A New Approach for Pattern Problem Detection

Nadia Bouassida[1] and Hanêne Ben-Abdallah[2]

Mir@cl Laboratory,
[1] Institut Supérieur d'Informatique et de Multimédia
[2] Faculté des Sciences Economiques et de Gestion
Sfax University, Tunisia
Nadia.Bouassida@isimsf.rnu.tn, Hanene.BenAbdallah@fsegs.rnu.tn

Abstract. Despite their advantages in design quality improvement and rapid software development, design patterns remain difficult to reuse for inexperienced designers. The main difficulty consists in how to recognize the applicability of an appropriate design pattern for a particular application. Design problems can be found in a design with different shapes, unfortunately, often through poor solutions. To deal with this situation, we propose an approach that recognizes pattern problems in a design and that assists in transforming them into their corresponding design patterns. Our approach adapts an XML document retrieval technique to detect the situation necessitating a pattern usage. Unlike current approaches, ours accounts for both the structural and semantic aspects of a pattern problem. In addition, it tolerates design alterations of pattern problems.

Keywords: Design pattern identification, design pattern problem, pattern instantiation.

1 Introduction

It is irrefutable that the quality of a design highly impacts the quality of the code and, possibly, the performance and maintenance of software products. This vital importance of design quality justifies several efforts invested to define design techniques based on reuse of *proven solutions*, *e.g.*, components, design patterns [6], frameworks and models [23]. Reusing such solutions allows a designer to profit from prior experiences to produce a higher quality design more rapidly. Design patterns are among the highly advocated reuse techniques, which are also exploited in all recent reuse techniques; they can be instantiated and composed by a designer in order to produce a component, a framework and/or an application model.

However, to benefit from design patterns, a designer must have a thorough understanding of and a good practice with design patterns in order to identify the pattern problem and thereafter the appropriate patterns to instantiate for its application. The absence of such high-level of expertise motivated several researchers to propose assistance through the detection of patterns (*cf.*, [13]), spoiled patterns or anti-patterns (*cf.*, [1], [2]) and pattern problems (*cf.*, [10]) in a design. The proposed approaches differ mainly in the pattern concepts they consider (*i.e.*, only the structure, the structure and the method invocations/declarations [21]) and the degree of structural discordance they tolerate: exact match [12] or partial match [16], [13] [21]. These differences can be justified by the objectives behind each approach: code improvement through re-engineering (*cf.*, [12],[9],[14]) *vs.* design improvement (*cf.*, [8],[10]).

In our work, we are interested in improving design during its construction. In this context, a designer (inexperienced with design patterns) may inadvertently specify a design fragment that "resembles" a design pattern. The resemblance can be manifested as: 1) a correct, but extended and/or reduced, structural instantiation of a pattern; 2) a close representation of a pattern problem; or 3) a poor/spoiled instantiation of a pattern. In fact, while a design pattern is considered as a "good" design solution for a frequent design problem, the second and third cases are considered as "bad" design solutions. In addition, a pattern problem can be found in different shapes and forms; unfortunately, they are often represented in designs through poor solutions [10]. Hence, we argue that a design pattern problem identification technique would help designers in all three resemblance cases. One limit of existing identification approaches is that they can handle only one of the above three resemblance cases with no tolerance to variability in the design.

Furthermore, contrary to design patterns, the detection of design pattern problems requires an analysis of the semantics in addition to the structure. Indeed, the names of the classes and their operations distinguish one pattern problem from another. For instance, even though the *State* and *Strategy* patterns have the same structure [3], they represent different problems through different class and operation names. Hence, to detect and correct a pattern problem, one needs to identify first the structure of the problem (or one that is similar to it) and validate the identification through the problem's semantics.

In this paper, we illustrate how our approach based on an XML document retrieval technique (presented in [21]) can be fine-tuned to identify all three design resemblance cases. Being based on an XML retrieval technique, our approach identifies the similarities between the structure of the pattern problem (*i.e.,* pattern misuses) and a given application design. It offers two advantages over graph/matrix matching and constraint propagation based approaches used by other researchers. First, it can detect similarities between a *group* of classes in the application and the pattern problem. Secondly, it can tolerate controlled alterations of the pattern problem in the design.

The remainder of this paper is organized as follows. Section 2 overviews currently proposed approaches for pattern and pattern problem identification. Section 3 presents our technique for pattern problem identification which adapts an XML retrieval approach. Section 4 presents the pattern problem identification through examples of pattern problems or bad solutions and their transformation into a pattern solution. Section 5 summarizes the paper and outlines future work.

2 Related Work

2.1 Current Works for Pattern Detection

Several works have been interested in pattern identification but for different purposes. The main purpose of detecting design patterns in a reverse engineering context is to obtain information to understand and possibly restructuring a program. As an example, the work of Tansalis [13] aims at the identification of design patterns as part of the reengineering process. It proposes a design pattern detection methodology based on similarity scoring between graph vertices. For this, it encodes the pattern and design information as matrixes and relies on a graph matching algorithm to detect the pattern. Source code analysis is applied too late in the development process to offer a valuable assistance in the design phase.

On the other hand, for design analysis purposes, Dong *et al.* [20] also use matrixes to encode the pattern information. They use a template matching method to calculate the normalized cross correlation between the two matrices. A normalized cross correlation shows the degree of similarity between a design pattern and an analyzed part of the design.

Albin-Amiot *et al.* [9] present a set of tools and techniques to help OO software practitioners design, understand, and re-engineer a piece of software, using design-patterns. A first prototype tool, Patterns-Box, provides assistance in designing the architecture of a new piece of software, while a second prototype tool, Ptidej, identifies design patterns used in existing software.

2.2 Current Works for Pattern Problem Detection

The work of Mili and Boussaidi [10] [15] represents the design problem that the pattern is meant to solve explicitly. This explicit representation is one step towards the automatic detection and application of patterns. In fact, this work aims at recognizing occurrences of the problem solved by the design patterns which are then transformed according to the solution proposed by the design pattern. It uses an MDA approach where it identifies instances of pattern problems in a given model using a meta-model of the problem, marks the appropriate entities, and finally applies an appropriate transformation to get the pattern solution. The models are considered as graphs and model transformations are a special kind of graph transformations. The transformation is seen as a rule-based implementation of graph grammars. The current implementation is in the ECLIPSE Modeling Framework.

El-Boussaidi *et al.* [10] consider that design problems solvable by design patterns are sometimes badly understood, which often produces poor solutions to modeling requirements. To limit bad designs, this work proposes a semi-automatic tool for marking models (with the variable parts), identifying pattern problems using constraint satisfaction techniques, and then transforming the design by instantiating the appropriate design patterns. The tool relies on an explicit representation of design problems solved by design patterns. It uses a meta-model of the pattern problem *structure* to look for an *exact* match of the pattern problem. In other words, this work does not handle incomplete designs or designs similar to the pattern problem. Moreover, since it does not capture the behavioral aspect of patterns, it may not be as efficient for behavioral patterns.

Alikacem *et al.* [19] propose an approach to detect design problems using metrics to evaluate the source code quality. Their objective is to detect violations of quality rules in object-oriented programs. On the other hand, Ciupke [20] model the quality rules using fuzzy rule based systems. This approach is used to detect automatically design problems in an OO reengineering context using logic programming. In both of these works, design problems are defined as violations of OO code-quality rules and, hence, are of a lower level of abstraction than the design problems solved by design patterns [10].

Bouhours *et al.* [2] propose a detection approach for "bad smells in a design" that can be remodeled through the use of design patterns. The automatic detection and the explanation of the misconceptions are performed thanks to *spoiled* patterns. A spoiled pattern is a pattern that allows to instantiate inadequate solutions for a given problem: requirements are respected, but architecture is improvable. Thus, this work considers only the structural information, while the behavior is very important.

Moha *et al.* [4][5] are interested in design smells which are poor solutions to recurring design problems. They present detection algorithms for design smells and validate the detection algorithms in terms of precision and recall.

Overall, none of the proposed approaches and tools allows pattern problem detection and its similar structures using structural and behavioral information.

3 Pattern Problem Identification

As argued previously, the structural similarity to a pattern problem is insufficient to decide upon the necessity of a particular design pattern. Semantics, in terms of pattern class names and method declarations within the classes, is also required to confirm the presence of a pattern problem and, hence, the necessity of a corresponding design pattern.

This justifies the operation of our approach in three steps:

- The structural pattern problem identification: As mentioned in the introduction, in order to tolerate structural variations of the pattern problem, we adapt an XML document retrieval approach: we consider a pattern problem as an XML query and the design as the target XML document where the pattern problem is searched. This adaptation is feasible since the transformation of UML diagrams into XML documents is straightforward and can be handled by all existing UML editors.
- The semantic correspondences of class names: it relies on linguistic and typing information to identify the classes of the pattern problem. It confirms the first step's results and resolves any non deterministic identification when necessary.
- The identification of method declaration patterns: one characteristic of pattern problems is the repeated declaration of certain methods, for instance, by redefining abstract versions, combining abstract methods, etc.

3.1 Structural Pattern Problem Identification

For this first step of pattern problem identification, we use the same XML technique for the identification of a design pattern and which we presented in [21]. For the sake of completeness of this paper, we next briefly review it and we will illustrate it through an example in the next section.

In XML document retrieval, the document can be considered as an ordered, labeled tree. Each node of the tree represents an XML element. The tree is analyzed as a set of paths starting from the root to a leaf. In addition, each query is examined as an extended query – that is, there can be an arbitrary number of intermediate nodes in the document for any parent-child node pair in the query. Documents that match the query structure closely by inserting fewer additional nodes are given more preference.

A simple measure of the similarity of a path c_q in a query Q and a path c_d in a document D is the following context resemblance function [12]:

$$C_R(c_q, c_d) = \begin{cases} \dfrac{1+|c_q|}{1+|cd|} & \text{if } c_q \text{ matches } c_d \\ 0 & \text{if } c_q \text{ does not match } c_d \end{cases}$$

where:

- $|c_q|$ and $|c_d|$ are the number of nodes in the query path and document path, respectively, and
- c_q matches c_d if and only if we can transform c_q into c_d by inserting additional nodes.

Note that the value of $C_R(c_q, c_d)$ is 1 if the two paths are identical. On the other hand, the more nodes separate the paths, the less similar they are considered, *i.e.*, the smaller their context resemblance value will be.

In XML document retrieval in general, the context resemblance function C_R is calculated based on an exact match between the names of the nodes in the query and the document paths. However, for pattern problem detection, the nodes representing the classes in the design are application domain dependent, while those in the design are generic. Thus, in our identification algorithm [21], we operate as follows:

- First, we need to calculate the resemblance values for the various matches between the class nodes in the query (pattern problem) and those in the design. In addition, for a given path, we consider that the match between the pattern problem path and the design path may not necessarily start at the root node; for this, we need to consider all possible sub-paths of the design.
- Secondly, we need to take into account: 1) the number of times a given match between two class nodes is used to calculate C_R; and 2) the importance of each relation in the pattern.

The structural resemblance between a pattern problem and a design starts by calculating the resemblance scores between each path of the pattern problem to all the paths in the design (stored as matrix called *CRMatrix*). In this calculation, we assume that the structural variability should be limited between the pattern problem and a potential instantiation in the design. That is, we assume that a design path may differ from a pattern problem path by adding at most *N* nodes compared to the longest path of the pattern problem. The larger the *N*, the more scattered the pattern problem would be in the design.

In addition, to account for multiple matches of a class, the *CRMatrix* is normalized with respect to the total number of classes in the design. Thus, the final result of the structural identification algorithm is the matrix *NormalizedCRMatrix* whose columns are the pattern problem classes and whose rows are the classes of the design. Given this matrix, we can decide upon which correspondence better represents the pattern problem instantiation: For each pattern problem class, its corresponding design class is the one with the maximum resemblance score in the *NormalizedCRMatrix*. There might be more than one such class. This non-deterministic correspondence could be resolved through the following step, which is the semantic correspondence of class names.

Note that in a worst case instantiation, each pattern problem class must be matched to at least one class in the design; thus, on average, the sum of the normalized resemblance scores of the matched classes should not be less than the number of classes in the pattern divided by the number of classes in the design.

3.2 Semantic Correspondences of Class Names

In order to identify different pattern problems, we have to determine the correspondences between the class names. This part differs from our previous pattern identification approach [21]. For design pattern identification, recognizing the structure and the behavior of the pattern is sufficient. However, in a pattern problem, the class names reflect semantic relationships and the presence of methods in general and in some cases the presence of the same method in different classes is very important.

Thus, we propose to determine semantic correspondences between the class names by using an adapted version of our class name comparison criteria to construct frameworks [22]. These latter express linguistic relationships between class names. We define five types of relations between classes (however, the list can be extended):

- N_equiv(C_1,...,C_n): implies that the names are either identical or synonym, *e.g.*, Person-Person and Individual-Person.
- N_variation(C_1,...,C_n): implies that the names are a variation of a concept, *e.g.,* employee-contractual, employee-permanent, employee-vacationer.
- N_comp(C_1; C_2,...,C_n): implies that the name C_1 is a composite of the components C_2,…,C_n, *e.g.*, House- Room.
- Gen_Spec(C_1; C_2,...,C_n): implies the name C_1 is a generalization of the specific names C_2 ,…, C_n, e.g., Person-Employee.
- Str_extension (C_1; C_2): implies that the name C_1 is a string extension of the name of the class C_2, *e.g.*, XWindow- IconXWindow.

The determination of the above linguistic/semantic relations can handled through either a dictionary (*e.g.,* Wordnet [24]), or a domain ontology when available.

3.3 Method Declaration Pattern Identification

The structural resemblance and the name comparison steps managed in the previous sections are insufficient to identify the design problems. To determine the overall resemblance, we combine theses steps with method identification. In fact, in some pattern problems, the presence of methods repeated within a subset of classes in a particular pattern is essential. This last step should reinforce the quality of the identification results.

To determine the presence of methods in a design D and a pattern problem Pb, we will compare the method names for each pair of classes that were already identified as similar during the structural and semantic identification steps. First, note that, according to our DTD, each XML path for the class methods is composed of the class name node, the XML tag "method" node and the method name. Thus, the resemblance score function is not very useful.

Instead, to compare the method declaration pattern, we proceed as follows: First, for each class C_p in Pb that is matched to a class C_d in D, we derive from the path starting at C_p a set of paths representing all possible matches with the methods in C_d; these paths will be directly compared to all the paths in D. Secondly, the comparison results are collected into a matrix where the columns are the derived pattern problem paths and the rows are the design paths. A perfect match occurs when each column of this matrix contains at least one. A column with all zeros indicates a missing method; that is, the design lacks the redefinition of this pattern problem method and the pattern problem should therefore be reconsidered.

Note that, in addition to detecting missing re-declaration of methods, our approach tolerates the declaration of additional methods in the design.

4 Pattern Problem Identification Examples

To illustrate the steps of our approach for pattern problem identification, let us consider two examples: the bridge pattern problem and an altered version of this problem.

4.1 The Bridge Problem Case

The Bridge pattern (Figure 1) generally consists of three roles: Abstraction, Implementor, and ConcreteImplementor. The Implementor class provides an interface for the Abstraction class. Its children, ConcreteImplementor, need to implement the interface.

The Bridge pattern applies when an abstract class defines the interface to the abstraction and concrete subclasses implement it in different ways. Consider the implementation of a window abstraction in a user interface toolkit (Figure 2). This

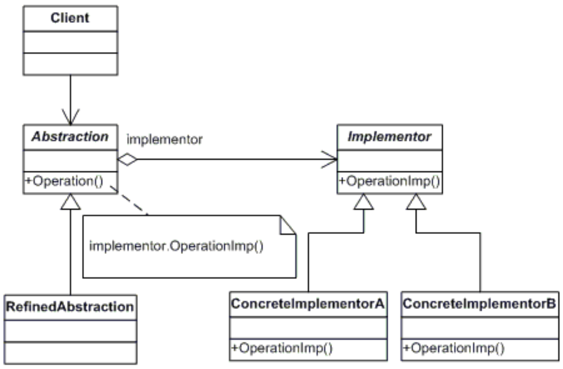

Fig. 1. The Bridge pattern[6]

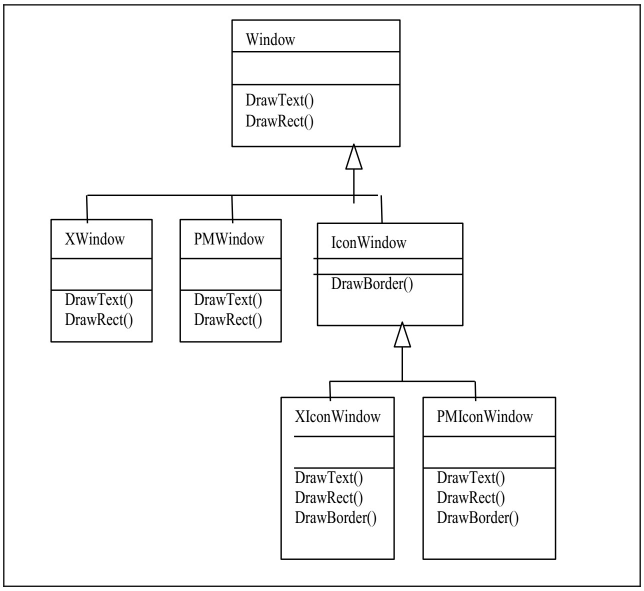

Fig. 2. A design that could be improved by the Bridge pattern

abstraction should enable us to write applications that work on both the XWindow system and IBM's presentation manager (PM), for example. Using inheritance, we could define an abstract class Window and subclasses XWindow and PMWindow that implement the window interface for different platforms. However, it is inconvenient to extend the window abstraction to cover different kinds of windows or new platforms. In the example, the IconWindow is a subclass of Window that specializes the window abstraction for icons. To support IconWindows for both platforms, we have to implement two new classes XIconWindow and PMIconWindow. The bridge pattern solves this problem by putting the window abstraction and its implementation in separate class hierarchies as shown in Figure 3.

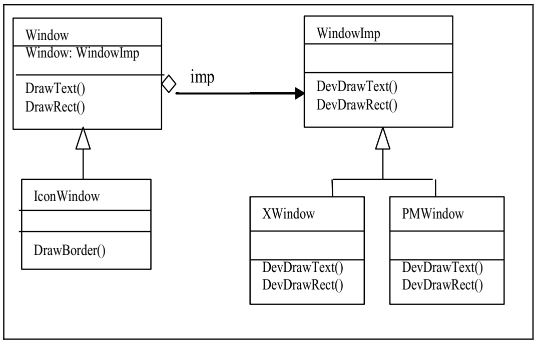

Fig. 3. The solution with the bridge pattern

Note that, at an abstract level, the Bridge pattern problem applies when : 1) there is an abstract class *A* with at least two levels of inheritance underneath it, and 2) one of the inheriting classes, say *B*, adds specific methods and has one or more inheriting classes (say *C* and *D*) redefining the methods of *A* and *B*.

An abstraction of the Bridge pattern problem is illustrated in Figure 4. We next illustrate the steps of our identification approach, which can be applied to improve the design fragment shown in Figure 2.

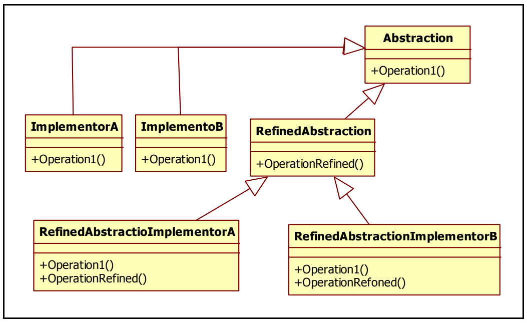

Fig. 4. Abstraction of the pattern problem

A) The structural identification step

The first step relies on the computation of resemblance function scores between the XML paths of the pattern problem and the design. The XML tree of the design is illustrated in a graphical format in Figure 5.

A sample of the context resemblance scores of the paths of the design (Figure 2) with the Bridge pattern problem paths (Figure 4) are shown in Table1. The class names are abbreviated because of space limitations.

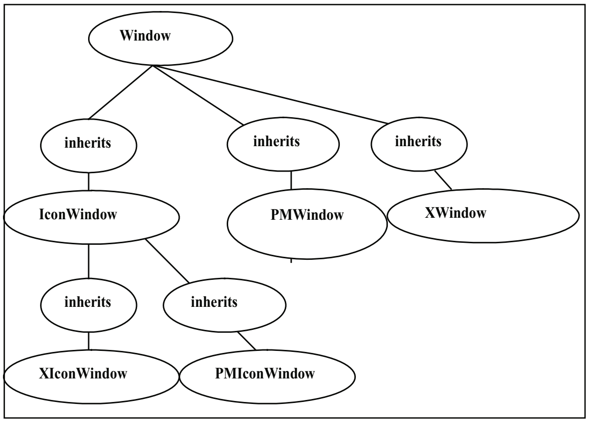

Fig. 5. XML trees for the design (Figure 2)

Table 1. Context similarity function scores

	Abst $\xrightarrow{\text{inherits}}$ *RefAbst*	*RefAbst* $\xrightarrow{\text{inherits}}$ *RefAbstImpA*	*RefAbst* $\xrightarrow{\text{inherits}}$ *RefAbstImpB*
Window $\xrightarrow{\text{inherits}}$ *IconWindow*	$CR(c_q , c_d) =1$ *if* Abst=Window RefAbst=IconWindow	$CR(c_q , c_d) =1$ *if* *R*efabst=Window RefAbstImpA= IconWindow	$CR(c_q , c_d)=1$ *if* *Ref*Abst=Window RefabstImpB=IconWindow
IconWindow $\xrightarrow{\text{inherits}}$ *XIconWindow*	$CR(c_q , c_d)=1$ *if* Abst=IconWindow RefAbst=XIconWindow	$CR(c_q , c_d) =1$ *if*RefAbst= IconWindow RefAbstImpA=XIconWindow	$CR(c_q , c_d)) =1$ *if* RefAbst=IconWindow RefAbstImpB=XIconWindow
IconWindow $\xrightarrow{\text{inherits}}$ *PMIconWindow*	$CR(c_q , c_d)=1$ *if* Abst= IconWindow RefAbst=PMIconWindow	$CR(c_q , c_d)=1$ *if* RefAbst = IconWindow RefAbstImpA= PMIconWindow	$CR(c_q , c_d)) =1$ *if* *R*efAbst=IconWindow RefAbstImpB=PMIconWindow

Once the above context resemblance scores are computed, we find the normalized similarity matrix which sums up the values of the context resemblance scores for each class in the design with respect to a class in the pattern problem and divides it by the number of classes in the design. In our example, we obtain the following normalized similarity matrix:

$$\text{Normalized CRMatrix(P atternPb, Design)} = \begin{array}{c} \\ \text{Window} \\ \text{IconWindow} \\ \text{XIconWindo w} \\ \text{PMIconWind ow} \\ \text{PMWindow} \\ \text{XWindow} \end{array} \begin{array}{cccccc} \text{Abst} & \text{RefAbst} & \text{RefAbstImp A} & \text{RefAbstImp B} & \text{ImpA} & \text{ImpB} \\ \end{array} \left[\begin{array}{cccccc} 15 & 10 & 0 & 0 & 0 & 0 \\ 6 & 5 & 1 & 1 & 0 & 0 \\ 0 & 1.75 & 2.75 & 1.75 & 1.75 & 1.75 \\ 0 & 1.75 & 1.75 & 2.75 & 1.75 & 1.75 \\ 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 \end{array} \right] /6$$

The normalized CR matrix identifies the Bridge pattern problem and indicates that the class *Window* matches the *Abst* class, the class *IconWindow* is the *RefAbst* class, *XI-conWindow* is *refAbstImpA*, *PMIconWindow* is *refAbstImpB*. However, note that the class *PMWindow* and *XWindow* were identified as either *ImpA* or *ImpB* since the resemblance score of the classes is equal to 1, while their resemblance score to *re-fAbstImpA* is equal to 1.75 which was already identified as *XIconWindow*. The next, semantic-based correspondence step will assist in deciding the best match and validating the determined matches.

B) Semantic correspondence

In the pattern problem abstraction, the semantic correspondence of class names uses the following relations:

- Str_extension(implementorA;RefinedAbstractionImplementorA),
- Str_extension (RefinedAbstraction;RefinedAbstractionImplementorA),
- Str_extension (implementorB;RefinedAbstractionImplementorB),
- Str_extension (RefinedAbstraction; RefinedAbstractionImplementorB)

Now, we verify that these semantic relations exist for the classes of the design that were identified by the structural step. In fact, in the design fragment (Figure 2), we noticed that the class names of the inheritance hierarchy at the second level (*i.e.*, *XIconWindow* and *PMIconWindow*) are *composed* of the class names of the hierarchy at the first level (*i.e.*, *IconWindow* et *PMWindow*). In addition, the following semantic relations hold:

- Str_extension(XWindow; XIconWindow),
- Str_extension(IconWindow; XIconWindow),
- Str_extension (PMWindow; PMIconWindow), and
- Str_extension (IconWindow; PMIconWindow)

Thus, we can deduce that this is a Bridge pattern problem.

C) Method declaration pattern identification

The third step in our pattern problem identification examines the respect of the method presence pattern among the identified classes. That is, this step identifies if the methods are present in the design as required in the pattern problem abstraction.

In the Bridge problem abstraction, the classes *XIconWindow* and *PMIconWindow* implement the methods of the abstract class *IconWindow* and those of *XWindow* and *PMWindow*, respectively.

Due to space limitation, Table 2 shows a sample of the resemblance function scores comparing the Design of Figure 2 and the Bridge pattern problem abstraction (Figure 4).

Recall that this step uses the identification results of the previous two steps. For example, since the class *Window* was identified as *Abstraction*, then we have either *operation1=DrawText()* or *operation1=DrawRect()*. Also since *RefinedAbstraction* is *IconWindow*, then we have *Operationrefined= DrawBorder()*.

Table 2. Identification of method declaration patterns

	Abstraction $\xrightarrow{\text{contains}}$ ***DrawText***	***Abstraction*** $\xrightarrow{\text{contains}}$ ***DrawRect***	***ImplementorA*** $\xrightarrow{\text{contains}}$ ***DrawText***	***ImplementorB*** $\xrightarrow{\text{contains}}$ ***DrawText***	…
Window $\xrightarrow{\text{contains}}$ ***DrawText()***	1	0	0	0	
Window $\xrightarrow{\text{contains}}$ ***DrawRect()***	0	1	0	0	
IconWindow $\xrightarrow{\text{contains}}$ ***DrawBorder()***	0	0	1	0	
XWindow $\xrightarrow{\text{contains}}$ ***DrawText()***	0	0	0	1	
PMWindow $\xrightarrow{\text{contains}}$ ***DrawText()***	0	0	0	0	
XWindow $\xrightarrow{\text{contains}}$ ***DrawRect()***	0	0	0	0	
PMWindow $\xrightarrow{\text{contains}}$ ***DrawRect()***	0	0	0	0	

Now, to conduct this last identification step, we substitute the method names of the problem abstraction by their potentially equivalent methods in the design (the various paths derivation step). Then, we verify that the presence of the methods in the design is conforming to the pattern problem abstraction: For a design to cover all the methods of the pattern problem, we need to find, in each column, at least one entry in this table that has the value of 1. Otherwise, the column that is missing a one indicates that the corresponding message exchange is missing in the design and the pattern problem identification is reconsidered.

On the other hand, our approach tolerates additional, methods in the design. In our example, we tolerated the additional method DrawRect().

At the end of the method declaration pattern identification step, the designer has more confidence in the results of the structural and semantic name identification steps.

4.2 Identification of an Altered Bridge Pattern Problem

In this section, we show how our approach (unlike others) can detect altered/similar pattern problems. Let us consider the altered version of the Bridge pattern problem as abstracted in Figure 4; and let us consider the design fragment of Figure 6 to analyze. In fact, the classes *Emptystyle*, *WithIconstyle* and *Applicativestyle* are added, yet it remains an instance of the Bridge pattern problem. In addition, we omitted the methods in Figure 6 and we illustrate the pattern problem identification using the structural determination and the semantic correspondences of class names.

After collecting the context resemblance scores, the normalized similarity matrix is computed.

$$\text{NormalizedCRMatrix(PatternPb, Design)} = \begin{array}{c|cccccc|} & \text{Abst} & \text{RefAbst} & \text{RefAbstImpA} & \text{RefAbstImpB} & \text{ImpA} & \text{ImpB} \\ \text{Window} & 26.75 & 12.5 & 0 & 0 & 0 & 0 \\ \text{EmptyWindow} & 11.5 & 8 & 1 & 1 & 1 & 1 \\ \text{WindowwithIcon} & 11.5 & 8 & 1 & 1 & 1 & 1 \\ \text{ApplicativeWindow} & 11.5 & 8 & 1 & 1 & 1 & 1 \\ \text{EmptyStyle} & 6 & 8 & 2.75 & 2.75 & 1.75 & 1.75 \\ \text{WithIconStyle} & 6 & 8 & 2.75 & 2.75 & 1.75 & 1.75 \\ \text{ApplicativeStyle} & 6 & 8 & 2.75 & 2.75 & 1.75 & 1.75 \\ \text{XWindowEmpty} & 0 & 1.75 & 2.75 & 2.75 & 1.75 & 1.75 \\ \text{PMEmpty} & 0 & 1.75 & 2.75 & 2.75 & 1.75 & 1.75 \\ \text{XWindowWithIcon} & 0 & 1.75 & 2.75 & 2.75 & 1.75 & 1.75 \\ \text{PMWithIcon} & 0 & 1.75 & 2.75 & 2.75 & 1.75 & 1.75 \\ \text{XWindowApplicative} & 0 & 1.75 & 2.75 & 2.75 & 1.75 & 1.75 \\ \text{PMApplicative} & 0 & 1.75 & 2.75 & 2.75 & 1.75 & 1.75 \end{array} /15$$

We noticed in the normalized *CRMatrix* that the class *Abstraction* matches *Window* since it has the most important score (26.75). Moreover, the match score of the classes *EmptyWindow*, *WindowWithIcon*, *ApplicativeWindow*, *EmptyStyle*, *WithIconStyle*, *ApplicativeStyle* to *RefAbs*t is equal to (8), while the score matching these classes to *Abstraction* is 11.5; however these six classes are identified as *RefAbst*, since *Window* has been already identified as *Abstraction* with a greater matching score. Finally, the classes *PMEmpty*, *PMWIcon*, *PMApplicative* are identified as *RefAbstImpB* and the classes *XwindowEmpty*, *XWindowWIcon*, *XWindowApplicative* are identified as *RefAbstImpA*.

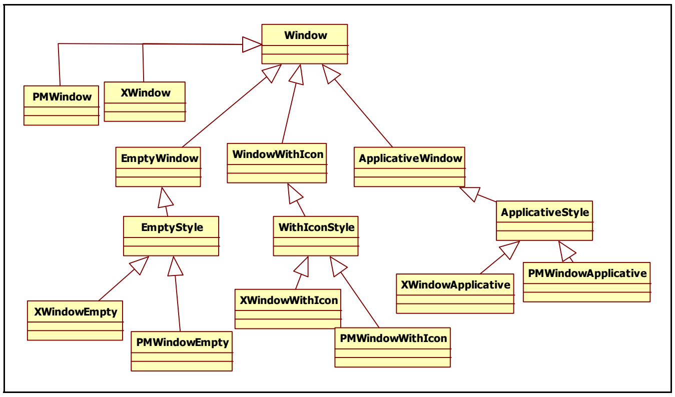

Fig. 6. Altered Bridge pattern problem

The results of the structural determination will be confirmed by the semantic correspondence step. For the semantic correspondence of class names, we remark that the following semantic relations hold:

Str_extension(EmptyWindow;XWindowEmpty),

Str_extension (EmptyWindow;PMWindowEmpty)

Str_extension(WindowWithIcon;XWindowWithIcon)

Str_extension (WindowWithIcon;PMWindowWithIcon)

Str_ extension(ApplicativeWindow;PMWindowApplicative) and

Str_ extension(ApplicativeWindow;XWindowApplicative)

Thus, the class names of the inheritance hierarchy at the fourth level (*i.e.*, XWindowEmpty) are *composed* of the class names of the hierarchy at the second level (*i.e.*, EmptyWindow, XWindow). As a conclusion, this step confirms that the design is also a representation of the Bridge pattern problem.

Note that, even though the above examples use the inheritance relation, our approach works in the same way for all types of UML class relations. The reader is referred to [21] where the Composite pattern is illustrated with the composition relation.

6 Conclusion

The paper first overviewed existing works for pattern problem detection. Secondly, it presented a new approach based on a structural identification followed by a semantic identification of class names (based on linguistic correspondences) and method presence pattern verification. This approach is applicable for all patterns and pattern problems that can be modeled through UML class diagrams or any UML profile language whose class diagrams can be stored as XML documents.

The paper illustrated the proposed approach through the Bridge pattern problem. In addition, to emphasize one advantage of our approach, it was applied to show its capacity to detect also an altered version of the Bridge pattern problem. The illustrated examples were automatically treated through our prototype toolset.

The current version of our structural identification algorithm runs in an exponential time in terms of the number of relations among the classes of the design. One practical way to manage this complexity is to decompose a large design and treat it one fragment at a time; a simple heuristic decomposition strategy is to consider the connected components of the class diagram, starting from an abstract class.

We are examining how to add more intelligence in the structural identification algorithm: how to alleviate the analysis of paths by adding priorities, and how to reduce the number of possible method correspondences in the last step.

References

1. Bouhours, C., Leblanc, H., Percebois, C.: Structural variants detection for design pattern instantiation. In: 1st International Workshop on Design Pattern Detection for Reverse Engineering, Benevento, Italy (October 2006)
2. Bouhours, C., Leblanc, H., Percebois, C.: Bad smells in design and design patterns. Journal of Object Technology 8(3) (May-June 2009)
3. Kampffmeyer, H., Zschaler, S.: Finding the Pattern You Need: The Design Pattern Intent Ontology. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 211–225. Springer, Heidelberg (2007)
4. Moha, N., Guéhéneuc, Y.G., Leduc, P.: Automatic generation of detection algorithms for design defects. In: Uchitel, S., Easterbrook, S. (eds.) Proceedings of the 21st Conference on Automated Software Engineering, September 2006, pp. 297–300. IEEE Computer Society Press, Los Alamitos (2006)
5. Moha, N., Guéhéneuc, Y.-G., Le Meur, A.-F., Duchien, L.: A domain analysis to specify design defects and generate detection algorithms. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 276–291. Springer, Heidelberg (2008)
6. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: Elements of reusable Object Oriented Software. Addisson-Wesley, Reading (1995)
7. Florijin, G., Meijers, M., Van Winsen, P.: Tool support for object oriented patterns. In: Aksit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 472–495. Springer, Heidelberg (1997)
8. Bergenti, F., Poggi, A.: Improving UML design pattern detection. In: Proceedings of the 12th International Conference on Software Engineering and Knowledge Engineering SEKE (2000)
9. Albin Amiot, H., Cointe, P., Guéhéneuc, Y.G.: Un meta-modele pour coupler application et détection des design patterns. L'objet 8, 1–18 (2002)
10. El Boussaidi, G., Mili, H.: Detecting patterns of poor design solutions by using constraint propagation. In: MODELS, Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems (September 2008)
11. Pagel, B.U., Winter, M.: Towards pattern-based tools. In: Proceedings of EuropLop (1996)
12. Brown, K.: Design reverse-engineering and automated design pattern detection in Smalltalk. Technical Report TR- 96-07, University of Illinois at Urbana-Champaign (1996)
13. Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., Halkidis, S.T.: Design pattern detection using similarity scoring. IEEE Transactions on Software Engineering 32(11) (2006)

14. Lee, H., Youn, H., Lee, E.: A design pattern detection technique that aids reverse engineering. International Journal of Security and Applications 2(1) (January 2008)
15. Mili, H., El Boussaidi, G., Salah, A.: Représentation et Mise en oeuvre de patrons de conception par représentation explicite des problèmes. In: LMO, Suisse (2005)
16. Dong, J., Sun, Y., Zhao, Y.: Design pattern detection by template matching. In: SAC 2008, Ceara, Brazil, March 16-20 (2008)
17. XML Metadata Interchange: OMG Document ad/98-07-03 (July 6, 1998)
18. Manning, C.D., Raghavan, P., Schütze, H.: An introduction to information retrieval. Cambridge University Press, England (2008)
19. Alikacem, E., Sahraoui, H.A.: Détection d'anomalies utilisant un langage de description règle de qualité. In: Rousseau, R., Urtado, C., Vauttier, S. (eds.) LMO 2006, pp. 185–200 (2006)
20. Ciupke, O.: Automatic Detection of Design Problems in Object-Oriented Reengineering. In: TOOLS 30, pp. 18–32. IEEE Computer Society Press, Los Alamitos (1999)
21. Bouassida, N., Ben-Abdallah, H.: Structural and behavioral detection of design patterns. In: International Conference on Advanced Software Engineering & Its Applications (ASEA), Jeju Island, Korea, December 10-12. LNCS Proceedings. Springer, Heidelberg (2009)
22. Bouassida, N., Ben-Abdallah, H., Gargouri, F.: Stepwise framework design by application unification. In: IEEE International Conference on system man and Cybernetics, Tunisia (2003)
23. `http://www.omg.org/mda/`
24. `http://wordnet.princeton.edu/`