JADEPT: Dynamic Analysis for Behavioral Design Pattern Detection

Francesca Arcelli, Fabrizio Perin, Claudia Raibulet and Stefano Ravani

Università degli Studi di Milano-Bicocca
DISCo – Dipartimento di Informatica Sistemistica e Comunicazione
Viale Sarca, 336, Ed. U14, 20126, Milan, Italy

Abstract. In the context of reverse engineering, the recognition of design patterns provides additional information related to the rationale behind the design. This paper presents our approach to the recognition of design patterns based on dynamic analysis of Java software. The idea behind our approach is to identify a set of rules capturing information necessary to identify a design pattern instance. Rules are characterized by weights indicating their importance in the detection of a specific design pattern. The core behavior of each design pattern may be described through a subset of these rules forming a macrorule. Macrorules define the main traits of a pattern. JADEPT (JAva DEsign Pattern deTector) is our software for design pattern identification based on this idea. It captures static and dynamic aspects through a dynamic analysis of the software by exploiting the JPDA (Java Platform Debugger Architecture). The extracted information is stored in a database. Queries to the database implement the rules defined to recognize design patterns. The tool has been validated with positive results on different academic implementations of design patterns and on systems as JADEPT itself.

1 Introduction

The use of design patterns in the context of forward engineering is a powerful mean to create effective software solutions. They guarantee the creation of transparent structures which allow software to be easily understood and extended. The description of design patterns [7] provides information about the structure, the participants' roles, the interaction between participants and, above all, the intent for which they should be used. Information related to the presence of a pattern is useful to understand not only the code, but to realize also the concepts behind its design. This has a significant implication for further improvement or adaptive changes. Implicitly, it leads to an enhancement of the life cycle with low maintenance costs.

In the context of design pattern detection, it is possible to use different approaches both for the identification logic (e.g., searching for subcomponents of design patterns, identifying the entire structure of a design pattern at once) and for the information extraction method (e.g., static, dynamic, or both). Static analysis consists in the analysis of static data gathered directly from source code or, if possible, from compiled code [4]. Dynamic analysis deals with data obtained during the execution of a system, gathered by means of third party applications as debuggers or monitoring interfaces.

Arcelli F., Perin F., Raibulet C. and Ravani S. (2009).
JADEPT: Dynamic Analysis for Behavioral Design Pattern Detection.
In *Proceedings of the 4th International Conference on Evaluation of Novel Approaches to Software Engineering - Evaluation of Novel Approaches to Software Engineering*, pages 95-106
DOI: 10.5220/0001951800950106
Copyright © SciTePress

One of the advantages of using static analysis is the complete coverage of the software under examination. This is not always achieved through dynamic analysis. Exploiting static analysis, it is not possible to determine properly the behavior of a system. One of the advantages of using dynamic analysis is the capability to monitor each of the functionalities of a software independently of the others. In this way it is possible to consider a smaller part of code to increase precision and limit false positive or false negative results.

Problems raised by the identification of design patterns are related not only to the search aspect, but also to design and development choices. There are at least three important decisions that should be taken when developing a design pattern detection tool. These decisions may influence significantly the final results. The first issue regards the evaluation of how to extract the interesting data from the examined software, including the type of the analysis to be performed. The second issue considers the data structure in which to store the gathered information: it may not model in a proper way the aspects of the software under investigation. The most important risk is related to the loss of knowledge at the data or the semantic level: this would generate inferences about something that is no more the analyzed software, but an incorrect abstraction of it. The third one highlights the importance to find a way to process the extracted data and to identify design pattern instances. Independently of the adopted data structure for the extracted information (e.g., a text file, XML, database), the following three issues should be considered: memory occupation, processing rate and, most important, the effective recognition process of design patterns with a minimum rate of false positives and false negatives. While the first two issues could be solved through an upgrade of the machine on which elaboration is performed, the last is strictly related to the efficiency of the recognition logic applied for design pattern detection due to the significant number of possible implementation variants.

In this paper, we present a new approach based on dynamic analysis to detect behavioral design patterns. We define a set of rules describing the properties of each design pattern. Properties may be either structural or behavioral and may define relationships between classes or families of classes. We define a family of classes as a group of classes implementing the same interface or extending a common class. Weights have been associated to rules indicating how much a rule is able to describe a specific property of a given design pattern. Rules have been written after we have deeply studied the books on design patterns of [7] and [5], evaluated different implementations of patterns, and implemented ourselves various variants of patterns.

JADEPT (JAva Design Pattern deTector) is our software prototype for design pattern detection based on these rules. An early version of JADEPT has been presented in the context of the PCODA workshop [2]. JADEPT collects structural and behavioral information through dynamic analysis of Java software by exploiting JPDA (Java Platform Debugger Architecture). Nevertheless part of the extracted information can be obtained by a static analysis of the software, JADEPT extracts all the information during the execution of the software adopting an approach based exclusively on dynamic analysis. The extracted information is stored in a database. The advantages of having information stored in database are: (1) the possibility to perform statistics and (2) the possibility to memorize information about various executions of the same software. A rule may be implemented by one or more queries. The presence of a design pattern is verified through the validation of its associated rules.

There are various approaches that aim to detect design patterns based on a static analysis of source code such as: FUJABA RE [11], [12], SPQR [18], [19], PINOT [17], PTIDEJ [9] or MAISA [21]. The design pattern detection mechanisms search for information defining an entire pattern or for sub-elements of patterns which can be combined to build patterns [1] or evaluate the similarity of the code structure with higher-level models as UML diagrams [3] or graph representations [20]. Other approaches exploit both static and dynamic analysis as in [10], [15] or only dynamic analysis as in [16], [22]. There are also several tools performing dynamic analysis of Java applications [6], [8], [23], but their main objective is not to identify design patterns. For example, Caffeine is a dynamic analyzer for Java code which may be used also to support design patterns detection.

The rest of the paper is organized as follows. Section 2 presents the identification rules and an example of their application on a behavioral design pattern, Section 3 introduces the JADEPT prototype, Section 4 describes several aspects concerning the validation of JADEPT. Conclusions and further work are dealt within Section 5.

2 Rules for Design Pattern Detection

Our idea is to develop a new approach for design pattern detection exploiting dynamic analysis. This kind of analysis allows the monitoring of the Java software at runtime, thus it is strictly related to the behavior of the system under analysis. A set of rules capturing the dynamic properties of design patterns and the interactions among classes and/or interface of design patterns are necessary for the detection of patterns through dynamic analysis.

Our identification rules consider those static aspects which provide information further exploited in dynamic rules. For example, to check the existence of a particular behavior, it is necessary to verify in the software under analysis the presence of a method having a specific signature.

2.1 Rules, Weights, Relationships and Macrorules

We consider behavioral design patterns because they are particularly appropriate for dynamic analysis. In fact, their traces may be better revealed at runtime by analyzing all the dynamic aspects including: object instantiation, accessed/modified fields, method calls flows.

In the first step of our work, the identification rules have been written using natural language. This approach avoids introducing constraints regarding the implementation of rules. In JADEPT, rules are translated into queries, but they can be used also outside the context of our tool and hence, represented through a different paradigm (e.g., graphs).

At the second step weights have been added to the rules. Weights denote the importance of a rule in the detection process of a pattern. Weights' range is 1 to 5. These values are used to compute the probability score indicating the probability of the presence of a pattern instance. A low weight value denotes a rule that describes a generic characteristic of a pattern like the existence of a reference or a method with a specific signature. A high weight value denotes a rule that describes a specific characteristic of a pattern like a particular method call chain that links two class families.

Even if each behavioral design pattern has its own particular properties, an absolute scale for the weights value has been defined. Rules whose weight value is equal to 1 or 2 describe structural and generic aspects of code (e.g., abstract class inheritance, interface implementation or the presence of particular class fields). Rules whose weight value is equal to 3 or higher, describe a specific static or dynamic property of a pattern. For example, the fifth rule of Chain of Responsibility in Table 1, specifics that each call to the *handle()* method has always the same caller-callee objects pair. This is the way objects are linked in the chain. A weight whose value is equal to 5 describes a native implementation of the design pattern we are considering. The weights of rules are used to determine the probability of the pattern presence in the examined code.

The next step regarded the definition of the relationship between rules [14]. There are two types of relationships. The first one is *logical*: if the check of a rule does not have a positive value, it does not make sense to proof the rules related to it. For example, the fifth rule of Chain of Responsibility in Table 1 cannot be proved if the fourth rule has not been proved first. The second one is *informative*: if a rule depends on another one, and the latter is verified by the software detector, its weight increases. The second type of relationship determines those rules which are stronger for the identification of design patterns.

Table 1. Detection rules for the Chain of Responsibility design pattern.

Nr.	Rule	Weight Specificity	Type	Dependencies
1	Some classes implement the same interface.	1	S	
2	Same classes extend the same class.	1	S	
3	All classes that implement the same interface or extend the same class, contain a reference whose type is the same of the implemented interface or the extended class.	3	S	
4	Each class has one method that contains a call to the same method in another class of the same family and this method must contain a parameter.	3	S-D	
5	"*handle*" is defines as the name of the method identified by the forth rule. The call to *handle* method of one object is always originated by the same caller object. This property is true for each object of the family.	3	D	
6	The name of an implemented interface or extended class contains the *chain* or *handler* word.	3?	S-D	if (4 and 5) = +1

Finally we have introduced macrorules. A macrorule is a set of rules which describes a specific behavior of a pattern. If the rules that compose a macrorule are verified, the core behavior of a pattern has been detected so the final probability value increases. The value added to the probability is different for each pattern because the number of rules which belong to a macrorule varies from one macrorule to another.

2.2 Chain of Responsibility Detection Rules

The rules we have defined for the Chain of Responsibility pattern are shown in Table 1. In the first column we assign a unique identifier to the rule in the context of a specific design pattern. The second column contains a textual description of each rule. In the third column are indicated the weights associated to each rule. A question mark after a weight value indicates a variable weight. For example, the sixth rule has a variable weight because of its relation with rule number four and number five. If the fourth or the fifth rule or both are verified then the weight of the sixth rule is increased by one, hence associating a higher probability to the pattern instance recognition.

The fourth column indicates the type of information needed to verify a rule. If a rule describes a static property, which can be verified through an analysis of static information, then the value in this column is S (indicating *static*). If a rule describes a dynamic property, which can be verified through an analysis of dynamic information, then the value in this column is D (indicating *dynamic*). In the case we have to verify a property by performing analysis of static and dynamic information, then the value specified is S-D (indicating static and dynamic). However, in JADEPT both static and dynamic information are extracted through a dynamic analysis of the software under inspection.

A relationship among two or more rules is indicated in the fifth column.

Table 2 shows the name of the macrorule, called *sequential redirection* defined for the Chain of Responsibility pattern and its consisting rules. For this pattern it is important to identify clues which capture its chain structure and behavior. Rules from 1 to 4 define the static properties related to its structure, while rules 5 and 6 the dynamic ones related to its behavior.

Rules 1 and 2 require that chain classes must implement a common interface or extend a common class.

For the Chain of Responsibility pattern the common class/interface must declare a method for sending a request to its successor in the chain. In the following we call this method *handle()*.

Rule 3 claims for the presence in each chain class of a field whose type is the implemented interface or the extended class. At runtime, this reference indicates the successor of an instance in the chain, and it is used to call the *handle()* method. This reference is assigned to the successor during execution and it is used to call the *handle()* method.

Rule 4 is verified if the interface or considered class declares a method which can be a *handle()* method. Rules 3 and 4 are preconditions for the fifth rule. These rules define the Chain of Responsibility peculiar behavior.

Rule 5 specifies that each object in the chain must always be called by the same object, which is its predecessor if objects are unchanged during execution. In fact, each time a request management is needed, if the chain elements have not been modified, the chain is preserved

Eventually, rule 6 checks if the name of the common interface or common class contains the *chain* or *handle* string.

A logical dependency is between rule four and five. Rule five cannot be proved if rule four is not previously verified.

Table 2. The Macrorule for the Chain of Responsibility Pattern.

Macrorule	Rules
Sequential redirection	4, 5, 6

The informative dependency we have defined for this pattern involves the 4th, 5th and 6th rules. Rule 6 can increment by one its weight if rules 4 and 5 are verified.

The macrorule for this pattern is called *sequential redirection* (see Table 2). It includes the rules that describe the core of the Chain of Responsibly pattern. If rules 4 and 5 are checked the macrorule is verified. This means that code satisfies the basic criteria for the recognition of this pattern. If the macrorule is proved the total probability score increases. This score indicates the confidence of the presence of the Chain of Responsibility pattern in the examined software.

3 JADEPT

JADEPT is a Java application composed of four main modules: Graphic User Interface (GUI), Launcher and Capture Module (LCM), Design Pattern Detector Module (DPDM) and JDEC Interface Module (see Figure 1).

JADEPT's GUI allows users: (1) to set up a JADEPT XML configuration file, (2) to launch the software to be monitored, (3) to start the analysis on the stored information and (4) to create the JDEC database.

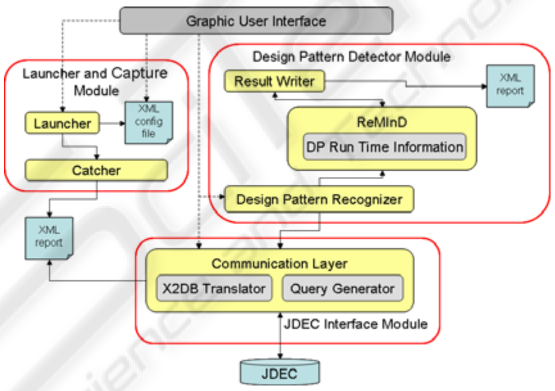


Fig. 1. JADEPT Architecture.

The Capture and Launcher Module is composed of (1) the Launcher, which starts the execution of the software under analysis and the execution of the Catcher Module, and (2) the Catcher, which captures events occurred in the JVM created by the Launcher for the analyzed software. Events regard classes and interfaces loading, method calls, field accesses and modifications. Through these events JADEPT extracts various types of information exploited in the detection process.

At the end of user's software execution the Catcher Module writes the XML Report File containing all the collected information and invokes the Communication

Layer insertion method. Thus, the XML Report File is inserted in the JDEC database. In this way, information is available to the Design Pattern Detector Module (DPDM).

The Communication Layer represents the link between JDEC and the other software modules. Its functions are provided by XML2DBTranslator and Query Generator. XML2DBTranslator interprets the XML Report File created by the Catcher Module and creates inserting queries to fill JDEC. The Query Generator is used during the analysis phase to create the appropriate queries.

The Design Pattern Detector Module is composed of: the Design Pattern Recognizer, the Result and Metric Information Dispenser (ReMInD) and the Result Writer.

The Design Pattern Recognizer contains the classes used in the detection process. Each class defines a thread representing a specific pattern role and each thread performs the analysis on a class family. A thread checks the rule on the family assigned to it and writes the results on an object metaphorically called *whiteboard* [13]. The ReMInD module provides objects which support the analysis threads. Each ReMInD object is a whiteboard used by a thread to store information about temporary results obtained during the analysis. The Result Writer receives results coming from the Design Pattern Recognizer and it disposes them into an output file, called Result File. The last is divided into various sections; each of them referring to a different pattern.

For more details on JADEPT see [2].

4 Validation

JADEPT has been validated using different implementation samples of design patterns more or less closer to their GoF's definitions [7].

The results of the analysis on different implementations of design patterns are shown in Tables 3, 4 and 5 and are related to the detection of three of the behavioral design patterns: Chain of Responsibility, Observer and Visitor. Table 6 shows the results of JADEPT that analyzes itself.

The first column of each table contains the identification name for the implementations considered. The remaining columns show the results provided by the Chain of Responsibility, Observer and Visitor detectors. The `-' symbol means that JADEPT has not detected any instance for a given design pattern. The `X' symbol indicates that the considered sample does not provide any implementation of a specific pattern.

Table 3 illustrates the results obtained during the detection of Chain of Responsibility pattern. The second column indicates the results obtained through the Chain of Responsibility detector, while column three and four the results obtained through the Observer and Visitor detectors. The last two detectors have been used to verify if they provide false negatives. The same approach has been applied in Table 4 and Table 5.

JADEPT recognizes the Chain of Responsibility pattern in three implementations with reliable values. The Chain implementation in fluffycat is detected as a false negative because JADEPT is not able to find a good *handle()* candidate in this pattern instance. This argument indicates the request that should be managed by one of the classes which implements the interface. Moreover, each class implementing the interface declares a field whose type is the type of the common interface. The successor element in the chain is assigned to this field during execution.

Table 3. Chain of Responsibility implementation analyzed by three design pattern detectors.

Chain of Responsibility	Chain of Responsibility Detector	Observer Detector	Visitor Detector
composite	X	X	X
composite3	X	X	X
cooper	100%	10%	17%
earthlink	76%	-	-
earthlink2	X	X	X
fluffycat	7%	-	-
kuchana	69%	-	-
sun	X	X	X
vis1	X	X	X
visitorcontact	X	X	X

Figure 2 shows the class diagram related to the implementation of the Chain of Responsibility in the *fluffycat* example. According to the GoF's definition, this pattern should define a common interface (e.g., called *Chain*) which is implemented further by two or more classes. The interface defines a method (e.g., called *sendToChain(String)*) which accepts only one argument.

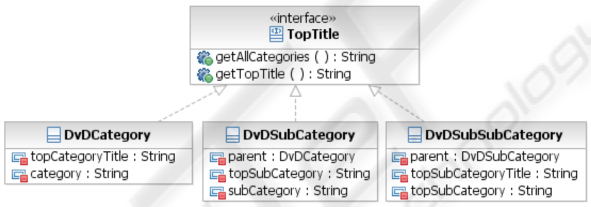


Fig. 2. Class diagram for the Chain of Responsibility pattern in fluffycat.

The fluffycat implementation is not closed to the GOF's definition: it defines a common interface called *TopTitle*, but this interface declares methods which accept no arguments. One of the purposes of the Chain of Responsibility pattern is to build a structure which is able to handle requests generated by a sender. Hence, it is not possible that the *handle()* method accepts no parameters. Moreover, the three classes do not declare any field for a successor whose type is the interface type. *DvdCategoryClass* does not declare any field which indicates a reference to its successor. The *DvdSubCategory* class declares a field of *DvdSubSubCategory* type. The instances of these classes can be chained only in one way: the knowledge indicating which object must be used as a successor of another is built-in the classes and not in an external class which should define how the chain must be created. Hence, such an implementation is unacceptable and does not comply with the GoF's definition. This is reflected in the low values associated to the fluffycal implementation in Table 3.

The Observer and Visitor detectors obtain satisfying results. There are cases (e.g., earthlink, fluffycat and kuchana) in which detectors have not even start their analysis due to the absence of a common class or interface in these implementation instances. Moreover, the Chain structure is deeply different from the Observer and Visitor structures. It requires only one role, while the Observer and Visitor require two. This is the main reason why the two detectors cannot perform the analysis. In the Observer and Visitor implementations of the cooper sample it is revealed a Chain instance with a very low probability score, hence it cannot be even considered significant.

The results of the detection of the Observer pattern are shown in Table 4.

Table 4. Observer implementation analyzed by three design pattern detectors.

Observer	Chain of Responsibility Detector	Observer Detector	Visitor Detector
composite	X	X	X
composite3	X	X	X
cooper	15%	100%	20%
earthlink	15%	-	37%
earthlink2	15%	10%	17%
fluffycat	-	-	-
kuchana	23%	90%	40%
sun	23%	47%	40%
vis1	X	X	X
visitorcontact	X	X	X

The Observer detector provides significant results for kuchana and cooper implementations. It obtains false negative values for earthlink2 and sun implementations and it does not provide any result for earthlink and fluffycat. The reason why the Observer detector cannot perform the analysis is related to the correctness of the implementations and to how JADEPT partitions the software system under examination to perform analysis. Even if Observer and Visitor are similar, the Visitor detector provides false positive results. The highest values were obtained for kuchana and sun implementations. It may be possible that the *notify()* and *update()* methods are considered by the static rules as *accept()* and *visit()* candidates. The fluffycat implementation cannot be analyzed due to the lack of the common classes/interfaces. The Chain of Responsibility detector provides very low probability scores, and also it cannot analyze the fluffycat implementation.

Table 5 shows the results obtained during the detection of the Visitor pattern.

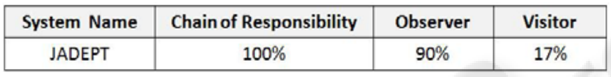
Table 5. Visitor implementation analyzed by three design pattern detectors.

Visitor	Chain of Responsibility Detector	Observer Detector	Visitor Detector
composite	7%	-	93%
composite3	69%	20%	93%
cooper	76%	10%	37%
earthlink	15%	-	37%
earthlink2	15%	10%	17%
fluffycat	-	-	-
kuchana	23%	-	100%
sun	23%	26%	100%
vis1	23%	26%	20%
visitorcontact	-	26%	100%

The Visitor detector provides satisfying results for five implementations. The Observer detector provides low false positive results, and it cannot analyze kuchana and composite implementations. The Chain of Responsibility detector obtains low false positive results, except for the composite3 and cooper implementations. In these cases, one method is wrongly considered as a *handle()* candidate. The Chain of Responsibility detector cannot perform the analysis on the visitorContact implementation. This result depends on the differences between the two pattern structures.

Table 6 shows the results of JADEPT that analyzes itself. JADEPT is composed of 151 classes. Analysis reveals the presence of Chain of Responsibility and Observer, which are actually implemented in the code. In JADEPT there are no Visitor instances, and this analysis was performed only to test if any false positives are revealed.

Table 6. JADEPT analyzed by JADEPT.



System Name	Chain of Responsibility	Observer	Visitor
JADEPT	100%	90%	17%

To summarize, there are two main reasons why JADEPT cannot perform analysis on some implementations. The first is related to the quality of implementations themselves because they are very different from the UML structure of patterns defined by GoF. For example, classes do not implement the same interface or extend the same class. We mean that such implementations cannot be retained as valid ones. Common interfaces and classes are used to easily extend software and their use is a principle of good programming as much as other design pattern features.

The second problem concerns the information partitioning technique of JADEPT. Our tool can work on families retrieved from the information collected in JDEC. Before starting the analysis, JADEPT identifies all the possible families and assigns to each family a specific role, according to the design pattern it is looking for. If the analyzed system is unstructured, meaning that common interfaces or classes are absent, JADEPT cannot build correctly the families and perform further analysis.

5 Conclusions and Further Work

In this paper we have presented our approach to detect design patterns in Java applications. As mentioned in the introduction section, there are several main issues which should be addressed during the development of a design pattern detection tool. Considering these issues, our contribution includes the definition of the recognition rules, the use of dynamic analysis to extract all the information needed in the detection process and the specification of an entity-relationship schema to store and organize the extracted information from Java applications to be easily used for the recognition of design pattern instances.

Our recognition rules regard the dynamic nature of patterns. Rules focus on the behavior of the design patterns and not on their static aspects. Rules capturing static properties have been introduced because they express pre-conditions for the dynamic ones. Furthermore we have defined logical and informative dependencies among rules, established the importance of rules in the detection process through scores, and identified a group of rules characterizing the particular behavior of each pattern through macrorules.

Dynamic analysis may obviously provide significant information for design pattern recognition. Through dynamic analysis it is possible to observe objects, their creation and execution during their entire life-cycle and overcome part of the limitations of the static analysis (i.e., polymorfism) which may be determinant in pattern recognition. There are also two disadvantages of the dynamic approach. The first is related to the reduced performance of the analyzed application. To improve its performance we have used a filtering system to trace only the meaningful events. Nevertheless the execution time of the monitored applications is still longer than the ordinary execution time, especially for software having a Graphic User Interface. If the software under examination does not require user interaction, execution time should not be a critical factor. The second, concerns the code coverage problem. If the analyzed software needs a user interaction, it could be necessary a human-driven selection of code functions to reveal all possible behaviors. Thus, it is necessary to cover the entire code and test all code functions one by one.

We have validated our idea through the implementation of the JADEPT prototype. Modularity is one of the main characteristic of the JADEPT architectural model. Furthermore, the tool can be easily extended to other programming languages. It may use alternative ways to extract information or to perform analysis. It is possible to exclude the database and to use another approach to detect design patterns due to existence of the XML Report file. Or, the database model can be used in another design pattern detector or a software architecture reconstruction tool.

The decision to use a database to store the extracted information is due to two main reasons. The first is related to the large amount of information which should be extracted during software execution and which should be further considered to identify design patterns. The second is related to the traceability/persistence in time of the extracted information, the comparison among two or more executions of the software code or among executions of different applications, and the statistics which may be done. The issues related to this second aspect are not implemented in the current version of our prototype but will be addressed in the future developments of JADEPT.

Further work will regard also the extension of JADEPT to the creational and structural design patterns, as well as to its validation on systems of larger dimensions.

References

1. Arcelli, F., Masiero, S., Raibulet, C., Tisato, F.: A Comparison of Reverse Engineering Tools based on Design Pattern Decomposition. In Proceedings of the IEEE Australian Software Engineering Conference. (2005) 262-269
2. Arcelli, F., Perin, F., Raibulet, C., Ravani, S.: Behavioral Design Pattern Detection through Dynamic Analysis. In Proceedings of the 4th International Workshop on Program Comprehension through Dynamic Analysis –Tech. Report TUD-SERG-2008-036. (2008) 11-16
3. Bergenti, F., Poggi, A.: Improving UML Designs Using Automatic Design Pattern Detection. In Proceedings of the 12th International Conference on Software Engineering and Knowledge Engineering. (2000)

4. Byte-Code Engineering Library (BCEL), http://jakarta.apache.org/bcel/
5. Cooper, J. W.: The design pattern Java companion. Addison-Wesley (1998)
6. Demeyer, S., Mens, K., Wuyts, R., Guéhéneuc, Y.-G., Zaidman, A., Walkinshaw, N., Aguiar, A., Ducasse, S:.Workshop on Object-Oriented Reengineering (2005)
7. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: elements of reusable object-oriented software, Addison Wesley, Reading MA, USA (1994)
8. Guéhéneuc, Y.-G., Douence, R., Jussien, N.: No Java without Caffeine. A Tool for Dynamic Analysis of Java Programs. In Proceedings of the 17th IEEE International Conference on Automated Software Engineering. 117-126 (2002)
9. Guéhéneuc, Y. G.: PTIDEJ: Promoting Patterns with Patterns. In Proceedings of the 1st ECOOP Workshop on Building Systems using Patterns, Springer-Verlag, (2005)
10. Heuzeroth, D., Holl, T., Löwe, W.: Combining Static and Dynamic Analyses to Detect Interaction Patterns. In Proceedings the 6th World Conference on Integrated Design and Process Technology (2002)
11. Nickel, U., Niere, J., Zündorf, A.: The FUJABA Environment. In Proceedings of the 22nd International Conference on Software Engineering, 742-745 (2000)
12. Niere, J., Schäfer, W., Wadsack, J. P., Wendehals, L., Welsh, J.: Towards Pattern-Based Design Recovery. In Proceedings of the 24th International Conference on Software Engineering, 338-348 (2002)
13. Perin, F., Dynamic analysis to detect the design patterns in Java: gathering information with JPDA. MSc Thesis, University of Milano-Bicocca, Milan, (2007)
14. Ravani, S.: Dynamic analysis for Design Pattern detecting on Java code: information relationship modelling, MSc Thesis, University of Milano-Bicocca, Milan, (2007)
15. Pettersson, N.: Measuring Precision for Static and Dynamic Design Pattern Recognition as a Function of Coverage. In Proceedings of the Workshop on Dynamic Analysis, ACM SIGSOFT Software Engineering Notes, Vol. 30, No. 4, 1-7 (2005)
16. Shawky, D. M.., Abd-El-Hafiz, S. K., El-Sedeek, A.-L.: A Dynamic Approach for the Identification of Object-oriented Design Patterns. In Proceedings of the IASTED Conf. on Software Engineering, 138-143 (2005)
17. Shi, N., Olsson, R. A.: Reverse Engineering of Design Patterns from Java Source Code. In Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering, 123-134 (2006)
18. Smith, J. McC., Stotts, D.: Elemental Design Patterns: A Formal Semantics for Composition of OO Software Architecture. In Proceedings of the 27th Annual NASA Goddard Software Engineering Workshop, 183 (2002)
19. Smith, J. McC., Stotts, D.: SPQR: Flexible Automated Design Pattern Extraction From Source Code. In Proceedings of the 2003 IEEE International Conference on Automated Software Engineering, 215-224 (2003)
20. Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., Halkidis, S. T.: Design Pattern Detection Using Similarity Scoring. In IEEE Transactions on Software Engineering, Vol. 32, No. 11, 896-909 (2006)
21. Verkamo, A. I., Gustafsson, J., Nenonen, L., Paakki, J.: Design patterns in performance prediction. In Proceedings of the ACM Second International Workshop on Software and Performance, 143-144 (2000)
22. Wendehals, L.: Improving Design Pattern Instance Recognition by Dynamic Analysis. In Proceedings of the ICSE 2003 Workshop on Dynamic Analysis, 29-32 (2003)
23. Zaidman, A., Hamou-Lhadj, A., Greevy, O.: Program Comprehension through Dynamic Analysis. In Proceedings of the 1st International Workshop on Program Comprehension through Dynamic Analysis Tech. Rep. 2005-12 (2005)