International Journal of Research Studies in Computer Science and Engineering (IJRSCSE)
Volume 1, Issue 1, May 2014, PP 42-51
www.arcjournals.org

Detection of Design Patterns Using Design Pattern Nearness Marking (DPNM) Algorithm

Shanker Rao A
M.Tech Student/Department of CSE
Aurora's Engineering College
Bongiri, nalgonda .India
shankerrao22@gmail.com

M.A. Jabber
Associate professor/Department of CSE
Aurora's Engineering College
Bongiri, nalgonda .India
jabbar.meerja@gmail.com

Mayank Sharma
HOD/Department of CSE
Aurora's Engineering College
Bongiri, nalgonda .India
mayanksharma@aurora.ac.in

Abstract: *Detection of design pattern methodology is based on match gain between graph vertices. This approach has the combined action power of detection normal, changed version patterns and nearness marking. This approach make use of the fact that patterns reside in one or additional inheritance hierarchies, reducing the dimensions of the graphs to that the algorithm is applied. Finally, the algorithm doesn't suppose any pattern-specific heuristic, facilitating the extension to novel style structures. Analysis on 3 open-source comes demonstrated the accuracy and therefore the potency of the proposed methodology.*

Keywords: *Design patterns, hits, Gof, DPNM, jHotDraw, jRefactory, jUni*

1. INTRODUCTION

Design patterns abstract reusable software design. Each pattern solves design problems that occur in every day software development. The detection of design patterns during the process of software reverse engineering can provide a better understanding of the software system.

Each design patterns contains set of categories with reference to one another in structural and behavioral. Detection of the design pattern from large commercial project and legacy system faces challenges like mostly poorly documented or missing of the document change of the pattern version detection, to overcome such challenges, completely different approaches are proposed to recover design patterns from software systems. However, there lack approaches based mostly on machine learning techniques. As a popular inductive machine learning algorithm, decision tree learning algorithm [1,2, 3] has been successfully applied to many applications, such as pattern matching, weather forecast and virus classification. Decision tree algorithm extracts useful rules and pattern information from the training examples that must be pre-classified by an expert (or a supervisor). It is normally easy to collect the training examples by recording the related attribute values as one single record. For example, whether a person will play tennis is decided by the outlook, temperature, humidity, and wind of a day. Thus, the outlook, temperature, humidity and wind attribute values can be collected together as one entry. Based on the classification of these collected entries, a decision tree prediction model can be constructed by applying the decision tree algorithms. However the learning problem of design pattern detection is more complicated which involves a group of classes with relationships. Each class corresponds to a record. The potential relationships among the group of classes (records) of a pattern can be a large number. Nevertheless, only one set of relationships is typically valid for the pattern. In other words, the training example for decision tree algorithm here is a combination of training records (classes) connected by some relationships/links, instead of a single record (class). This is a compound records learning problem involving multiple training records. It is not easy to classify all the potential combinations within a set of classes, especially when considering the roles of different relationship.

2. RELATED WORK

Detecting design patterns enhances program understanding as well as existing source code and documentation. There is a great deal of use in having this knowledge about a software system. The time developers need to understand a software system decreases if they have access to good documentation about the design of the application. There has been a lot of research in the area of design pattern detection over the last years. Many different approaches have been developed and some of them were implemented in tools and tested with software systems to show their effectiveness.

Detecting design patterns was written by Kraemer and Prechelt in 1996 it was the first paper. They introduced an approach detecting design information directly from C++ header files. This information is stored in a repository. The design patterns are expressed as PROLOG rules which are used to query the repository with the extracted information. Their work focused on detecting five structural design patterns: Adapter, Bridge, Composite, Decorator, and Proxy. The precision of their developed tool is 14 to 50 percent which they claim is acceptable for discovering design information. They suggest a more detailed approach that considers method call delegation during structural analysis which would lead to overall better results

Detecting design patterns in C++ code was introduced by Espinoza, Esqueer and Cansino. They formulate a canonical model to represent design patterns. They developed a system called Design Patterns Identification of C++ programs (DEPAIC++). This tool is composed by two modules that first transform the C++ to a canonical form and then recognize design patterns. It is implemented in Java. They use the structural relationships between classes to identify design patterns in the source code. They tested their work with different software systems that used design patterns and were able to detect them using DEPAIC++. Their future work includes extending this tool as well as detecting design patterns in Java source code.

In the field of automatic pattern detection Keller [1] describes a static analysis to discover design patterns (Template Method, Factory Method and Bridge) from C++ systems. The authors identify the necessity for human insight into the problem domain of the software at hand, at least for detecting the Bridge pattern due to the large number of false positives. The Pat system [2] detects structural design patterns by extracting design information from C++ header files and storing them as Prolog facts. Patterns are expressed as rules and searching is done by executing Prolog queries. Brown [4] uses dynamic information, analyzing the flow of messages. His approach is restricted to detecting design patterns in Smalltalk, since he only regards flows in Visual Works for Smalltalk. He therefore annotates the Smalltalk runtime environment. Another drawback is that he only gathers type information at periodic events. Carriere[3] also employ code instrumentation to extract dynamic information to analyze and transform architectures.

The presented approach only identifies communication primitives, but no complex protocols. The present paper extends our previous results [5] in two ways. Firstly, it implements more than the Observer Pattern analyzer and extends the experiments to unknown code. These extensions show that the results (and shortcomings in the dynamic analysis) can be generalized. Secondly, it sketches our approach to and first results of automatic generation of analyses

3. PROPOSED SYSTEM

3.1. Design Pattern Nearness Marking (DPNM) Algorithm

The DPNM algorithm is the core of the proposed design pattern detection methodology. Therefore, a brief outline of the underlying theory will be presented.

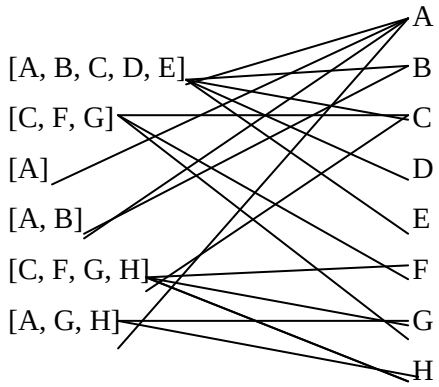
The proposed algorithm derived based on the link analysis algorithm called HITS [6]. In HITS algorithm, Hub and authority weights will be obtained. The authority score of vertex j of a graph G can be thought of as a nearness score between vertex j of G and vertex authority of the graph hub à authority and, nearness, the hub score of vertex j of G can be seen as nearness score between vertex j and vertex hub. Within the context of design pattern detection, the DPNM algorithm can be used for calculating the nearness between the vertices of the graph describing the pattern (G1) and the corresponding graph describing the system (G2). This will lead to a number of correspondent matrices of size n2 X n1. In order to obtain an overall picture for the

nearness between the pattern and the system, one has to exploit the information provided by all matrices. To preserve the validity of the results, any DPNM score must be bounded within the range. Therefore, individual matrices are initially summed and the resulting matrix is normalized by dividing the elements of column 'C' (corresponding to DPNM scores between all system classes and pattern role 'C') by the number of matrices (mC) in which the given role is involved. This is equivalent to applying an affine transformation in which the resulting matrix is multiplied by a square n2 X n1 diagonal matrix, where element (C C) is equal to 1=mC.

3.2. Ranking Transactions with Hits [14]

TID	Transaction
100	[A,B,C,D,E]
200	[C,F,G]
300	[A]
400	[A,B]
500	[C,F,G,H]
600	[A,G,H]

(a)



(b)

Fig 1: *Ranking Transaction with Hits*

A database of transactions can be depicted as a bipartite graph without loss of information. Let *D= {T1, T2, . . . , Tm}* be a list of transactions and *I={i1, i2, . . . , in}* be the corresponding set of items. Then, clearly D is equivalent to the bipartite graph G = (D, I, E) where

E = {(T,*i*) : *i* ∈T, T ∈ D, *i* ∈I}

Example: Consider the database shown in Fig. 1a. It can be equivalently represented as a bipartite graph, as shown in Fig. 1b. The graph representation of the transaction database is inspiring. It gives us the idea of applying link-based ranking models to the evaluation of transactions. In this bipartite graph, the support of an item i is proportional to its degree, which shows again that the classical support does not consider the difference between transactions. However, it is crucial to have different weights for different transactions in order to reflect their different importance. The evaluation of item sets should be derived from these weights. Here comes the question of how to acquire weights in a database with only binary attributes. Intuitively, a good transaction, which is highly weighted, should contain many good items; at the same time, a good item should be contained by many good transactions. The reinforcing relationship of transactions and items is just like the relationship between hubs and authorities in the HITS model. Regarding the transactions as "pure" hubs and the items as "pure" authorities, we can apply HITS to this bipartite graph. The following equations are used to perform each iteration*:*

$$Auth\ (i) = \sum_{T:i\in T} hub(T)\ ,\ hub(T) = \sum_{i:i\in T} auth(i)$$

When the HITS model eventually converges, the hub weights of all transactions are obtained. These weights represent the potential of transactions to contain high-value items. A transaction with few items may still be a good hub if all component items are top ranked. Conversely, a transaction with many ordinary items may have a low hub weight.

3.3. W-Support: A New Measurement

Item set evaluation by support in classical association rule mining is based on counting. In this section, we will introduce a link-based measure called w-support and formulate association rule mining in terms of this new concept. The previous section has demonstrated the application of the HITS algorithm to the ranking of the transactions. As the iteration converges, the authority weight *auth* (*i*) = $\sum_{T:i\in T} hub(T)$ represents the "significance" of an item *i*. Accordingly, we

generalize the formula of *auth* (*i*) to depict the significance of an arbitrary item set, as the following definition shows:

The w-support of an item set X is defined as

$$Wsupp(X) = \frac{\sum_{T:X \subset T \wedge T \in D} hub(T)}{\sum_{T:T \in D} hub(T)}$$

Where *hub (T)* is the hub weight of transaction T. An item set is said to be significant if its w-support is larger than a user specified value.

4. REPRESENTATION OF SYSTEM AND PATTERNS

Prior to the pattern detection process, it is necessary to define a representation of the structure of both the system under study and the design patterns to be detected. Such a representation should incorporate all information that is vital to the identification of patterns. We have opted for modeling the relationships between classes (as well as other static information) in an object-oriented design using matrices. The key idea is that the class diagram is essentially a directed graph that can be perfectly mapped into a square matrix. The main two advantages of this approach are 1) that matrices can be easily manipulated and 2) that this kind of representation is intuitively appealing to engineers and computer scientists. The relationships or attributes of the system entities to be represented depend on the specific characteristics of the patterns that the designer wishes to detect. The information that we have chosen to represent includes associations, generalizations, abstract classes, object creations, abstract method invocations, etc. However, the nearness algorithm does not depend on the specific types of matrices that are used. The designer can freely set as input any kind of information, provided that he/she can describe the system and the pattern as matrices in terms of this information. Concerning the Similar Abstract Method Invocation Graph, each edge represents the invocation from a method's body (in the starting node) of a similar abstract method (in the ending node). Two methods are considered similar if they have the same signature. For example, the edge between the Decorator and Component nodes implies that a method in the Decorator class invokes a similar abstract method in the Component class through reference. Moreover, similar method invocations can also occur when explicitly stating the base class method (e.g., via the super identifier in Java), as in the case of classes Concrete-Decorator and Decorator.

5. METHODOLOGY

One issue that requires careful treatment is that the convergence of the nearness algorithm depends on the system graph size. As a result, the time needed for the calculation of nearness scores between all the vertices of the system and the pattern can be prohibitive for large systems. In order to make the approach more efficient, one must find ways to reduce the size of the graphs to which the algorithm is applied without losing any structural information that is vital to the design pattern detection process. By taking the advantage of the fact that most design patterns involve class hierarchies (since they usually include at least one abstract class/interface in one of their roles), a solution would be to locate communicating class hierarchies and apply the nearness algorithm to the classes belonging to those hierarchies. The overall methodology for the detection of implemented design patterns in an existing system can be outlined as follows:

a. Reverse engineering of the system under study. Each characteristic of the system under study (i.e., association, generalization, similar method invocation, etc.) is represented as a separate n X n adjacency matrix, where n is the number of classes.

b. Detection of inheritance hierarchies. All kinds of generalization relationships are considered for building the inheritance trees (i.e., concrete or abstract class inheritance, interface implementation). Since hierarchies are represented as trees, multiple inheritances cannot be modeled as a single tree because a node cannot have more than one parent. Therefore, each node that has multiple parents participates (including all its descendants) in a number of trees equal to the number of its direct ancestors. This is diagrammatically shown in classes C, C1, and C2 are considered as classes belonging to both hierarchies. Classes that do not participate in any

hierarchy are listed together in a separate group of classes since, in a number of design patterns, some roles might be taken

by classes that do not belong to any inheritance hierarchy (e.g., Context role in the State/Strategy pattern).

c. Construction of subsystem matrices. A subsystem is defined as a portion of the entire system consisting of classes belonging to one or more hierarchies. As already mentioned, the role of the subsystems in the pattern detection methodology is to improve the efficiency. Experimental results have shown that the cumulative time required for the convergence of the nearness algorithm applied on all subsystems is less than the time required for the entire system. The set of matrices that represent a subsystem is constructed by preserving from the matrices of the entire system the information concerning only the classes of the corresponding hierarchies. According to the number of hierarchies in the pattern to be detected, one of the following two approaches is taken. In a case where the pattern contains only one hierarchy (e.g., Composite, Decorator), each hierarchy in the system forms a separate subsystem. Thus, the number of subsystems is equal to the number of hierarchies in the system. In a case where the pattern contains more than one hierarchy (the design patterns that we have studied contain at most two hierarchies, e.g. State, Visitor), subsystems are formed by combining all system hierarchies, taken two at a time. Thus, the number of subsystems is equal to m (m-1)/2, where m is the number of hierarchies in the system. Next, the number of exchanged messages between the hierarchies of each pair is calculated, and the pairs in which the hierarchies are not communicating are filtered out. Since the system is partitioned based on hierarchies, pattern instances involving characteristics that extend beyond the subsystem boundaries (such as chains of delegations) cannot be detected.

d. Application of nearness algorithm between the subsystem matrices and the pattern matrices. Normalized nearness scores between each pattern role and each subsystem class are calculated. This corresponds to seeking patterns in each subsystem separately.

e. Extraction of patterns in each subsystem. Usually, one instance of each pattern is present in each subsystem (i.e., one or two hierarchies), which means that each pattern role is associated with one class. There are two cases in which more than one pattern instance exists within a subsystem:

i. One pattern role is associated with one class while other pattern roles are associated with multiple classes. Such a case is depicted where Strategy role is associated with interface Strategy while Context role is associated with classes Context1 and Context2. In this case the nearness algorithm assigns a score of "1" to the interface Strategy and classes Context1, Context2. The two instances of the Strategy pattern are correctly identified as (Strategy, Context1) and (Strategy, Context2) by combining the classes corresponding to discrete roles.

ii. All pattern roles are associated with more than one class. Since design patterns involve abstractions, in order for this to happen, multiple levels of abstract classes/interfaces must exist in the same hierarchy. The application of the nearness algorithm in the subsystem of would assign a score of "1" to classes Context1, Context2 as well as interfaces Strategy1 and Strategy2. It becomes obvious that the problem now is how to decide (based only on scores), which classes to pair in order to identify all pattern instances. Since there are four possible combinations, the methodology would end up in two true positives (Context1- Strategy1, Context2-Strategy2) and two false positives (Context1-Strategy2, Context2-Strategy1). It should be mentioned that such a case has not been encountered in the systems that we have examined. Therefore, the extraction of pattern instances is performed as follows: The nearness scores for each subsystem are sorted in descending order. For each pattern

role, a list is created. The subsystem classes having scores that are equal to the highest score for each role are added to the corresponding list. The detected pattern instances are extracted by combining the entries of the lists. The selection of the highest score for each role is based on the observation that a class assigned a score that is less than the score of another class (for a given role) definitely satisfies fewer criteria according to the sought pattern description. As a result, the class with the lower score is a worse candidate for the specific pattern role. An exception would be a class satisfying the same set of criteria, but with a lower score due to modification. This rare

case that would result in a false negative has not occurred in the systems that we have examined. According to the nearness algorithm, exact matching for a given pattern role results in scores which are equal to "1."

However, as already explained, modified pattern roles result in scores which are less than "1." The consideration of such "not absolute" scores would pose difficulties in distinguishing true from false positives. Consequently, a threshold value is required. Values below or equal to that threshold would signify that the sought pattern role is likely not to be present. The proposed approach is based on the assumption that no more than one pattern characteristic is modified for a given instance. According to this assumption,

the threshold value for a pattern role involving x characteristics must guarantee the presence of x-1 unmodified characteristics and the presence of the other one either as modified or unmodified. A threshold value of (x- 1)/x ensure that for a pattern role with x characteristics, (x- 1) are not modified. Moreover, the range ((x-1)/ x, 1) is covered by nearness values for pattern roles with one modified characteristic. The larger extend of the modification (e.g., the number of intermediate inheritance levels) the closer the nearness value gets to (x-1)/ x. Consequently, the threshold value of (x-1)/x guarantees the detection of a pattern role with (x-1) unmodified characteristics and one modified, regardless of the extent of the modification. In the steps that have been described above, the following optimizations have been applied in order to improve the efficiency of the pattern detection process:

a) Minimization of number of roles for each pattern. As already mentioned, the description of each pattern consists of a number of matrices, each one describing a different attribute. Some of these attributes are quite common in a system while others are less common. These uncommon characteristics are the ones that distinguish a pattern from other structures. Therefore, for the description of a pattern, the roles with the most

unique characteristics should be preferred. For example, roles participating only in the generalization matrix (e.g., concrete children inheriting their abstract patterns) should be excluded. Their inclusion to the pattern description would lead to numerous false positives, since there are many classes in a subsystem that simply inherit another class without being part of any pattern instance. In the results that will be presented in the next section, only the roles that are important for each pattern have been considered. However, the excluded roles can easily be found after the pattern detection process since they are closely related to the detected pattern roles. An alternative handling would be to assign weights to each matrix according to the importance of the corresponding attribute. However, assuming that all roles are sought, roles corresponding to common characteristics will eventually obtain very low nearness scores, hindering the detection of those roles.

b) Exclusion of irrelevant subsystems. In a case where one of the required attributes is not present at all in a subsystem (i.e., the corresponding matrix is a zero matrix), the pattern detection process is terminated for the specific subsystem.

6. IMPLEMENTATION

A tool has been implemented in Java that encompasses all steps of the proposed methodology. The program employs a Java byte code manipulation framework [13], which enables the detailed analysis of the system's static structure. The information retrieved is

a. Abstraction (whether a class is concrete, abstract, or interface)

b. Inheritance (parent class, implemented interfaces)

c. Class attributes (type, visibility, and static members)

d. Constructor signatures (parameter types)

e. Method signatures (method name, return type, parameter types, abstract or not)

f. Method invocations (origin class and signature) and

g. Object instantiations.

The above information is used to extract more advanced properties such as

a. Collection element type detection (type of elements contained in a collection) and identification of iterative method invocation on the elements of a collection used for detecting Observer and Composite)

b. Similar abstract method invocation (invocation of an abstract method within a method having the same signature used for detecting Decorator and Composite)

c. Abstract method adaptation (invocation of another class' method in the implementation of an inherited abstract method used for detecting Adapter/Command),

d. Template method (invocation of an abstract classes method in a method of the same class),

e. Factory method (instantiation of an object in the implementation of an inherited abstract method),

f. Static self-reference (private static attribute having as type the class that it belongs to use for detecting Singleton), and

g. Double or dual dispatch (used for detecting Visitor).

The extracted information is used to generate the matrices that describe the system under study. In the current implementation, pattern descriptions are hard-coded within the program. However, the information required for describing a design pattern (role names, adjacency matrices for the attributes of interest, and the number of hierarchies that the pattern involves) could be easily provided as external input. Once the system has been analyzed, the user can select a design pattern to be detected from the graphical user interface. Next, the similarity algorithm is applied as described in the section on methodology and the detected patterns are presented to the user without further human intervention.

7. RESULT

To evaluate the effectiveness of any pattern detection methodology, one should interpret the results by counting the number of correctly detected patterns (True Positives -TP), False Positives (FP), and False Negatives (FN). False positives are considered identified pattern instances which do not comply with the pattern description that has been specified. On the other hand, false negatives are actual pattern instances (according to the documentation or an inspector) that are not being detected by the applied methodology [10]. The sum of true positives and false negatives is equal to the total number of actual pattern instances in the system.

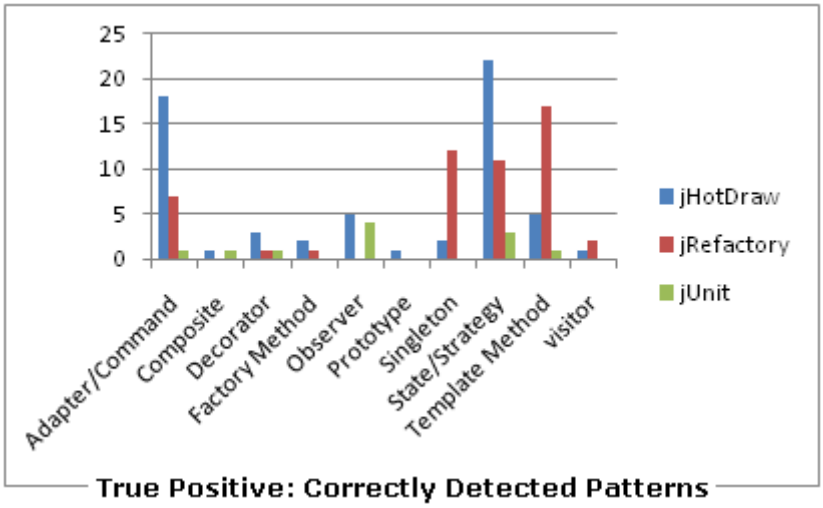
The results of the pattern detection process for the three systems are summarized in Fig 1. The recall values (sensitivity), defined as TP/ (TP+ FN), and is also given. Results are given for GoF patterns [11] that, according to the internal documentation and the relevant literature, exist in these three projects. Concerning Observer and Visitor, whose representation in the catalog by Gamma [11] includes sequence diagrams (referring to dynamic information) their static description is strong enough to allow the identification of these patterns.

The classification of the results has been performed by manually inspecting the source code and referring to the internal and external documentation of the projects. The precision (TP/ (TP + FP)) for all the examined patterns is 100 percent since there are no false positives. That is mainly because the pattern descriptions focused on the essential information of each pattern (by eliminating roles with common characteristics as explained in Section 4). False negatives occurred only in two patterns. In the Factory Method pattern (JHotDraw and JRefactory) the internal documentation mentions cases where a class method is considered a factory method only because it returns a reference to a created object. However, according to the literature, the pattern description includes the requirement that an abstract method with the same signature exists in one of the super classes. In the State pattern (JHotDraw and JRefactory), a State hierarchy actually exists; however, there is no Context class with a persistent reference to it (the reference is declared as a local variable within the scope of a method). The usual pattern description of State foresees the existence of a Context class with an association for holding the current state. As can be observed from the Table (1, 2 & 3) and Fig (1,2 & 3), the results for patterns Object Adapter/Command and State/Strategy have been grouped. That is because the structure of the

corresponding patterns is identical, prohibiting their distinction by an automatic process (e.g.,without referring to conceptual information).

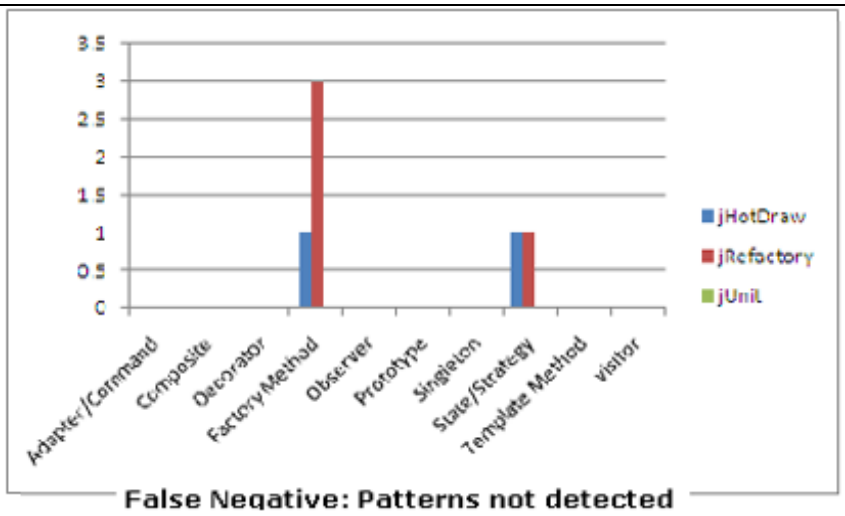
Design Pattern	jHotDraw	jRefactory	jUnit
Adapter/Command	18	7	1
Composite	1	0	1
Decorator	3	1	1
Factory Method	2	1	0
Observer	5	0	4
Prototype	1	0	0
Singleton	2	12	0
State/Strategy	22	11	3
Template Method	5	17	1
visitor	1	2	0

True Positive: Correctly detected Patterns



Design Pattern	jHotDraw	jRefactory	jUnit
Adapter/Command	0	0	0
Composite	0	0	0
Decorator	0	0	0
Factory Method	1	3	0
Observer	0	0	0
Prototype	0	0	0
Singleton	0	0	0
State/Strategy	1	1	0
Template Method	0	0	0
visitor	0	0	0

False Negative: Patterns not detected



Design Pattern	jHotDraw	jRefactory	jUnit
Adapter/Command	100%	100%	100%
Composite	100%	100%	100%
Decorator	100%	100%	100%
Factory Method	66.70%	25%	100%
Observer	100%	100%	100%
Prototype	100%	100%	100%
Singleton	100%	100%	100%
State/Strategy	95.60%	91.60%	100%
Template Method	100%	100%	100%
visitor	100%	100%	100%

Recall Value: TP/(TP+FN)

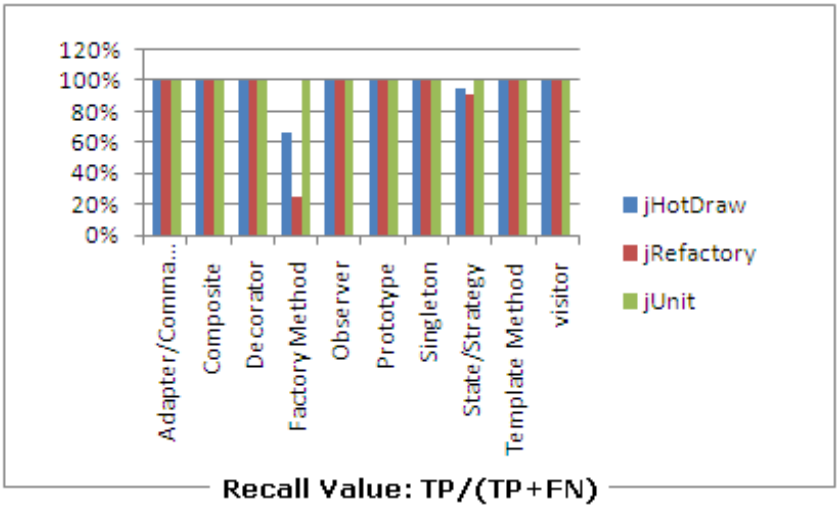


Fig 2: *Results extracted from jHotDraw, jReactory, jUnit*

8. CONCLUSION

The detection of design patterns in software systems is a crucial task within the re-engineering method, exploiting solely UML diagrams and designers' expertise, is extremely troublesome within the absence of automated help tools. The proposed methodology totally automates the pattern detection method by extracting the particular instances in a very system for the patterns that the user is curious about. The most contribution of the approach is that the use of a similarity algorithm, that has the inherent advantage of additionally detecting patterns that seem in a very

kind that deviates from their customary illustration. The appliance of the proposed methodology in 3 open-source systems demonstrated the accuracy and precision of the approach. Few of the targeted patterns were missed (false negatives), with no false positives.

REFERENCES

[1] Keller, R. K., R. Schauer, S. Robitaille, and P. Page (1999),"Pattern-Based Reverse-Engineering of Design Components", In Proc. ISCE, pp. 226-235.

[2] Prechelt, L. and C. Krämer (1998), "Functionality versus Practicality: Employing Existing Tools for Recovering Structural Design Patterns", J.UCS: 4, 12, 866ff.

[3] Carriere, S. J., S. G. Woods, and R. Kazman (1999),"Software Architectural Transformation", In Proc. 6th WCRE.

[4] Brown, K. (1997), "Design Reverse-Engineering and Automated Design Pattern Detection in Smalltalk", Master Thesis, Univserity of Illinois at Urbana-Champaign.

[5] Heuzeroth, D., T. Holl, and W. Löwe (2002), "Combining Static and Dynamic Analyses to Detect Interaction Patterns", In Proc. 6th Int. Conf. IDPT.

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

[7] J.R. Ullman, "An Algorithm for Subgraph Isomorphism," J.ACM, vol. 23, no. 1, pp. 31-42, Jan. 1976.

[8] E. Bengoetxea, "Inexact Graph Matching Using Estimation of Distribution Algorithms," PhD thesis, Ecole Nationale Supe´rieure des Te´le´communications, France, Dec. 2002.

[9] B.T. Messmer and H. Bunke, "Efficient Subgraph Isomorphism Detection: A Decomposition Approach," IEEE Trans. Knowledge and Data Eng., vol. 12, no. 2, pp. 307-323, Mar./Apr. 2000.

[10] M. Vokac, "Defect Frequency and Design Patterns: An Empirical Study of Industrial Code," IEEE Trans. Software Eng., vol. 30, no. 12, pp. 904-917, Dec. 2004.

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

[12] D.J. Cook and L.B. Holder, "Substructure Discovery Using Minimum Description Length and Background Knowledge," J. Artificial Intelligence Research, vol. 1, pp. 231-255, Feb.1994.

[13] ASM Home Page, http://asm.objectweb.org/, 2006. [14]. Pettersson, N.; LoXwe, W.; Nivre, J "Evaluation of Accuracy in Design Pattern Occurrence Detection," Software Engineering, IEEE Transactions on , vol.36, no.4, pp.575-590, July-Aug. 2010 doi: 10.1109/TSE.2009.92

[14] Ke Sun and Fengshan Bai " Mining Weighted Association Rules without Preassigned Weights"