss the number of attributes that class A contains, n is the number of operations that class A contains, and l = |Class|.

Definition 3.6 (System Weight Vector): The weights of all classes in a system are defined by the following vector:

Vector Bm = (bi)m

where bi = ω(Ci), Ci ∈ Class, 1 ≤ i ≤ m, and |Class| = m.

Consider a set of design patterns that need to be discovered from a software system

PATTERN={adapter, bridge, strategy, composite, . . .}.

Definition 3.7 (Pattern Class): All classes that participate in a design pattern are defined as its pattern classes, and ∀p ∈ PATTERN, Class(p) represents the set of classes participating pattern p.

Definition 3.8 (Pattern Matrix): The relationships between the classes of a design pattern are defined as a square matrix

DM(p) = Am = (aij)m, p∈ PATTERN

where aij = γ(Ci, Cj), Ci, Cj ∈ Class(p), 1 ≤ i, j ≤ m, and |Class(p)| = m.

Definition 3.9 (Pattern Weight Vector): The weights of all classes in a design pattern are defined by the following vector:

Vector DW(p) = Bm = (bi)m, p∈ PATTERN

where bi = ω(Ci), Ci ∈ Class(p), 1 ≤ i ≤ m, and

|Class(p)| = m.

Definition 3.10 (Matrix Match): Consider a system matrix

Am = (aij)m, |Class| = m, and a design pattern p ∈PATTERN, |Class(p)| = n with its matrix DM(p) =(dij)n. If there exists a submatrix of Am

sub An = (sij)n = A[k1, k2, . . . , kn; k1, k2, . . . , kn],

(1 ≤ k1, k2, . . . , kn ≤ m)

such that

sij mod dij = 0, 1 ≤ i, j ≤ n

then the pattern matrix matches the system matrix.

Definition 3.11 (Weight Match): Consider a system weight

vector Bm = (bi)m, |Class| = m, and a design pattern p ∈

PATTERN, |Class(p)| = n with its weight vectorDW(p) =

(di)n. If there exists a subvector of Bm

sub Bn = (si)n = B[k1, k2, . . . , kn],

(1 ≤ k1, k2, . . . , kn ≤ m)

such that

si mod di = 0, 1 ≤ i ≤ n

then the pattern weight vector matches the system weight vector.

Definition 3.12 (Pattern Structure Match): When both the matrix and weight of a design pattern match those of a system, it is defined as structure match. This definition can be derived directly from the previous two definitions. Informally speaking, the previous definitions use matrices and weights to represent the structural information of systems and patterns and define the matching of a pattern structure with a system structure. More specifically, the system matrix (Definition 3.4) or pattern matrix (Definition 3.8) describes the relationships, such as generalization and association, between the classes in a system or a pattern, respectively. Pattern matrix and weights serve as the criteria of structural analysis. When there is a matrix match (Definition 3.10) between a system matrix and a pattern matrix, it shows that the system

includes some classes having the same relationships as those in the pattern. If the weights of these classes in the system also match those of the classes in the pattern, which is called a weight match (Definition 3.11), it shows that these classes have the required numbers of attributes and operations by the corresponding pattern. Therefore, these classes, whose matrix and weights match those of the pattern, can be considered as a structure match with the pattern (Definition 3.12) and, thus, a candidate instance of the pattern.

4. EXPERIMENTS & RESULT ANALYSIS

4.1 Experiments for Adapter Pattern

The DPI is given as an input the system which has adapter design pattern implemented in its design. Our DPI should be able to extract pattern related information i.e. the name of the pattern and its components and their role names.

The UML file which is given as input is as shown in Fig 4-1

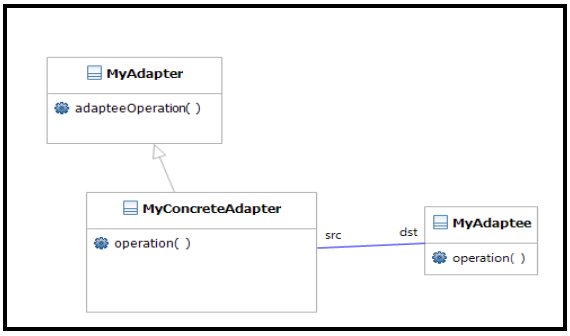

Fig 4-1 Class diagram containing Adapter Design Pattern

The source code for the same is as shown in the following listing

MyAdaptee.java

```
public class MyAdaptee {
        public void xyz() {
                // TODO Auto-generated method stub
        }
}
```

MyConcreteAdapter.java

```
public class MyConcreteAdapter extends MyAdapter {
        private MyAdaptee adaptee = null;
        @Override
        public void operation() {
                adaptee.xyz();
        }
}
```

MyAdapter.java

```
public class MyAdapter {
        public void operation() {
                // TODO Auto-generated method stub
        }
}
```

The Results found by DPI is as shown below

UML file : C:\DPIfinalnew\res\uml\my_adapter.uml

---------- Identifying Design Patterns ----------

Source Folder : C:\DPIfinalnew\res\source\adapter

Matrix

	MyConcreteAdapter	MyAdapter	MyAdaptee
MyConcreteAdapter	1	17	5
MyAdapter	1	1	1
MyAdaptee	1	1	1

Weight

MyConcreteAdapter : 255

MyAdapter : 3

MyAdaptee : 3

Structural Analysis

Below design patterns are found :-

.....

Pattern Name : ADAPTER PATTERN

Adapter : MyAdapter

ConcreteAdapter : MyConcreteAdapter

Adaptee : MyAdaptee

Behavioral Analysis

Below design patterns successfully passed Behavioral Analysis test -

.....

Pattern Name : ADAPTER PATTERN

Adapter : MyAdapter

ConcreteAdapter : MyConcreteAdapter

Adaptee : MyAdaptee

Semantic Analysis

Below design patterns successfully passed Semantic Analysis test :-

.....

Pattern Name : ADAPTER PATTERN

Adapter : MyAdapter

ConcreteAdapter : MyConcreteAdapter

Adaptee : MyAdaptee

---------- Done ----------

We can see that in structural, behavioral and semantic analysis Adapter design pattern is identified.

5. CONCLUSION AND FUTURE WORK

In our system, we have presented the formal specification of our design pattern recovery approach and experimental results. Our approach uses XMI as the intermediate representation format to represent the UML diagram information extracted from source code. XMI is a standard for metadata exchange, which allows our approach to work with existing software design and development tools, such as Eclipse. Our approach includes structural, behavioural, and semantic analyses, each of which refines the results from the previous phases. Our approach also uses matrices and weights encoded by prime numbers to represent object-oriented design information, which facilitates the pattern-matching processes. Based on our approach, we have developed a tool, called Design Pattern Identifier, to recover design patterns.

During structural analysis phase, our tool extracts the structural information of the pattern and encodes it into a matrix and weights in a similar way as we encode the system. Thus, the structural analysis can be reduced to the matching of the design pattern matrix with the system matrix as well as the weights of the design pattern classes with the weights of the system classes.

We have implemented it for Composite, Adapter and Strategy Design Patterns and getting good results. The same approach can be implemented for all the 23 Design Patterns. The Pattern Software metrics is also implemented by using matrix approach and if they match the quality attributes as specified by quality assurance norms they qualify the quality check. If the values of Efferent and Afferent Coupling if less than 3, then the design is called good else it is termed as bad. If the install ability is 0, the design is evaluated as a stable package.

6. REFERENCES

[1] C. Kramer and L. Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In Proc. of the 3 Working Conference on Reverse Engineering (WCRE), Monterey, A, pages 208-215. IEEE Computer Society Press, November 1996.

[2] G. Antoniol, G. Casazza, M. di Penta, and R. Fiutem, "Object-Oriented Design Patterns Recovery," J. Systems and Software,vol.59,pp.181-196, http://web.soccerlab.polymtl.ca/~antoniol/publications/index.html, Nov. 2001.

[3] R. K. Keller, R. Schauer, S. Robitaille, and P. Page. Pattern based reverse-engineering of design components. In ICSE,pages 226–235, 1999.

[4] J. Niere, J.P. Wadsack, and A. Zudorf, "Recovering UML Diagrams from Java Code Using Patterns," Proc. Second Workshop Soft Computing Applied to Software Eng., J.H. Jahnke and C. Ryan,eds., pp. 89-97, http://trese.cs.utwente.nl/scase/scase-2/Proceedings.pdf, Feb. 2001.

[5] Sergiu Dascalu, Ning Hao, Narayan Debnath "Design Patterns Automation with Template Library" 2005 IEEE International Symposium on Signal Processing

[6] Gennaro Costagliola, Andrea De Lucia, Vincenzo Deufemia, Carmine Gravino, Michele Risi" Design Pattern Recovery by Visual Language Parsing" Proceedings of the Ninth European Conference on Software Maintenance and Reengineering (CSMR'05)1534-5351/05 $20.00 © 2005 IEEE

[7] D. Heuzeroth, T. Holl, G. Hogstrom, and W. Lowe, "Automatic design pattern detection," in Proc. 11th IWPC, 2003, pp. 94–103.

[8] G. Antoniol, R. Fiutem and L. Cristoforetti" Design Pattern Recovery in Object-Oriented Software" Istituto per la Ricerca Scientifica e TecnologicaPovo (Trento), Italy I-38050

[9] Francesca Arcelli, Stefano Masiero, Claudia Raibulet" Elemental Design Patterns Recognition In Java" Proceedings of the 13th IEEE International Workshop on Software Technology and Engineering Practice (STEP'05)0-7695-2639-X/05 $20.00 © 2005T.

[10] Taibi, D. Check, and L. Ngo. Formal specification of design patterns-a balanced approach. Journal of Object Technology,2(4), July-August 2003.

[11] Kim and W. Shen"Using Role Based Modelling Language (RBML)to characterise Model Families"2002

[12] Bayley and H. Zhu. Formalising design patterns in predicate logic. In Proc. of SEFM'07

[13] D. Mapdlsden, J. Hosking, and J. Grundy, "Design Pattern Modelling and Instantiation Using DPML," Proc. 40th Int'l Conf.Object-Oriented Languages and Systems (TOOLS Pacific '02), 2002

[14] W. P. Stevens, G. J. Myers, and L. L.Constantine, "Structured design," IBM Systems Journal, vol. 13, pp. 115–139, 1974.

[15] Model Driven Architecture. [Online]. Available: http://www.omg.org/mda/

[16] G. Booch, J. Rumbaugh, and I. Jacobson, The Unified Modeling Language User Guide. Reading, MA: Addison-Wesley, 1999.

[17] Jing Dong, Senior Member, IEEE, Yajing Zhao, and Yongtao Sun" A Matrix-Based Approach to Recovering Design Patterns"IEEE TRANSACTIONS ON SYSTEMS, MAN, AND CYBERNETICS—PART A: SYSTEMS AND HUMANS, VOL. 39, NO. 6, NOVEMBER 2009

[18] Chidamber S.R., Kemerer C.F., A metrics suite for object oriented design, Software Engineering, IEEE Transactions, Vol 20, (1994) 476 -493

[19] Chitra S. Atole and K. V. Kale**,"** Assessment of Package Cohesion and Coupling Principles for Predicting the Quality of Object Oriented Design**"** 1-4244-0682-X/06/$20.00 ©2006 IEEE

[20] W Rebecca, W Brian, W Lauren, "Designing Object Oriented Software" Prentice Hall 2000.

AUTHOR'S PROFILE

MR. KISHOR R. KOLHE is persuing his PhD from JJTU, Jhunjhunu (Rajasthan), India. He obtained his M. Tech. in Information Technology degree from the Bharati Vidyapeeth Deemed University, Pune [M.S.] and B.E (Hons) Electronics Engineering from S.G.G.S. Institute of Engineering & Technology, Nanded [M.S.] in year 2011 and 1996 respectively.

He is currently working as Assistant Professor in Information Technology Department at Trinity College of Engineering and Research, Pune, India. He has more than 5 years teaching and 10 years of industry experience. His areas of interest are Software Engineering, Computer Network and Artificial Intelligence. He has published more than five research papers in journals and conferences. He has also guided fifteen undergraduate students.

DR. AKHILESH R. UPADHYAY obtained Ph.D. degree from the Swami Ramanand Teerth Marathwada University, Nanded in 2009, M.E. (Hons.) and B.E. (Hons.) in Electronics Engineering from S.G.G.S. Institute of Engineering & Technology, Nanded [M.S.] in year 2004 and 1996 respectively.

He is currently working as Vice Principal and Head of Electronics and Communication Engineering Department at Sagar Institute of Research and Technology, Bhopal, India.

He has more than 12 years teaching and 3 years of industry experience. He is Associate Editor of Journal of Engineering, Management & Pharmaceutical Sciences, Ex-Editor of International Journal of Computing Science and Communication Technologies and member of editorial boards/review committee of various reputed journals and International conferences. He has more than 50 research publications in various international/national journals and conferences; he also authored more than 16 text/reference books on electronics devices, instrumentation and power electronics. He is recognized Ph.D. Supervisor for various Universities in India and presently guiding 11 Ph.D. scholars.