

A Review of Design Pattern Mining Techniques

Jing Dong, Yajing Zhao, Tu Peng

*Department of Computer Science, University of Texas at Dallas, Richardson, TX 75083, USA
{jdong, yxz045100, txp051000}@utdallas.edu*

Abstract

The quality of a software system highly depends on its architectural design. High quality software systems typically apply expert design experience which has been captured as design patterns. As demonstrated solutions to recurring problems, design patterns help to reuse expert experience in software system design. They have been extensively applied in industry. Mining the instances of design patterns from the source code of software systems can assist the understanding of the systems and the process of re-engineering them. More importantly, it also helps to trace back to the original design decisions, which are typically missing in legacy systems. This paper presents a review on current techniques and tools for mining design patterns from source code or design of software systems. We classify different approaches and analyze their results in a comparative study. We also examine the disparity of the discovery results from different approaches and analyze possible reasons with some insight.

Keywords: design pattern, reverse engineering, discovery

1. Introduction

During the past decade, design patterns [12][7][24] have been widely adopted by software industry to reuse expert design experience. Software developers become routinely reusing design patterns to solve their problems that are frequently encountered by them. Patterns document expert experience on solving recurring problem. The reuse of patterns may help the software architect to avoid pitfalls and mistakes in software design, and thus improve the quality of software products. Design patterns may help to record architectural tradeoff, capture design decisions, and reuse expert experience. Many current software systems embed instances of design patterns in system source code. After patterns are applied and implemented in a system, however, the pattern-related knowledge is generally no longer available in the source code. It is hard to trace such design information in source code. Even though the design information and diagrams are available, furthermore, it is typically not easy to find the patterns applied in a large software design. Pattern-related knowledge is normally buried under the descriptions because the participants of a pattern are not tagged with any knowledge on what roles they play in the design patterns. Without such knowledge, software designer cannot communicate with each other in terms of the patterns used. A software designer may forget the patterns he/she used over a long period of time. Thus, the benefits of applying patterns may be compromised. The important design decisions may be lost. Mining design pattern instances from system source code or design can greatly help to understand the systems and change them in the future. It also helps to trace back to the original architectural design decisions which are generally lost in system source code. Although recent model-driven development (MDD) techniques improve the documentation of design and allow the maintenance at model design level,

pattern-related knowledge is still missing in high-level software design models [19]. Even the high-level design models are available, thus, mining design patterns is still essential.

Designing a software system is hard; designing high-quality reusable system is even harder. Design patterns may help to design large high-quality systems in forward engineering. On the other hand, pattern-related knowledge is generally not available in software systems. Understanding large software systems is difficult. Many legacy systems lack design documents. With the evolution of software systems, design documents may not be consistent with their source code anymore. Reverse-engineering design information can recover the early decisions and assist the understanding of the systems, and thus, improve the systems with higher quality. Many existing techniques and tools can help mining software design from system source code. These reverse engineering tools become the foundation for design pattern discovery that typically do not search the source code from scratch. The results of these tools are the source of pattern discovery.

A number of pattern mining techniques and tools have been proposed in the literature. These approaches apply different methods and present different results. To the best of our knowledge, however, there is no study to classify these approaches and analyze their results. It is important to study different approaches due to the following reasons. First, this study may help to understand the differences and similarities between different approaches. Second, it assists to identify challenging issues in pattern discovery. Third, we found that different approaches may render different results while mining the same pattern from the same system. It is interesting to know the reason for such disparity. Fourth, this study may lead to benchmarking of pattern discovery techniques.

In this paper, we present a comparative study to classify different pattern mining approaches. We study their different technical bases and summarize their results with the numbers of patterns discovered and the names of the systems experimented. In addition, we analyze the disparity of different results and the potential reasons and insights for such differences. Furthermore, we evaluate the relative strength and weakness of different approaches. We also discuss the issues related to the precision and recall.

The rest of this paper is organized as follows. Section 2 presents a comparative study based on different characteristics of design pattern discovery techniques. Section 3 introduces the different discovery results from different approaches and analyzes the reasons. We conclude the paper in Section 4.

2. Comparative Study on Design Pattern Mining Techniques

Mining the instances of design patterns from system source code can help to understand and trace back to the original design decisions and reengineer the systems. There have been a number of different approaches proposed to solve this problem in the literature. In this section, we present a study that compares these approaches from different perspectives. The main goals are to summarize what have been done and to discuss the current issues.

In particular, our study of current approaches includes the following items. First, each design pattern is generally described from different perspectives, such as structure aspect and behavior aspect. Thus, different approaches may choose to check either structural aspect, behavioral aspect, or both. Some approaches may include other aspects, such as semantics. Second, current approaches typically take advantages of some existing reverse engineering tools to get an intermediate representation of the system source code. Design pattern discovery is actually done based on these intermediate representations, instead of source code. Current approaches use different tools to get different intermediate representations. The choice of intermediate representation format directly affects the choice of the algorithms for discovery. Third, some approaches require exact matching to the patterns whereas others may allow approximate matches. Fourth, the final discovery results are also presented differently. Most approaches just present the number of discovered patterns, whereas some approaches show the positions of discovered patterns graphically. Fifth, most approaches provide tool supports to automate the discovery processes. Some of the tools may require user interactions. Sixth, the discovery tool of each study

generally only supports the discovery of certain patterns. Seventh, different approaches conduct experiments on different open-source systems.

Figure 1 displays an overview of our classification scheme. The diagram shows the entities involved in our classification, which are denoted by rectangle boxes. The relations between the entities are denoted by ovals in the diagram. For example, design patterns are the subjects of design pattern activity, such as pattern mining. AUTOMATION DEGREE (fully automated or user interaction involved) is a character of the pattern mining tools; and MATCHING DEGREE (exact matching or approximate matching) is another character of the mining approach. Each pattern mining approach normally presents experiments on mining some selected patterns in some systems. The selected systems and patterns are subjects in the experiments, as well as characters of the experiments. The subject patterns are typically chosen from patterns defined by GoF [24]. In addition, we show the entities of our classification scheme in capital letters. For example, the intermediate representation FORMAT of design patterns, DEFINITION ASPECT of design patterns, and so forth are all considered to be the entities of our classification schema to be analyzed on each approach. In the following sub-sections, we provide detailed descriptions of our study.

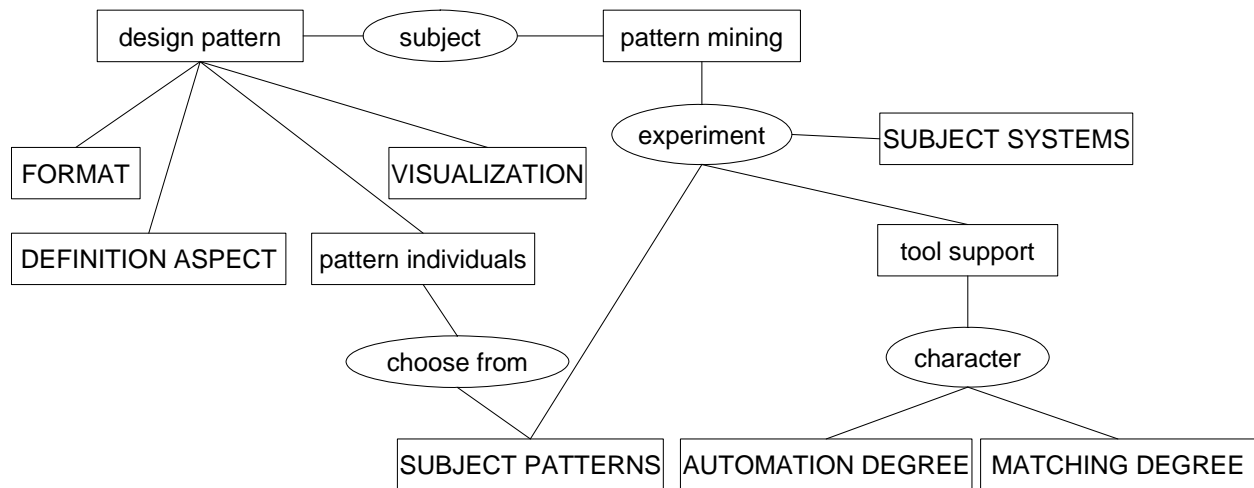


Figure 1 Classification Scheme

2.1 Pattern Aspects Checked

Each design pattern has its own characteristics. There are typically three aspects of pattern characteristics that distinguish them from each other -- structural characteristics, behavioral characteristics, and semantic characteristics. Pattern mining techniques normally try to check these characteristics in system source code or design to identify the matching patterns. Among the three aspects, the structural aspect is often available in most of design patterns. It is also relatively easy to detect from system source code or architectural design. Therefore, many approaches focus on structural aspect of patterns, such as Pat [32], DP++ [6], JBOORET [33], the tool based on FUJABA framework [34][35][44], SPQR [39], CroCoPat [8], DPRE [13][14], and the tool developed by Guéhéneuc *et al.* [25]. Most of these approaches consider the relationships between classes, such as generalization, association, aggregation, as the main properties that need to be checked. A few others inspect class attributes and methods. Therefore, the typical modeling elements that are considered in the structural aspect are generalization, association, aggregation, attributes, and methods. The checking of these modeling elements is important because they are the essential structural characteristics of design patterns. For instance, a design pattern may require two (or a group of) classes to have generalization, association, or aggregation relationships. It may also require a class to contain at least a certain number of attributes or

methods. For example, Guéhéneuc *et al.* [25] consider new, inherited and overridden methods, the total number of methods, and the count of methods weighted with their number of method invocations as valuable metrics that characterize patterns. Antoniol *et al.* [2] consider the number of public, private and protected attributes, number of public, private and protected operations, and the number of association, aggregation and generalization relations in which a class is involved as important metrics. Ferenc *et al.* [23] use appropriate number of attributes, public, private and protected methods, virtual and concrete methods as characters to initially collect the classes that might be involved in pattern instances. Dong *et al.* [16] examines the number of attributes and the number of methods defined in each class in terms of weight and the relations between every pair of classes in terms of matrix. Table 1 visualizes the categorization of different approaches based on the kind of pattern aspects they checked. The majority of the existing approaches analyze only the structural information.

Pattern Aspect Checked	Authors	Tools
Structural Aspect Only	Kramer 1996 [32]	Pat
	Bansiya 1998 [6]	DP++
	Mei 2001 [33]	JBOORET
	Asencio 2002 [3]	
	Smith 2003 [39]	SPQR
	Beyer 2003 [8]	CroCoPat
	Zhang 2004 [46]	
	Guéhéneuc 2004 [25]	
	Costagliola 2005 [13][14]	DPRE
	Streitferdt 2005 [40]	
	Ferenc 2005 [23]	Columbus
	Kaczor 2006 [30]	Ptidej
Tsantalis 2006 [42]		
Behavioral Aspect Only	Park 2004 [36]	
Both Structural and Behavioral Aspects	Antoniol 1998 [2]	
	Tonella 1999 [41]	
	Wendehals 2003 [44]	
	Heuzeroth 2003 [27][28]	
	Wang 2005 [43]	DPVK
	Huang 2005 [29]	PRAssistor
	Shi 2006 [38]	PINOT
Structural and Semantic Aspect	Seemann 1998 [37]	
Structural, Behavioral and Semantic Aspects	Blewitt 2001 [9]	Hedgehog
	Dong 2007 [16][20]	DP-Miner
Pattern Composition	Hericko 2005 [26]	

Table 1 Categorization of Current Discovery Method Based on the Pattern Aspect Checked

Although comparatively harder to gather from source systems, behavioral aspect contains important knowledge to judge whether or not a candidate is a real instance. Table 1 shows that a number of existing tools perform behavioral analysis based on the results from the structural analysis, aiming at eliminating false positives¹. Behavioral aspect of design patterns is typically described by method invocations. Some of the studies choose to check the behavior in a static way whereas others check it dynamically at system

¹ Note that the empty cells in the tables of this paper are because the corresponding information is not presented in the literature.

run time. Antoniol *et al.* [2] check method delegations statically. Dong *et al.* [16] check method delegations and dependency links between classes statically. Park *et al.* [36] propose to use reference call relations to statically estimate run time behavior among pattern participants. Shi *et al.* [38] propose using control flow graph analysis to check the behavior of patterns, especially the Singleton and Flyweight patterns. Besides these static ways, some researchers propose to dynamically analyze pattern behavior. As some patterns require specific sequences of actions and interactions among the objects of the participating classes, Heuzeroth *et al.* [27][28] propose to monitor the execution of the classes. They further track the effects of the executed classes to check whether the candidate satisfies the dynamic pattern rules. Huang *et al.* [29] keep track of the entries or exits of operations, the allocation or destruction of an object at run time. Wendehals [44] improves their previous work [34][35] based on FUJABA framework by doing behavioral checking dynamically. In his approach, information is gathered during program execution by debugging the program manually. Breakpoints should be set for the pattern-related methods and method traces are recorded at run time. Sequence diagrams can therefore be generated based on the method trace.

The definition of semantic characteristic is somewhat not unique. Although it refers to the semantic meaning of some entities in the system, the entities of interest vary among different approaches. Seemann and Wolff von Gudenberg [37] take advantage of naming conventions and programming guidelines that lead to the class names with particular keywords. For example, Vector and HashTable normally indicate the multiplicity of an aggregation. Blewitt *et al.* [9] also use semantic analysis to determine whether classes are related to each other by one-to-one or one-to-many relations, and whether these relations are required or optional. Dong *et al.* [16] explore the pattern-related knowledge from the naming conventions of each class for indications of the roles that the class may play in the potential patterns. For instance, a class with its name including “strategy” is highly likely to be part of the Strategy pattern.

Besides the aspects mentioned previously, there is also work focusing on the discovery of the case where a class may participate in multiple pattern instances and a composition of patterns may exist. Hericko and Beloglavec [26] propose two metrics, pattern coverage and overlapping which help to discover the composition of multiple pattern instances.

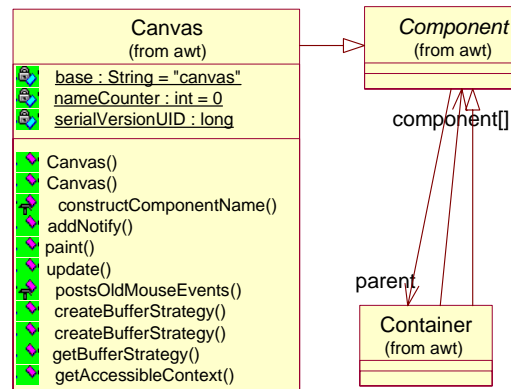


Figure 2 Class Diagram of an Example System

2.2 Intermediate Representations

There are many existing tools that discover design information from source code. Thus, there is no point of discovering design patterns from scratch. Current approaches on pattern mining usually use some existing tools to transform the source code into some intermediate representations to reduce the search complexity. Similarly, design patterns are represented in some format, such as predicates and grammar. Pattern instances are actually discovered from these intermediate representations.

In this section, we discuss several options for the intermediate representation of software systems and design patterns, such as Abstract Syntax Tree (AST), Abstract Semantic Graph (ASG), bit, vector, matrix, and etc. We do not claim that we present a complete list of all possible representation formats. For example, the XML, Control Flow Graph (CFG), and Data Flow Graph are not discussed in detail because they are standard representation formats. We provide several examples here so that it is clear how the intermediate representation is obtained and how the information is captured. As a running example, we use a piece of system shown in Figure 2 from Java.AWT. For simplicity, we do not show all the attributes and operations of each class, but just those of the Canvas class.

2.2.1 Abstract Syntax Tree

Abstract Syntax Tree is a finite, labeled, directed tree, where each interior node represents a programming language construct and the children of that node represent meaningful components of the constructs. It can be created in a parser for a language described by a context free grammar and used as an intermediate between a parse tree and a data structure. It usually contains all the key information in the source code. The AST has been extensively used in compiler construction. For example, the AST for the system shown in Figure 2 can be obtained by some automated AST generator. The tree shown in Figure 3 gives an idea how the AST looks like. Note that this AST does not include the package information since Figure 2 does not show the package related information for the purpose of simplicity.

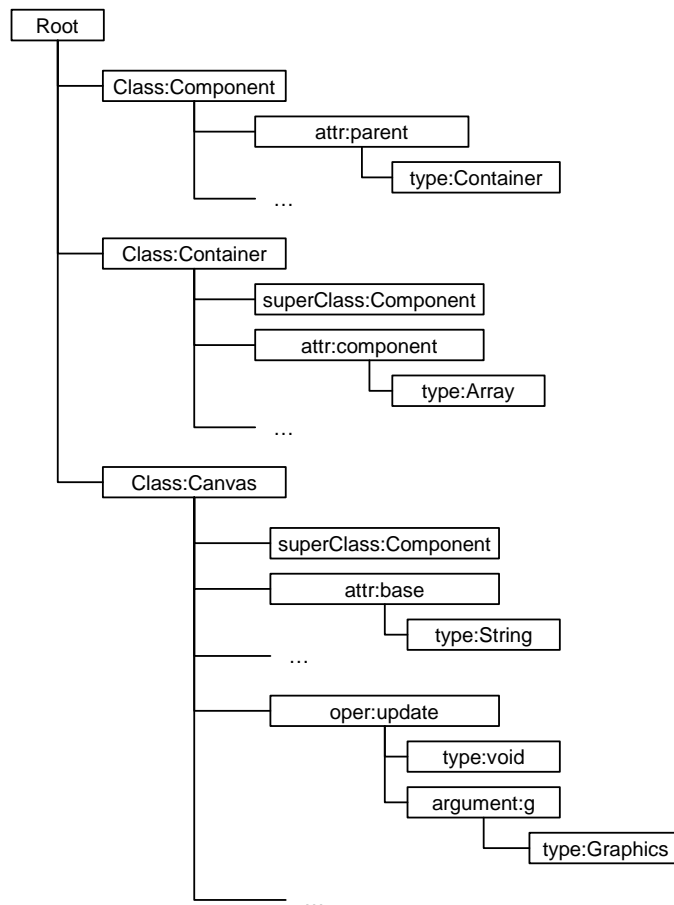


Figure 3 AST of the Example System

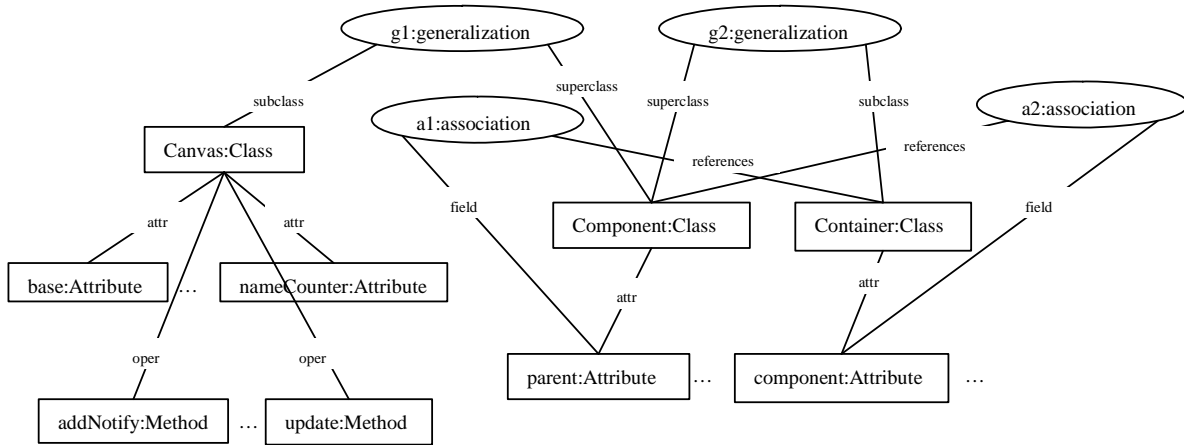


Figure 4 ASG of the Example System

2.2.2 Abstract Semantic Graph

Abstract Semantic Graph (ASG) is also a good format that the system information can be stored. There are usually two types of nodes in ASG, element nodes and relation node. Element nodes can be used to denote the elements in the system, such as classes, attributes, and operations. Relation nodes can be used to denote the relations between system elements, such as generalization and association relations. Figure 4 shows part of the ASG of the example system in Figure 2 so as to show the general idea.

2.2.3 Matrix and Vector

Design patterns are good designs in Object-Oriented software systems which normally consist of classes and relations between them. One way to extract the useful information for design pattern detection from complex systems is to use matrix and weight. Matrix can be used to represent the relations between the classes in the system by using cell (i, j) to represent the relation from class i to class j. Since there are different kinds of relations in object-oriented systems, such as generalization, association, dependency, one matrix can be used for each relation.

For the example systems shown in Figure 2, the following matrix can be constructed to represent the association relations between classes in the system.

	Component	Container	Canvas
Component	0	0	0
Container	1	0	0
Canvas	1	0	0

Table 2 Association Relation Matrix

Similarly, the following matrix can be constructed to represent the generalization relations between classes in the system.

	Component	Container	Canvas
Component	0	1	0
Container	1	0	0
Canvas	0	0	0

Table 3 Generalization Relation Matrix

The value of each cell in these matrices represents the existence of such relationship. In both matrices, for example, 1 indicates that there is such a relation between the two classes, and 0 indicates that there is no such relation between them.

There are normally more than one kind of relations exists in an object-oriented system. To save memory space, some methods can help to reduce the number of matrices, such as by using prime numbers to combine different relation matrices into one. For example, if we use 2 to represent the association relation and 3 to represent the generalization relation, the relations in the above example can be represented by the following matrix.

	Component	Container	Canvas
Component	0	2	0
Container	2*3	0	0
Canvas	3	0	0

Table 4 Combined Matrix by Using Prime Numbers

In this table, 2*3 indicates that there exist one association relation and one generalization relation from the Container class to the Component class; 2 indicates that there exists one association relation from the Component class to the Container class; and 3 indicates that there exists one generalization relation from the Canvas class to the Component class.

In addition, vector can be used to represent the number of elements defined in each class by using each value in the vector to represent the number of elements.

For example, the vector that defines the number of attributes in each class of the example system is

Component	Container	Canvas
5	5	3

Table 5 Attribute Vector

The vector that defines the number of operations in each class of the example system is

Component	Container	Canvas
19	17	11

Table 6 Operation Vector

Similarly, these two vectors can also be reduced into one by using prime numbers. For example, if we use 2 to represent the attribute and 3 for the operation, then we get the following vector.

Component	Container	Canvas
$2^5 * 3^{19}$	$2^5 * 3^{17}$	$2^3 * 3^{11}$

Table 7 Combined Vector by Using Prime Number

2.2.4 Eulerian Model

A useful intermediate representation is Eulerian model. An Eulerian digraph contains an Eulerian circuit, which is a cycle traversing each edge exactly once. A UML class diagram is a directed graph and can be easily converted to an Eulerian model without losing information. Dummy relations, denoted by *dm*, can be added into the diagram so as to form cycle between some of the classes when there is the need. For example, Figure 2 can be converted to an Eulerian model as shown in Figure 5. Relations denoted by *in* are inheritance relations and those denoted by *as* are association relations in Figure 5.

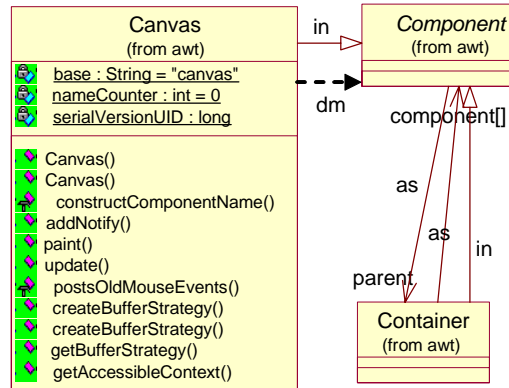


Figure 5 Eulerian Model of the Example System

The circuit character of Eulerian model makes it easy to be represented by a string. For example, the above model can be represented by the following string:

Component in Canvas dm Component in Container as Component as Container

Since a string usually can be represented by a binary string (bit vector). By using Eulerian model helps to reduce the complex information of a system into an easy format of representation.

2.2.5 Intermediate Representation Usage Comparison

Table 8 lists the format that each approach uses to represent source system and to represent patterns. It shows that most of the approaches use the Abstract Syntax Tree (AST) or Abstract Syntax Graph (ASG) as the intermediate representation. For example, Antoniol *et al.* [2] extract software metrics from AST produced by parsing Abstract Object Language (AOL) representation of code and design. Niere *et al.* [34][35] define patterns and code with respect to the ASG representation, and propose top-down-bottom-up two-way search on the graph to improve performance and precision. Heuzeroth *et al.* [27][28] use AST to define the static aspect of patterns and the Temporal Logic of Actions (TLA) to define their dynamic aspect.

Some approaches follow XML format. Balanyi and Ferenc [4] introduce the Design Pattern Markup Language, an XML-based language, to describe design patterns, and ASG to represent source code. Dong *et al.* [16] transform design information of source code into XML format using Rational Rose. Osprey [3] uses Imagix to parse source code and store the result in XML format which helps validating the output.

Kaczor *et al.* [30] express the problem of design pattern identification with operations on finite sets of bit-vectors, which improves space performance. The bit-vector algorithm can find solution to the problem in a limited number of vector operations, which is independent of the length of the program.

Costagliola *et al.* [13][14] propose to use graph to represent class diagrams of systems and visual grammar to specify the pattern rules. They use visual language parsing technique to perform the matching. Seemann *et al.* [37] also apply their algorithm on graphs; however, their approach uses set theory and predicates to define patterns. Zhang *et al.* [46] extend the concept of traditional graph for

describing object-oriented design and design patterns, and use sub-graph isomorphic technology to perform the matching.

Authors	Tools	System Representation	Pattern Representation
Kramer 1996 [32]	Pat	Prolog	Prolog
Seemann 1998 [37]		graph	Predicate
Bansiya 1998 [6]	DP++	class hierarchy	text
Antoniol 1998 [2]		AST	AOL
Keller 1999 [31]	SPOOL		UML/CDIF
Blewitt 2001 [9]	Hedgehog		Spine
Albin-Amiot 2001 [1]	Ptidej	CSP	
Niere 2002 [34]	FUJABA	ASG	ASG
Wendehals 2003 [44]		ASG & call graph	
Smith 2003 [39]	SPQR	OML & OTTER	otter rules
Heuzeroth 2003 [27][28]		AST	AST & TLA
Beyer 2003 [8]	CroCoPat	BDDs	predicates
Park 2004 [36]		Class Diagram	
Zhang 2004 [46]		Graph (matrix)	Graph (matrix)
Guéhéneuc 2004 [25]		XML tree & PADL	PADL
Costagliola 2005 [13][14]	DPRE	SVG & AOL->AST	Grammar-based Pattern Specification
Huang 2005 [29]	PRAssistor		Prolog
Ferenc 2005 [23]	Columbus	ASG, XML DOM tree	DPML
Wang 2005 [43]	DPVK	REQL static & RSF dynamic	REQL script
Kaczor 2006 [30]	Ptidej	bit representation	bit representation
Shi 2006 [38]	PINOT	AST	DFG & CFG
Tsantalis 2006 [42]		matrix	matrix
Dong 2007 [16][20]	DP-Miner	XMI/matrix and vector	XMI/matrix and vector

Table 8 the Intermediate Representation used by Existing Approaches

SPQR [39] uses GCC parse tree, Object XML, and OTTER as intermediate representation in different phases. Pat system [32] uses Prolog for both code and design. SPOOL [31] uses CDIF transfer format as the syntax and UML metamodel 1.1 [53] as the semantic model of the intermediate format. Hedgehog [9] uses Spine, typed first-order logic. Blewitt *et al.* [9] create Spine to describe design patterns and their constraints in terms of variants, mini-patterns, and artifacts. Streitferdt *et al.* [40] propose minimal key element structure definition for patterns, with which the required, forbidden and don't-care elements are defined for patterns.

2.3 Exact vs. Approximate Matches

Since there is lack of research on the set of characteristics of each pattern which need to be checked, most researchers tend to clarify the problem by making their own definitions. Most approaches search a piece of architectural design that structurally confirms to the structural characteristics of the pattern that they defined and behaviorally exhibits the expected actions that they defined. As long as they find all matching rules are conformed, they claim to find a match. If one of them is violated, on the other hand, they treat it as mismatch. This seems to solve the problem. Nevertheless different definitions of the same pattern apparently render different search results. In addition, some pattern candidates which might be

real instances are filtered out due to strict exact matches. Therefore, some approaches perform approximate matches by computing the similarity degree. Approximate matches retain more candidates which mostly conform to the matching rules. The following section introduces how approximate matching algorithms work for design pattern discovery by applying algorithms to calculate matching degree, such as similarity scoring and template matching.

Matching Technique	Authors	Tools
Exact Match	Kramer 1996 [32]	Pat
	Seemann 1998 [37]	
	Bansiya 1998 [6]	DP++
	Antoniol 1998 [2]	
	Tonella 1999 [41]	
	Keller 1999 [31]	SPOOL
	Blewitt 2001 [9]	Hedgehog
	Mei 2001 [33]	JBOORET
	Albin-Amiot 2001 [1]	Ptidej
	Asencio 2002 [3]	
	Wendehals 2003 [44]	
	Smith 2003 [39]	SPQR
	Heuzeroth 2003 [27][28]	
	Beyer 2003 [8]	CroCoPat
	Park 2004 [36]	
	Zhang 2004 [46]	
	Costagliola 2005 [13][14]	DPRE
	Streitferdt 2005 [40]	
	Huang 2005 [29]	PRAssistor
	Wang 2005 [43]	DPVK
Kaczor 2006 [30]	Ptidej	
Shi 2006 [38]	PINOT	
Dong 2007 [16][20]	DP-Miner	
Approximate Match	Niere 2002 [34]	FUJABA
	Guéhéneuc 2004 [25]	
	Ferenc 2005 [23]	Columbus
	Tsantalis 2006 [42]	

Table 9 Categorization of Current Discovery Method Based on Matching Techniques

2.3.1 Similarity Scoring

Blondel *et al.* [10] introduce an algorithm to calculate a similarity score between two graphs. Let G_A and G_B be two directed graphs with, respectively, n_A and n_B vertices. The similarity matrix S is defined as an $n_B \times n_A$ matrix whose real entry s_{ij} expresses how similar vertex j (in G_A) is to vertex i (in G_B) and is called the similarity score between the two vertices. The algorithm used for calculating the similarity matrix S is as follows:

1. Set $Z_0 = 1$.
2. Iterate an even number of times $Z_{k+1} = (BZ_k A^T + B^T Z_k A) / (||BZ_k A^T + B^T Z_k A||_1)$ and stop upon convergence.
3. Output S is the last value of Z_k where
 - A, B are the adjacency matrices of graphs G_A and G_B , respectively,

- Z_0 is an $n_B * n_A$ matrix filled with ones,
- $\|\cdot\|_1$ is the 1-norm of a matrix, and convergence refers to the subsequence of even iterations.

As discussed in Section 2.2.2, UML class diagrams can be represented by matrix. Applying this algorithm on two matrices, one is that of a system piece and the other is that of a design pattern, the similarity degree can be calculated.

2.3.2 Template Matching

Template matching is a popular method in computer vision [5] to detect a template in a graph. It takes the cross correlation value of template f and graph g as an estimate of the degree of match. The formula, $CC(u) = \sum f(x-u) \bullet g(x)$ where $f(x)$ and $g(x)$ are two vectors, $x = 1, \dots, n$ and u is an offset, shows how to calculate cross correlation. The larger the value is the higher potential they match. When $u = 0$, the cross correlation formula is $CC = \sum f(x) \bullet g(x)$. The basic idea is that if f and g match each other, their product is amplified. Thus, the sum of their products at every x is larger than that of f and g when they do not match. However, there are two problems. If $g(x)$ is a vector with much larger values than $f(x)$, the large value of cross correlation does not guarantee a match. If $g(x)$ is a multiple function of $f(x)$, in addition, the cross correlation of f and g is much larger than the cross correlation of f and f (exact match), which should be the same from matching perspective. To overcome these problems, normalized cross correlation (CCn) is introduced by dividing the cross correlation value by the product of the norms of f and g .

$$CCn = \frac{\sum f(x) \cdot g(x)}{|f(x)| \cdot |g(x)|}$$

The normalized cross relation is a value between 0 and 1, where 1 means exact match and 0 means no match. The higher the score is the higher similarity is. Applying this algorithm on system matrix and pattern matrix also gives us the similarity degree between them, which indicates the pattern detection result.

2.3.3 Exact Matching or Approximate Matching Preference Comparison

Exact matching requires that the system piece is exactly the same as the pattern. In other words, a range in $[0, 1]$ can be used as the matching degree. Therefore exact matching selects system pieces with a matching degree of 1, and eliminates those with a degree less than 1. Since only exact matched system pieces are retained, however, exact matching approaches usually omit the effort of calculating matching degrees. On the other hand, approximate matching sets a number between 0 and 1 as the threshold. For example 0.8 can be a reasonable threshold. Thus, any system piece with a matching degree equal to or higher than 0.8 is retained and others are filtered out when matching a particular pattern. A lot of research can be done on selecting and optimizing the threshold to get a perfect result, which is not our focus in this paper.

Niere *et al.* [35] introduces fuzzy-belief, a value between 0 and 1, for each structural rule that express its precision. To limit the rule applications to reasonable cases, they introduce thresholds to the rules. The match with a fuzzy value lower than the threshold will be excluded. This helps to minimize computation load and therefore improve the scalability. To compute matching degree between system under study and the pattern, Tsantalis *et al.* [42] applied a similarity scoring algorithm, an inexact graph matching algorithm, when an exact isomorphism between two graphs cannot be found. Dong *et al.* [17] applied a template matching algorithm to calculate the matching degree between a piece of system and a design pattern. Gu  h  neuc *et al.* [25] propose a machine learning algorithm to compute software metrics and use the degree of confidence to infer the rules by a rule learner. Ferenc *et al.* [23] introduce predictors for each pattern and use machine learning algorithm training the pattern recognizer to acquire the value of the predictors. These predictor values are then used as the standards to compare with the predictor values

obtained for system under study. A pre-processing algorithm is also proposed by Dong *et al.* [18] for generating the training set of machine learning algorithms for pattern mining. Table 9 categorizes the preferences of different tools on exact or approximate matching.

2.4 Visualization

Visualization is important in reverse engineering since it can help to envision the mining results of a system so that the user can easily understand the system. Some tools provide visualization support that can present the discovery results in a user friendly way. Design pattern are typically visualized in the UML diagrams or class tree hierarchy.

2.4.1 Visualization in System UML Diagrams

To demonstrate how visualization can improve the understanding of a system, we include the approach presented in [19] as an example. Figure 6 shows the design pattern visualization effect on part of the Java.awt package which is the same part of the system as shown in Figure 2. Design patterns are visualized in UML class diagram. After the pattern instances are discovered from the system, the pattern related information, such as the roles a class, operation, or attribute plays in a pattern, is attached to the corresponding modeling element in the original system representation. The visualization tool can dynamically show or hide pattern-related information based on the current location of the cursor. For example, when the cursor is moved onto a class in the UML diagram, the classes involved in the same pattern instance are highlighted in the same color. The role that each modeling element plays is also displayed in a textbox. Because a class may participate in multiple pattern instances at the same time, different colors are displayed to represent different pattern instances where the class is the overlapping part.

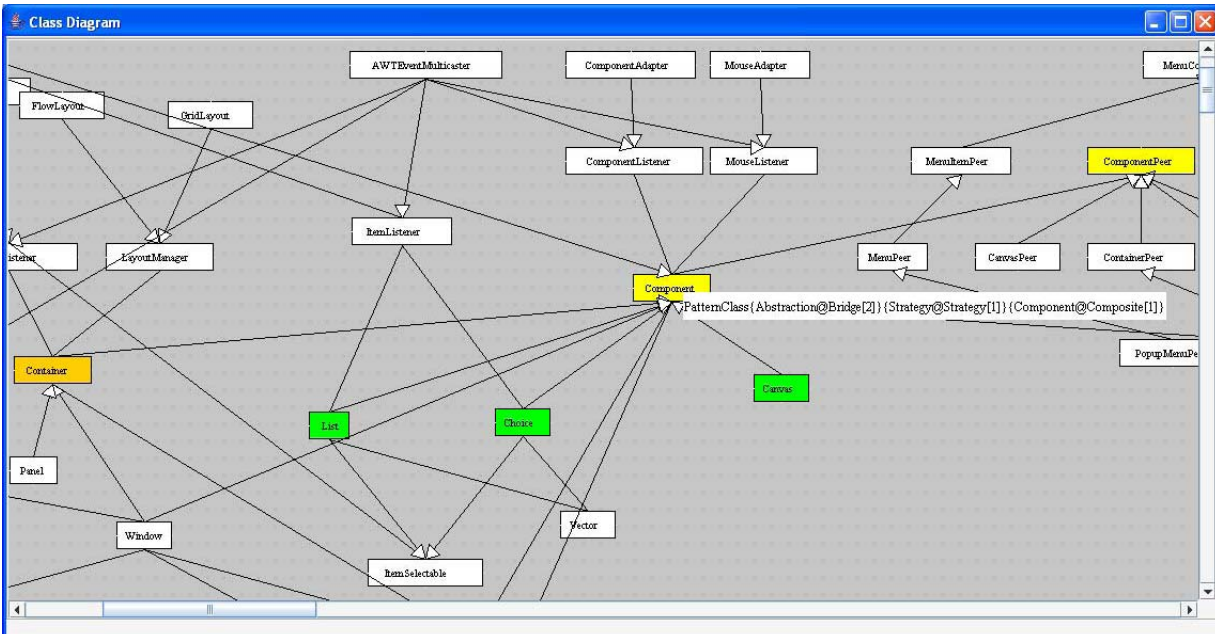


Figure 6 Visualization of Design Patterns in Class Diagram

2.4.2 Visualization in Class Hierarchy

Design patterns can be also visualized in class tree hierarchy in addition to UML diagrams. Each pattern instance is represented in a tree hierarchy, where the first level nodes under the root are the classes participating in the pattern. The role each class plays is defined in one of its children. The attributes and operations of each class and their roles in a pattern instance are also defined in its children. For example, Figure 7 shows the AST of a system, where all classes, their attributes and their operations are listed in a tree structure. We omit the package information for the purpose of simplicity. Under each class, a pattern instances node is included. Before pattern mining, the node can be empty, i.e. containing no child nodes, as shown in Figure 3. After pattern mining, the pattern related information will be added to the AST representation. The AST will be updated to reflect it. This figure shows that the Component class plays the role of Component in the first instance of the Composite pattern. Note that the pattern instances are ordered randomly and indexed sequentially for distinguishing the instances of the same pattern.

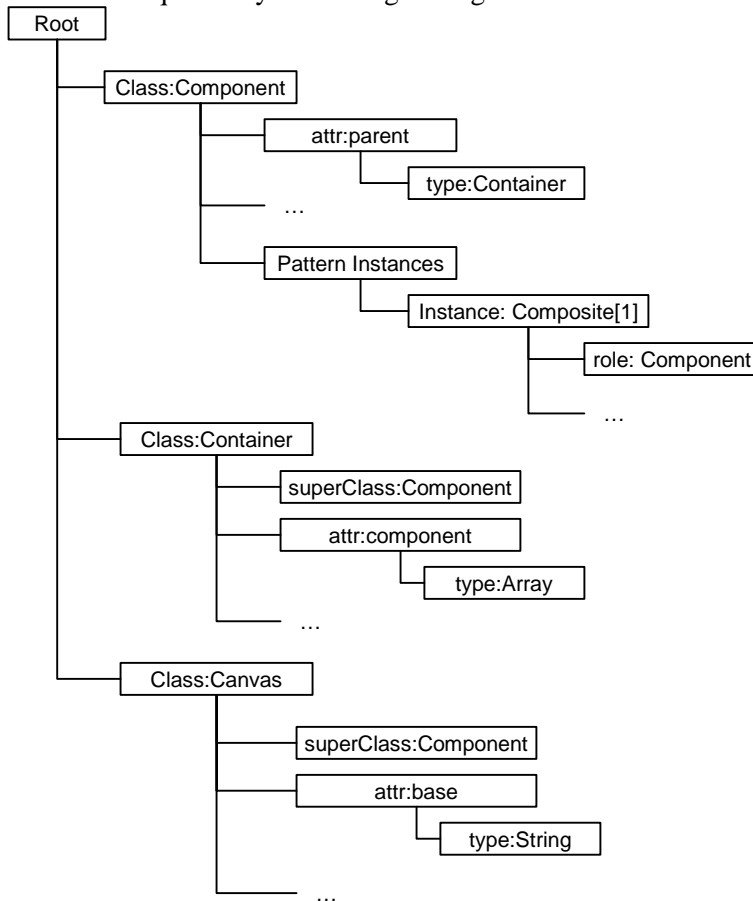


Figure 7 Visualization of Design Patterns in AST

2.4.3 Pattern Visualization Approach Comparison

SPOOL [31] includes a graphical user interface that displays generalization hierarchy diagrams with tree or spring layouts, provides property sheet to control the content of a diagram interactively, and displays the partial class diagram of pattern instances. Dong *et al.* [19] propose a UML profile containing new stereotypes, tagged values and constraints for visualizing pattern-related knowledge in UML diagrams. Dynamic visualization is implemented to identify patterns by changing color when the cursor is on pattern participants. DPRE [13][14] is a visual environment which allows visualizing the imported

UML class diagram. As a supporting environment of pattern recognition, it automatically obtains knowledge on the discovered design patterns and highlights them in the class diagram. However, if visualization support is not provided by the pattern discovery tool, it is better to give the exact location/path of each discovered pattern, rather than to give only the number of discovered pattern.

2.5 Automated or Human Interactive Tool Support

Human judgment in pattern mining process is sometimes more precise since people know what they are looking for. Some researchers take advantage of this and have the users involved in the analyzing process. FUJABA [34] integrates the reverse engineering algorithms into a semi-automatic process. The analyzer has the opportunity to check the results generated by the tool, delete the inappropriate ones, and annotate the results. The analyzer can also resume the process rather than restart it. SPOOL [31] also accepts interruption from the analyzer who may confirm or decline the existence of a pattern, and manually supply specific implementation details. However, the downside of having human interaction in pattern discovery is that it slows down the process.

Authors	Abstract Factory	Adapter/Command	Builder	Bridge	Chain of Responsibilities	Command	Composite	Decorator	Facade	Factory Method	Flyweight	Mediator	Observer/MVC	Prototype	Proxy	Singleton	Strategy/State	Template Method	Visitor
Kramer 1996 [32]		×		×		×	×								×				
Seemann 1998 [37]				×		×											×		
Antoniol 1998 [2]		×		×		×	×								×				
Keller 1999 [31]				×						×								×	
Blewitt 2001 [9]				×	×					×					×	×			
Asencio 2002 [3]	×			×				×		×					×	×	×		
Niere 2002 [34]				×		×											×		
Heuzeroth 2003 [27][28]					×	×	×					×	×						×
Balanyi 2003 [4]	×	×		×	×			×		×				×	×	×	×	×	×
Guéhéneuc 2004 [25]	×	×	×			×	×	×		×			×	×		×	×	×	×
Costagliola 2005 [13][14]		×		×		×	×								×				
Ferenc 2005 [23]		×															×		
Hericko 2005 [26]						×							×				×		
Huang 2005 [29]	×	×	×	×	×	×	×	×		×	×	×	×		×	×	×		×
Kaczor 2006 [30]	×					×													
Shi 2006 [38]	×	×		×	×	×	×	×	×	×	×	×	×		×	×	×	×	×
Tsantalis 2006 [42]		×		×		×	×			×			×	×		×	×	×	×
Dong 2007 [16][20]		×		×		×											×		

Table 10 Design Patterns Discovered by Different Approaches

2.6 Discovered Patterns

Due to the large number of available patterns, each approach often considers only a small number of patterns. A summarization of patterns discovered by different tools is shown in Table 10. This table suggests that the approaches that focus on the structural aspect of patterns, such as Kramer [32] and Costagliola [13][14], usually discover structural patterns with few behavioral patterns. Approaches that take into account both structural and behavioral aspects of patterns, such as Heuzeroth *et al.* [27][28], are able to discover more behavioral patterns.

Authors	Galib	LEDA	Libg++	Java AWT	Java Swing	JDK	JEdit	JHotDraw	JRefactory	JUnit	Mec	QuickUML	socket	zApp class library	Other
Kramer 1996 [32]														×	×
Seemann 1998 [37]				×											
Antoniol 1998 [2]	×	×	×									×	×		×
Tonella 1999 [41]												×			
Keller 1999 [31]															×
Blewitt 2001 [9]				×											
Albin-Amiot 2001 [1]				×			×	×		×					
Asencio 2002 [3]															×
Niere 2002 [34]				×											×
Niere 2002 [35]				×											
Wendehals 2003 [44]				×											
Heuzeroth 2003 [27][28]					×										×
Beyer 2003 [8]				×		×		×							×
Balanyi 2003 [4]		×													×
Guéhéneuc 2004 [25]								×	×	×		×			×
Costagliola 2005 [13][14]	×		×								×				×
Streitferdt 2005 [40]				×											×
Ferenc 2005 [23]															×
Wang 2005 [43]															×
Huang 2005 [29]								×		×					
Kaczor 2006 [30]								×				×			×
Shi 2006 [38]				×	×			×							×
Tsantalis 2006 [42]								×	×	×					
Dong 2007 [16][20]				×			×	×		×					

Table 11 Experiments in Different Studies

Authors	Tools	Structural (ST) Behavioral (BE) Semantic (SE)	Exact (EX) Approximate (AP) match	Autom-atic(AT) Inter-active(IT)	Languages	Techniques	System Representation	Pattern Representation
Kramer 1996 [32]	Pat	ST	EX	AT	C++	Prolog	Prolog	Prolog
Seemann 1998 [37]		ST	EX	AT	Java	first order logic, graph	graph	Predicate
Bansiya 1998 [6]	DP++	ST	EX	AT	C++		class hierarchy	text
Antoniol 1998 [2]		ST&BE	EX	AT	C++	metrics	AST	AOL
Tonella 1999 [41]		ST&BE	EX	AT	C++	concept analysis		
Keller 1999 [31]	SPOOL		EX	IT	C++			UML/CDIF
Blewitt 2001 [9]	Hedgehog	ST&BE&SE	EX	AT	Java			Spine
Albin-Amiot 2001 [1]	Ptidej		EX	AT	Java		CSP	
Asencio 2002 [3]		ST	EX	IT	C++			
Niere 2002 [34]	FUJABA		AP	IT	Java	bottom-up & top-down	ASG	ASG
Wendehals 2003 [44]		ST&BE	EX	AT	Java	dynamic runtime data	ASG & call graph	
Smith 2003 [39]	SPQR	ST	EX	AT	C++	OTTER rho-calculus	OML & OTTER	otter rules
Heuzeroth 2003 [27][28]		ST&BE	EX	AT	Java	SanD and SanD-Prolog	AST	AST & TLA
Beyer 2003 [8]	CroCoPat	ST	EX	AT	Java	predicate calculus	BDDs	predicates
Park 2004 [36]		BE	EX	AT			Class Diagram	
Zhang 2004 [46]		ST	EX	AT			Graph (matrix)	Graph (matrix)
Guéhéneuc 2004 [25]		ST	AP	AT	Java		XML tree & PADL	PADL
Costagliola 2005 [13][14]	DPRE	ST	EX	AT	C++ Java	XPG formalism & LR-based parsing	SVG & AOL->AST	Grammar-based Pattern Specification
Huang 2005 [29]	PRAssistor	ST&BE	EX	AT	C++			Prolog
Streitferdt 2005 [40]		ST	EX	AT	Java			
Ferenc 2005 [23]	Columbus	ST	AP	AT	C++		ASG, XML DOM tree	DPML
Wang 2005 [43]	DPVK	ST&BE	EX	AT		REQL query	REQL static & RSF dynamic	REQL script
Kaczor 2006 [30]	Ptidej	ST	EX	AT	Java		bit vector	bit vector
Shi 2006 [38]	PINOT	ST&BE	EX	AT	Java	Data/Control Flows	AST	DFG & CFG
Tsantalis 2006 [42]		ST	AP	AT	Java	Similarity Matrix	matrix	matrix
Dong 2007 [16][20]	DP-Miner	ST&BE&SE	EX	AT	Java	Matrix and Weight	XMI	XMI

Table 12 Comparison of Pattern Discovery Approaches

2.7 Experiments

Experiment is a good way to evaluate approaches. Table 11 presents a list of software systems used in the experiments by different studies. Due to the lack of documents of these systems, especially open-source systems, it is hard to evaluate the precision of different approaches. Some in-house systems may have documents that can help the analysis of the precision and recall of their approaches. For example, Streitferdt *et al.* [40] experiment on their own systems and Heuzeroth *et al.* [27][28] use Java code of their own mining tool as experiment subjects. Although the precision and recall is relatively easy to compute for these in-house systems, they are not representative experiment samples comparing with other publicly available systems. Most approaches experiment on open-source systems, such as Java.AWT [49], JHotDraw [51], JEdit [50], and JUnit [52]. Due to the lack of documents of these open-source systems, it is hard to compute the precision and recall of the mining tools. Although manual checking has been conducted by some approaches for this purpose, variations of patterns and human errors may bias the values of precision and recall as discussed in the next section.

2.8 Summary

We compare the existing approaches based on the seven dimensions mentioned previously. They all differentiate themselves from others in some dimensions. Table 12 summarizes the comparative study on all different dimensions.

3. Analysis of Experiment Results

As shown in Table 10 and Table 11, most approaches presented experiments on mining different patterns from some application systems. Based on our study in the previous section, we found that different approaches render different results when mining the same pattern in the same system. For instance, Table 13 shows the comparison of the mining results of the same design patterns from the same systems by two different approaches². We investigated the test data presented by different approaches and found some reasons for such disparity. Since most approaches just provide the numbers of discovered patterns whereas only a few of them actually identify the participating classes of each discovered pattern, our following discussions are mostly based on the discovery results of these approaches with identified participants of the discovered patterns. Nevertheless, our analysis is applicable to other approaches because it is based on the foundation of design patterns, instead of each individual mining approach.

3.1 Missing Roles

Each pattern typically includes a group of classes, each of which plays some role. The discovery of pattern generally needs to match all these roles by the classes in the systems. Due to the flexibility of patterns, however, some approaches may choose partial matches of the roles of each design pattern as discussed in Section 2.3. Thus, more instances of a given pattern may be reported than other approaches.

For example, the Adapter pattern normally includes three roles, Target, Adapter, and Adaptee, as shown in Figure 8. The Adapter class inherits from the Target class. Its Request method delegates tasks to the SpecificRequest in the Adaptee class. In the experiment results of JHotDraw [51] reported by [42], there is a case where the AbstractConnector and Figure classes are discovered as the Adapter and

² The main reason for presenting the results from these two approaches is because such results are explicitly presented in their papers.

Adaptee, respectively. However, the AbstractConnector class does not have any super class. Therefore the Target role is missing.

Systems	JHotDraw5.1		JRefactory2.6.24		JUnit3.7	
Authors	Tsantalis <i>et al.</i> [42]	Guéhéneuc <i>et al.</i> [25]	Tsantalis <i>et al.</i> [42]	Guéhéneuc <i>et al.</i> [25]	Tsantalis <i>et al.</i> [42]	Guéhéneuc <i>et al.</i> [25]
Adapter	18	1	7	7	1	0
Composite	1	1	0	0	1	1
Decorator	3	1	1	0	1	1
Factory Method	3	3	4	1	0	0
Observer	5	2	0	0	4	3
Prototype	1	2	0	0	0	0
Singleton	2	2	12	2	0	2
State	23	2	12	2	3	0
Template Method	5	2	17	0	1	0
Visitor	1	0	2	2	0	0

Table 13 Different Results from the Same System of the Same Version

As another example, the Bridge pattern generally consists of three roles, Abstraction, Implementor, and ConcreteImplementor. The Implementor class provides an interface for the Abstraction class. Its children, ConcreteImplementor, need to implement the interface. Figure 9 presents an instance of the Strategy pattern discovered in Java.AWT reported by [34], where the Component and ComponentPeer classes play the roles of Abstraction and Implementor, respectively. However, the ComponentPeer class is an interface without any concrete classes implementing it. One of the possible reasons for such missing roles may be that Java.AWT is an object-oriented framework including many abstract classes and interfaces. The developers who use the framework are supposed to provide the concrete implementations of these interfaces.

These are two representative examples that show the corresponding approaches miss checking some roles. These examples are not accidental. There are many other such examples. For simplicity, we do not present all. In the rest of this section, we use examples to demonstrate our analysis. These examples reflect the algorithms and methods of the corresponding approach. Therefore, our analysis is able to generalize to these and other similar approaches.

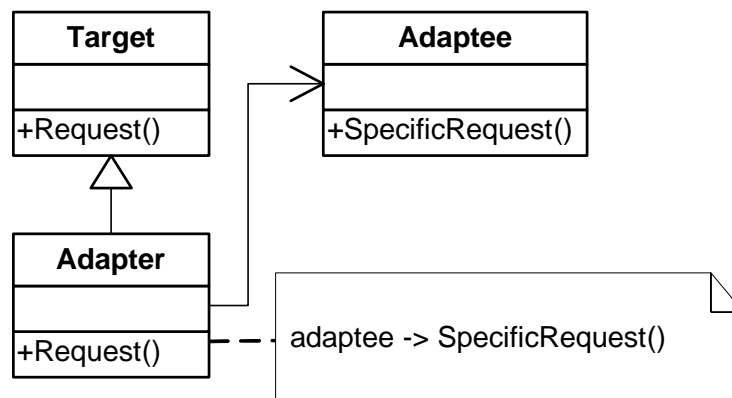


Figure 8 the Adapter Pattern

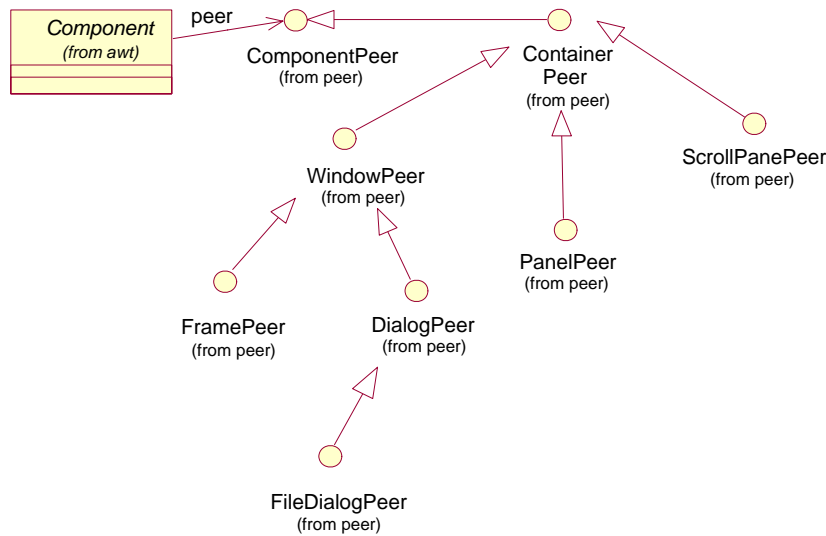


Figure 9 Partial Class Diagram for Java.AWT

3.2 Role Types

The classes in each pattern are often restricted to some types, such as abstract classes, interfaces, or concrete classes. Such restriction allows a pattern to define its interfaces and implementations so that it is easy to make changes. For example, the Target class is typically an abstract class or an interface, whereas the Adapter and Adaptee classes are concrete. In this way, the Target class provides an interface (Request) to the client, whose implementation is provided by its child (Adapter). In the Adapter pattern, the Adapter class does not provide the implementation of the Request methods, instead, it delegates to the SpecificRequest method in the Adaptee class. Thus, the Adapter and Adaptee classes should be concrete classes. In system source code, however, the situation can be more complex. For example, an Adapter pattern instance was identified in JHotDraw by [42], which considered the DrawingView class as the Adaptee. However, the DrawingView is actually an interface, instead of a concrete class. All operations in DrawingView are implemented in its children classes, such as the StandardDrawingView. Another Adapter pattern instance identified in JHotDraw by [38] includes the MDI_DrawApplication, JavaDrawApp, and Animator classes, which play the roles of Target, Adapter, and Adaptee respectively. However, MDI_DrawApplication is a concrete class with its own implementation for the Request method. Therefore, it cannot delegate to the Adapter or Adaptee. These examples actually show the corresponding approaches miss checking the type of pattern roles, which is not an isolated problem in other approaches.

3.3 Missing Relationships

The relationships, such as association, generalization, and delegation, between classes are normally important parts of a design pattern. Without certain relationships between the classes, a group of classes may not be considered as a given pattern. Thus, the pattern discovery approaches need to check the important relationships among the classes while mining a design pattern.

Delegation between two classes is generally an important characteristic of a design pattern. One operation of a class may delegate its responsibility to the other operation in another class. For example, the Adapter pattern requires the Request method in the Adapter class delegates its implementation to the SpecificRequest method in the Adaptee class. Delegation is sometimes simplified to association by some

pattern mining tools. That is, if a reference/pointer to the Adaptee class is defined in the Adapter class, the delegation requirement is considered to be satisfied. However, such simplification may actually introduce some problems: first, even if a reference/pointer to the Adaptee class is defined in the Adapter class, it may not be used to call any method in Adaptee at all. Second, even if it is used to call a method in Adaptee, the invocation may not appear in the right Request method. We illustrate these two problems in the following false positive instances of the Adapter pattern discovered from JHotDraw by [42].

The first instance of the Adapter pattern identified the TextFigure and OffsetLocator classes as the Adapter and Adaptee, respectively. The TextFigure class has an association relationship with the OffsetLocator class, where the reference fLocator of type OffsetLocator is defined in the TextFigure class. However, there is no method invocation using the fLocator that is found in TextFigure. Even if the association between the TextFigure and OffsetLocator classes exists, thus, the delegation does not.

The second instance of the Adapter pattern considered the PaletteListener, DrawApplet, and ToolButton as the Target, Adapter, and Adaptee, respectively. The PaletteListener and DrawApplet classes have two common methods, paletteUserOver() and paletteUserSelected(), which play the role of Request in the Adapter pattern. Although there exists an invocation from a method in DrawApplet (Adapter) to another method in ToolButton (Adaptee), the method invocation is not initiated from either paletteUserOver() or paletteUserSelected(). Therefore, this is not a valid instance of the Adapter pattern since the SpecificRequest method does not really exist.

Generalization is also an important relationship that appears in many patterns. For instance, the Adapter class inherits from the Target class in the Adapter pattern, and the Composite class inherits from the Component class in the Composite pattern as shown in Figure 11. Some approaches may miss checking some of these important relationships between the classes, which may result in false positive cases. For example, an instance of the Composite pattern was discovered in JHotDraw by [42], where the Figure and CompositeFigure classes are considered as the Component and Composite, respectively. However, there is no generalization relationship between the Figure and CompositeFigure classes. An Adapter instance identified in JHotDraw by [38] contains the Tool, PolygonTool, and PolygonFigure classes playing the roles of Target, Adapter, and Adaptee, respectively. However, no generalization relationship exists between the PolygonTool (Adapter) and Tool (Target) classes.

Association is also an important relationship in a pattern. An instance of the Bridge pattern identified in JHotDraw by [38] includes the DrawApplet and VersionControlStrategy classes with the roles of Abstraction and Implementor, respectively. However, there is no (association) relationship between the DrawApplet and VersionControlStrategy classes.

3.4 Delegation Implementations

As discussed in the previous section, delegation is an important mechanism that appears in many patterns. Essentially, it refers to the invocation from one method in a class to another method in another class. It can be implemented in many different ways, which complicates its discovery process. For the same example of the Adapter pattern, there should be a delegation from the Request method of the Adapter class to the SpecificRequest method in the Adaptee class. In order to make such invocation, there should be a reference/pointer to the Adaptee class defined in the Adapter class. However, this reference/pointer can be accessed by many different ways. For instance, it can be returned by a getter() method; it can be copied to another variable with the same Adaptee type; it can be cast to another variable with the parent type of Adaptee. Given so many various ways that the reference/pointer to the Adaptee can be when it is used to call the SpecificRequest, it is hard to find all possible cases. Hence, some approaches may choose to simply consider whether the reference/pointer to Adaptee is defined in the Adapter class. If the Request method in the Adapter class also uses this particular reference/pointer to call the SpecificRequest method in the Adaptee method, then it satisfies the delegation requirement. Nevertheless, this simplification may rule out some correct pattern instances that are false-negative cases.

Besides the Adapter pattern, many patterns have the same issue with delegation. In the Strategy pattern, for example, the ContextInterface() in the Context class needs to invoke the AlgorithmInterface() method in the Strategy class, which is dynamically bound to an AlgorithmInterface() in a ConcreteStrategy class. Figure 10 shows two false-negative instances of the Strategy pattern appeared in Java.AWT, which should not have been eliminated by [34]. One instance includes the Component class playing the role of Context, the ComponentListener class playing the role of Strategy, and the NativeInLightFixer and AWTEventMulticaster classes playing the role of ConcreteStrategy. The other instance consists of the Container class playing the role of Context, the ContainerListener class playing the role of Strategy, and the NativeInLightFixer and AWTEventMulticaster classes playing the role of ConcreteStrategy. The componentListener is an attribute with the type of ComponentListener defined in the Component class. One of the AlgorithmInterface operations is named componentShown that is defined in the ComponentListener class. Therefore, the simplified approach may only check whether there is a method invocation: componentListener.componentShown() in the Component class. However, the source code actually copies the componentListener to another variable, named “listener”, which is used to invoke the componentShown method:

```
ComponentListener listener = componentListener;
listener.componentShown();
```

This variation of delegation implementation seems to be the main reason that this Strategy pattern instance is not discovered by some approaches. For the same reason, the other Strategy pattern instance shown in Figure 10 flies under the radar of some approaches.

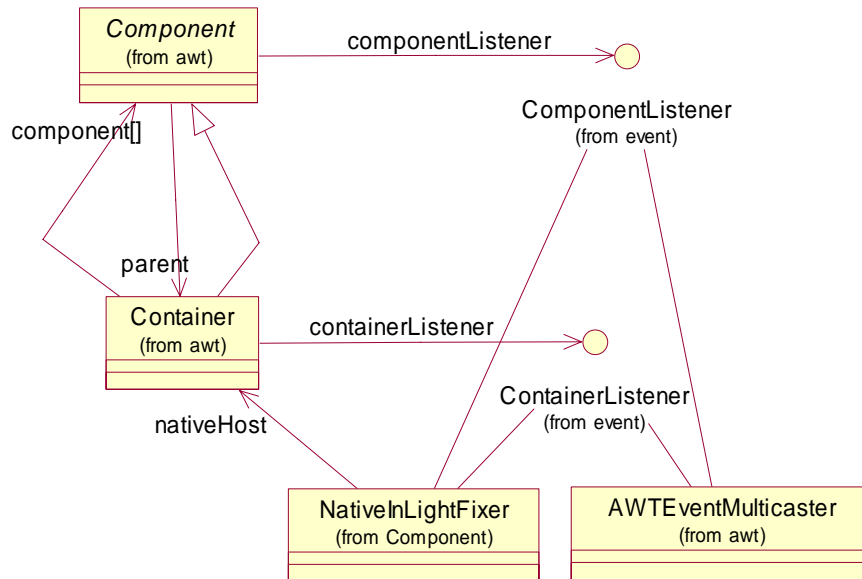


Figure 10 Class Diagram for Partial Java.AWT Package

One possible solution to this problem is to just check the operation name (e.g. AlgorithmInterface in the Strategy pattern) and omit its reference/pointer. For the previous example, we can just check whether there exists an invocation to componentShown(), instead of worrying about which reference/pointer, e.g., listener or componentListener, it belongs. In this way, both false-negative instances of the Strategy pattern shown in Figure 10 can be discovered. However, this solution may introduce some other problem that leads to false-positive cases. Different classes may include the same operation name. For example, the componentShown() operation may be defined in classes other than the ComponentListener, NativeInLightFixer, and AWTEventMulticaster classes, and be invoked by the Component class. In fact,

such situation does exist in some systems, e.g., in JHotDraw, where a potential Strategy pattern instance includes the JavaDrawViewer, Drawing, and StandardDrawing classes as the Context, Strategy, and ConcreteStrategy, respectively. The *add(Figure)* operation possibly playing the role of AlgorithmInterface is defined in the Strategy and ConcreteStrategy classes. However, it is not invoked by any methods in the JavaDrawViewer (Context) class. Instead, another *add("Center", fView)* operation with the same operation name but different number of parameters, which is not defined in the Strategy and ConcreteStrategy classes, is invoked in the JavaDrawViewer (Context) class. Therefore, this is not a correct instance of the Strategy pattern. It is instead a false-positive case.

To reduce such kind of false-positive instances, a solution [16] is to check the number of parameters of the required operation, as well as its name. This solution can eliminate the previous false-positive case. However, there are still false-positive situations that may occur. Although the number of parameters of two operations is the same, for instance, their parameters may have different types. In addition, the reference/pointer that is used to invoke the particular operation may not be the desired reference/pointer.

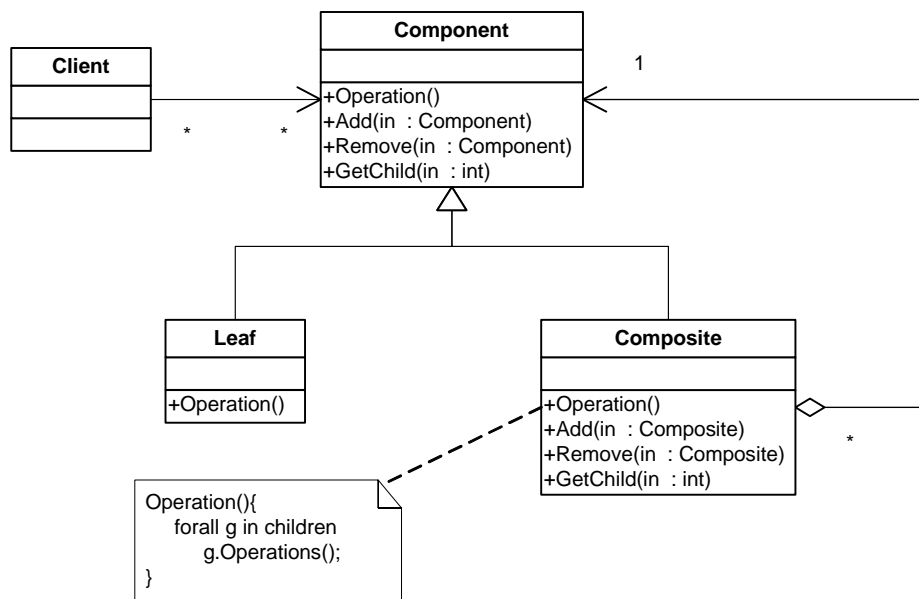


Figure 11 Class Diagram for the Composite Pattern

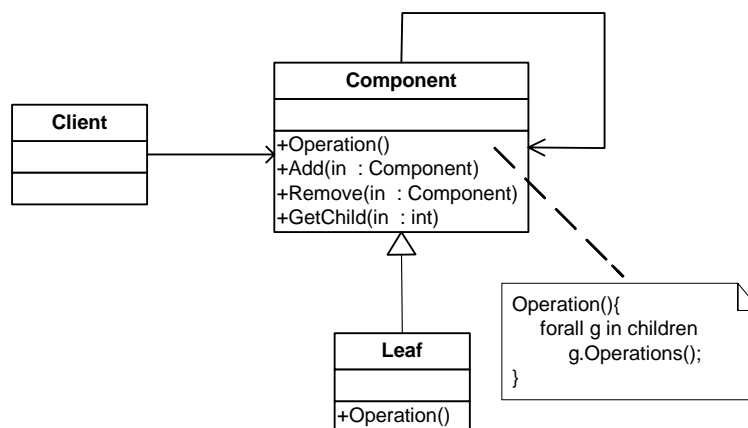


Figure 12 A Variation of the Composite Pattern

3.5 Implementation Variations

When a pattern is applied in a system, it can be implemented in many different ways. Some programming languages even provide special language constructs or library that facilitates the pattern implementation. Using such library, nevertheless, may make the pattern instances harder to discover because the library code is generally not in the scope of pattern discovery. For instance, Figure 11 shows the class diagram of the typical Composite pattern. The Add, Remove, and GetChild methods are used to manage an aggregate of components. Some programming languages, e.g., Java, provide aggregation framework, such as LinkedList, ArrayList, HashMap, and Hashtable, which can be used to manage any type of aggregated components. Such library facilities can be used to replace the Add, Remove, and GetChild methods, which make pattern mining more difficult because these library classes are not part of system source code. This issue has also been recognized by Wirfs-Brock [45]. When the library facilities are used in the Composite class, additionally, there is no need to keep the interface of the Add, Remove, and GetChild operations in the Component class. Thus, the pattern discovery processes should also take this issue into consideration. This is also the reason that some of the existing pattern discovery approaches miss instances of such cases. For instance, one of the instances of the Composite pattern in JUnit missed by [42] includes the IMoney, Money, and MoneyBag classes that play the roles of the Component, Leaf, and Composite, respectively.

3.6 Merging Roles

One of the difficulties of pattern discovery is that a design pattern may have several variations and can be implemented differently. For example, Figure 11 displays a typical class diagram of the Composite pattern that may have several variations. One of such variation is shown in Figure 12, where the roles of the Component and Composite are merged into one single class. Such variations of design patterns may be discovered by some approaches and missed by others.

3.7 Summary

In this section, we present an analysis of the experiment results from different approaches and study the reasons for the disparity among the results. We found that the main reasons are missing roles, role types, and relationships, such as association, generalization, and delegation, as well as implementation variations. Such omissions are actually part of underlying theory and principles of design patterns. We backed up our analysis with some representative examples that are generalizeable to other approaches. We have not enumerate all cases for each problem because the main goal of our analysis is to identify the problems and challenges of current pattern mining techniques and to show the causes of the problems to be missing checking important object-oriented characteristics underlying the design patterns. Our study demonstrates the importance of identifying the essential characteristics of each design patterns [21].

4. Conclusions

Software design patterns encapsulate good ideas for solving high level problems in large software systems. They have been widely adopted by software industry as best practice. They benefit not only forward engineering when software architect needs to design and develop software systems from scratch, but also reverse engineering when mining and understanding of a software system is critical to its change and evolution. Patterns help to improve software architectural design by reusing expert solutions. However, patterns are normally not traceable from system source code. The design and its system may evolve and migrate independent from each other making them inconsistent. Mining design patterns from

a system may greatly assist the understanding and further improvement of system quality. It also visualizes the evolution and migration of system design over time.

Many pattern mining techniques and tools have been proposed in the literature. In this paper, we presented a comparative study on these approaches in terms of the pattern aspects checked, the intermediate representations, exact or approximate matches, visualization, automated or human interactive supports, mining results and experiments based on our initial work in [22]. In addition, we discussed the results rendered by these approaches and analyzed the disparity of these results and the potential reasons. Although most of experiments conducted by these approaches are on different systems, some of them use the same systems. Our study demonstrates that there is disparity among the experiment results from these approaches even on the same systems. This signifies a key challenge in pattern discovery, that is, there is no standard and rigorous definition of the characteristics of each pattern. Patterns are normally described informally for easy understanding. Moreover, many patterns present flexible architectural design that can be instantiated into several different applications. Although formal methods have been used to specify design patterns and their integrations [15], these approaches tend to focus on the introduction of formal methods for patterns specifications, instead of concentrating on the identification of the essential characteristics of each pattern which are important architectural design features to be discovered. Initial work addressing this challenging issue in pattern discovery has been conducted in [21].

Most of existing approaches on pattern discovery present some experiments on open-source and/or in-house systems. Few approaches discussed the precision and recall of their techniques with the following possible reasons. First, the experimental systems, especially the open-source systems, do not provide any architecture and design documents that clearly identify the numbers and exact locations of the pattern instances. Second, patterns are generally flexible design templates that may have several variations, which may lead to different implementations. Third, design patterns are typically described informally, which may cause ambiguity, imprecision, and misunderstanding. Due to these reasons, different approaches may present different results for mining the same pattern from the same system. To the best of our knowledge, there is no benchmark system available so far. Benchmarking is important because it can help to compare different approaches with the same environment and criteria. We plan to work on benchmarking the pattern discovery techniques [20].

New methods and tools are continuously proposed with the new trend on applying interdisciplinary technologies, such as computer vision [17] and machine learning [18], into design pattern mining. While existing approaches and tools are typically evaluated in an ad-hoc manner, the comparison among these approaches may not result in a best choice because they are not evaluated with a common benchmark as discussed previously. A benchmark may allow the study of the best choices of tools in different situations.

Software architectural design is an abstraction of a large complex system, which aids the understanding and analysis of the system. Although the study of software architectural design may be independent from its system, maintaining the connection between a system and its architectural design is critically important. The study of software architectural design without concerning the connection to its system may lose touch of the reality. In summary, there are several benefits of our study and analysis in this paper. Our study allows a deep understanding of the problems of pattern discovery. It also identifies the challenging issues in mining patterns from both the system source code and architectural design. Furthermore, our analysis provides the insight on the effectiveness of different solutions by comparing different approaches. Moreover, our study helps to recognize new related problems, such as benchmarking and pattern charactering.

References

- [1] H. Albin-Amiot, P. Cointe, Y. Guéhéneuc, 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.
- [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] Asencio, S. Cardman, D. Harris, and E. Laderman, “Relating expectations to automatically recovered design patterns.” *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE)*, 2002.
- [4] 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.
- [5] D.H. Ballard and C.M. Brown, *Computer Vision*, Prentice Hall, 1982.
- [6] J. Bansiya, Automating design-pattern identification – DP++ is a tool for C++ programs. *Dr. Dobbs Journal*, 1998.
- [7] L. Bass, P. Clements and R. Kazman, *Software Architecture in Practice*, Addison Wesley. 2003.
- [8] D. Beyer, A. Noack, and C. Lewerentz, “Simple and efficient relational querying of software structures.” In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE’03)*, 2003.
- [9] Blewitt and A. Bundy, “Automatic verification of Java design patterns.” *Proceedings of International Conference on Automated Software Engineering*, pp. 324-327, 2001.
- [10] V.D. Blondel, A. Gajardo, M. Heymans, P. Senellart, and P. Van Dooren, “A Measure of Similarity between Graph Vertices: Applications to Synonym Extraction and Web Searching,” *SIAM Rev.*, vol 46, no. 4, pp. 647-666, 2004.
- [11] G. Booch, J. Rumbaugh, I. Jacobson. *The Unified Modeling Language User Guide*, Addison-Wesley, 1999.
- [12] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad and M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns*, Wiley. 1996.
- [13] 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.
- [14] G. Costagliola, A. Lucia, V. Deufemia, C. Gravino, and M. Risi, “Case studies of visual language based design patterns recovery.” In *Proceedings of the Conference on Software Maintenance and Reengineering (CSMR)*, 2006.
- [15] Jing Dong, Paulo Alencar, and Donald Cowan, A Formal Framework for Design Component Contracts, *the Proceedings of the IEEE International Conference on Information Reuse and Integration (IRI)*, pages 53-60, Las Vegas, US, October 2003.
- [16] 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.

- [17] Jing Dong, Yongtao Sun, Yajing Zhao, Design Pattern Detection By Template Matching, *the Proceedings of the 23rd Annual ACM Symposium on Applied Computing (SAC)*, pages 765-769, Ceará, Brazil, March 2008.
- [18] Jing Dong, Yongtao Sun, Yajing Zhao, Compound Record Clustering Algorithm for Design Pattern Detection by Decision Tree Learning, *the Proceedings of the IEEE International Conference on Information Reuse and Integration (IRI)*, USA, July 2008.
- [19] J. Dong, S. Yang and K. Zhang, Visualizing Design Patterns in Their Applications and Compositions, *IEEE Transactions on Software Engineering*, Volume 33, Number 7, pp. 433-453, July 2007.
- [20] J. Dong and Y. Zhao, Experiments on Design Pattern Discovery, *the Proceedings of the 3rd International Workshop on Predictor Models in Software Engineering (PROMISE)*, in conjunction with ICSE, USA, May 2007.
- [21] Jing Dong, Yajing Zhao, Classification of Design Pattern Traits, *the Proceedings of the Nineteenth International Conference on Software Engineering and Knowledge Engineering (SEKE)*, Boston, USA, July 2007.
- [22] Jing Dong, Yajing Zhao, and Tu Peng, Architecture and Design Pattern Discovery Techniques – A Review, *the Proceedings of the 6th International Workshop on System/Software Architectures (IWSSA)*, USA, June 2007.
- [23] R. Ferenc, A. Beszedes, L. Fulop, and J. Lele, “Design pattern mining enhanced by machine learning.” In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM’05)*, 2005.
- [24] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
- [25] Y. Guéhéneuc, H. Sahraoui, and F. Zaidi, “Fingerprinting design patterns.” *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE)*, 2004.
- [26] M. Hericko and S. Beloglavec, “A composite design-pattern identification technique.” *Informatica* 29, pp 469–476, 2005.
- [27] D. Heuzeroth, T. Holl, G. Hogstrom, and W. Lowe, “Automatic design pattern detection.” In *Proceedings of the 11th International Workshop on Program Comprehension (IWPC 2003)*, pp 94-103, 2003.
- [28] 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.
- [29] Heyuan Huang, Shensheng Zhang, Jian Cao, Yonghong Duan, A practical pattern recovery approach based on both structural and behavioral analysis, *Journal of Systems and Software*, 75(1-2), pp.69-87, February 2005
- [30] Kaczor, Y. Guéhéneuc, and S. Hamel, Efficient Identification of Design Patterns with Bit-vector Algorithm, *Proceedings of the Conference on Software Maintenance and Reengineering (CSMR)*, 2006.
- [31] R. K. Keller, R. Schauer, S. Robitaille, and P. Page, Pattern-based reverse-engineering of design components. *Proceedings of the 21st International Conference on Software Engineering (ICSE)*, pp 226-235. 1999.

- [32] C. Kramer and L. Prechelt, "Design recovery by automated search for structural design patterns in object-oriented software." In *Proceedings of 6th International Workshop on Program Compression (IWPC'98)*, 1998.
- [33] H. Mei, T. Xie, and F. Yang, "JBOORET: an automated tool to recover OO design and source models." In *Proceedings of the 25th Annual International Computer Software & Applications Conference (COMPSAC)*, 2001.
- [34] 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.
- [35] 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.
- [36] C. Park, Y. Kang, C. Wu, and K. Yi, "A static reference analysis to understand design pattern behavior." In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE'04)*, 2004.
- [37] J. Seemann and J. Wolff. 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.
- [38] 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.
- [39] J. M. Smith and D. Stotts, "SPQR: Flexible automated design pattern extraction from source code." In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE)*, 2003.
- [40] 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)*, 2005.
- [41] P. Tonella, G. Antoniol, "Object oriented design pattern inference." In *Proceedings of International Conference on Software Maintenance (ICSM'99)*, 1999.
- [42] 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.
- [43] W. Wang and V. Tzerpos, "Design pattern detection in Eiffel systems." In *Proceedings of 12th Working Conference on Reverse Engineering (WCRE'05)*, 2005.
- [44] L. Wendehals, Improving design pattern instance recognition by dynamic analysis. *Proceedings of the ICSE workshop on Dynamic Analysis*, pp. 29-32, 2003.
- [45] R. J. Wirfs-Brock, "Refreshing Patterns," *IEEE Software*, vol.23, no.3, pp. 45-47, May/June, 2006.
- [46] Z. Zhang, Q. Li, and K. Ben, "A new method for design pattern mining." In *Proceedings of the 3rd International Conference on Machine Learning and Cybernetics*, 2004.
- [47] Design Pattern Detection using Similarity Scoring, <http://java.uom.gr/~nikos/pattern-detection.html>
- [48] Fujaba User Documentation
<http://wwwcs.uni-paderborn.de/cs/fujaba/documents/user/manuals/FujabaDoc.pdf>
- [49] Java.awt resource information, September 2006,
<http://java.sun.com/j2se/1.5.0/docs/guide/awt/index.html>.

- [50] JEdit – Programmer’s Text Editor. <http://www.jedit.org/>
- [51] JHotDraw Start Page. <http://www.jhotdraw.org/>
- [52] JUnit, Testing Resources for Extreme Programming. <http://www.junit.org/>
- [53] UML metamodel version 1.1 documentation,
http://www.cse.msu.edu/~cse870/Materials/UML11_Metamodel_Diagrams.pdf, September, 1997.
- [54] W3C, Extensible Markup Language (XML), <http://www.w3.org/>