Innovations Syst Softw Eng (2015) 11:39–53
DOI 10.1007/s11334-014-0241-3

ORIGINAL PAPER

Using metric-based filtering to improve design pattern detection approaches

Imène Issaoui · Nadia Bouassida · Hanêne Ben-Abdallah

Received: 29 September 2013 / Accepted: 24 November 2014 / Published online: 9 December 2014

Abstract Design patterns represent high-level recurring abstractions that reflect the know-how of experts. Their detection is a key task in the context of software engineering; it is very useful in program comprehension, design recovery and also for re-documenting object-oriented systems. Despite their usefulness, current design pattern detection approaches have a high time complexity that hinders their application. This complexity is further aggravated with the absence of guiding principles in determining which pattern to look for first among the various patterns. To overcome this practical limit, we propose to optimize any pattern detection approach by foregoing it by a filtering phase that detects possible existence of patterns and that orders the candidate patterns in terms of their degree of resemblance to the analyzed design fragments. The herein proposed filtering approach exploits semantic and structural design metrics to look for the semantic and structural symptoms of design pattern instances. Its performance is experimentally demonstrated through our design pattern detection method MAPeD (Multi-phase Approach for Pattern Discovery) applied on the open source system JUnit.

Keywords Object-oriented metrics · Design patterns · Quality assurance · Filtering

I. Issaoui (✉) · N. Bouassida
University of Sfax, Sfax, Tunisia
e-mail: imene.issaoui@gmail.com

N. Bouassida
e-mail: nadia.bouassida@isimsf.rnu.tn

H. Ben-Abdallah
King Abdulaziz University, Jeddah, Saudi Arabia
e-mail: hbenabdallah@kau.edu.sa

1 Introduction

Design patterns [1] capture quality-proven solutions to recurring problems. In software development, design patterns reuse accelerates the development process and improves the quality of the design and produced code. In software maintenance, design patterns can be used in program comprehension, design recovery and also for re-documenting object-oriented systems. In both contexts, identifying design patterns instantiations in existing code and design is very important.

Several researchers proposed multiple approaches for design pattern detection (e.g., [2–5]) using different techniques: graph matching [2,4], constraint satisfaction problem [3], or XML document retrieval [6]. Independently of their technique, all existing design pattern detection approaches suffer from a high time complexity—exponential in terms of the size of the design for each design pattern identification. In practice, this complexity further hinders their application since most often, with no a priori knowledge about one particular design pattern or the design, designers end up using the existing approaches to try the identification of all patterns in an ad hoc manner, even if no pattern exists. A more judicious identification should have a strategy to eliminate those patterns that may not exist and order candidate patterns that it should look for in the design. The proposition of such strategy is the main contribution of this paper.

More specifically, we propose a filtering approach that can be applied as a preliminary step to any existing pattern identification method to improve its performance in terms of both time and recall and precision rates. Our method first filters the design patterns that are probably present in the design, secondly it orders those candidate patterns in terms of their degree of resemblance to the design fragments, and finally it delimits the design fragment possibly containing

each candidate pattern. Therefore, any pattern identification approach can be used to detect patterns beginning by those that are most likely to be present and within the delimited design fragments.

The main basis of our method is the fact that any design pattern is devoted to a certain objective and it improves a particular aspect of design quality, e.g., loose structural coupling, high semantic dependency among structurally related elements, etc. Hence, the presence of a design pattern depends on the presence of its corresponding quality indicators or symptoms. Consequently, an important step in defining our approach consists of capturing each pattern objective and quality improvements with a set of structural and semantic metrics. Thanks to these metrics, a design can be analyzed both to predict the existence of design pattern instances and to locate their positions within the design. For instance, the pattern Mediator is devoted to reduce coupling [7] while it centralizes control in the class Concrete Mediator whose complexity becomes high. In addition, the class Mediator of this pattern has a high Coupling Between Objects (CBO) [8]. Hence, if the CBO is low for *all* the classes of a design, then the pattern Mediator does not exist and it is useless to try to identify it in such a design.

In addition to existing structural design metrics [9], we propose a new metric, called *semantic_coverage*, to measure the semantic similarity between design patterns and a design fragment. This semantic metric refines the ranking of candidate patterns determined through the structural metrics. Evidently, besides their formulas, the structural and semantic metrics' values are essential to a meaningful characterization of design patterns. We think that an appropriate design pattern instance is one that respects (i.e., does not fall under the thresholds of) the values of the metrics that reflect the intention of the pattern. In order to characterize the patterns with quality values and to determine the thresholds that reflect the intention of the design pattern, we conducted an empirical study on four open source systems: JHotDraw v5.1 [10], JRefactoty v1.0 [11], QuickUML 2001 and Lexi. Collecting the characterizing structural metrics from the literature, defining the new ranking semantic metric, and empirically determining the metrics' thresholds represent the second contribution of this paper.

Finally, to show the efficiency of the proposed filtering approach, we applied it as a preliminary step to our method for patterns identification called MAPeD (Multiphase Approach for Pattern Discovery) [12]. Using an XML document retrieval technique, MAPeD has the advantage of tolerating structural differences between the examined design and identified design patterns; as shown experimentally [12], this advantage gave MAPeD higher recall and precision ratios compared to other existing design pattern identification approaches. As we report in this paper, the filtering approach further increases the precision of MAPeD.

The remainder of this paper is organized as follows. Section 2 overviews currently proposed approaches for pattern identification, presents the basics of design metrics and reports on works that evaluate the quality of design patterns with object-oriented metrics. Section 3 presents the characterization of design patterns with metrics along their thresholds. Section 4 presents the overall steps of our method for patterns filtering and its tool support. Section 5 first presents our pattern identification approach MAPeD, secondly it shows the advantages of our filtering approach when applied before MAPeD. Section 6 summarizes the paper and outlines our future work.

2 Related work

2.1 Design pattern identification approaches: an overview

Several techniques are used in existing approaches to design pattern identification: static design analysis (e.g., [3,4]), dynamic analysis (e.g., [5,13]), or a combination of static and dynamic analyses [14].

Within the static analysis approaches, Gueheneuc et al. [3] propose a multilayered approach for design motif identification based on constraint satisfaction problems. They introduce the concept of motif to express the structure and behavior of a pattern; they argue that patterns are different from motifs since patterns have more information such as intent, motivation, etc. Their approach, called Design Motif Identification Multilayered Approach (DeMIMA), detects motifs similar to micro-architectures in source code. It consists of three layers: two layers to recover an abstract model of the source code, including binary class relationships, and a third layer to identify motifs in the abstract model. In order to identify micro-architectures similar to a motif, DeMIMA transforms the motif into a constraint system, and then it solves the resulting constraint satisfaction problem. One advantage of DeMIMA is that it tolerates a partial match of the pattern in the design; however, it focuses only on the structural aspect of the pattern, while neglecting the behavioral and semantic aspects.

Also following a static analysis approach, Tsantalis et al. [2] consider that recognizing a pattern in a design is a graph matching problem. They propose a design pattern detection approach based on similarity scoring [15] between graph vertices; the graphs of the searched pattern and the examined design are encoded as matrices from which a similarity matrix is derived. This approach can only calculate the similarity between two vertices (representing classes/relations), instead of two graphs (representing the whole pattern and design), thus it has a low precision ratio in the identification. This drawback is bypassed in the approach and tool (called SGFinder) proposed by Belderrar et al. [4]. SGFinder derives

OO micro-architectures from the class diagram; a micro-architecture represents a connected sub-graph induced from the design's class diagram.

Because a static analysis may not be sufficient to correctly identify a design pattern within a design, some approaches combine it with a dynamic analysis technique. As an example of methods adopting this hybrid approach, De Lucia et al. [5] identify behavioral design patterns. First, a static analysis is used to identify candidate instances of a behavioral pattern. Then, a dynamic analysis is performed over the automatic instrumentation and monitoring phase of the method calls involved in the identified candidate pattern instances. The dynamic information obtained from a program monitoring activity is matched against the definitions of the pattern behaviors expressed in terms of monitoring grammars.

Also following a combination of static and dynamic analysis approach, Bouassida et al. propose an identification approach [6] based on XML document retrieval techniques where the pattern is seen as the XML query and the design as the XML document in which the query is searched. It relies on a context resemblance function to compute the similarity potential between the design structure and behavior, and the pattern. The proposed approach is applicable to account for both the structure and dynamic aspects of the pattern and it accommodates design variability with respect to the pattern structure without losing the pattern essence.

Among the dynamic analysis approaches, Arcelli et al. [16] propose a design pattern detection technique for Java codes that relies on data collection through JPDA (Java Platform Debugger Architecture). The approach is rule based and it is designed for behavioral design patterns. A rule is a set of conditions verified by a given design pattern. All Rules are based on definitions given by the Gang of Four (GoF) [1]. They describe both structural and behavioral properties of selected patterns. The affinities of rules to different patterns are defined via specific weights.

In summary, most approaches proceed by a structural identification of design patterns; only a few consider the behavioral aspects of design patterns. In addition, except for MAPeD, none of the existing approaches explore the semantics encoded in the patterns' elements. Furthermore, all proposed approaches suffer from a high temporal complexity which is exponential in the size of the design. MAPeD proposes a practical solution to alleviate this complexity by starting with a design fragmentation step that can be done either manually, or automatically based on some heuristics derived from the analysis of GoF design patterns. Finally, all existing approaches presume that the designer knows which design pattern he/she is looking for in the design. In the absence of such information, all existing approaches would end up looking for all design patterns in an ad hoc manner—this in turn aggravates the temporal complexity.

2.2 Useful existing design metrics

Several metrics are defined in the area of OO software engineering. The most known metrics are those presented by Chidamber and Kemerer (C&K) [9] and classified into four categories: coupling, cohesion, complexity and inheritance.

1. Coupling measures the degree of interdependency between classes; it should be minimized in order to allow easy maintainability. It is measured thanks to Coupling Between Objects (CBO) and Response for a Class (RFC).
2. Cohesion indicates if the operations and methods of a class are closely related. Evidently, the cohesion should be maximized to ensure a high encapsulation and a good design quality. Cohesion is measured thanks to Lack Of Cohesion in Methods (LCOM).
3. Complexity measures the degree of simplicity of a design. A reduced complexity insures an increased readability and a better understanding. Many complexity measures have been proposed in the literature, amongst which we find Weighted Methods per Class (WMC).
4. Inheritance measures the tree of inheritance and the number of children. Inheritance decreases complexity. The two metrics used to measure the amount of inheritance are the Depth of Inheritance of a class (DIT) and the Number Of Children (NOC).

Table 1 briefly presents the metrics that we found useful for design pattern characterization.

Note that NOC, DIT, WMC, CBO and RFC metrics are calculated at the design level, while LCOM is calculated at the code level.

2.3 Using metrics to measure the quality of design patterns

Several works examine the relationships between design patterns and OO design metrics in an attempt to evaluate the impact of design patterns on the quality of the design [7,8,17–25]. For example, Reißing [22] compares the values of classic OO design metrics to two similar designs—one using design patterns while the second does not; the metrics show that the first design is better than the second because it has less classes, operations, inheritance, associations, etc. Reißing therefore proposes a more appropriate notion of quality that includes both views: the traditional design metric view based on size, coupling, and other complexity criteria; and the flexibility considerations inherent to design patterns.

Masuda et al. [7] use the C&K metrics suite of [9] to evaluate the efficiency of applying design patterns in two applications developed by their research group. They notice that some design patterns have a tendency to degrade the values of certain metrics. However, they caution that this does not necessarily mean that these design patterns always

Table 1 Object-oriented metrics [9]

Metric name	Abbr.	Description
Number of children	NOC	Counts the number of immediate descendants (via an inheritance relationship) of the class
Coupling between objects[9]	CBO	Counts the number of classes coupled to a given class. A coupling can occur through method calls, field accesses, inheritance, arguments, return types, and/or exceptions
Response for a class [9]	RFC	Counts the number of methods that can be executed when an object of that class receives a message (i.e., when a method is invoked for that object). Ideally, we would want to find for each method of the class, the methods that class will call and repeat this for each called method, calculating what is called the transitive closure of the method's call graph
Lack of cohesion in methods [9]	LCOM	Counts the sets of methods in a class that are not related through the sharing of some of the class's methods
Depth of inheritance of a class [9]	DIT	It is the depth in the inheritance tree. If multiple inheritances are involved, then the DIT will be the maximum length from the node to the root of the tree
Weighted methods per class [9]	WMC	It is simply the sum of the complexities of the methods. As a measure of complexity we can arbitrarily assign a complexity value of 1 to each method in which case the value of the WMC is equal to the number of methods in the class

induce quality degradation. In addition, Masuda et al. [7] note that because the metrics WMC, DIT, NOC and LCOM are defined for single classes, they are not suitable to measure relationships among classes. Only the RFC and CBO metrics can capture the degree of communication between classes in a limited way—they reflect the one-to-many relationship. Instead, Masuda et al. suggest that new metrics should be proposed for the evaluation of the efficiency of applying design patterns.

Ampatzoglou et al. [25] propose a methodology for comparing pattern designs to alternative designs. The proposed method attempts to formulate several quality attributes as functions of functionality addition on multiple, equivalent solutions to a design problem. Then the functions are compared so as to identify cut-off points during system maintenance when one solution gets better or worse than the other. In this study Ampatzoglou et al. conclude that patterns typically improve certain aspects of software quality, while they might weaken some other. Moreover, they state that there exist thresholds that when surpassed the design pattern is getting more or less beneficial. More specifically, the identification of such thresholds can become very useful for decision-making during system design.

Even though the verdict on the relationships between design patterns and specific OO metrics is not clear, the various works seem to agree about which subset of metrics that can be used to indicate the presence of design patterns.

3 Characterizing design patterns with metrics

As discussed in [1,7,19,20], design patterns do not simultaneously improve all quality aspects, but each design pattern is devoted to particular quality aspects. For example, the Strategy pattern mainly promotes polymorphism [26]; in addition, each created subclass is focused on only one job, which increases cohesion. In fact, the strategy promotes the "low cohesion principle" while reducing complexity [26]. Translated into metrics, these characteristics mean that Strategy reduces WMC [27] and decreases CBO and RFC. Consequently, if the CBO and the RFC are high in a design, then the Strategy pattern is highly unlikely to be correctly instantiated in the design.

In Table 2, we synthesize the relationships between design patterns and OO design quality properties. Note that, in general, using design patterns promotes object composition rather than class inheritance; hence the DIT values are not very high in a design instantiating design patterns. We note that an inappropriate/incorrect instantiation of design patterns leads to quality deterioration. In our opinion, an appro-

Table 2 Metric-based properties of design patterns symptoms

D. P	OO design properties	Design Quality attributes	Metrics					
			Inheritance		Coupling		Complexity	Cohesion
			DIT	NOC	CBO	RFC	WMC	LCOM
Mediator	Decrease coupling, increase complexity, increase reusability [8,20]	Increase reusability [1]	DIT value does not become very high while applying design patterns [22]	High for the class Colleague	High class Mediator [7]	High for the class Concrete Mediator [7]	High for the class Concrete Mediator [7]	High for the class Concrete Mediator [7]
Command	Decrease complexity [1]	Increase flexible [1]		High for the class Command [7]	Low for the class Invoker	Low for the class Invoker	Low for the class Concrete Command [7]	–
Strategy	Increase cohesion [26,27], decrease complexity [17]	Increase flexibility, increase maintainability [1], increase polymorphism[20]		High for the class Strategy [7]	Low for the class Strategy	Low for the class Concrete Strategy	Low for the class Concrete Strategy and context [7,17,27]	–
State	Increase cohesion [26], decrease complexity	Increase flexibility [1], increase polymorphism [20]		High for the class State [22]	Low for the class State	Low for the class Concrete State	Low for the class Concrete State [27]	Does not change when applying State [22]
Factory method	Decrease complexity [7]	Increase flexibility [1]		High for the classes Product and Creator	High for the class Concrete Creator [7]	High for the class Concrete Product	Low for the class Concrete Factory [7]	–
Visitor	Decrease complexity [17], increase polymorphism [1]	–		High for the class Visitor [7]	Low for the class Concrete Visitor	High for the class Concrete Visitor [7]	Low for the class element [17]	–
Abstract factory	Increase complexity [28], decrease coupling [20]	Increase reusability [1]		High for the classes Abstract Factory and Abstract Product [7,28]	High for the class Concrete Factory [7,28]	High for the classes Abstract Factory and Concrete Factory [28]	High for the classes Abstract Factory and Concrete Factory [28]	–

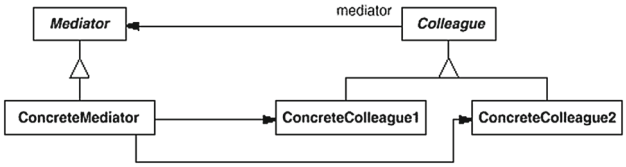


Fig. 1 Mediator design pattern [1]

priate instance is one that respects the values of the metrics for which the pattern is devoted. For example, according to Table 2, an appropriate instance of the Mediator pattern (shown in Fig. 1) implies a high coupling for the class Mediator since it plays a central role in the application and it needs to know about other Colleagues, its CBO value therefore tends to be high. In addition, the Mediator pattern centralizes control in the class Concrete Mediator, which increases the complexity of this class and produces a high WMC value for it. Moreover, its abstract classes Colleagues, which essentially have a large number of children Concrete Colleage, have high NOC values. Also the RFC value of Concrete Mediator tends to be high because it communicates with many other colleague objects via message passing. Furthermore, the class Concrete Mediator receives a lot of different requests in place of its colleague objects; as a result, it has various methods which are unrelated to one another which manifests itself with a high LCOM value.

Besides the set of metrics characterizing the intention of each design pattern, one must have a means to interpret the values of these metrics. In other words, one must have the appropriate thresholds based on which the "high" and "low" characterizing of Table 2 is interpreted. We should caution that, in the software engineering field, in general, there is not yet a precise guideline for how to fix thresholds. In fact, the threshold problem is far from being new.

The empirical study we conducted used four open source projects and software architectures that are known to instantiate various design patterns separately and/or in combination:

- QuickUML 2001 is an object-oriented design tool that supports the design of a core set of UML models. It contains 293 classes and approximately 3,187 methods, 1,312 uses relation, 2,380 associations, 583 aggregations and 277 inheritances.
- Lexi v0.1.1 alpha is an open source system is implemented with 23 classes, 601 methods, 336 use relationships, 551 associations, 111 aggregations.
- JRefactory v1.0 [11] is a tool designed to refactor and restructure Java source files. It contains 512 classes and approximately 8,804 methods, 3,736 uses relation, 5,004 associations, 765 aggregations and 570 inheritances.
- JHotDraw v5.1 [10] is a two-dimensional graphics framework for structured drawing editors. It includes several examples of editors. It is implemented with 191 classes and approximately 2,824 methods, 1,006 uses relation, 1,454 associations, 292 aggregations and 234 inheritances.

Table 3 presents a detailed list of the design patterns, their number of occurrences and metrics' values in the above systems. For these open source systems, the pattern instances are extracted from documentation and also identified manually by experts as presented in [29,30]. The metric values calculated on the four open source systems show that different patterns have different values. In addition, the value trends confirms the symptoms listed in Table 2. For example, for the Composite instances in the JHotDraw v5.1 and QuickUml, we find: the DIT values of all classes of the design are not very high (in the interval [0,3]); the NOC of the classes playing the role Component are high (in the interval [2,5]) and their CBO are low (in the interval [1,2]); and the RFC of the classes playing the role Composite is in the interval [4,48]) and their WMC are in the interval [4,33].

Furthermore, for each design pattern, the intervals of the metrics' values of Table 3 are used to define the structural rules where the pattern elements are matched to design elements.

4 The design patterns filtering approach

As mentioned in the Sect. 1, the main goal of our design patterns filtering is to provide designers with a mechanism that allows them to forecast the existence or nonexistence of design patterns using metrics. It should be noted that filtering is meant to be applied before the identification process for two purposes: (1) limit the list of candidate design patterns; and (2) delimit the design fragments which are structurally and semantically likely to match existing patterns.

Our design pattern filtering method operates in two analysis steps: a Semantic analysis that exploits the semantics encoded in the design patterns' elements, followed by a structural analysis only when the first step succeeds in identifying candidates design pattern. Each step uses metrics to determine an ordered list of candidate design patterns; the order reflects the degree of resemblance between the design patterns and design fragments. Thus, any detection approach can focus on identifying only the design patterns in the final ordered list as opposed to trying all design patterns.

4.1 The semantic filtering

Semantic filtering consists in predicting the potential existence of design patterns. The potential is determined by measuring the degree of the *semantic_coverage* of the patterns in the design. To define this new metric, we rely on the names

Table 3 Design patterns metrics values for JHotDraw v5.1, JRefactory v1.0, QuickUML2001 and Lexi v0.1

Design patterns	Number of instances	Classes	Metrics										
			Inheritance					Coupling				Complexity	
			NORoot	DIT		NOC		CBO		RFC		WMC	
				Min	Max	Min	Max	Min	Max	Min	Max	Min	Max
Adapter	18	Target	1	0	0	1	10	0	3	11	11	1	11
		Adapter		1	4	0	1	1	2	2	21	1	11
		Client		0	3	0	1	1	4	4	83	1	61
		Adapter		0	4	0	4	0	1	1	71	1	33
Builder	4	Director	1	0	1	0	0	1	1	1	35	1	32
		Builder		0	0	1	3	1	1	1	31	1	31
		Concrete Builder		1	2	0	0	0	1	6	37	4	35
		Product		0	2	0	1	0	1	0	28	0	28
Command	2	Command	1	0	0	9	4	1	1	3	3	3	3
		Concrete Command		1	2	0	1	2	2	1	4	0	3
		Invoker		0	2	0	0	1	2	0	19	0	61
		Receiver		0	1	0	2	2	2	3	71	3	40
		Client		0	2	0	4	2	5	7	95	7	61
Composite	3	Component	1	0	1	2	5	1	2	4	5	4	5
		Composite		1	3	0	5	1	1	4	48	4	33
		Leaf		1	3	0	5	0	2	1	57	1	38
		Client		0	3	0	0	1	1	2	95	2	52
Decorator	1	Component	1	0	1	2		9		71		32	
		Concrete Component		1	5	0	2	0	1	5	44	5	34
		Decorator		1	3	2		1		32		31	
		Concrete Decorator		2	4	0	0	0	0	17	18	8	10
Factory method	3	Product	2	0	1	1	17	0	0	6	11	0	11
		Concrete Product		1	4	0	3	1	1	4	11	2	8
		Creator		0	1	1	26	0	3	11	71	1	32
		Concrete Creator		1	3	0	4	1	1	6	71	6	38
Observer	4	Subject	2	0	1	1	4	1	4	3	3	3	32
		Concret Subject		1	3	0	1	0	2	1	44	0	34
		Observer		0	0	0	4	0	1	0	5	0	5
		Concrete Observer		0	5	0	0	1	5	4	83	1	38
Prototype	2	Prototype	1	1	2		1	0	4	15	71	15	32
		Concrete Prototype		2	3	0	2	0	2	4	57	5	38
		Client		1	3	0	2	1	1	9	71	8	18
Singleton	7	Singleton	0	0	0	0	0	0	0	4	66	4	66
State	4	State	1	0	0	1	10	1	2	7	7	9	14
		Concrete State		1	4	0	11	0	0	3	25	2	18
		Context		0	2	0	1	1	1	12	83	4	61
Strategy	2	Strategy	1	0	1	1	3	1	2	1	6	1	6
		Concrete Strategy		1	4	0	3	0	0	1	11	1	8
		Context		1	4	0	2	1	1	5	83	5	61
TemplateMethod	2	Abstract Class	1	0	2	4	6	0	1	26	42	12	35
		Concrete Class		1	3	0	2	0	2	5	57	4	61
Vistor	2	Visitor	2	0	0	6		1		13	90	13	86
		Concrete Visitor1		1	4	0	1	0	0	3	106	2	106
		Element		0	0	9		1		10	15	9	10
		Concrete Element		1	2	0	8	0	0	4	60	2	33

used in defining the pattern elements. Being well chosen, these names represent the semantics of the problem solved by the design patterns.

We suppose that each design pattern is represented by the list of its class and method names. In addition, since classes in a design pattern represent its essence (e.g., "observer" and "subject" in the Observer pattern) and since method names are less important than class names, we associate to each class name and to each method name a weight to reflect its importance in the design pattern. We manually analyzed the documentation of each design pattern to construct the lists of weighted names characterizing the design patterns. These lists are used to compare them with the names used in the design.

More specifically, the new metric *semantic_coverage* counts the number of classes and method names in the design that are "related" to a name list characterizing a particular design pattern. We consider that a class is related to a name list characterizing a design pattern, if the name of the class and/or the names of its methods are semantically related to those in the list through the name comparison criteria presented below and initially proposed in [31].

4.1.1 Class name semantic relationships

We propose the following six criteria to express the semantic relationships among the class names of the design and keywords from the name list characterizing a design pattern:

- *Is_a kind_of(C,K)*: means that the class C is a type or a variation of the keyword K. Example: *Is_a kind_of (student, intellectual).*
- *Is_one way_ to(C,K)*: means that C is one of the several manners to do K. Example: *is_one_way_to(help, support).*
- *Synonym(C,K)*: means that the name C is a synonym of the name K. Example: *synonym(student, pupil).*
- *Inter_Def(C,K)*: means that the definitions of C and K (as given by the WordNet dictionary (33)) have common words. The list of common words is obtained after eliminating the stop words such as 'a', 'and', 'but', 'how', 'or', and 'what'. Example: *Inter_Def(Figure,Composite)*
- *Def_Contain(C,K)*: implies that the definition of the name C contain the keyword K. Example*: Def_Contain(Paper, Observation).*
- *Name_Includ(C,K)*: implies that C is a string extension of the string K. Example: *Name_Includ* ("XWindow", IconXWindow).

Note that, the semantic criteria *Is_a kind_of, Is_one way_to*, and *Synonym* exist already in the WordNet dictionary [32]. The above criteria are presented in an increasing order of importance: When comparing a design class to a given pattern's classes, *Is_a kind_of* is evaluated first, if it does not hold, then *Is_one way_ to* is evaluated next, and so on so forth.

4.1.2 Method name semantic relationships

The method name comparison criteria explicit the relations among the method names in the design and the keywords characterizing the design patterns. Presented in an increasing order of importance, they are:

- *Synonym_Meth* (K, M_C): means that the method M_{C1} of the class C in the design is identical or synonym to the keyword K, i.e., they have the same name or synonym names. Example: (Build, Construct).
- *Inter_Def_Meth* (K, M_C): implies that the definition of the keyword K and the definition of the method M_C of the design class C have common words. Example: (handle, mouseDown).
- *Meth_name_Includ* (K, M_C): implies that the name of the method M_C of the design class C1 contains the keyword K. Example: (BuildPart, Build).
- *Meth_Def_Contain* (K, M_C): implies that the definition of the method M_C of the design class C contains the keyword K. Example: (Behavior, Move).

4.1.3 Semantic_coverage metric

The *semantic_coverage* metric reflects the semantic similarity between a design fragment and a particular design pattern. A design fragment is represented as set of classes and methods names and a design pattern is represented by a set of characterizing keywords collected from its documentation. The *semantic_coverage* is calculated as a weighted sum of the number matches among class names and among method names; the matches are determined by the semantic criteria defined above. The weights are used to reflect the fact that a class name carries more semantic information than its method names.

Let D be a design with C_D its set of classes and M_D its set of methods; and let K_P be the set of names characterizing a pattern P. The *semantic_coverage* between the design D and the pattern P is calculated as follows:

$$Semantic\_coverage = w_C * |C_D \text{ Rel } K_P| + w_M * |M_D \text{ Rel } K_P|$$

where:

- $|C_D \text{ Rel } K_P|$ is the number of classes in the design D which are related to the name list K_P characterizing the pattern P, through the semantic relationships defined above;

- $|M_D \text{ Rel } K_P|$ is the number of methods in the design D which are related to the name list K_P characterizing the pattern P, through the semantic relationships defined above; and
- w_C, w_M are the weighing factors for the class names and method names, respectively.

The exact values of the weighting factors can be fixed by the designer; appropriate interval values should however be determined empirically. In our experiment, we fixed the weighing factor of class names to 70 % and that of method names to 30 %.

The semantic coverage metric is used to calculate the linguistic similarity between a design fragment and a particular design. The fragment with the highest semantic coverage is the most similar.

4.2 The structural filtering

The structural filtering starts with the list of candidate design patterns identified by the semantic filtering step. This step uses the metric-based characterization of the design properties (inheritance, coupling, complexity,…) of design patterns, where each design pattern is represented through a set of correlated symptoms (see Sect. 3).

For each design pattern P semantically identified as a candidate to a design fragment D, we first calculate the values of its corresponding Structural metrics in D. Secondly, we check if the calculated values are coherent with the metrics' thresholds that reflect the intention of the design pattern, already presented in Table 3. If that is the case, we presume that the pattern has a high probability of being adequately instantiated in the design; in other words, it is justifiable to apply a pattern detection method on the design fragment D to look for this particular pattern P.

We retained from the metrics suite of [9] the following set of metrics: Depth of Inheritance Tree (DIT), Number Of Children (NOC), Coupling Between Objects (CBO), RFC (Response For Call), Weighted Methods per Class (WMC). In addition to the above existing metrics, we propose a new metric to filter further candidate patterns. The new metric, called Number Of Roots (NORoot), calculates the number of super classes. This metric is inspired from our observation that the number of hierarchies is one salient characteristic of GoF design patterns [1]. For instance, State and Strategy have a single hierarchy; Mediator, and Observer have two hierarchies, whereas Abstract Factory has three hierarchies. Our empirical study shows (Table 3) that the open source systems respect these values of NORoot. Thus, filtering first according to NORoot eliminates several patterns depending on the value of NORoot of the design fragment.

Besides confirming the metric-based characterization of design patterns shown in Table 2, our empirical study of Table 3 has two additional contributions: it complements existing metric-based characterizations for missing design patterns (Adapter, Builder, Composite, Decorator, Observer, Prototype, Singleton and Template Method), and it offers detailed interval values of the metrics thresholds. These latter can be used to structurally filter the design patterns.

For example, for the Composite instances in the JHotDraw v5.1 and QuickUml, we find that NORoot is equal to 1, the DIT values of all classes of the design are not very high (in the interval [0,3]), and the NOC of the class playing the Component role are high (in the interval [2,5]). The CBO of the classes playing the Component role are low (in the interval [1,2]), the RFC of the classes playing the role of Composite is in the interval [4,48]), and the WMC of the classes playing the Composite role are in the interval [4,33]. Thus, to detect the Composite pattern, we apply the following rule:

RuleComposite.
If there is a set of classes, $C_1,\ldots,C_n$, belonging to the design fragment D, such that :
NORoot (D) =1 and
$0< DIT(C_1,\ldots,C_n) <3$ and
(*Synonym*(C_1,*Component*) or *Name_Includ*("Component", C_1)) and $(2< NOC(C_1) < 5)$ and $(1< CBO(C1)<2)$
(*Synonym*(C_2,*Composite*) or *Name_Includ*("Composite", C_2)) and $(4< RFC(C_2)<48)\}$ and $(4 < WMC(C_2)<33)$
→ the Composite design pattern is susceptible to be present in D.

To summarize, the structural filtering begins by the NORoot metric. Next, it calculates the structural metrics for these classes. Afterwards, it checks if the metrics' values of these classes comply with the symptoms of the pattern according to Table 2. If that is the case, the structural filtering indicates the eventual existence of the corresponding design pattern.

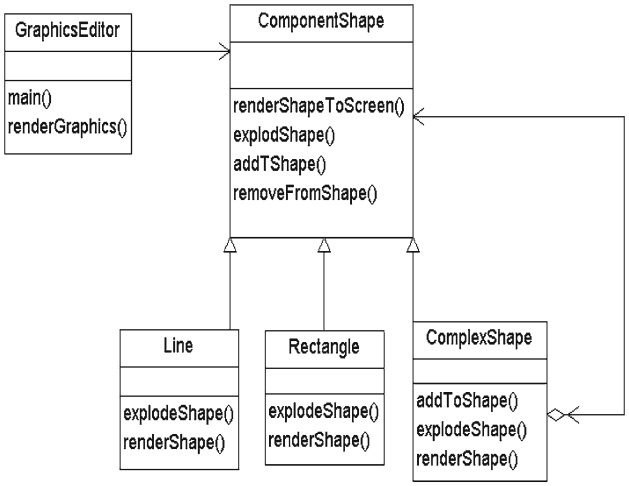


Fig. 2 Design example showing an application instantiating the composite design pattern

4.3 Example: structural filtering of the composite design pattern

In the semantic filtering of the design D of Fig. 2, after calculating the *semantic_coverage* metric for all the patterns, we find that the *semantic_coverage* for the Composite design pattern has the highest value: *semantic_coverage (D, composite)* = 2.3

After the semantic filtering, we proceed with the structural step by applying the rule **RuleComposite**. Using class name semantic relationships, we find that:

- Name_Includ(ComponentShape,Component) which gives us the ComponentShape class as Component;
- Synonymy(ComplexShape, Composite) which gives us the class ComplexShape as Composite;
- NORoot of the design is equal to 1, the DIT values of all its classes are not very high (in the interval [0,1]); the NOC of the class ComponentShape playing the Component role is high (equal to 3), its CBO is low (equal to 1); and the WMC of the class ComplexShape is low (equal to 3).

4.4 Tool support

To further examine the performance of our filtering approach, we have developed for it a tool support. As illustrated in Fig. 3, the functional architecture of the *PatternSniffer* toolset is essentially composed of four parts: (1) the extraction of design tree; (2) the design decomposition; (3) the Seman-

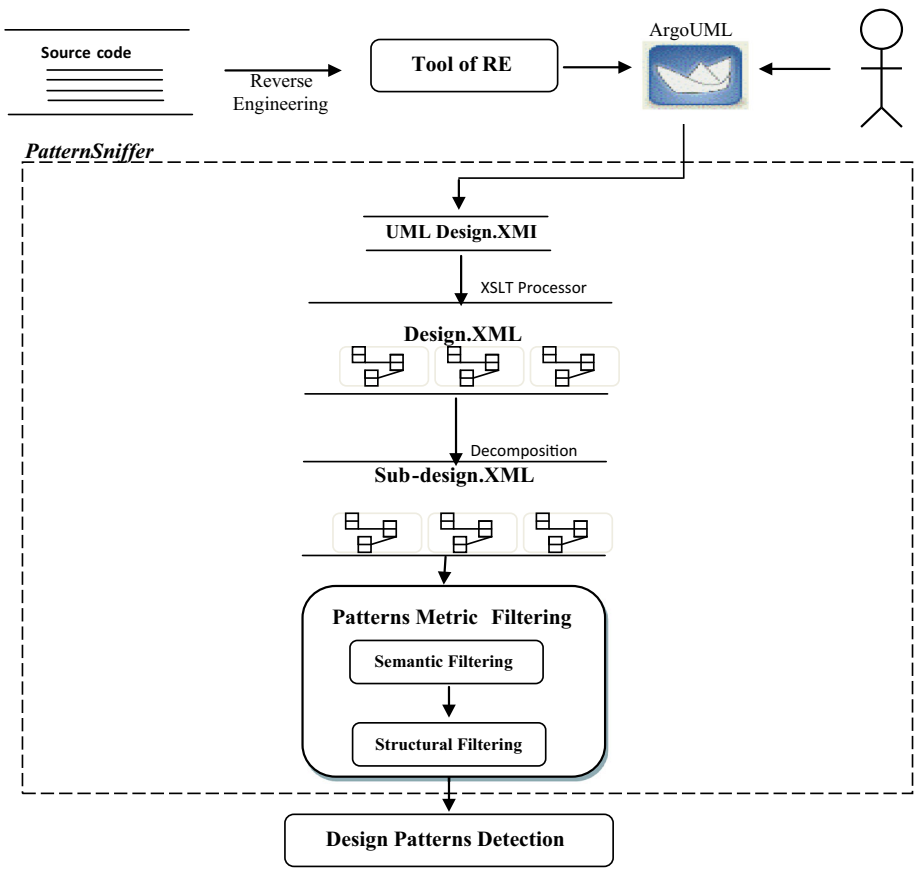


Fig. 3 Functional architecture of the *PatternSniffer* toolset

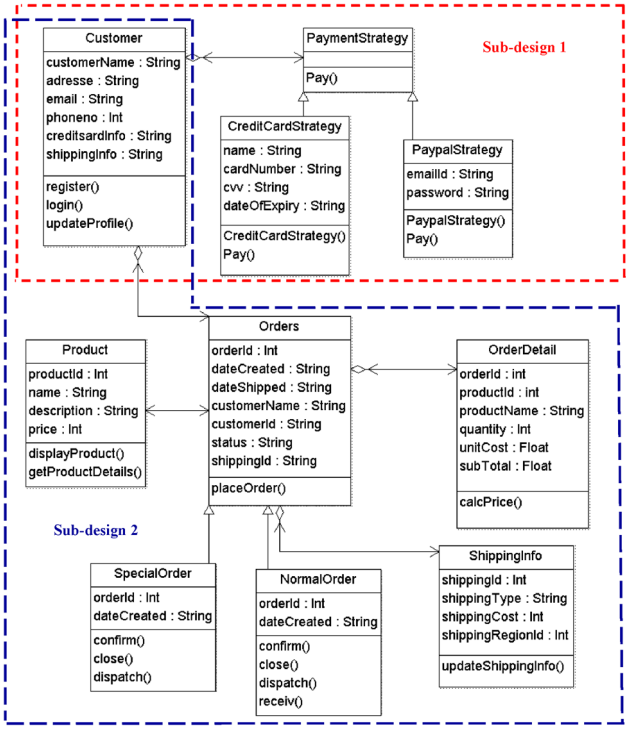


Fig. 4 Design fragment

tic pattern filtering; and (4) the Structural pattern filtering. The current *PatternSniffer* prototype reuses the ArgoUML toolset [33] for the design specification in UML and uses XSLT Transformations to convert the XMI file produced by ArgoUML to an XML file. *PatternSniffer* imports UML diagrams saved as XMI files or XML documents. Once the design is read (as an XML document), the decomposition and filtering steps can proceed for various design patterns. The design decomposition starts from the abstract class and its descendants, then for each class belonging to the obtained hierarchies, its associated classes are also chosen.

To illustrate the various functionalities of *PatternSniffer*, let us illustrate the filtering process on the Strategy design pattern within the design fragment shown in Fig. 4.

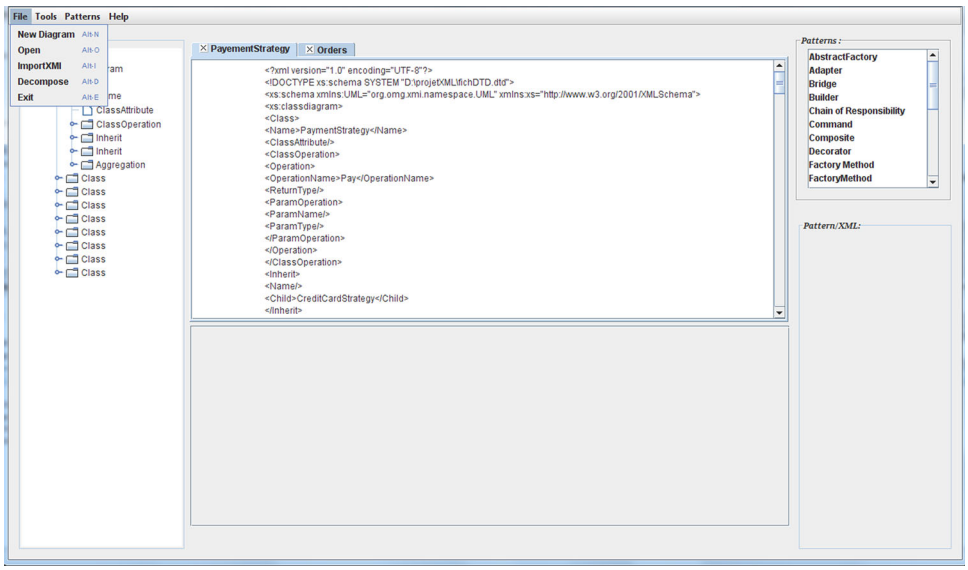
Figure 5 illustrates a snapshot of *PatternSniffer* presenting the XML document corresponding to the design fragment shown in Fig. 4. In addition, it shows in the right panel the GOF patterns [1] that could be filtered with our tool. Moreover, in the File Menu the Decompose button constructs the sub-designs. The decomposition of this design fragment produced two sub-designs. Recall that, our decomposition strategy considers the connected components of the class diagram, starting from an abstract class. In our example, the abstract classes chosen as roots of the class hierarchies in the sub-designs are, respectively, PayementStrategy and Orders. Moreover, for each class belonging to the obtained hierarchies, the decomposition collects its associated classes.

The third step is the semantic and structural filtering of the sub-designs. After decomposing the design fragment the filtering process is performed for each sub-design. Due to space limitation, in the remainder of the paper, we will present the filtering of the sub design 1.

The semantic filtering uses *semantic_coverage* metric. The *semantic_coverage* is the count of classes which are related to keywords, multiplied by a weighting factor associated with every class. Figure 6 illustrates a snapshot of *PatternSniffer* presenting the values of *semantic_coverage* metrics calculated for sub design 1.

In order to calculate the *semantic_coverage* of the sub-design 1 for the Strategy design pattern, we are interested in the classes, attributes and methods' semantic aspect.

Fig. 5 *PatternSniffer* the 2 sub-designs, decomposition menu and pattern list



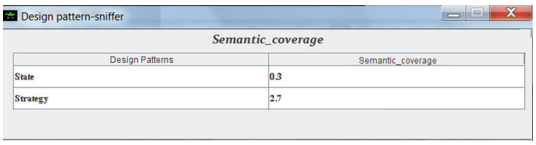


Fig. 6 Semantic_coverage metric of the sub design 2

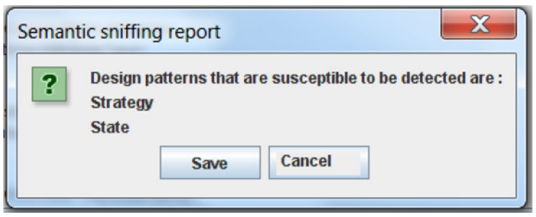


Fig. 7 Semantic filtering report

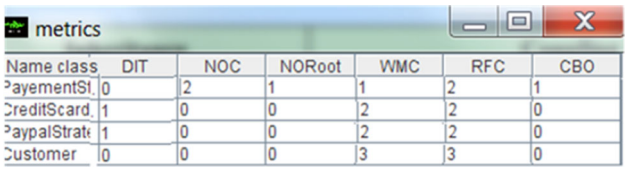


Fig. 8 Calculated metrics on the sub design 1

Using the WordNet dictionary, we find that: Name_includ (PayementStrategy,"Strategy",), Name_includ(CreditCard Strategy,"Strategy"), Name_includ(PaypalStrategy,"Strat egy"), Meth_Name_includ("Strategy",CreditCardStrategy) and Meth_Name_includ("Strategy",PaypalStrategy). The *semantic_coverage*(SD2,Strategy) = 3 * 70% + 2 * 30 % = 2.7 In the same way, the *semantic_coverage* of the sub design 1 for the State design pattern is automatically calculated: Meth_Def_Contain("State", UpdateProfile). The *semantic_coverage* (SD1, State) = 0 * 70 % + 1 * 30 % = 0.3. As shown in Fig. 7, the final list will be organized in terms of the *semantic_coverage* (suitability) of each design pattern.

After the semantic filtering, our tool calculates the metrics values that are necessary for the structural phase. Figure 8 shows the values of calculated metrics.

Figure 8 illustrates a screen shot of *PatternSniffer* presenting the calculated metrics for the sub design 1. Note that the structural filtering proves that the sub-design 1 is likely to contain the strategy pattern. In fact, when calculating the C&K metrics for the classes of the sub-design, we find that NORoot is equal to 1 and the DIT values of all classes of the design are not very high, they are in the interval [0,1] and the NOC of the classPayementStrategy is equal to 2. The CBO of the class PayementStrategy is low, it is equal to 1, and the WMC of the classes CreditCardStrategy and PaypalStrategy is considered low since it equals 2.

5 Evaluation of the effects of design patterns filtering on their identification

In this section, we briefly overview the MAPeD [12] approach for design patterns identification and present its evaluation on the open source system JUnit [34] without the filtering phase. Afterwards, we reevaluate MAPeD applied after the filtering and compare the identification results in terms of recall and precision. The choice of MAPeD is motivated by its advantage of tolerating structural and dynamic differences between the examined design and the identified pattern. In addition, MAPeD distinguishes the important concepts representing the essence of a design pattern from its optional concepts. Furthermore, it can be used to identify both the structure and the behavior of the pattern.

MAPeD allows design patterns identification through the following three main steps:

1. Decomposition of the design into sub-designs in order to manage the time complexity of the pattern identification. This step cuts the design into parts without losing any structural information. It reduces the searched space in the design.
2. Identification of the structural features of the design pattern (the classes, generalizations, aggregations, compositions, etc.) and identification of the method declarations to confirm the determined class correspondence.
3. Identification of the dynamic aspect of the design pattern. It relies on the correspondence results determined and confirmed in the second step. It also adapts an XML document retrieval approach to examine the conformity of the design behavior to that of the pattern. It supposes that the behavior is specified in terms of sequence diagrams.

In order to assess the efficiency of MAPeD, we performed an experimental evaluation on the open source system Junit v3.7 which contains many patterns. Table 4 shows the number of patterns that exist in JUnit v3.7, the number of true patterns found by MAPeD, and the precision and recall ratios.

To facilitate the presentation of results, we adopt the following notations:

- T: the number of true design pattern instances;
- I: the number of patterns identified by our approach;
- E: the number of existing real design pattern
- P: Precision;
- C: Recall;
- TP: true or correct design pattern instances found by MAPeD;
- FP: false or incorrect design pattern instances found by MAPeD;

Table 4 Design patterns identification results in JUnit v3.7

Design patterns	I	T	E	P	R	TP	FP	FN
Abstract Factory	0	0	0	100	100	0	0	0
Adapter	9	0	0	0	100	0	9	0
Command	8	0	0	0	100	0	8	0
Composite	1	1	1	100	100	1	0	0
Decorator	1	1	1	100	100	1	0	0
Factory method	23	0	0	0	100	0	23	0
Observer	4	1	1	25	100	1	3	0
Prototype	0	0	0	100	100	0	0	0
Singleton	0	0	0	100	100	0	0	0
State/strategy	8	0	0	0	100	0	8	0
Template method	11	0	0	0	100	0	11	0
Visitor	0	0	0	100	100	0	0	0
Average				52.1	100			

Table 5 Design patterns identification results in JUnit v3.7 after filtering

Design patterns	I	T	E	P (%)	R (%)	TP	FP	FN
Abstract Factory	0	0	0	100	100	0	0	0
Adapter	0	0	0	100	100	0	0	0
Command	0	0	0	100	100	0	0	0
Composite	1	1	1	100	100	1	0	0
Decorator	1	1	1	100	100	1	0	0
Factory method	0	0	0	100	100	0	0	0
Observer	1	1	1	100	100	1	0	0
Prototype	0	0	0	100	100	0	0	0
Singleton	0	0	0	100	100	0	0	0
State/stategy	0	0	0	100	100	0	0	0
Template method	0	0	0	100	100	0	0	0
Visitor	0	0	0	100	100	0	0	0
Average				100	100			

– FN: true or correct design pattern instances not found by MAPeD.

In the MAPeD evaluation illustrated in Table 4, the value of the precision is 52.1 % which is explained by the fact that we found some false positive classes (i.e., incorrectly detected classes). For example, MAPeD identifies nine incorrect instances of the Adapter design pattern. This is in part due to the fact that MAPeD identifies the patterns in an ad hoc manner since it applies the approach on all the patterns without any order or probability of existence. It applies also the identification approach even if no pattern exists.

To overcome this problem of false positives, we propose using a filtering approach before starting the detection. Indeed, when we applied our filtering process of JUnit v3.7, the first (semantic) step returns a negative report indicating that no Adapter design pattern is susceptible to be detected.

Table 5 shows the evaluation of MAPeD applied after using our filtering approach. Thanks to the filtering, MAPeD eliminates all the false negatives to get the highest precision (100 %) without losing any design patterns (the recall ratio is at its maximum).

Figure 9 summarizes the precision values of the MAPeD with and without filtering for all the design patterns on JUnit v3.7. This experimental evaluations shows that the performance of MAPeD is improved on average by 47.9 % once it is applied after our filtering approach. This improvement is explained by the elimination of the false negatives. In this experiment, the false negatives are semantically eliminated in the first filtering step.

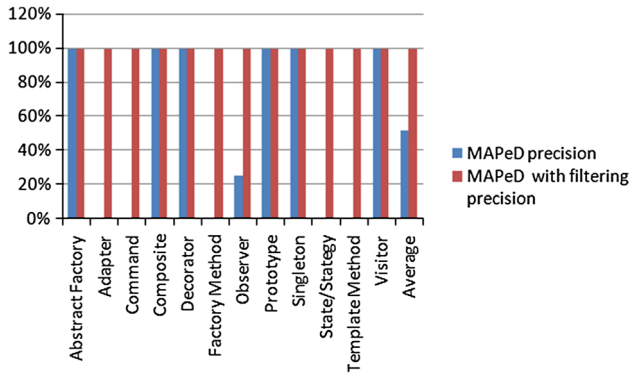


Fig. 9 Comparison of MAPeD results with and without filtering

6 Conclusion

Design patterns filtering consists of detecting design patterns symptoms. These latter encode the design pattern problem through structural and semantic concepts. We propose to use filtering as a step prior to any existing design patterns identification method, all of which suffer from a high temporal complexity and limited performance (precision and recall ratios).

This paper proposes a new filtering approach that uses a set of structural metrics proposed by C&K [9] and defines a *semantic-coverage* metric to determine the most probable correspondences between the design elements and the design patterns. The proposed filtering relies on a characterization of the design patterns symptoms in terms of structural metrics whose thresholds are determined empirically on a set of open source systems. In addition, it exploits the semantics encoded in the names of the design patterns' elements to order the list of candidate design patterns according to their *semantic_coverage* within the design.

The performance gains of our filtering approach are shown through an experimental evaluation with the MAPeD design patterns identification method. This experimental evaluation on the JUnit v3.7 open source application shows that, when preceded by our filtering approach, MAPeD gains an average of 47.9 % in its precision without losing in its recall ratio. The precision improvement is due to the elimination of false negatives—non-existing design patterns which were discarded in the filtering. Besides this performance gain, thanks to the filtering, the application of MAPeD is accelerated for two reasons: MAPeD is applied for only those design patterns that the filtering identified as candidate (as oppose to all design patterns) and according to their degree of semantic similarity to the design; and the application of MAPeD is delimited to the design fragment where the filtering identified candidate patterns (as opposed to all the design).

While the herein presented evaluation shows encouraging results, it is important to generalize the above-mentioned gains through a larger set of case studies. On the one hand, such extended experimental evaluation would validate the performance of our filtering approach in terms of recall and precision; on the other hand, it would allow us to complete the characterization of the structural symptoms of design patterns through the set of OO design metrics.

References

1. Gamma E, Helm R, Johnson R, Vlissides J (1995) Design patterns: elements of reusable object oriented software. Addisson-Wesley, Reading
2. Tsantalis N, Chatzigeorgiou A, Stephanides G, Halkidis ST (2006) Design pattern detection using similarity scoring. IEEE Transactions on Software Engineering, vol 32, p 11
3. Gueheneuc Y, Antoniol G (2008) DeMIMA: a multilayered approach for design pattern identification. IEEE Transactions on Software Engineering
4. Belderrar A, Kpodjedo S, Guéhéneuc Y, Antoniol G, Galinier P (2011) Sub-graph mining: identifying micro-architectures in evolving object-oriented software. 15th European Conference on Software Maintenance and Reengineering (CSMR'11), pp 171–180
5. De Lucia A, Deufemia V, Gravino C, Risi M (2010) Improving behavioral design pattern detection through model checking. 14th European conference on software maintenance and reengineering (CSMR'10), pp 176–185
6. Bouassida N, Ben-Abdallah H (2009) Structural and behavioral detection of design patterns. (ASEA). International conference on advanced software engineering and its applications, CCIS proceedings. Springer, Berlin
7. Masuda G, Sakamoto N, K Ushijima (1999) Evaluation and analysis of applying design patterns. In: Proceedings of the international workshop on principles of software evolution
8. Huston B (2001) The effects of design pattern application on metric scores, pp 261–269
9. Chidamber S, Kemerer C (1994) A metrics suite for object oriented design. IEEE Transactions on Software Engineering, pp 476–493
10. Gamma E, Eggenschwiler T (2007) http://www.jhotdraw.org
11. JRefactory (2007) http://jrefactory.sourceforge.net/
12. Bouassida N, Ben-Abdallah H, Issaoui I (2013) Evaluation of an automated multi-phase approach for patterns discovery. Int J Softw Eng Knowl Eng
13. Arcelli F, Maggioni S (2009) Metrics-based detection of micro patterns to improve the assessment of software quality. 1st Symposium on emerging trends in software metrics
14. Lee H, Youn H, Lee E (2008) A design pattern detection technique that aids reverse engineering. Int J Secur Appl 2(1)
15. Blondel VD, Gajardo A, Heymans M, Senellart P, Van Dooren P (2004) A measure of similarity between graph vertices. SIAM, Applications to synonym extraction and web searching
16. Arcelli F, Perin F, Raibulet C, Ravani S (2009) JADEPT: dynamic analysis for behavioral design pattern detection. 4th International conference on evaluation of novel approaches to software engineering (ENASE'09), Milan, pp 95–106
17. Aydinoz B (2006) The effect of design patterns on object oriented metrics and software error-proneness. MS Thesis, The Graduate School of Natural and Applied Sciences of Middle East Technical University
18. Hernandez J, Kubo A, Washizaki H (2011) Selection of metrics for predicting the appropriate application of design patterns. AsianPLoP 2011: 2nd Asian conference on pattern languages of programs, Tokyo

19. Abul Khaer M, Hashem M, Raihan Masud M (2008) On use of design patterns in empirical assessment of software design quality. In: Proceedings of the international conference on computer and communication engineering
20. Hsueh N, Chu P, Chu W (2008) A quantitative approach for evaluating the quality of design patterns, pp 1430–1439
21. Maggioni S, Arcelli F (2009) Metrics-based detection of micro patterns. Emerging Trends in Software Metrics
22. Reißing R (2001) The impact of pattern use on design quality. OOPSLA 2001 workshop "Beyond Design: Patterns (mis)used"
23. Ampatzoglou A, Charalampidou S, Stamelos I (2013) Research state of the art on GoF design patterns: a mapping study. J Syst Softw, pp 1945–1964
24. Antoniol G, Fiutem R, Cristoforetti L (1998) Using metrics to identify design patterns in object-oriented software. In: IEEE proceedings of the 5th international symposium on software metrics (METRICS 1998). IEEE Computer Society, Maryland, pp 23–34
25. Ampatzoglou A, Frantzeskou G, Stamelos I (2012) A methodology to assess the impact of design patterns on software quality. J Inf Softw Technol, 331–346
26. Larman C (2004) Applying UML and patterns: an introduction to object-oriented analysis and design and iterative development, 3rd edn. Addison Wesley
27. Ayata M (2010) Effect of some software design patterns on real time software performance. A Master's Thesis, the Graduate School of Imfomatics of Middle East Technical University
28. Vernazza T, Granatella G, Succi G, Benedicenti L, Mintchev M (2000) Defining metrics for software components. In: Proceedings of the world multiconference on systemics, cybernetics and informatics
29. Guéhéneuc Y-G, Sahraoui H, Zaidi F (2004) Fingerprinting design patterns. In: Proceedings of the 11th working conference on reverse engineering. IEEE Computer Society Press, pp 172–181
30. Bieman J, Straw G, Wang H, Munger PW, Alexander RT (2003) Design patterns and change proneness: an examination of five evolving systems. In: Berry M, Harrison W (eds) Proceedings of the 9th international software metrics symposium. IEEE Computer Society Press, pp 40–49
31. Issaoui I, Bouassida N, Ben-Abdallah H (2012) A design pattern detection approach based on semantics. In: Proceedings of the the 10th international conference on software engineering research, management and applications (SERA2012) selected papers to appear in Studies in Computational Intelligence. Springer, Berlin
32. Fellbaum C http://wordnet.princeton.edu/ [En ligne]
33. ArgoUML (2000) http://argouml.tigris.org/ [En ligne]
34. JUnit (2007) http://www.junit.org