Architecture and Design Pattern Discovery Techniques – A Review

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

Architecture and design patterns, as demonstrated solutions to recurring problems, have proved practically important and useful in the process of software development. They have been extensively applied in industry. Discovering the instances of architecture and design patterns from the source code of software systems can assist the understanding of the systems and the process of re-engineering. More importantly, it also helps to trace back to the original architecture and design decisions, which are typically missing for legacy systems. This paper presents a review on current techniques and tools for discovering architecture and design patterns from object-oriented systems. We classify different approaches and analyze their results. We also discuss the disparity of the discovery results from different approaches and analyze possible reasons with some insight.

1. Introduction

Architecture and design patterns [10][6][16] have been widely adopted by software industry. Software developers become routinely applying architecture and design patterns to solve their problems. When patterns are applied and implemented in a system, however, the pattern-related information is generally no longer available in the source code. It is hard to trace such architectural design information in source code. Recovering pattern instances from system source code 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 legacy systems.

Many existing techniques and tools can help recover the architecture and design from system source code. These reverse engineering tools become the foundation for architecture and 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 discovery 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 may generalize common processes for pattern discovery. Third, we found that different approaches may render different results while discovering 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 classify different pattern discovery 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 discuss the disparity of different results and the potential reasons and insights for such differences. We also discuss the issues related to the precision and recall. Since most of existing approaches are applied on object-oriented systems, our study has the same focus.

The rest of this paper is organized as follows. Section 2 presents a comparison on different characteristics of pattern discovery techniques. Section 3 introduces the different discovery results from different approaches and discusses the reasons. Section 4 concludes the paper.

2. Comparative Study

Discovering the instances of architecture and design patterns from system source code can help to trace back and understand the original architectural decisions and tradeoffs and reengineer the systems. There have been a number of different approaches proposed to solve this problem in the literature. In this section, we compare these approaches in different aspects. 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 pattern is generally described by different aspects, such as structure and behavior. Thus, different approaches may choose to check either structural, behavioral, or both aspects. 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. Pattern discovery is actually done from these intermediate representations, instead of source code. Current approaches use different tools to get different intermediate representations. The choice of intermediate representations 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 a certain number of patterns. Seventh, different approaches conduct experiments on different open-source systems. Table 1 through Table 3 provide detailed descriptions of our study.

3. Discussions on Experiment Results

Based on our study in the previous section, we found that different approaches render different results when detecting the same pattern in the same system as, e.g., shown in Table 4. We investigated the test data presented by different approaches and found some reasons for such disparity. Since most approaches just provide the numbers of detected patterns and only a few of them actually provided the participating classes of each detected pattern, our following discussions are mostly based on the discovery results of these approaches.

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 pattern. Thus, more instances of a given pattern may be reported than other approaches.

Authors	Tools	Structural (ST) Behavioral (BE) Semantic (SE)	Exact (EX) Approximate(AP) match	Automatic (AT) Interactive (IT)	Languages	Techniques	System Representation	Pattern Representation
Kramer 1996 [23]	Pat	ST	EX	AT	C++	Prolog	Prolog	Prolog
Seemann 1998 [28]		ST	EX	AT	Java	first order logic, graph	graph	Predicate
Bansiya 1998 [5]	DP++	ST	EX	AT	C++		class hierarchy	text
Antoniol 1998 [2]		ST&BE	EX	AT	C++	metrics	AST	AOL
Tonella 1999 [32]		ST&BE	EX	AT	C++	concept analysis		
Keller 1999 [22]	SPOOL		EX	IT	C++			UML/CDIF
Blewitt 2001 [8]	Hedgehog	ST&BE&SE	EX	AT	Java			Spine
Mei 2001 [24]	JBOORET	ST	EX	AT	C++			
Albin-Amiot 2001 [1]	Ptidej		EX	AT	Java		CSP	
Asencio 2002 [3]		ST	EX	IT	C++			
Niere 2002 [25]	FUJABA		AP	IT	Java	bottom-up & top-down	ASG	ASG
Wendehals 2003 [35]		ST&BE	EX	AT	Java	dynamic runtime data	ASG & call graph	
Smith 2003 [30]	SPQR	ST	EX	AT	C++	OTTER rho-calculus	OML & OTTER	otter rules
Heuzeroth 2003 [19][20]		ST&BE	EX	AT	Java	SanD and SanD-Prolog	AST	AST &TLA
Beyer 2003 [7]	CroCoPat	ST	EX	AT	Java	predicate calculus	BDDs	predicates
Park 2004 [27]		BE	EX	AT			Class Diagram	
Zhang 2004 [37]		ST	EX	AT			Graph (matrix)	Graph (matrix)
Gueheneuc 2004 [17]		ST	AP	AT	Java		XML tree & PADL	PADL
Costagliola 2005 [11][12]	DPRE	ST	EX	AT	C++ Java	XPG formalism & LR-based parsing	SVG & AOL->AST	Grammar-based Pattern Specification
Streitferdt 2005 [31]		ST	EX	AT	Java			
Ferenc 2005 [15]	Columbus	ST	AP	AT	C++		ASG, XML DOM tree	DPML
Wang 2005 [34]	DPVK	ST&BE	EX	AT		REQL query	REQL static & RSF dynamic	REQL script
Kaczor 2006 [21]	Ptidej	ST	EX	AT	Java		bit representation	bit representation
Shi 2006 [29]	PINOT	ST&BE	EX	AT	Java	Data/Control Flows	AST	DFG & CFG
Tsantalis 2006 [33]		ST	AP	AT	Java	Similarity Matrix	matrix	matrix
Dong 2007 [13][14]	DP-Miner	ST&BE&SE	EX	AT	Java	Matrix and Weight	XMI	XMI

Table 1 Comparison of Pattern Recovery Approaches

Authors	Abstract Factory	Adapter /Command	Builder	Bridge	Chain of Responsibilities	Command	Composite	Decorator	Facade	Factory Method	Flyweight	Mediator	Observer	Prototype	Proxy	Singleton	Strategy/State	Template Method	Visitor
Kramer 1996 [23]		×		×			×	×							×				
Seemann 1998 [28]				×			×										×		
Antoniol 1998 [2]		×		×			×	×							×				
Keller 1999 [22]				×						×								×	
Blewitt 2001 [8]				×		×				×					×	×			
Asencio 2002 [3]	×			×				×		×					×	×	×		
Niere 2002 [25]				×			×										×		
Heuzeroth 2003 [19][20]					×		×	×				×	×						×
Balanyi 2003 [4]	×	×		×	×			×		×				×	×	×	×	×	×
Gueheneuc 2004 [17]	×	×	×			×	×	×		×			×	×		×	×	×	×
Costagliola 2005 [11][12]		×		×			×	×							×				
Ferenc 2005 [15]		×															×		
Hericko 2005 [18]							×						×				×		
Kaczor 2006 [21]	×						×												
Shi 2006 [29]	×	×		×	×		×	×	×	×	×	×	×		×	×	×	×	×
Tsantalis 2006 [33]		×		×			×	×		×			×	×		×	×	×	×
Dong 2007 [13][14]		×		×			×										×		

Table 2 Architecture and Design Patterns Detected by Different Approaches

Authors	Galib	LEDA	Libg++	Java AWT	Java Swing	JDK	JEdit	JHotDraw	JRefactory	JUnit	Mec	QuickUML	socket	zApp class library	Other
Kramer 1996 [23]														×	Network Management Environment Library of Efficient Datatype Algorithms Automatic Call Distribution
Seemann 1998 [28]				×											
Bansiya 1998 [5]															DTK
Antoniol 1998 [2]	×	×	×								×		×		Groff
Tonella 1999 [32]											×				
Keller 1999 [22]															ET++
Blewitt 2001 [8]				×											
Albin-Amiot 2001 [1]				×			×	×		×					
Asencio 2002 [3]															MISM SUIF Mozilla(subset) GangOfFour ACE
Niere 2002 [25]				×											JGL Libraries
Niere 2002 [26]				×											
Wendehals 2003 [35]				×											
Heuzeroth 2003 [19][20]					×										Java code of their own tool
Beyer 2003 [7]				×		×		×							JWAM 1.6 Eclipse 2.02
Balanyi 2003 [4]		×													Jikes, Calc, Writer
Gueheneuc 2004 [17]								×	×	×		×			Lexi, NetBeans
Costagliola 2005 [11][12]	×		×								×				Socket
Streitferdt 2005 [31]				×											Drawlet Tomcat Patterns (they developed)
Ferenc 2005 [15]															StarWriter (of sum.com)
Wang 2005 [34]															Eiffel
Kaczor 2006 [21]								×							Juzzle Quick UML
Shi 2006 [29]				×	×			×							Apache Ant
Tsantalis 2006 [33]								×	×	×					
Dong 2007 [13][14]				×			×	×		×					

Table 3 Experiments Done in Different Studies

Systems	JHotDraw5.1		JRefactory2.6.24		JUnit3.7	
References	[33]	[17]	[33]	[17]	[33]	[17]
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 4 Different Results from the Same System-Version

For example, the Adapter pattern normally includes three roles, Target, Adapter, and Adaptee, as shown in Figure 1. 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 [39] reported by [33], there is a case where the AbstractConnector and Figure classes are recovered as the Adapter and Adaptee, respectively. However, the AbstractConnector class does not have any super class. Therefore the Target role is missing.

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 2 presents an instance of the Strategy pattern discovered in Java.AWT [38] reported by [25], 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 implementations of these interfaces.

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 [33], 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 [29] 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. Thus, it cannot delegate to the Adapter or Adaptee.

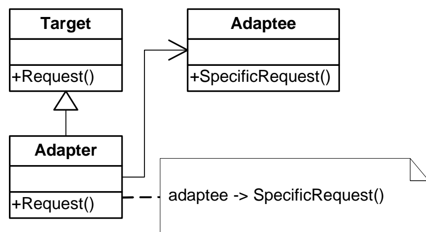

Figure 1 The Adapter Pattern

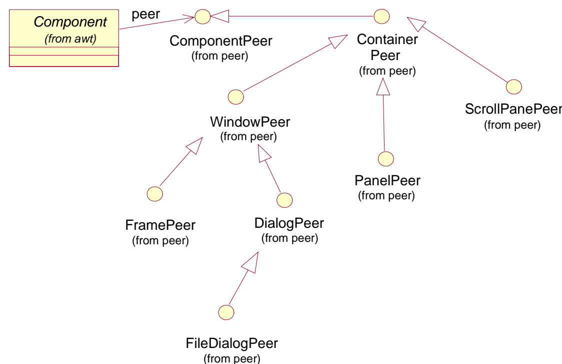

Figure 2 Partial Class Diagram for Java.AWT

3.3 Missing Relationships

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

Delegation between two classes is generally an important characteristic of a pattern. 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 discovery 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 [33].

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 4. 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 detected in JHotDraw by [33], 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 [29] 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 an important relationship in a pattern. An instance of the Bridge pattern identified in JHotDraw by [29] 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 3 shows two false-negative instances of the Strategy pattern appeared in Java.AWT, which should not have been eliminated by [25]. 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 3 flies under the radar of some approaches.

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 3 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.

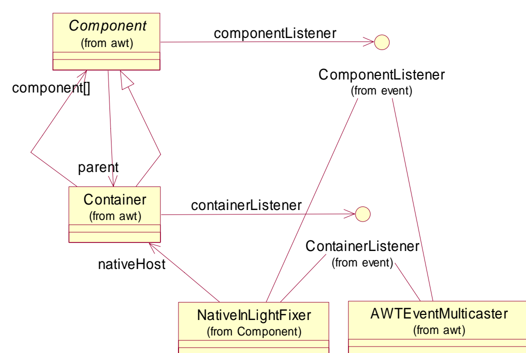

Figure 3 Class Diagram for Partial Java.AWT Package

To reduce such kind of false-positive instances, a solution [13] 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.

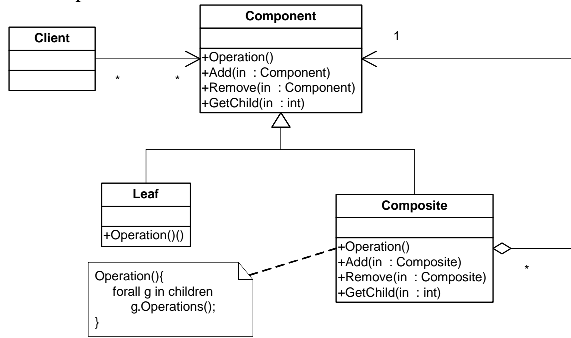

Figure 4 Class Diagram for the Composite Pattern

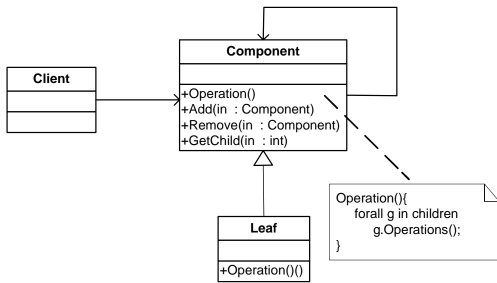

Figure 5 A Variation of the Composite Pattern

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 detect because the library code is generally not in the scope of pattern discovery. For instance, Figure 4 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 detection more difficult because these library classes are not part of system source code. This issue has also been recognized by Wirfs-Brock [36]. 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 [33] 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 pattern may have several variations and can be implemented differently. For example, Figure 4 displays a typical class diagram of the Composite pattern that may have several variations. One of such variation is shown in Figure 5, 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.

4. Conclusions

A number of pattern discovery techniques and tools have been proposed in the literature. These approaches use different intermediate representations, apply different analysis techniques, develop different algorithms, and provide different experiments on recovering different patterns. In this paper, we presented a comparative study on different aspects of these approaches. In addition, we discussed the results rendered by these approaches and analyzed the disparity of these results and the potential reasons.

Most of existing approaches on pattern discovery presented 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, patterns are typically described informally, which may cause ambiguity, imprecision, and misunderstanding. Due to these reasons, different approaches may present different results for discovering the same pattern from the same system. To the best of our knowledge, there is no benchmark system available so far, which may help to compare different approaches. We plan to work on benchmarking the pattern discovery techniques [14].

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.

[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] A. 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*, pp. 305-314, 2003.

[5] J. Bansiya, Automating design-pattern identification – DP++ is a tool for C++ programs. *Dr. Dobbs Journal*, 1998.

[6] L. Bass, P. Clements and R. Kazman, Software Architecture in Practice, Addison Wesley. 2003.

[7] 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*, 2003.

[8] A. Blewitt and A. Bundy, "Automatic verification of Java design patterns." Proceedings of International Conference on Automated Software Engineering, pp. 324-327, 2001.

[9] G. Booch, J. Rumbaugh, I. Jacobson. The Unified Modeling Language User Guide, Addison-Wesley, 1999.

[10] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad and M. Stal, Pattern-Oriented Software Architecture: A System of Patterns, Wiley. 1996.

[11] 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, pp. 102-111, 2005.

[12] G. Costagliola, A. De 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.

[13] 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.

[14] 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.

[15] R. Ferenc, A. Beszedes, L. Fulop, and J. Lele, Design pattern mining enhanced by machine learning. *Proceedings of the 21st International Conference on Software Maintenance*, 2005.

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

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

[18] M. Hericko and S. Beloglavec, "A composite design-pattern identification technique." *Informatica* 29, pp 469–476, 2005.

[19] D. Heuzeroth, T. Holl, G. Hogstrom, and W. Lowe, "Automatic design pattern detection." *Proceedings of the 11th International Workshop on Program Comprehension*, 2003.

[20] 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.

[21] 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)*, 2006.

[22] 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.

[23] 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)*, 1998.

[24] H. Mei, T. Xie, and F. Yang, "JBOORET: an automated tool to recover OO design and source models." In *Proceedings of the 25h Annual International Computer Software & Applications Conference (COMPSAC),* 2001.

[25] 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.

[26] 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.

[27] 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)*, 2004.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] P. Tonella, G. Antoniol, "Object oriented design pattern inference." In *Proceedings of International Conference on Software Maintenance (ICSM)*, 1999.

[33] 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.

[34] W. Wang and V. Tzerpos, "Design pattern detection in Eiffel systems." In *Proceedings of 12th Working Conference on Reverse Engineering (WCRE)*, 2005.

[35] L. Wendehals, Improving design pattern instance recognition by dynamic analysis. *Proceedings of the ICSE workshop on Dynamic Analysis*, pp. 29-32, 2003.

[36] R. J. Wirfs-Brock, "Refreshing Patterns," *IEEE Software*, vol.23, no.3, pp. 45-47, May/Jun, 2006.

[37] 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.

[38] Java.awt resource information, September 2006, http://java.sun.com/j2se/1.5.0/docs/guide/awt/index.html.

[39] JHotDraw. http://www.jhotdraw.org/