Experiments on Design Pattern Discovery

Jing Dong, Yajing Zhao
Department of Computer Science
University of Texas at Dallas
Richardson, TX 75083, USA
{jdong, yxz045100}@utdallas.edu

Abstract

Design patterns have been applied in many large software systems to help developers coping with recurring design problems. However, pattern-related information is generally lost in system source code. Discovering design pattern instances from source code can help to understand and analyze the software systems. In this paper, we present several experiments on design pattern discovery using our tool. We also compare the results of our experiments with other approaches and identify the need of benchmarks.

1. Introduction

Since the initial introduction of design patterns in [9], they have been widely adopted by software industry. Many software developers routinely apply design patterns in their software systems to reuse expert design experience and record design decisions. However, such high-level design information is typically lost in system source code when the systems are deployed. The architectural design document is normally not deployed with source code. Even the design document is available, it may not be consistent with the source code after the system has evolved and been changed due to new requirements. In addition, many legacy systems do not have documents available. It is generally hard to trace such design decisions in source code. Without such information, the benefits of using design patterns may be compromised. The developers may not be able to communicate in terms of design patterns and change the systems with the guidance of design patterns during the system maintenance and evolution. Thus, discovering the design patterns applied in a software system may help on not only the understanding of the system but also its maintenance and evolution.

The analysis and understanding of software systems treat system source code as data. Design patterns may be recognized from this source code data. Therefore, just like data mining and analysis, software source code become data that is subject to analysis. There are a large number of open-source software systems available, which can be excellent candidates for program analysis. In particular, many open-source systems embedded design patterns. Experiments on discovering design patterns from these open-source systems can not only help on system understanding, but also lead to benchmarks for comparing different approaches.

In this paper, we present several experiments on design pattern discovery from the open-source systems, including JUnit [28], JEdit [26], JHotDraw [27], and Java.AWT [25]. Our experiments use our design pattern discovery tool, DP-Miner, presented in [8]. Our results show the numbers of design patterns discovered from each open-source system. We also discuss our results and pinpoint the importance of benchmarks for design pattern discovery.

The rest of this paper is organized as follows. Section 2 presents the related work. Section 3 details our experiments on four open-source software systems. We compare our results with others and discuss some current issues in Section 4. The last section concludes this paper.

2. Related Work

Several approaches on design pattern discovery have been proposed in the literature. Experiments on open-source systems have also been conducted by these approaches. However, different approaches reported different results when discovering the same design patterns in the same open-source systems. It lacks the studies on the benchmarks of these systems.

Tsantalis *et al*. [19] applies an existing similarity score algorithm to detect design patterns. Structural relations between classes are encoded in multiple graphs and matrices. Similarly the design patterns are also encoded in matrices. The discovery of design patterns is achieved by calculating the similarity score between the matrices of system source and those of patterns. Currently their toolkit is able to detect the Adapter, Bridge, Composite, Decorator, Factory Method, Observer, Prototype, Singleton, Strategy, Template Method, and Visitor patterns. They experiment their toolkit on JUnit, JEdit and JRefactory.

Third International Workshop on Predictor Models in Software Engineering (PROMISE'07)
0-7695-2954-2/07 $20.00 © 2007 IEEE

A multi-stage searching process consisting structural metrics evaluation and method delegation constraints evaluation is proposed by Antoniol *et al.* [2]. In their structural metrics evaluation, they extract class metrics. Numbers of attributes, operations, and relations are all considered class metrics and are extracted from the intermediate representation in Abstract Object Language (AOL) Abstract Syntax Tree (AST). Their toolkit is capable of detecting the Adapter, Bridge, Composite, Decorator and Proxy patterns. It has been run on some public domain code, such as LEDA, libg++, galib, gruff, and socket, and some industrial systems.

Structure constraints is defined as Prolog predicates [11] and behavioral constraints as Prolog procedure based on temporal logic of actions (TLA) [12] by Heuzeroth *et al*. They use AST as the intermediate representation of their system. They conducted the experiment on the Java code of their own pattern recovery tool and discovered the Composite, Decorator and Observer pattern instances.

Niere *et al.* [14][15] implement their top-down-bottom-up approach on FUJABA platform [24], which uses AST as intermediate representation. In [20], behavioral analysis are emphasized and sequence diagrams are considered as pattern rules for behavioral characteristics. They conducted the experiments on the Java.AWT package and JGL libraries with the discovery of the Bridge, Strategy and Composite patterns.

Blewitt and Bundy [5] develop a proof system Hedgehog, which is capable of reasoning design patterns in Java language. They emphasize the importance of semantic constraints which describe how classes are related to each other and how implementations of particular methods operate. An experiment is done on Java.AWT package to recover the Command, Factory, Proxy, Singleton patterns.

Design Pattern Markup Language (DPML), an XML-based language, is presented in [3]. Balanyi and Ferenc introduce an approach with DPML as representation format for patterns. Source code, on the other hand, is analyzed and built into an Abstract Semantic Graph (ASG). Finding a design pattern in their approach is to match ASG sub-structures with the DPML description of the pattern. Some experiments are performed on some open-source C++ projects, such as Jikes, Leda, StarOffice Calc and StarOffice Writer. The patterns discovered by their tool are Abstract Factory, Adapter, Bridge, Builder, Chain of Responsibility, Decorator, Factory Method, Prototype, Proxy, Singleton, Strategy, Template Method, and Visitor.

Shi and Olsson [17] propose using dataflow diagram and control flow diagram which works perfect on Singleton and Flyweight patterns. Their tool, PINOT, is able to detect most of the GoF patterns, namely Abstract Factory, Adapter, Bridge, Chain of Responsibility, Composite, Decorator, Façade, Factory Method, Flyweight, Mediator, Observer, Proxy, Singleton, Strategy, Template Method, and Visitor. Experiments have been conducted on JDK, Java.AWT and JHotDraw.

Gueheneuc *et al.* [10] propose to fingerprint design patterns from source code using machine learning techniques. Size, filiations, cohesion, and coupling are the finger prints of classes. This algorithm eliminates the classes that do not match these metrics and therefore reduces searching space. Experiment on JHotDraw is done to detect the Abstract Factory, Adapter, Builder, Command, Composite, Decorator, Factory Method, Observer, Prototype, Singleton, State, Strategy, Template Method, Visitor patterns.

3. Experiments

In order to discover design pattern instances from object-oriented software systems, we introduced a novel approach based on matrix and weight in [8] and implemented it in our toolkit, DP-Miner. More specifically, DP-Miner builds a matrix of a source system with all classes in the system to be the rows and columns, and the relationships between each pair of classes to be the value of the corresponding cell in the matrix. Realizing that the product of prime numbers can be decoded into a unique group of prime numbers, we represent different relationships by different prime numbers and combination of relationships by product of prime numbers in each cell value. DP-Miner also encodes the numbers of attributes, operations, and relationships of each class in the system into weight by the product of prime numbers. Similarly, our approach encodes the information of each design pattern into its corresponding matrix and weights. Therefore, the discovery of design patterns is reduced to matching such matrices and weights in arithmetic computations.

Our discovery processes include the structural, behavioral, and semantic analysis. The structural analysis concentrates on the matching of the static design models between the design patterns and the software system. Because most of design patterns describe both structural and behavioral aspects of a design experience, only structural analysis may result in false positive cases. The behavioral analysis can help to check whether desired behavioral characteristics exist in the candidate results from the structural analysis. DP-Miner checks behavior characters, such as method delegation, based on structure analysis results. Some design patterns, such as the Bridge and Strategy patterns, are similar in their structures and behaviors, but only differ from their

intents and motivations. The semantic analysis aims at distinguishing some of these patterns by their naming conventions.

While our attention was mainly on the introduction of the algorithms and methods of our approach in [8], we present several experiments on large open-source systems in this paper. We also discuss our results on the discovery of four design patterns, Adapter, Bridge, Strategy, and Composite, from these systems and argue for the benchmarks of design pattern recovery. In our experiments, we concentrate on the analysis of a small number of design patterns because the discovery processes for different design patterns are generally quite similar. They only differ on the characteristic descriptions of each pattern. Focusing on a small number of patterns allows us to study deeper in the pattern discovery problem.

Most existing studies conduct experiments on open source systems, such as Java Swing package, Java Awt package, QuickUML, etc, which contain hundreds to thousands classes. Our experiments use four open-source systems, the Java Abstract Window Toolkit (AWT) [25], JUnit [28], JEdit [26], and JHotDraw [27]. Java.AWT is a library for developing graphical user interfaces for Java programs. JUnit is a regression test framework that helps developers to implement unit tests in Java. JEdit is a mature text editor for programmers, which provides many features for ease of use. JHotDraw is a two-dimensional graphics framework for technical and structured drawing editors written in Java. Table 1 shows the system information in our experiments, including the versions of systems and the number of classes and files each system has.

Table 1 Subject System Descriptions

Systems	Version	Class #	File #
Java.AWT	JDK1.4.2	570	345
JUnit	3.8.2	126	93
JEdit	4.2	1001	394
JHotDraw	6.0 beta 1	530	484

We select these open-source software systems as the subjects of our experiments because the idea of design patterns is already mature and widely applied in software industry at the time they were developed. Thus, these systems contain design patterns in their software design and their usages of patterns follow the pattern principles as introduced in [9]. In addition, other works on design pattern discovery use one or some of these systems to evaluate their approaches. It allows us to compare and evaluate our experiment results.

3.1 Results

Using DP-Miner, we conducted our experiments on these four systems. The number of discovered instances for each pattern is shown in Table 2, which shows that Java.AWT includes more instances of the Adapter, Bridge, Strategy, and Composite patterns than other systems do. In addition, the numbers of the instances of the Bridge and Strategy patterns are very similar. The main difference between them is their design intention, whether the pattern is to define a family of algorithms or it is to decouple the abstraction from the implementation. The candidate sets obtained after the structural and behavioral analysis are the same for both patterns. They only differ semantically.

Table 3 through Table 6 show the numbers of candidate pattern instances acquired at each analysis phase for all four systems. These tables illustrate how behavioral analysis and semantic analysis eliminate false positive candidates from the results of the structural analysis. We relax the criteria for structural analysis in order to reduce the number of false negative cases. Thus, the results of structural analysis may include a higher number of candidates. Our approach relies on the behavioral and semantic analysis to eliminate the false positives.

Our behavioral analysis is based on the results from the structural analysis. Thus, it deals with a smaller number of classes from the original systems. For example, the JUnit package contains a total of 126 classes. After structural analysis, only 6 to 15 candidate instances of the four patterns are discovered. Thus, the search space of the behavioral analysis is reduced to the classes of these 6 to 15 pattern instances. After behavioral analysis, 3 to 6 candidate instances are left, which limits the search space for the semantic analysis further down.

Table 2 Pattern Discovery Results

Systems	Adapter	Bridge	Strategy	Composite
Java.AWT	21	65	76	3
JUnit	3	6	6	3
JEdit	17	24	24	0
JHotDraw	4	58	64	0

Table 3 Pattern Recovery of Each Phase for Java AWT Package

System Java.AWT	Structural Analysis	Behavioral Analysis	Semantic Analysis
Adapter	57	21	N/A
Bridge	100	76	65
Strategy	100	76	76
Composite	92	3	N/A

Table 4 Pattern Recovery of Each Phase for JUnit

System JUnit	Structural Analysis	Behavioral Analysis	Semantic Analysis
Adapter	15	3	N/A
Bridge	6	6	6
Strategy	6	6	6
Composite	9	3	N/A

Table 5 Pattern Recovery of Each Phase for JEdit

System JEdit	Structural Analysis	Behavioral Analysis	Semantic Analysis
Adapter	80	17	N/A
Bridge	33	24	24
Strategy	33	24	24
Composite	0	0	N/A

Table 6 Pattern Recovery of Each Phase for JHotDraw

System JHotDraw	Structural Analysis	Behavioral Analysis	Semantic Analysis
Adapter	27	4	N/A
Bridge	74	64	58
Strategy	74	64	64
Composite	0	0	0

Our behavioral analysis results show significant reductions of the candidate instances of the Adapter and Composite patterns than those of the Bridge and Strategy patterns. The main reason is that the former patterns include more behavioral characteristics than the latter do. These distinct behavioral characteristics help to eliminate a large number of false positive cases.

As shown in these tables, the results of the structural and behavioral analysis of the Bridge and Strategy patterns are the same for all four systems. The main reason is that the structural and behavioral characteristics of the Bridge and Strategy patterns are pretty much the same. The major difference of these two patterns is their intents and motivations that are generally not available from the system source code. While such semantic information is hard to recover, we found that many developers follow some naming conventions, which may leave some traces of their original intents. Therefore, checking class names, attribute names, and method names may provide certain indications of pattern usages. If a clue suggests the candidate is for sure not an instance of some pattern, the semantic analysis removes it from the candidate set. For example, the behavioral analysis results in 64 candidate instances of the Bridge and Strategy patterns in JHotDraw6.0 beta 1. One of the instances includes the ETSLADisposalStrategy class. This class name is a good indication of the potential instance of the Strategy, rather than Bridge, pattern. Our Semantic analysis deems it as an instance of the Strategy pattern. After the semantic analysis, 6 candidate instances in JHotDraw are found to be Strategy instead of Bridge based on naming conventions by our tool.

Due to the lack of design documentation of these open-source software systems, it is generally hard to validate the experimental results. Even if there is such design document available, the system implementations may sometimes divaricates from its original designs such that the design and implementation are inconsistent. In order to determine the precision of our approach, therefore, we manually checked the results generated by our tool and see whether they are real pattern instances. This manual checking was carried out with the results of JHotDraw as shown in Table 7, where TP and FP stand for true positive and false positive, respectively. It shows that there are 5 and 6 false positives in our result for the Bridge and Strategy patterns, respectively. These false positives are not eliminated during behavioral analysis. The main reason is that a method invocation may take many different forms in object-oriented programming languages. It is sometimes required to check the existence of some method invocation of an object in the behavioral analysis. For example, a *foo()* method of an *object* can be invoked directly by *object.foo()*, where *object* can be a direct instance of the desired class, a copy of some other instance of the desired class, return value of a method with the desired class as return type, a cast of an object of the superclass of the desired class, etc. To deal with such complexity, DP-Miner only checks the existence of a method invocation to *foo()* without considering to which object it belongs. By including such checking in our behavioral analysis, it allows our tool to include more true positives. On the other hand, it may also introduce some false positives caused by the invocations of the *foo()* methods that belong to some objects of undesired classes.

Table 7 Recovery Precision for JHotDraw

JHotDraw	TP	FP	Precision
Adapter	4	0	100%
Bridge	53	5	91.38%
Strategy	58	6	90.63%
Composite	0	0	100%

3.2 Benchmark

Open source systems, such as JUnit and JHotDraw, typically do not have detail design documentation.

Hence, it is unclear which design patterns have been used and where they are applied in the systems. Without such information, it is hard to measure the precision and recall of design pattern discovery approaches. In this paper, we emphasize the importance of benchmarking design pattern instances existing in different open-source systems. Although existing approaches may recover design patterns from source code, there are discrepancies in these results when discovering the same pattern from the system open-source systems. Such discrepancies include different numbers of the same design patterns are recovered from the same system by different approaches. Even if the number of a particular design pattern recovered from the same system is the same for some approaches, the particular location of such pattern instances may differ as well. Thus, not only the numbers of the particular design pattern instances are discovered in the open source systems, but also their location shall be benchmarked. Benchmarking design pattern instances applied in open source systems may allow evaluating the results of different approaches.

Table 8 Instances found manually but missed by DP-Miner

	CONTEXT	STRATEGY
[1]	LineConnection	Connector
[2]	ChangeConnectionHandle	Connector
[3]	ConnectionTool	Connector
[4]	PolygonHandle	Locator
[5]	LocatorHandle	Locator
[6]	LocatorConnector	Locator
[7]	SelectionTool	DrawingView

Due to the flexibility of design pattern applications, there may be several different ways to implement a design pattern. Such implementation variations are one of the main causes of imprecision in design pattern discovery processes. Although our approach tries to tackle different variations of a design pattern, there still are false-positive and true-negative cases in the results of DP-Miner. In this case, we have to manually inspect the source code for such instances. Table 7 presents the precision of DP-Miner for JHotDraw based on our manual inspections. We provide a complete table of all Strategy pattern instances discovered from JHotDraw v6.0 beta1 by DP-Miner in [23], where the names of classes participating in each instance are listed. The precision of our tool on the Strategy pattern in JHotDraw is 90.63%, listed in Table 7. Besides precision, we also attempt to compute the recall of DP-Miner. We manually checked JHotDraw for the Strategy pattern instances and found 7 real instances, listed in Table 8, which are missed by DP-Miner. Although we could not claim we have manually found all the Strategy pattern instances applied, we roughly find out the recall of DP-Miner is at most 89.23%. Based on DP-Miner and our manual inspections, we present an initial benchmarking result of the Strategy pattern from JHotDraw v6.0 beta1 in [23].

4. Comparison and Discussions

Several other approaches on design pattern discovery have also included the experiments on Java.AWT [1, 5, 14, 15, 16, 17, 18, 19, 20], JUnit [1], [19], JEdit [1], and JHotDraw [1, 4, 10, 13, 17, 19] Some of these experiments discovered the instances of the Adapter [17], [19], Bridge [16, 17, 19], Strategy [16, 17, 19], and Composite [13, 16, 17, 19] patterns from these systems. We have studied the results of design pattern discovery from these other approaches and found different approaches reached different results for the same patterns in the same systems. We manually investigated the corresponding source code and discovered the following issues with the current state of design pattern discovery techniques.

Design patterns describe a general guidance of a design solution to recurring problems. The application of a design pattern is normally flexible with several variations. Thus, the implementation of a design pattern is typically not unique. For example, one of the variations of the Composite pattern is to collapse both the Component and Composite classes into a single class. Such variations may be considered by some approaches but not others. This may cause different discovery results from different approaches.

Each design pattern may include several classes that play some particular roles in the pattern. For instance, the Adapter pattern generally includes the Target, Adapter, and Adaptee classes. Some existing approaches tend to match all such roles whereas others may consider only partial matches. The former approaches may end up discovering less pattern instances than the latter do since they have stricter criteria for matching. The latter approaches may lead to more potential false positives.

Current object-oriented programming languages may provide some special language constructs that may greatly simplify the implementation of a design pattern. For example, Java provides aggregation framework, such as LinkedList, ArrayList, HashMap, and Hashtable. These language features may make the implementation of the aggregate elements in the Composite class of the Composite pattern easy. The developers do not need to write their own Add(), Remove(), and getChild() operations to manage these aggregate elements in the Composite class. They can simply choose one of the language features instead. This makes the pattern discovery processes trickier

because these language features provide their own ways of managing the aggregates which are difficult to parse. Similar issue has also been discussed in [21]. Quite a lot of current approaches missed some instances of the Composite pattern due to this reason.

Current design pattern discovery approaches typically do not search design patterns directly from source code. Instead, they normally apply some existing reverse engineering tools to obtain certain intermediate representations, such as the Abstract Syntax Tree (AST) of the source code. Different approaches may apply different algorithms and methods to search for pattern instances from these intermediate representations, rather than the source code. While it can save time and efforts for discovering patterns from these representations, it may also cause potential errors in the results because the intermediate representations are just an abstraction of the source code after all. Some essential characteristics of a pattern may be abstracted away from the intermediate representations, which causes missing pattern instances for some approaches. We found it essential to check also the source code no matter which intermediate representation is used. Our approach uses XMI-based intermediate representations for the structural analysis. Our behavioral and semantic analysis need to resort back to the system source code to reduce the false positives.

Although there have been several experiments on discovering design patterns from these open-source systems, there are yet consistent results due to various reasons presented previously. Above all, the key problem is that there is no benchmark available for design pattern discovery. Lacking such benchmarks is the main impediment to evaluate and compare design pattern discovery techniques and methods. To calculate the precision and recall of a pattern matching result, in particular, it is essential to know the pattern instances actually exist in a system. Without such information, it is hard to judge whether an approach actually discovers all existing instances of a given pattern in a system and whether the discovered instances are correct. Although open-source systems are publicly accessible, the correct instances of patterns are generally not available. We have manually checked the JHotDraw system to calculate the precision of our results. We are also working on manually checking other systems. Our such efforts may lead to initial benchmarks for design pattern discovery.

5. Conclusions

In this paper, we presented our experiments on design pattern discovery from open-source systems using our tool, DP-Miner. In particular, our experiments discover the Adapter, Bridge, Strategy, and Composite patterns from the Java.AWT, JUnit, JEdit, and JHotDraw systems. A number of design pattern instances are recovered from these open-source systems. Our experimental results show that design patterns have been widely applied in these systems and can be recovered.

In addition, we compared our experimental results with those of others and found several discrepancies. We analyzed this issue and discussed possible reasons for the discrepancies. More importantly, we argue for benchmarks for design pattern discovery.

References

[1] H. Albin-Amiot, P. Cointe, Y. Gueheneuc, and N. Jussien, "Instantiating and detecting design patterns: putting bits and pieces together." In *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, 2001.

[2] G. Antoniol, R. Fiutem, and L. Cristoforetti, "Design pattern recovery in object-oriented software." *Proceedings of the 6th IEEE International Workshop on Program Understanding (IWPC),* pp 153-160, 1998.

[3] Z. Balanyi and R. Ferenc, "Mining design patterns from C++ source code." *Proceedings of the 19th IEEE International Conference on Software Maintenance (ICSM)*, pp. 305-314, September, 2003.

[4] D. Beyer, A. Noack, and C. Lewerentz, "Simple and efficient relational querying of software structures." In P*roceedings of the 10th Working Conference on Reverse Engineering (WCRE'03)*, 2003.

[5] A. Blewitt and A. Bundy, "Automatic verification of Java design patterns." Proceedings of 16th Annual International Conference on Automated Software Engineering (ASE'01), pp. 324-327, 2001.

[6] G. Costagliola, A. De Lucia, V. Deufemia, C. Gravino, and M. Risi, Design Pattern Recovery by Visual Language Parsing, Proceeding of the Ninth European Conference on Software Maintenance and Reengineering. (CSMR'05), pp. 102-111, 2005.

[7] J. Dietrich, C. Elgar, A Formal Description of Design Patterns Using OWL, *Proceedings of the 2005 Australian Software Engineering Conference (ASWEC'05)*, pp. 243-250, 2005.

[8] J. Dong, D. S. Lad and Y. Zhao, DP-Miner: Design Pattern Discovery Using Matrix, *the Proceedings of the Fourteenth Annual IEEE International Conference on Engineering of Computer Based Systems (ECBS)*, Arizona, USA, March 2007. (to appear)

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

[10] Y. Gueheneuc, H. Sahraoui, and F. Zaidi, "Fingerprinting design patterns." *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE)*, 2004.

[11] D. Heuzeroth, T. Holl, G. Hogstrom, and W. Lowe, "Automatic design pattern detection." *Proceedings of*

the 11th International Workshop on Program Comprehension (IWPC), pp 94-103, 2003.
[12] D. Heuzeroth, S. Mandel, and W. Lowe, "Generating design pattern detectors from pattern specifications." *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE)*, 2003.
[13] O. Kaczor, Y. Gueheneuc, and S. Hamel, Efficient Identification of Design Patterns with Bit-vector Algorithm, Proceedings of the Conference on Software Maintenance and Reengineering (CSMR'06), Volume 00 22-24, 2006.
[14] J. Niere, W. Schafer, J. P. Wadsack, L. Wendehals, and J. Welsh, "Towards pattern-based design recovery." In *Proceedings of the 24th International Conference on Software Engineering (ICSE)*, pp 338-348, 2002.
[15] J. Niere, J. P. Wadsack, L. Wendehals, "Handling large search space in pattern-based reverse engineering." *Proceedings of the 11th IEEE International Workshop on Program Comprehension (IWPC)*, pp. 274-279, 2003.
[16] J. Seemann and J. W. von Gudenberg, "Pattern-based design recovery of Java software." In *Proceedings of 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 10-16. ACM Press, 1998.
[17] N. Shi and R. A. Olsson, "Reverse engineering of design patterns from java source code." *21st IEEE/ACM International Conference on Automated Software Engineering*, 2006.
[18] D. Streitferdt, C. Heller, I. Philippow, "Searching design patterns in source code." In *Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC'05)*, 2005.
[19] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. Halkidis, "Design Pattern Detection Using Similarity Scoring." *IEEE transaction on software engineering*, Vol. 32, No. 11, November 2006.
[20] L. Wendehals, Improving design pattern instance recognition by dynamic analysis. *Proceedings of the ICSE workshop on Dynamic Analysis (WODA)*, pp. 29-32, May 2003.
[21] R. J. Wirfs-Brock, "Refreshing Patterns," *IEEE Software*, vol.23, no.3, pp. 45-47, May/Jun, 2006.
[22] Design Pattern Detection using Similarity Scoring, http://java.uom.gr/~nikos/pattern-detection.html
[23] DP-Miner. http://www.utdallas.edu/~jdong/DP_Miner/
[24] Fujaba User Documentation http://wwwcs.uni-paderborn.de/cs/fujaba/documents/user/manuals/FujabaDoc.pdf
[25] Java.awt resource information, September 2006, http://java.sun.com/j2se/1.5.0/docs/guide/awt/index.html.
[26] JEdit – Programmer's Text Editor. http://www.jedit.org/
[27] JHotDraw Start Page. http://www.jhotdraw.org/
[28] JUnit, Testing Resources for Extreme Programming. http://www.junit.org/