Predicting the Existence of Design Patterns based on Semantics and Metrics

Imène Issaoui[1], Nadia Bouassida[2], and Hanêne Ben-Abdallah[3]
[1]Institut Preparatory to Engineering Studies, University of Monastir, Tunisia
[2]Department of Computer Science, University of Sfax, Tunisia
[3]Faculty of Computing and Information Technology, King Abdul-Aziz University, KSA

Abstract: *As part of the reengineering process, the identification of design patterns offers important information to the designer. In fact, the identification of implemented design patterns could be useful for the comprehension of an existing design and provides the grounds for further code/design improvements. However, existing pattern detection approaches generally have problems in detecting patterns in an optimal manner. They either detect exact pattern instantiations or have no guidelines in deciding which pattern to look for first amongst the various patterns. To overcome these two limitations, we propose to optimize any pattern detection approach by preceding it by a preliminary "sniffing" step that detects the potential existence of patterns and orders the candidate patterns in terms of their degree of resemblance to design fragments. Our approach uses design metrics to characterize the structure and semantics of the various design patterns.*

Keywords: *Metrics, design pattern, quality assurance, sniffer.*

Received September 3, 2013; accepted April 27, 2014; published online April 1, 2015

1. Introduction

The advantages of design patterns [17] as good quality generic solutions for recurring problems have been widely accepted. Reusing design patterns accelerates the development process and reduces the cost of software development. In addition, it improves the quality of the design and the produced code. Furthermore, the identification of instantiated design patterns could be useful for the comprehension of an existing design and/or code.

These advantages motivated the proposition of different approaches for design pattern detection [10, 11, 16, 18, 33]. Existing design pattern identification approaches tackled the problem in different manners: Some are based on graph matching techniques [10, 33] others are based on constraint satisfaction problems [18] yet others are based on XML retrieval techniques [13]. One common weakness of these approaches is their assumption that the designer knows which pattern he/she is trying to find in the design. In other words, if this assumption does not hold, the existing approaches would have to try the identification of all patterns in an ad hoc manner, even if no pattern exists. Evidently, this is impractical given their high time complexity (exponential in terms of the size of the design). A more judicious identification should have a strategy to eliminate those patterns that may not exist and order those 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 method that can be applied as a preliminary step to any existing pattern identification approach to make it more efficient (in time). To do so, our method: Filters the design patterns that are probably present in the design; orders those candidate patterns in terms of their degrees of resemblance to the design fragments and delimits the design fragment susceptible of containing each candidate pattern. As a consequence, any pattern detection approach will detect patterns beginning by those that are most likely to be present.

Our method relies on the fact that any pattern has an intention and that it improves particular aspects of design quality. Thus, it tries to capture the pattern intention through a set of structural and semantic metrics. These metrics are hence used to "sniff"/predict the existence of patterns in a design. We think that an appropriate pattern instance is one that respects (i.e., does not fall under) the thresholds of the values of the metrics which reflect the intention of the pattern. For instance, the pattern Mediator is devoted to reduce coupling [30] while it centralizes control in the class Concrete Mediator whose complexity becomes high. In addition, among the classes of this pattern, the class Mediator has a high Coupling Between Objects (CBO) [22]. Hence, if the CBO is low for all the classes of a given design, then the pattern Mediator does not exist and it is useless to try to identify it in such a design. Besides existing structural design metrics [11, 15], we propose a new metric, called semantic coverage, to measure the semantic similarity between design patterns and design fragments. This semantic metric refines the ranking of candidate patterns determined through the structural metrics.

Evidently, the thresholds of the metrics highly influence the performance of our prediction method. To determine the thresholds that reflect the intention of the design pattern, we conducted an empirical study on three open source systems JHotDraw v5.1, JRefactoty v1.0 and JUnit v3.7 [24, 25, 26].

The remainder of this paper is organized as follows: Section 2 overviews currently proposed approaches for pattern identification and works that relate object oriented metrics to design patterns, section 3 highlights the basic steps of our prediction method, section 4 illustrates our approach through an example and using the pattern sniffer toolset and section 5 summarizes the paper and outlines our future work.

2. Related Works

2.1. Design Pattern Identification Methods

Many researchers were interested in proposing approaches that recover design pattern instances by performing a static analysis of the design cf., [10, 18]. Other approaches carried out the design pattern identification using either dynamic analysis or a combination of static and dynamic analyses [5, 16].

Within the static analysis approaches, Gueheneuc and Antoniol [18] proposed a multilayered approach for design motif identification based on constraint satisfaction problems. One advantage of this approach (called 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 behavioural and semantic aspects. Also, among the static analysis approaches, Tsantalis *et al.* [33] consider that recognizing a pattern in a design is a graph matching problem. They propose a design pattern detection approach based on similarity scoring [12] 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.* [10]. SGFinder derives OO micro-architectures from the class diagram; a micro-architecture represents a connected sub-graph induced from the design's class diagram.

Some approaches combine the static analysis with the dynamic analysis technique, as an example, De Lucia *et al.* [16] identify behavioural design patterns. First, a static analysis is used to identify candidate instances of a behavioural 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 behaviours expressed in terms of monitoring grammars.

Also, combining static and dynamic analyses, Bouassida and Ben-Abdallah [13] propose an identification approach 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. Their approach relies on a context resemblance function to compute the similarity potential between the design structure and behaviour and the pattern.

Within the dynamic analysis approaches, Arcelli *et al.* [6] propose a design pattern detection technique for Java codes. The approach relies on data collection through Java Platform Debugger Architecture (JPDA). It applies a set of rules to detect behavioural design patterns.

Overall, depending on their technique, the proposed methods detect structural, creational and/or behavioural design patterns with different precision rates. They have two common limitations: A high (exponential) time complexity and the assumption that the designer knows which pattern he/she is trying to identify. To overcome the first limitation in practice, some methods presume that the design is fragmented (manually or automatically) before the identification is launched. The second limitation will be addressed for the first time in this paper.

2.2. Design Patterns and Design Metrics

Several works examined the relationships between design patterns and OO design metrics [1, 2, 3, 4, 9, 20, 21, 22, 29, 30, 31]. Some works were interested in patterns detection using design metrics. Others were interested in studying the effects of design patterns on the design quality through various metrics.

On the other hand, many works focus on evaluating the impact of patterns reuse on quality. For example, Reißing [31] applies classic OO design metrics to two similar designs A and B. Knowing that B uses design patterns and A does not use them, the metrics show that the design A is better than B because it has less classes, operations, inheritance, associations, etc., for this reason, Reibing [31] 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.* [30] use the C and K metrics suite of [15] to evaluate the efficiency of applying design patterns in two applications developed by their research group. They show that particular design patterns have a tendency to make a particular metric value poorer. However, they caution that this does not necessarily mean that those design patterns always induce quality degradation. In addition, Masuda *et al.* [30] presume that the Weighted Methods per Class (WMC), Depth of Inheritance (DIT), Number of Children (NOC) and Lack of Cohesion in Methods (LCOM) metrics are essentially used for single classes. Hence, they are not suitable to measure the relationship among classes. Only the Response For Call (RFC) and CBO metrics can capture the degree of communication between classes. However, the RFC and CBO metrics reflect the one-to-many relationship. Instead, Masuda *et al*. [30] suggest that new metrics should be proposed for the evaluation of the efficiency of applying design patterns. The main limitation of this work is that the authors used small applications which are developed by their research group to draw their conclusions.

Ampatzoglou *et al.* [3] propose an approach for comparing design patterns to alternative designs with an analytical method. In fact the designs are compared based on their possible number of classes and on equations representing the values of the various structural quality attributes as a function of these numbers of classes. For this purpose, the authors needed to predict quality attributes from code and design measurements. They choose code/design metrics and a model to calculate quality attributes from them and according to the proposed thresholds, the designer can opt for the design pattern solution.

3. A New Approach for Design Patterns Prediction

As mentioned in the introduction, the main goal of our design pattern "sniffing" 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 sniffing is meant to be applied before the identification process for two purposes: Limit the list of candidate design patterns, delimit the design fragments which are structurally and semantically susceptible to include patterns. Because it uses both semantic and syntactic information, our method can further assist the detection approaches by giving them the probable matches between the design elements and the pattern elements. This information can be used by the designers to improve their designs, for instance, by reading the pattern documentation in order to add/remove/rename/restructure their design fragments so that they fit the pattern problem. Once such modifications are done, the pattern would be reused according to its intention, which improves the quality of the design as discussed in the various works in the literature cf. [5, 18, 34].

Our design pattern sniffing method operates in two analysis steps: A semantic analysis that is followed by a structural analysis only when the first step succeeds in identifying design pattern candidates. Both steps use a filtering technique to determine an ordered list of candidate design patterns; the order reflects the degree of resemblance between the design patterns and design fragments.

3.1. Sniffing Patterns Semantically

Semantic sniffing 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 used in defining the pattern elements. Being well chosen, these names characterize the semantics encoded in 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. The lists of weighted names characterizing the design patterns are manually constructed based on their documentation. These lists are used to compare them with the names used in the design.

More specifically, we propose the new metric, called semantic coverage, to count the number of classes and method names in the design that are "related" to the 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 [34].

3.1.1. Class Name Semantic Relationships

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

- Is_a kind_of (*C*, *K*): Implies that there is a semantic relationship between the class *C* indicating that *C* a type or a variation of the keyword K. Example: Is_a kind_of (student, intellectual).
- Is_one way_to (*C*, *K*): Implies that there is a semantic relationship between the class *C* and keyword *K*. *C* is one of several manners to do *K*. Example: Is_one_way_to (help, support).
- Synonym (*C*, *K*): Implies that the name *C* is a synonym of the name *K*. Example: Synonym (student, pupil).
- Inter_Def (*C*, *K*): Implies that the definitions *C* and K given by WordNet dictionary [36] have common words. The common words list 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 a certain keyword K. Example: Def_Contain (Paper, Observation).
- Name_Includ (*C*, *K*): Implies that the name *C* includes the name *K*. *C* is a string extension of the name of the class *K*. Example: Name_Includ ("XWindow", IconXWindow).

Note that, the semantic criterion Is_a kind_of, Is_one way_to and Synonym exist, already, in the Word Net dictionary [36].

3.1.2. Method Name Semantic Relationships

The method name comparison criterion explicit the relation between the methods names in the design and the keywords characterizing the design patterns:

- Synonym_Meth (*K*, MC1): Implies that the method MC1 of the class C1 in the design is identical or synonym to the keyword K, i.e., they have the same or synonym names.
- Inter_Def_Meth (*K*, MC1): Implies that the definition of the keyword K and the definition of the method MC1 of the class *C*1 in the design have common words.
- Meth_name_Includ (*K*, MC1): Implies that the name of the method MC1 of the class *C*1 contains the keyword *K*.
- Meth_Def_Contain (*K*, MC1): Implies that the definition of the method MC1 of the class *C*1 contains the keyword *K*.

3.1.3. Semantic Coverage Metric

The semantic coverage metric is used to calculate the linguistic similarity between a design fragment (represented as a set of classes and set of methods names) and a particular design (represented as a set of characterizing key words, collected manually from its documentation). It is calculated as a weighted sum of the number of class names matches and method names matches. The weights are needed to reflect the fact that a class name carries more or coarser semantic information than its method names.

Let D be a design, C_D be a set of classes in the design D and M_D a set of methods in the design D and let K_P be the set of names characterizing a pattern P. The semantic coverage metric between the design D and the pattern P is calculated as follows:

$$SemanticCoverage\ (D, P) = w_C * |C_D\ Rel\ K_P| + w_M * |M_D\ Rel\ K_P|$$

Where $|C_D\ Rel\ K_P|$: 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, $|M_D\ Rel\ K_P|$: 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.

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

3.2. Sniffing Patterns Syntactically

Once the semantic sniffing produces a list of candidate patterns potentially present in a design, we proceed with the syntactic sniffing step. For this step, we characterize the design properties (inheritance, coupling, complexity, …) of design patterns through a correlated set of symptoms that can be captured by a set of metrics. Using the list of candidate patterns provided by the semantic sniffing step, we calculate the values of the syntactic metrics for the design fragment that are susceptible to be candidate patterns only. We check if these values are coherent with the metric's thresholds that reflect the intention of the design pattern. If that is the case, we presume that the pattern has a high probability of being present in the design.

Before illustrating this step, we first overview the metrics we used. Secondly, we propose a characterization of design patterns in terms of a set of metrics Table 1. Finally, we show how we determined empirically the thresholds of these metrics.

Table 1. Measuring the quality of design patterns with metric.

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 [21, 22]	Increase reusability [11]	DIT value does not become very high while applying design patterns. [31]	high for the Class Colleague	high Class Mediaor [30]	high for the class Concrete Mediaor [30]	high for the class Concrete Mediator [30]	high for the class Concrete Mediator [30]
Command	decrease complexity [17]	increase flexible [17]		high for the class Command [30]	low for the class Invoker	low for the class Invoker	low for the class Concrete Command [30]	-
Strategy	increase cohesion [9,9] decrease complexity [9]	increase flexibility increase maintainability [17] increase Polymorphism [21]		high for the class Strategy [30]	low for the class Strategy	low for the class Concrete Strategy	low for the class Concrete Strategy and context [8, 9, 30]	-
State	increase cohesion [28] decrease complexity	increase flexibility [17] increase polymorphism [21]		high for the class State [31]	low for the class State	low for the class Concrete State	Low for the class Concrete State [8]	does not change when applying State [31]
Factory Method	decrease complexity [30]	increase flexibility[17]		high for the classes Product and Creator	high for the class Concrete Creator [30]	high for the class Concrete Product	low for the class Concrete Factory [30]	-
Visitor	decrease complexity [9] increase polymorphism [17]			high for the class Visitor [30]	low for the class Concret Visitor	high for the class Concrete Visitor [30]	Low for the class element [9]	-
Abstract Factory	increase complexity [35] decrease coupling; [21]	increase reusability [17]		high for the classes Abstract Factory and Abstract Product [30, 35]	high for the class Concrete Factory [30, 35]	high for the classes Abstrac tFactory and Concrete Factory [35]	high for the classes Abstract Factory and Concrete Factory [35]	

3.2.1. The Syntactic Metrics Used

We retained from the metrics suite of [15, 27] the following set of metrics: DIT, NOC, CBO, RFC and WMC.

In addition, to the above existing metrics, we propose a new metric to filter further candidate patterns. The new metric, Number Of Roots (NORoot), calculates the number of super classes. This metric is inspired from our note that the number of hierarchies is one salient characteristic of GoF design patterns [17]. For instance, State and Strategy have a single hierarchy. Mediator and observer have two hierarchies, while AbstractFactory has three hierarchies.

3.2.2. Characterizing Patterns Through Metrics

As demonstrated in [1, 17, 21, 30] patterns do not improve all quality aspects, but each pattern is devoted to a certain quality aspect. For example, the Strategy pattern mainly promotes polymorphism [28] in addition, each created subclass is focused on only one job, this in turn increases cohesion. In fact, Strategy promotes the "low cohesion principle" while it reduces complexity [28]. Translated into metrics, these characteristics mean that Strategy reduces WMC [8] and decreases CBO and RFC. As a consequence, if the CBO and the RFC are high then the Strategy pattern does not exist and it is useless to try to identify it.

We synthesize the relationships of design patterns with OO design properties. In our opinion, an appropriate instance is one that respects the values of metrics required as shown in Table 1.

3.2.3. Metrics Threshold Values

Choosing useful metrics and relating them to patterns is not enough to ensure that our approach is beneficial, it is necessary to fix the threshold values which highly influence the efficiency of the sniffing. 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.

Table 2 shows threshold values for the CK metric suite [14]. The WMC threshold limit is set to 15 per class. The works in [14, 32] suggest a threshold value of 6 for DIT and NOC. The threshold limit for CBO metric is set to 8 per class. For the RFC metric the threshold limit is set to 35 per class, finally the threshold limit for the LCOM metric is set to 1 per class. Note that, a metric value is considered high if it is greater than half of its threshold and it is considered low if it is lower than half of its threshold fixed in Table 2.

Table 2. Threshold values for the CK metric suite [15].

Metric	Threshold
WMC	0-15
DIT	0-6
NOC	0-6
CBO	0-8
RFC	0-35
LCOM	0-1

3.2.4. The Empirical Study

We noticed that it is essential to make an empirical study on open source projects and software architectures reusing different and combined design patterns to determine the right thresholds values for metrics: JHotDraw v5.1, JRefactoty v1.0 and JUnit v3.7 [24, 25, 26] as shown in Tables 3, 4 and 5.

JHotDraw v5.1 [24] is a two-dimensional graphics framework for structured drawing editors. It involves many pattern occurrences. JRefactory v1.0 [25] is a tool designed to refactor and restructure Java source files. JUnit v3.7 [26] is a unit-test framework developed to ease the implementation and running of unit tests for Java systems.

Table 3. Calculated metrics for the JUnit v3.7.

JUnit v3.7													
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**
Composite	1	Component	1	0		3		3		30		2	
		Composite		1	2	0	1	1	1	4	48	4	20
		Leaf		1	3	0	2	0	0	4	57	2	24
		Client		0		0		1		21	95	21	
Decorator	1	Component	1	0		3		1		71		2	
		Concrete Component		1	3	0	2	0	0	5	44	0	24
		Decorator		0	0	2		1	0	32		2	
		Concrete Decorator		1	2	0	2	0	0	17	18	4	6
Iterator	1	Aggregate	2	0		1	2	1		2		1	
		Concrete Aggregate		1	2	0	0	2		4	10	2	6
		Iterator		0		1		1		2		0	
		Concrete Iterator		1		0	0	4		4		2	
		Client		0		0		2		11		9	
Observer	3	Subject	2	0	1	1	3	1		27	71	0	6
		Concrete Subject		1	2	0	0	2	3	9	44	9	21
		Observer		0	0	1	3	1		2	5	0	4
		Concrete Observer		1	2	0	0	1		4	83	0	21
Singleton	2	Singleton	0	0	0	0	0	0	0	2	18	0	2

Table 4. Calculated metrics for the JRefactory v1.0.

JRefectory v1.0													
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	17	Target	1	0		1	10	1	3	11		1	11
		Adapter		1		0	0	1		2	21	1	11
		Client		0	3	0	1	1		4	83	1	33
		Adaptee		0	2	0	4	1		1	71	1	33
Builder	2	Director	1	0	1	0	0	1	1	14	35	12	32
		Builder		0	0	1	1	1	1	6	20	4	16
		ConcreteBuilder		1	1	0	0	1	1	6	15	4	10
		Product		0	0	0	1	1	1	2	15	0	10
FactoryMethod	1	Product	2	0		1		0		6		0	
		ConcreteProduct		1		0		1		7		7	
		Creator		0		1		0		11		1	
		ConcreteCreator		1		0		1		6		6	
Singleton	2	Singleton	0	0	0	0	0	0	0	6	18	6	
State	2	State	1	0	0	5	10	1	1	7	7	9	14
		ConcreteState		1	2	0		0	0	3	25	3	8
		Context		0	1	0		1	1	12	83	4	29
Vistor	2	Visitor	2	0	0	6		1		13	90	13	86
		ConcreteVisitor1		1	4	0	1	0	0	3	106	2	106
		Clement		0	0	9		1		10	15	9	10
		ConcreteElement		1	2	0	8	0	0	4	60	2	33

Table 5. Calculated metrics for the JHotDraw v5.1.

JHotDraw v5.1													
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	1	Target	1	0		1		0		11		11	
		Adapter		1	4	0	1	1	2	2	21	2	10
		Client		0	2	0	0	1	4	4	83	3	61
		Adaptee		0	4	0	2	0	1	1	71		
Command	1	Command	1	0		9		4		4		4	
		ConcreteCommand		1	2	0		2	2	2	4	2	3
		Invoker		0	2	0	0	1	2	5	19	2	61
		Receiver		0	1	0	2	2	2	3	71	3	40
		Client		0	2	0	4	2	5	36	95	16	61
Composite	2	Component	1	0	1	2		9		71		32	
		Composite		3	5	0	3	1	1	4	48	4	33
		Leaf		1	6	0	2	0	2	4	57	3	38
		Client		0	5	0	4	1	1	2	95	2	52
Decorator	1	Componenet	1	0	1	2		9		71		32	
		ConcreteComponent		1	5	0	2	0	1	5	44	5	34
		Decorator		1	3	2		1		32		31	
		ConcreteDecorator		2	4	0	0	0	0	17	18	8	10
FactoryMethod	3	Product	2	0	1	4	17	0	0	6	11	6	11
		ConcreteProduct		2	4	0	3	1	1	4	11	2	8
		Creator		1	1	11	26	0	3	11	71	11	32
		ConcreteCreator		1	6	0	4	1	1	12	71	7	38
Observer	2	Subject	2	0	1	1	2	1	4	27	71	27	32
		ConcreteSubject		1	6	0	2	0	1	4	44	4	34
		Observer		0	0	1	4	0	0	2	5	2	5
		ConcreteObserver		1	5	0	0	1	5	4	83	1	38
Prototype	2	Prototype	1	1	2		1	0	4	15	71	15	32
		ConcretePrortype		2	6	0	2	0	2	4	57	5	38
		Client		1	3	0	2	1	1	9	71	8	18
Singleton	2	Singleton	0	0	0	0	0	0	0	4	18	4	9
State	2	State	1	0	0	1	1	1	2	7	7	7	7
		ConcreteState		1	4	0	11	0	0	3	25	2	18
		Context		2	2	0	1	1	1	12	83	7	61
Strategy	2	Strategy	1	0	1	1	3	1	2	1	6	1	6
		ConcreteStrategy		1	4	0	3	0	0	1	11	1	8
		Context		1	4	0	2	1	1	5	83	5	61
TemplateMethod	2	AbstractClass	1	0	2	4	6	0	1	26	42	12	35
		ConcreteClass		1	6	0	2	0	2	5	57	4	61

Tables 3, 4 and 5 present the patterns existing in each open source systems, their number of occurrences and the calculated metrics for each design pattern. Note that, for these open source systems, the pattern instances are extracted from documentation and also, identified manually by experts as presented in [11, 19]. We noticed from Tables 3, 4 and 5 that, for each design pattern, we got different metric's values which are in perfect agreement with the thresholds shown in Table 5. For example, for the command instance in the JHotDraw v5.1, 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, 2]) and the NOC of the class playing the role of Command is high (it is equal to 9). The CBO of the classes playing the role of Invoker is low (in the interval [1, 2]), the RFC of the classes playing the Invoker role is low (in the interval [5, 19]), and the WMC of the classes playing the role of Concrete Command is low (in the interval [2, 3]). Hence, to detect the Command pattern, we apply the following rule:

- RuleCmd: The Command design pattern is susceptible to be present in a design fragment D, if there is a set of classes, C_1, …, C_n, in to the design fragment D such that:

  {*NORoot* (*D*)=1}
  {*0<DIT*(*C*1, ..., *Cn*)< 3}
  {*Synonym*(*C*1,*Command*) *or Name_Includ* (*"Command"*, *C*1) *and* (2*<NOC(C*1*)<*9)}
  {*Synonym*(*C*2*, Invoker) or Name_Includ*(*"Invoker"*, *C*2) *and/or* (0*<RFC*(*C*2)<17)} *and* {(0*<CBO*(*C*2)<4)} *and* {*Synonym* (*C*3*, ConcreteCommand*) *or Name_Includ* (*"ConcreteCommand"*, *C*3) *and* (2< *WMC*(*C*2< 3)}.

3.2.5. Example: Syntactic Sniffing of the Command Design Pattern

In the semantic sniffing of the design D of Figure 1 with the command design pattern, we calculate the semantic coverage metric. Semantic_Coverage (D, Command)=5. After the semantic sniffing, we proceed to the syntactic phase. To illustrate the syntactic sniffing of the Command design pattern, we apply the rule RuleCmd which allows the detection of the Command design pattern.

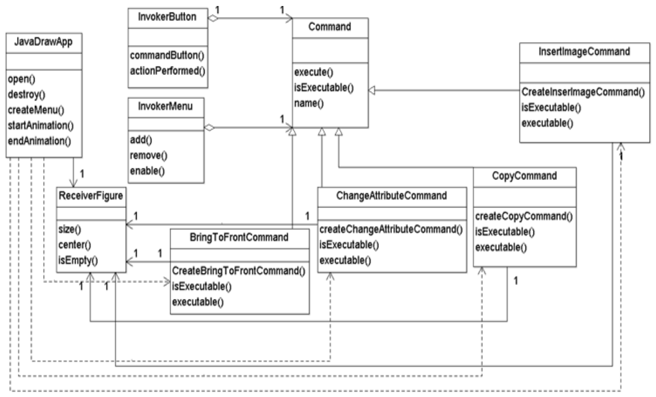


Figure 1. Design example showing an application instantiating the design pattern.

Using class name semantic relationships, we found that: Synonym (Command, Command) then the Command class is identified as: Command, Name_Includ ("Invoker", InvokerButton) Name_Includ ("Invoker", InvokerMenu) therefore, the classes InvokerButton and InvokerMenu are identified as Invoker. In the same manner the classes Bring To Front Command, Change Attribute Command, Copy Command and Insert Image Command are identified as Concrete Command. Simultaneously, we calculate the *CK* metrics for the classes of the design *D*1, 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 class Command playing the role of Command is high since, it is equal to 4. The CBO of the classes Invoker Button and Invoker Menu playing the role of Invoker are low, it is equal to 1, and WMC of the classes Bring To Front Command, Chang Attribute Command, Copy Command, Insert Image Command playing the role of Concrete Command is low, it is equal to 3.

4. Tool Support

The Pattern Sniffer toolset whose functional architecture is shown in Figure 3 reuses the Argo UML editor [7]. The principal activities performed by Pattern Sniffer, are essentially composed of four parts: The extraction of design tree, the design decomposition, the semantic pattern sniffing and the Syntactic pattern sniffing.

The module of "extraction of design tree" parses the XMI file representing the UML design (class diagram and sequence diagrams) and extracts important information (class name, attribute name, operation name, message name, message sender, message receiver, …). This extraction is done thanks to an XSLT processor.

Once the XML tree is generated, the decomposition process begins. 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.

After decomposing the design fragment, the semantic sniffing process is performed for each sub design. It calculates the semantic coverage metric using the already generated XML file.

To illustrate the steps of our method and the various functionalities of the Pattern Sniffer, let us consider the design fragment illustrated in Figure 2. We first decompose it; the decomposition produced the two sub designs as shown in Figure 2. The third step is the semantic and syntactic sniffing of the sub-designs. After decomposing the design fragment, the sniffing process is performed for each sub design. Due to space limitation, in the remainder of the paper, we will present our approach on the Sub-Design 2 (SD2).

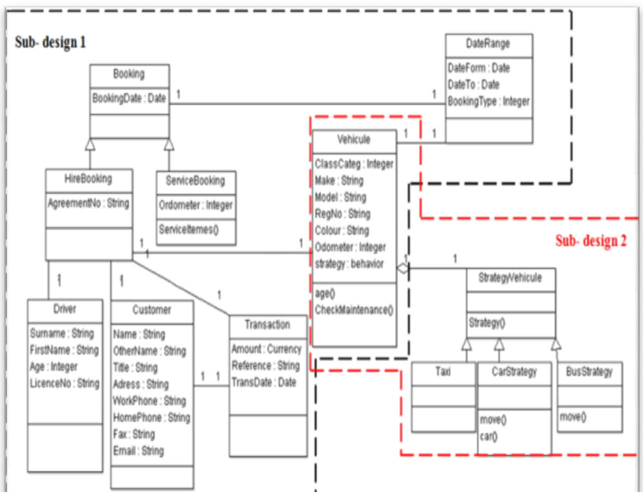


Figure 2. A design fragment example.

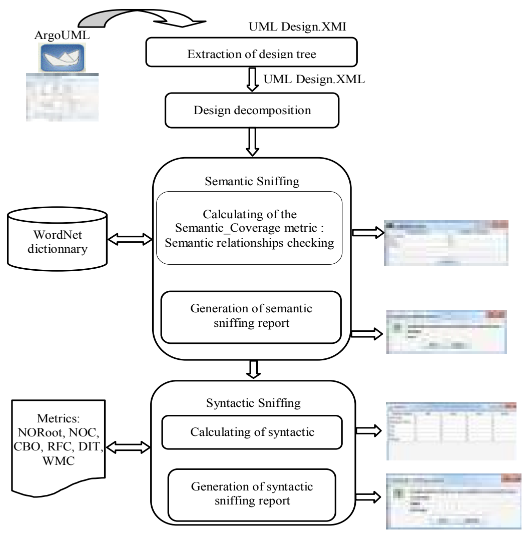
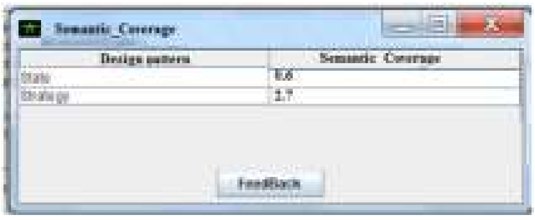

ArgoUML
UML Design.XMI
Extraction of design tree
UML Design.XML
Design decomposition
Semantic Sniffing
Calculating of the Semantic_Coverage metric : Semantic relationships checking
WordNet dictionnary
Generation of semantic sniffing report
Syntactic Sniffing
Calculating of syntactic
Metrics: NORoot, NOC, CBO, RFC, DIT, WMC
Generation of syntactic sniffing report

Figure 3. Conceptual architecture of the patternsniffer tool.

The semantic sniffing on SD2 produces the semantic coverage metric values shown in Figure 4.

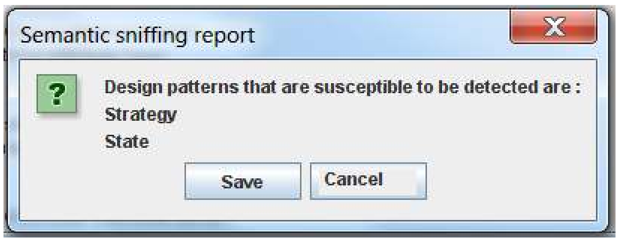
In order to, calculate the semantic-coverage of the SD2 with the Strategy design pattern, we are interested in the classes, attributes and method's semantic aspect. Thus, the tool used the Word Net dictionary. We found that: Name_includ ("Strategy", Strategy Vehicule), Def_Contain(BusStrategy,"way")Meth_Def_Contain(" Behavior", MoveCarStrategy).SemanticCoverage(SD2, Strategy)= 3*0.7+2*0.3= 2.7.


Semantic_Coverage

Design pattern	Semantic Coverage
State	0.6
Strategy	2.7

Figure 4. Semantic coverage metrics for the sub-design 2.

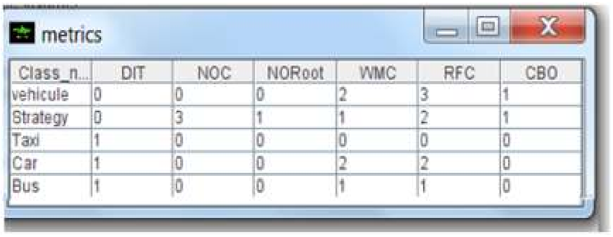
In the same way, we calculated the semantic coverage of the SD2 with the state design patterns. Using WordNet dictionary, we found that: Meth_Def_Contain ("State", Move Car Strategy) and Meth_Def_Contain ("State", Move Bus Strategy). The semantic coverage (SD2, State)= 0*70%+2*30%= 0.6. The final list will be organized in terms of the semantic coverage (suitability) as shown in Figure 5 of each design patterns. Consequently, the detection step will focus on identifying the final, ordered list of candidate patterns is (Strategy, State).


Semantic sniffing report

Design patterns that are susceptible to be detected are :
Strategy
State

Figure 5. Semantic sniffing report.

Figure 6 illustrates a screen shot of Pattern Sniffer presenting the calculated metrics for the SD2. We noticed that the syntactic sniffing proves that the SD2 is likely to contain the strategy pattern instances.


metrics

Class_n...	DIT	NOC	NORoot	WMC	RFC	CBO
vehicule	0	0	0	2	3	1
Strategy	0	3	1	1	2	1
Taxi	1	0	0	0	0	0
Car	1	0	0	2	2	0
Bus	1	0	0	1	1	0

Figure 6. Calculated metrics on the sub design 2.

5. Conclusions

This paper proposes a new method that uses the *C* and *K* metrics proposed by [15] and defines a semantic coverage metric to determine the most probable correspondences between the design elements and the patterns. In addition, to predicting the possibility of existence of design patterns, our method also determines the existence probability of each candidate design pattern.

The semantic resemblance determination obviously assumes that the design uses the same language as the design patterns. To widen the applicability of our approach, we will look into integrating a translation preliminary step to harmonize the languages of the design and patterns. Such a step can make use of existing powerful automatic translators; nevertheless, designers would need to intervene to validate the translation results because the design terminology often depends on the application domain.

The presented experimental evaluation on open source code showed that our approach can predict design patterns. To demonstrate its efficiency quantitatively, we are currently evaluating the performance of our prediction method in terms of recall and precision. This evaluation will cover both open source applications (where pattern reuse is often well applied by experienced developers) and new designs produced by non-experienced developers and hence the probability of pattern reuse would be lower.

References

[1] Abul Khaer M., Hashem M., and Masud R., "On Use of Design Patterns in Empirical Assessment of Software Design Quality," *in Proceedings of International Conference on Computer and Communication Engineering*, Kuala Lumpur, Malaysia, pp. 133-137, 2008.

[2] Ampatzoglou A., Charalampidou S., and Stamelos I., "Research State of the Art on GoF Design Patterns: A Mapping Study," *Journal of Systems and Software*, vol. 86, no. 7, pp. 1945-1964, 2013.

[3] Ampatzoglou A., Frantzeskou G., and Stamelos I., "A Methodology to Assess the Impact of Design Patterns on Software Quality," *Journal of Information and Software Technolo*gy, vol. 54, no. 4, pp. 331-346, 2012.

[4] Antoniol G., Fiutem R., and Cristoforetti L., "Using Metrics to Identify Design Patterns in Object-Oriented Software," *in Proceedings of the 5th International Symposium on Software Metrics*, Maryland, USA, pp. 23-34, 1998.

[5] Arcelli F. and Maggioni S., "Metrics-Based Detection of Micro Patterns to Improve the Assessment of Software Quality," *in Proceedings of the 1st Symposium on Emerging Trends in Software Metrics*, Srdinia, Italy, pp. 50-59, 2009.

[6] Arcelli F., Perin F., Raibulet C., and Ravani S., "JADEPT: Dynamic Analysis for Behavioral Design Pattern Detection," *in Proceedings of the 4th International Conference on Evaluation of Novel Approaches to Software Engineering*, Milan, Italy, pp. 95-106, 2009.

[7] ArgoUML., available at: http://argouml.softonic.fr/, last visited 2013.

[8] Ayata M., "Effect of Some Software Design Patterns on Real Time Software Performance," *A Master's Thesis*, the Graduate School of Informatics of Middle East Technical University, 2010.

[9] Aydinoz B., "The Effect of Design Patterns on Object Oriented Metrics and Software Error-Proneness," *Master's Thesis*, The Graduate School of natural and applied sciences of Middle East Technical University, 2006.

[10] Belderrar A., Kpodjedo S., Guéhéneuc Y., Antoniol G., and Galinier P., "Sub-Graph Mining: Identifying Micro-architectures in Evolving Object-oriented Software," *in Proceedings of the 15th European Conference on Software Maintenance and Reengineering*, Oldenburg, Germany, pp. 171-180, 2011.

[11] Bieman J., Straw G., Wang H., Munger W., and Alexander T., "Design Patterns and Change Proneness: An Examination of Five Evolving Systems," *in Proceedings of the 9th International Software Metrics Symposium*, Sydney, Australia, pp. 40-49, 2003.

[12] Blondel D., Gajardo A., Heymans M., Senellart P., and Dooren V., "A Measure of Similarity between Graph Vertices," *Applications to Synonym Extraction and We*b Searching, vol. 46, no. 4, pp. 647-666, 2004.

[13] Bouassida N. and Ben-Abdallah H., "Structural and Behavioral Detection of Design Patterns," *in Proceedings of International Conference on Advanced Software Engineering and its Applications*, Jeju Island, Korea, pp. 16-24, 2009.

[14] Chandra P. and Edith L., "Class Break Point Determination using CK Metrics Thresholds," *Global Journal of Computer Science and Technology*, vol. 10, no. 14, pp. 73-7, 2010.

[15] Chidamber S. and Kemerer C., "A Metrics Suite for Object Oriented Design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476-493, 1994.

[16] De Lucia A., Deufemia V., Gravino C., and Risi M., "Improving Behavioral Design Pattern Detection through Model Checking," *in Proceedings of the 14th European Conference on Software Maintenance and Reengineering*, Madrid, Spain, pp. 176-185, 2010.

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

[18] Guéhéneuc Y. and Antoniol G., "DeMIMA: A Multilayered Approach for Design Pattern Identification," *IEEE Transactions on Software Engineering*, vol. 34, no. 5, pp. 667-684, 2008.

[19] Guéhéneuc Y., Sahraoui H., and Zaidi F., "Fingerprinting Design Patternséé," *in Proceedings of the 11th Working Conference on Reverse Engineering*, Eindhoven, Netherlands, pp. 172-181, 2004.

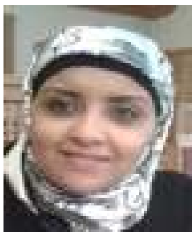
[20] Hernandez J., Kubo A., and Washizaki H., "Selection of Metrics for Predicting the Appropriate Application of Design Patterns," *in Proceedings of the 2nd Asian Conference on Pattern Languages of Programs*, Tokyo, Japan, pp. 5-8, 2011.

[21] Hsueh N., Chu P., and Chu W., "A Quantitative Approach for Evaluating the Quality of Design Patterns," *the Journal of Systems and Software*, vol. 81, no. 8, pp. 1430-1439, 2008.

[22] Huston B., "The Effects of Design Pattern Application on Metric Scores," *the Journal of Systems and Software*, vol. 58, no. 3, pp. 261-269, 2001.

[23] Issaoui I., Bouassida N., and Ben-Abdallah H., "A Design Pattern Detection Approach Based on Semantics," *in Proceedings of the 10th International Conference on Software*

Engineering Research, *Management and Applications*, Shangai, China, pp. 49-63, 2012.
[24] JHotDraw., available at: http://www.jhotdraw.org, last visited 2013.
[25] JRefactory., available at: http://jrefactory.sourceforge.net/, last visited 2013.
[26] JUnit., available at: http://www.junit.org, last visited 2013.
[27] Kuljit K. and Hardeep S., "Investigation of Design Level Class Cohesion Metrics," *the International Arab Journal of Information Technology*, vol. 9, no. 1, pp. 66-73, 2012.
[28] Larman C., *Applying UML and Patterns*: *An Introduction to Object-Oriented Analysis and Design and Iterative Development*, Addison Wesley, 2004.
[29] Maggioni S. and Arcelli F., "Metrics-Based Detection of Micro Patterns," *in Proceedings of ICSE Workshop on Emerging Trends in Software Metrics*, Cape Town, South Africa, pp. 39-46, 2010.
[30] Masuda G., Sakamoto N., and Ushijima K., "Evaluation and Analysis of Applying Design Patterns," available at: http://nanotsu.ait.kyushu-u.ac.jp/IWPSE99/Proceedings/27.pdf, last visited 2013.
[31] Reißing R., "The Impact of Pattern Use on Design Quality," available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.21.9968&rep=rep1&type=pdf, last visited 2001.
[32] Riel J., *Object-Oriented Design Heuristics*, Addison Wesley, 1996.
[33] Tsantalis N., Chatzigeorgiou A., Stephanides G., and Halkidis T., "Design Pattern Detection using Similarity Scoring," *IEEE Transactions on Software Engineering*, vol. 32, no. 11, pp. 896-909, 2006.
[34] Venners B., "How to Use Design Patterns-a Conversation with Erich Gamma, part I," available at: http://www.artima.com/lejava/articles/gammadp.html, last visited 2013.
[35] Vernazza T., Granatella G., Succi G., Benedicenti L., and Mintchev M., "Defining Metrics for Software Components," *in Proceedings of World Multi Conference on Systemics*, *Cyberneti58cs and Informatics*, Florida, USA, pp. 16-23, 2000.
[36] WordNet., available at: https://wordnet.princeton.edu/, last visited 2013.



Imene Issaoui is preparing a Doctorate degree in Computer Science at the Faculty of Economic Sciences and Management of Sfax, Tunisia. She is a Teaching Assistant at the Institut Preparatory to engineering studies of the University of Monastir, Tunisia.

Nadia Bouassida received a Phd in Computer and Information Science from the University of Science of Tunis, Tunisia. Currently, she is Assistant Professor at the Department of Computer Science of the Institut Supérieur d'Informatique et du Multimédia at the University of Sfax, Tunisia. She is a member of the Multimedia, Information systems and Advanced Computing Laboratory, University of Sfax Her research interests include reuse techniques, such as design patterns, Frameworks and Software Product Lines.

Hanene Ben-Abdallah received a BS degree in Computer Science and BS degree in Mathematics from the University of Minnesota, MPLS, MN, a MSE and PhD degrees in Computer and Information Science from the University of Pennsylvania, PA. She worked at University of Sfax, Tunisia from 1997 until 2013. She is now full professor at the Faculty of Computing and Information Technology, King Abdulaziz University, Kingdom of Saudi Arabia. She is a member of the Multimedia, Information Systems and Advanced Computing Laboratory, University of Sfax. Her research interests include software design quality, reuse techniques in software and business process modelling.