Design Pattern Detection by Template Matching

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

Yongtao Sun
American Airlines
4333 Amon Carter Blvd
Fort Worth, TX 76155, USA
Yongtao.Sun@aa.com

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

ABSTRACT

In this paper, we adopt a template matching method to detect design patterns from a software system by calculating their normalized cross correlation. Because design patterns document flexible design ideas, there can be various ways of implementing them. In our approach, not only the exact matches of pattern instances are detected from system source code, but also the variations of pattern candidates can be identified. Based on our method, we provide tool support and perform experiments on different large open-source systems.

Keywords

Design Pattern, Matrix, Template Matching, Cross Correlation

1. INTRODUCTION

Design pattern is a recurring solution to a standard software problem. Modern software industry has widely adopted design patterns to reuse the best practices and improve the quality of software systems. However, design document is often missing in many legacy systems. Even if the document is available, it may not exactly match the source code that may be changed and migrated over time. Due to the time-to-market pressure, some developers may not adequately document their software. Missing pattern-related information may compromise the benefits of using design patterns. Therefore detecting a design pattern from existing source code become a key issue for many research areas, such as reverse engineering and code refactoring, because it helps on program comprehension and design visualization.

Many approaches [12][4][7][9][1][10] have been proposed to detect design patterns from source code or a design model, such as the UML diagrams. Most existing approaches tend to search pattern instances by checking system source code or their extracted structural and behavioral models with individual features of each pattern. Machine learning algorithms, such as decision tree and neural network, have been applied to classify the potential pattern candidates in [7][9]. However, their approaches cannot be fully automated, because manual collection of certain features and human interaction are required. Pattern and system information has been encoded into matrixes in [12][4]. Thus, the matching of a design pattern in a system is conducted by matching a pattern matrix with a system matrix. In this paper, we propose an approach that adopts a template matching method from computer vision by calculating the normalized cross correlation between pattern matrix and system matrix. A normalized cross correlation shows the degree of similarity between a design pattern and the part of a system. Our approach also encodes the pattern and system information into matrixes. However, we use a new method to match the patterns.

The rest of this paper is organized as follows. Section 2 introduces the background and motivations of our approach. Section 3 details our pattern detection method via template matching. We conclude our work in Section 4.

2. Background and Motivations

Graph theory studies the properties and relationships of the vertices and edges in a graph. It has been widely applied to solve practical problems in network topology, traffic routing, and software structure analysis. It has also been applied in design pattern detection [12] to calculate the similarity between the classes (vertices) in different systems (graphs) using the similarity score and iterative algorithm proposed in [3]. Kleinberg [11] proposed link analysis method to find the main hub and source nodes for web pages. Blondel [3] generalized this idea to an iterative algorithm for computing the similarity score of two vertices. This similarity score algorithm for design pattern detection has been applied in [12] by first encoding the source code and design patterns into different feature matrixes. For example, the value of X_{ij} in the generalization matrix represents the inheritance relationship between classes *i* and *j* (1 means true and 0 means false). The generalization matrixes can be created to represent the generalization relationships between any pair of classes in both the source code and the design patterns to be matched. Besides generalization, other features are encoded in matrixes similarly. Second, a similarity matrix *S* is defined that the value of S_{ij} represents the similarity score between class *i* in a design pattern and class *j* in the source code. *S* can converge via Blondel's algorithm.

The limitation of this approach is that the algorithm can only calculate the similarity between two vertices, instead of two graphs. High similarity score of two vertices does not guarantee a match between two sets of vertices. Let us use an example to illustrate this limitation.

Figure 1 shows two graphs, A and B, and their corresponding graph matrixes, respectively. Using Blondel's algorithm, the similarity matrix can be calculated as shown in Figure 1, where the

SAC'08, March 16-20, 2008, Fortaleza, Ceará, Brazil.
Copyright 2008 ACM 978-1-59593-753-7/08/0003…$5.00.

higher the cell value is, the more similar the corresponding pair of vertices are. For example, $S_{X1} = 1$ represents that vertex X of Graph A is the most similar to vertex *1* in Graph B. Similarly, vertex Y of Graph A is the most similar to vertex *2* and *3* in Graph B. However, high similarity score of each pair of vertices does not guarantee a high degree of similarity of two graphs with multiple vertices and edges. As shown in Figure 1, the result of the similarity matrix S points out that the sub-graph 2->1 or 3->1 of Graph B matches Graph A (Y->X), due to their high similarity scores. However, such result has ruled out an exact matching sub-graph, 5->4, of Graph B, whose vertices have much lower similarity scores: 0.00097. The main reason for this problem is that the similarity score only represents the degree of similarity between a vertex in the source graph and a vertex in the target graph. It does not show the degree of similarity between a group of vertices in the source graph and those in the target graph. Even though a pair of vertices, one of which is in the source graph and the other is in the target graph, may have low similarity score, they may exactly match each other as shown in our example.

To solve this problem, we propose a new approach based on the template matching method in this paper. Our approach allows calculating the similarity between the sub-graphs of two graphs, instead of just pair of vertices.

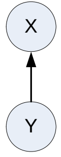
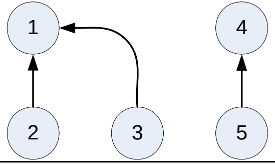
	Graph A	Graph B
Graph Structure		
Graph Matrix	$\begin{array}{ccc} & X & Y \\ X & 0 & 0 \\ Y & 1 & 0 \end{array}$	$\begin{array}{cccccc} & 1 & 2 & 3 & 4 & 5 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 & 0 & 0 \\ 3 & 1 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 1 & 0 \end{array}$
Similarity Matrix	$S = \begin{array}{cccccc} & 1 & 2 & 3 & 4 & 5 \\ X & 1 & 0 & 0 & 0.00097 & 0 \\ Y & 0 & 1 & 1 & 0 & 0.00097 \end{array}$	

Figure 1 Graph Similarity on Pattern Detection

3. Pattern detection by template matching

In this section, we first give a brief introduction to the template matching method [1]. We then present our approach that applies template matching method to design pattern detections. In addition, we implemented our method into a tool and conducted several experiments on different large open-source systems.

3.1. Template matching

Template matching method has been extensively applied in computer vision [1] 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, $x = 1, \ldots, n$ 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.

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

In other words, normalized cross correlation defines the $\cos\theta$ value, where θ is the angle between vector f and g. The maximum value is 1 when f and g is an exact match, i.e., $\theta = 0$. Figure 2 shows an example of template matching in computer vision. The values in Figure 2(a) and (b) are the pixel values of a template and a target image, respectively. For simplicity and better visualization, we omit the cell values in the matrixes in Figure 2(b), (c), and (d) when they are 0. Figure 2(c) shows the cross correlation values, where four possible matches are found with the maximum cross correlation value 3 highlighted. However, three of the four possible matches are not exact matches. The values of normalized cross correlation shows only one match with the maximum normalized cross correlation value 1 highlighted in Figure 2(d), which identifies the only exact match in this case.

3.2. Matrix representations

Each design pattern documents a unique set of design features that identify the pattern. Detecting a pattern from system source code or design document is to search for such unique set of design features of the pattern. For example, the Strategy pattern [8] involves at least three classes, one of which is an abstract class or interface as shown in Figure 3. There also exist the generalization and association relationships among those classes. These design features are encoded into matrixes in our approach. More specifically, we create an $n \times n$ matrix for the generalization relationship, where n is the number of classes. If the i^{th} and j^{th} classes have the generalization relationship, the corresponding cell of the matrix is set to 1. Otherwise, it is set to 0. For instance, the generalization matrix of the Strategy pattern is shown in Figure 3, where the ConcreteStategy class inherits from the Strategy class. Similarly, we create a matrix for each design feature. Figure 3 also shows the association and abstract matrixes.

Our approach currently encodes eight design features, such as generalization, association, abstract, and several different kinds of invocations, into matrixes.

In addition to the eight matrixes that encode the design features of a design pattern, we need to encode the design features of a software system into eight matrixes similarly. We can separately match the corresponding matrixes of a system by those of a design pattern, and then combine the results. As discussed in Section 2, however, the problem with this approach is that individual matches do not guarantee group matches, and vise versa.

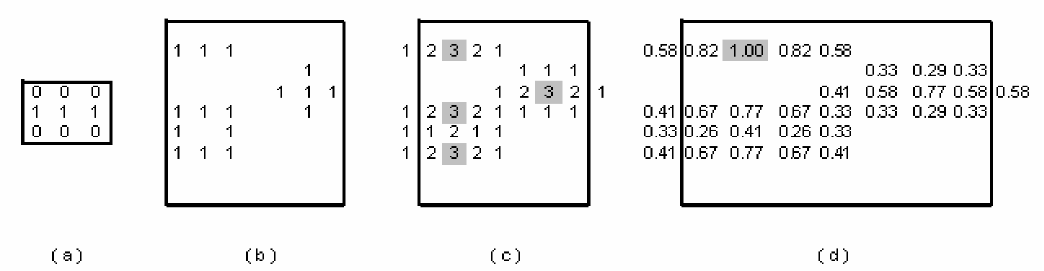


Figure 2 (a) Template graph (b) Target graph (c) Cross correlation (d) Normalized cross correlation

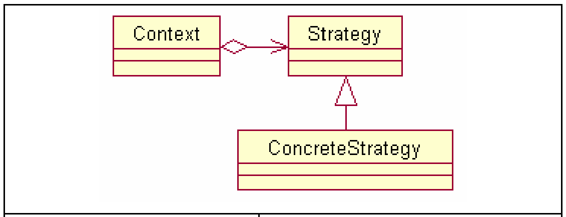
Therefore, we combine the eight matrixes into a single matrix for both the design pattern and the system. In this way, the detection of a design pattern can be done by a single match between two matrixes, instead of eight matches between sixteen matrixes. To combine the eight matrixes into one matrix, we give each matrix a root value of a different prime number. The cell value of each matrix is then changed to the value of its root to the power of the old cell value. Thus, the overall matrix is calculated with each cell holding the value of the product of the corresponding cell values in these eight new matrixes. For instance, we give the root values of the generalization, association, and abstract matrixes to be the prime numbers, 2, 3, and 5, respectively, as shown in Figure 3. Before we merge these three matrixes, our approach converts every cell value (x) of each matrix to a new value ($root^x$). The overall matrix is obtained by multiplying the corresponding cells of all converted matrixes. For example, there is one generalization relationship between classes 2 and 1 in Figure 3. There is no association between them. Class 2 is not abstract. Thus, the corresponding cell value of the overall matrix is $M_{21} = 2^1 3^0 5^0 = 2$. An example of overall matrix is shown in Figure 3.

3.3. Preprocessing for optimization

The overall matrix of a software system is an $n \times n$ matrix, where n is the number of classes in the system. The overall matrix of a design pattern is an $m \times m$ matrix, where m is the number of classes in the pattern. Our approach reduces design pattern detection problem into a matrix matching problem. In other words, our approach search the system overall matrix for an $m \times m$ sub-matrix that match the pattern matrix. There can be $n(n-1)(n-2)\ldots(n-m)$ possible matching sub-matrixes. Thus, the complexity is $O(n^m)$. To reduce the complexity of our method, we conduct the following preprocessing to optimize our approach before calculating the normalized cross correlations of the pattern matrix and the system sub-matrixes.

First, the classes participating in each design pattern are normally connected with other classes by some relationships. They typically are not independent, except for the Singleton pattern. For example, the classes of the Strategy pattern shown in Figure 3 are all connected. For this reason, our approach eliminates the system sub-matrixes that contain a row or column whose values are all ones, which means the corresponding class does not relate to others. This optimization applies to most of design patterns.

Second, each design pattern may include a limited number of design features that need to be matched with the system sub-matrixes that may include much more design features. Our approach filters out the unnecessary design features from the system sub-matrixes to reduce the template matching complexity. For example, there is only one generalization relationship between the Leaf and Component classes in the Composite pattern. Since the root value of the generalization matrix is 2, the $P_{Leaf, Component}$ cell of the overall pattern matrix is $2^1 = 2$. In addition to the generalization relationship, suppose there is also an association between the candidate Leaf and Component classes in the system sub-matrix whose cell $M_{Leaf, Component} = 2^1 \times 3^1 = 6$ (the root value of the association matrix is 3). In this case, the association relation in the system sub-matrix is not necessary to match the pattern matrix. Thus, our approach filters it out before calculating the normalized cross correlation.



Generalization

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \quad \text{root=2}$$

Association

$$\begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad \text{root=3}$$

Abstract

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad \text{root=5}$$

Class 0 : Context
Class 1 : Strategy
Class 2 : ConcreteStrategy

Overall Matrix

$$\begin{bmatrix} 2^0 3^0 5^0 & 2^0 3^0 5^0 & 2^0 3^0 5^0 \\ 2^0 3^1 5^0 & 2^0 3^0 5^1 & 2^0 3^0 5^0 \\ 2^0 3^0 5^0 & 2^1 3^0 5^0 & 2^0 3^0 5^0 \end{bmatrix}$$

$$\Downarrow$$

$$\begin{bmatrix} 1 & 1 & 1 \\ 3 & 5 & 1 \\ 1 & 2 & 1 \end{bmatrix}$$

Figure 3 Matrix Representation

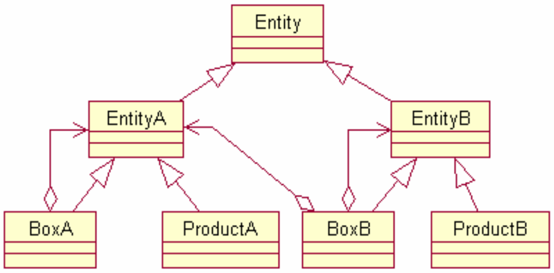
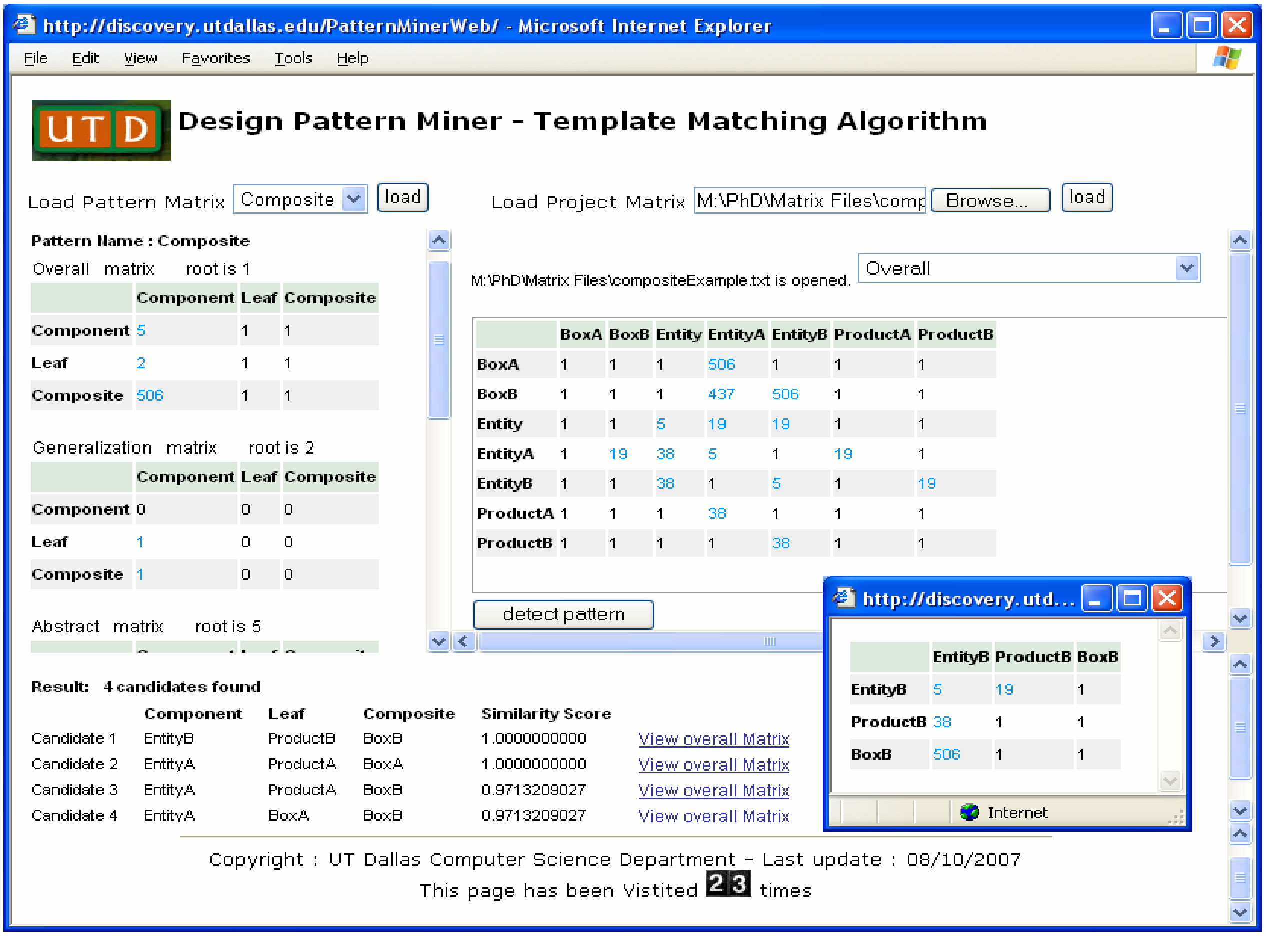


Figure 4 A Pattern Detection Example



Design Pattern Miner - Template Matching Algorithm

Pattern Name : Composite

Overall matrix root is 1

	Component	Leaf	Composite
Component	5	1	1
Leaf	2	1	1
Composite	506	1	1

Generalization matrix root is 2

	Component	Leaf	Composite
Component	0	0	0
Leaf	1	0	0
Composite	1	0	0

Abstract matrix root is 5

M:\PhD\Matrix Files\compositeExample.txt is opened.

	BoxA	BoxB	Entity	EntityA	EntityB	ProductA	ProductB
BoxA	1	1	1	506	1	1	1
BoxB	1	1	1	437	506	1	1
Entity	1	1	5	19	19	1	1
EntityA	1	19	38	5	1	19	1
EntityB	1	1	38	1	5	1	19
ProductA	1	1	1	38	1	1	1
ProductB	1	1	1	1	38	1	1

	EntityB	ProductB	BoxB
EntityB	5	19	1
ProductB	38	1	1
BoxB	506	1	1

Result: 4 candidates found

	Component	Leaf	Composite	Similarity Score	
Candidate 1	EntityB	ProductB	BoxB	1.0000000000	View overall Matrix
Candidate 2	EntityA	ProductA	BoxA	1.0000000000	View overall Matrix
Candidate 3	EntityA	ProductA	BoxB	0.9713209027	View overall Matrix
Candidate 4	EntityA	BoxA	BoxB	0.9713209027	View overall Matrix

Copyright : UT Dallas Computer Science Department - Last update : 08/10/2007
This page has been Vistited 23 times

Figure 5 Design Pattern Detection Tool Web Interface

3.4. Similarity score calculation

Our approach matches the pattern matrix with the system sub-matrixes defined in the previous section by calculating their normalized cross correlations. Since cross correlation is normally used for two vectors, our approach converts each matrix into a vector by appending the *i*th row to the end of *(i-1)*th row of the matrix. For example, the matrix in Figure 2(a) can be represented in a vector "000111000". Eventually, our approach calculates the $cos\theta$ value between each pair of vectors based on the formula shown in Section 3.1 and renders a single similarity matrix, instead of multiple ones in [12]. Our approach calculates the similarity between a sub-system and a pattern along all features of the pattern rather than just the similarity of two classes. In this way, our approach can solve the problem described in Section 2.

Let us consider an example adopted from [13] shown in Figure 4, which contains two instances of the Composite pattern: {EntityA, ProductA and BoxA} and {EntityB, ProductB and BoxB}. However, the approach presented in [12] cannot correctly identify these two instances of the Composite pattern. Instead, it identified {EntityA, BoxB} as a candidate of the Composite pattern. As discussed previously, the main reason is that the EntityA and BoxB classes are good candidates of Component and Composite, respectively. However, they do not form a valid instance of the Composite pattern. They actually belong to two different instances of the Composite pattern and are mixed together to form a Composite pattern instance by [12]. On the other hand, our approach is able to correctly detect the two instances of the Composite pattern as shown in the following section.

3.5. Prototype tool support

We develop a tool to implement our method for design pattern detections. Figure 5 shows the web interface of our tool. The users can select the design pattern to be detected and a software system on the top part of the user interface. For instance, consider the user selected the Composite pattern and the software system shown in Figure 4. All matrixes encoded the selected pattern (the Composite pattern in this case) are displayed in the left panel with the overall matrix on top. Similarly, all matrixes encoded the selected system are shown in the right panel, where the user can select the particular matrix to display.

With both the pattern and system loaded into our pattern detection tool, the detection results are shown in the bottom panel listed in descending order of their similarity scores, when the user click on the "detect pattern" button. In this case, two exact matches are detected with the maximum similarity score 1. They are the true positive results we mentioned previously. Additionally, two more candidates with high similarity score (0.97) are also listed in the result panel. They can be further checked whether they are variants of the desired pattern.

By clicking the “View Overall Matrix” link on the right of the select candidate, a window will popup displaying the overall matrix of the corresponding candidate instance. The small window in the bottom right corner of Figure 5 shows the overall matrix of the first candidate. Comparing it with the overall matrix of the Composite pattern in the left panel, we can see that the candidate is an exact match of the Composition pattern although it may contain more design features than the Composite pattern.

System	**JUnit**	**JHotDraw**	**JRefactory**	**Log4j**
Version	3.8.2	6.0	2.6.24	1.2
Class #	102	600	576	259
Composite	1	0	0	0
Adapter	5	51	27	23
State	3	29	12	8
Decorator	1	1	0	0

Table 1 Experiment results

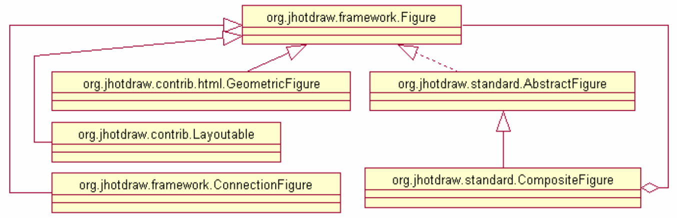

org.jhotdraw.framework.Figure
org.jhotdraw.contrib.html.GeometricFigure
org.jhotdraw.standard.AbstractFigure
org.jhotdraw.contrib.Layoutable
org.jhotdraw.framework.ConnectionFigure
org.jhotdraw.standard.CompositeFigure

Figure 6 Variant Pattern Instance in JHotDraw

3.6. Pattern detection results

In this section, we present several experiments on large open-source systems, JUnit [16], JHotDraw [14], JRefactory [15] and Log4j [17], to evaluate our approach and tool. Table 1 shows our detection results with the versions, sizes, and the numbers of detected patterns of each system.

Table 1 lists the number of exact match cases, which have the maximum similarity score 1. Sometimes a variant pattern instance may exist in the source code. In this situation, our tool can still identify the possible match by examining candidates with a high similarity score (but less than 1). Even though a variant may not exactly match a pattern, it should match the majority features of a pattern because of the small angle between the candidate and the pattern template based on the normalized cross correlation. Let us consider an example instance of the Composite pattern in JHotDraw shown in Figure 6. Since our tool currently does not support the transitive closure on relationships yet, it will reject this case as an exact match because there is no direct generalization relationship between the Figure and CompositeFigure classes, which is an important feature of the Composite pattern. However, this case matches all other features of the Composite pattern with a high similarity score of 0.999932. Thus, our tool can still identify this instance by lowering the similarity score threshold from 1 to 0.99 even though it is not an exact match.

4. CONCLUSIONS

In this paper, we apply the template matching method to solve the pattern detection problem. Our approach encodes both pattern and system knowledge into two overall matrixes and calculates their similarity score by cross correlation. Our approach can not only find the exact matches of pattern instances, but also identify their possible variants. The results show that our approach can avoid the false positives caused by individual matches. We also conducted several experiments on large open-source systems. In the future, we plan to apply machine learning methods to analyze inexact matching results. We will also further optimize our tool.

Although our approach encodes eight design features, we do not claim that these eight design features are complete. More or different design features can be introduced in our approach, which may render different pattern detection results. However, there will be minor effect on our method and approach by adding new feature matrixes. The design features of each design pattern are not currently defined in a standard as discussed in [5][6].

REFERENCES

[1] G. Antoniol, G. Casazza, M.D. Penta, R. Fiutem, Object-oriented design pattern recovery, Journal of Systems and Software, 59 (2), p. 181–196, 2001.

[2] D. H. Ballard and C. M. Brown, Computer Vision, Prentice Hall, 1982.

[3] 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.

[4] 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), USA, March 2007.

[5] J. Dong, Y. Zhao, Classification of Design Pattern Traits, Proceedings of the Nineteenth International Conference on Software Engineering and Knowledge Engineering (SEKE), Boston, USA, July 2007.

[6] J. Dong, Y. Zhao, and T. Peng, Architecture and Design Pattern Discovery Techniques – A Review, Proceedings of International Conference on Software Engineering Research and Practice (SERP), USA, June 2007.

[7] R. Ferenc, A. Beszedes, L. Fulop and J. Lele, Design Pattern Mining Enhanced by Machine Learning, 21st IEEE International Conference on Software Maintenance, 2005.

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

[9] Y.-G. Gueheneuc, H. Sahraoui, and F. Zaidi, Fingerprinting Design Patterns, Proc. 11th Working Conf. on Reverse Eng. (WCRE’04), Nov. 2004.

[10] H. Huang , S. Zhang , J. Cao , Y. Duan, A practical pattern recovery approach based on both structural and behavioral analysis, Journal of Systems and Software, 75(1-2), 2005.

[11] J.M. Kleinberg, Authoritative Sources in a Hyperlinked Environment, J. ACM, vol.46, no. 5, pp. 604-632, Sept. 1999

[12] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. Halkidis, Design Pattern Detection Using Similarity Scoring, IEEE transaction on software engineering, 32(11), 2006.

[13] Hudson Pattern Code Example http://home.earthlink.net/~huston2/dp/patterns.html

[14] JHotDraw, http://www.jhotdraw.org/, 2007.

[15] JRefactory, http://jrefactory.sourceforge.net/, 2007.

[16] JUnit, http://www.junit.org/, 2007

[17] Log4j, http://logging.apache.org/log4j/, 2007.