Alternative Models for Structural Design Patterns

Cédric Bouhours, Hervé Leblanc, Christian Percebois

IRIT – MACAO
Université Paul Sabatier
118 Route de Narbonne
F-31062 TOULOUSE CEDEX 9
{bouhours, leblanc, perceboi}@irit.fr

ABSTRACT. To give a consistent and more valuable feature on models, we propose that model-driven processes should be able to reuse the expert knowledge generally expressed in terms of patterns. We focus our study on a detailed design review activity, which precedes a coding stage in an object-oriented language. We make the hypothesis that design models lack in good object design practices. As we do not find any model inspection tool that urges the use of patterns in the most automatic possible way, we present here the concept of an "alternative model" and its usage within our design review. An "alternative model" is a model which solves the same problem as the pattern, but with a more complex or different structure than the pattern. Each pattern has many "alternative models". Structural characterization permits us to detect model fragments substitutable with a pattern. Criteria of object-oriented architecture and software engineering quality permit us to construct a knowledge base dedicated to design practices and to explain why the substitution makes the design more attractive.

1. Introduction

The emergent MDE community, aiming at giving a productive feature on models, has proposed model-driven process development. However, to obtain guarantees on model relevance at the end of each activity, these processes should promote the reuse of the knowledge of experts generally expressed in terms of analysis (Fowler, 1997), design (Gamma et al., 1995) or architectural (Buschmann, 1996) patterns approved by the community. As we are focused on the design stage, we limit our approach to design patterns, because we consider that the use of analysis patterns is business domain specific, and that the use of architectural patterns must be planed before the design stage. Up to now, in spite of the efforts to improve reusability of design patterns, thanks to assistance tools to guide pattern integration in models by precise modeling (Guennec, 2000) (Mak, 2004) (France, 2004), and thanks to pattern wizards dedicated to integrate patterns by code refactorings (Eden, 1997) (O'Cinnéide, 1999), we do not find a tool inspecting models that urges the use of patterns in the most automatic possible way. Given the existence of "code review" activities (Dunsmore, 1998) in some development processes, we would like to introduce a "design review" activity directed by design patterns and oriented to object model quality. In this activity, we propose to parse models to find fragments substitutable with design patterns and to replace them if the intent of the designer is compatible.

In this paper, after presenting the course of this activity, we focus on the concept of "alternative model" which is the basis of the step named "alternative models detection". In order to illustrate this concept, we present some experimental results dedicated to structural design patterns. As we want to constitute a knowledge base of design practices, we introduce these alternative models in detailing their design defects.

2. Our design review activity

This activity, described in Figure 1, may be decomposed into four steps. In order to work with models in an "efficient" quality, the first step checks good basic object oriented design practices. For example, "one class must implement at least one interface or inherits from an abstract class", or "all attributes in a class are private or protected". To obtain a consensus on an exhaustively checklist of good design practices is not simply, then we intend to use GRASP patterns (Larman, 2004) and the Bertrand Meyer design principles (Meyer, 1993). This checking is done thanks to OCL rules, and, the designer has to rework his model to put it in an "efficient" quality state, if the model to review does not verify all rules.

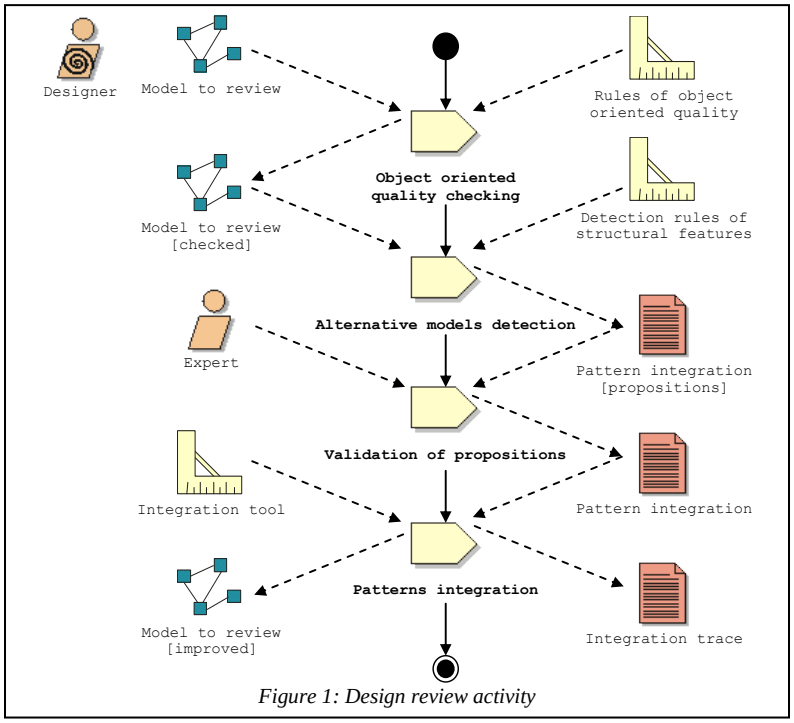


Figure 1: Design review activity

When the model to review is checked in an "efficient" quality state, a research based on structural and behavioral similarities to "alternative models" determines the model fragments which may be substitutable with a pattern. To do the detection of these substitutable fragments, we choose to use a match method. Rather than using an approximate design pattern match detection based on a similarity research (Arcelli Fontana, 2004), we choose to do exact pattern matching of models substitutable with a design pattern. Then, we seek a set of substitutable models for each structural pattern proposed by Gamma et al. According to the taxonomy proposed by Chikofsky and Cross

(Chikofsky, 1990), our detection technique can be connected to a redocumentation technique so as to permit model restructuring.

Each “alternative model” detected in the model to review represents propositions of fragments substitutable with a design pattern. Then, a design expert works with the designer to determine if his intention matches with the pattern intention and if the propositions are needed in the model to review.

With the expert authorization, the last step consists in integrating the validated propositions into their models. This integration is dealt with automatic parameterized transformations.

3. Alternative models

An alternative model is a model which solves the same problem as the pattern, but with a more complex or different structure than the pattern. Therefore, in agreement with the hypotheses on design patterns and class design defects (Guéhéneuc, 2001), it is a candidate model for substitution with a pattern. After obtaining these alternatives, we search for remarkable features that allow their detection in each model. Then, we automate this detection using OCL rules supported by the NEPTUNE platform (Neptune, 2003). Lastly, we test these rules on industrial models (NEPTUNE 2 platform) and standard meta-models (SPEM 1.0 and core UML 1.4).

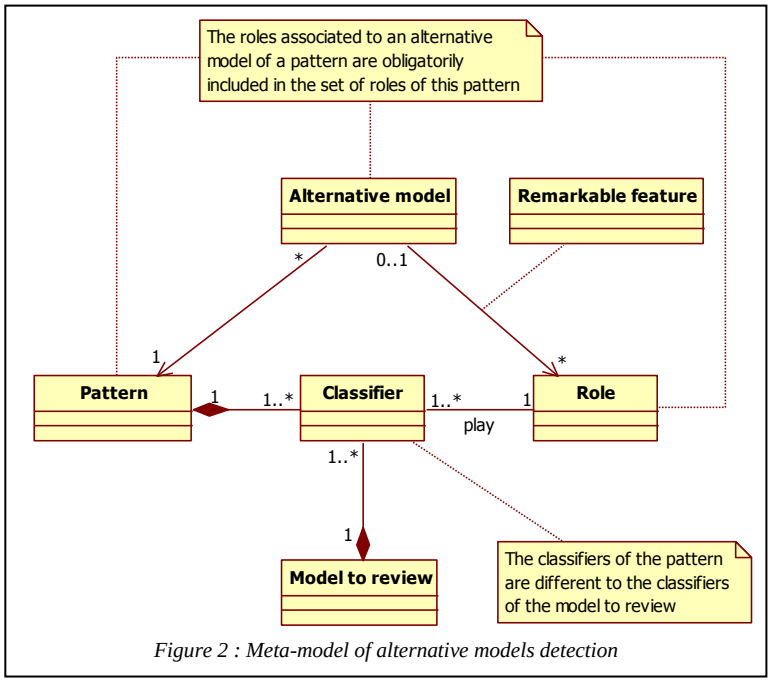


Figure 2 : Meta-model of alternative models detection

Each alternative model is characterized like a pattern. A set of structural features is associated with each alternative model role, as described in Figure 2. For the moment, these features concern inter-class relations only: i.e., associations, generalizations and aggregation/composition links, but neither interfaces nor class semantics. We deduce these features both associating corresponding pattern roles to each class of alternative models, and studying their structure.

3.1. Gathering

For this study, we choose to collect a set of models which do not use any patterns. If a model solves a problem solvable with a particular pattern, it may be considered as an alternative model.

We have organized an experiment which consists of the design of the seven standard problems solvable with the seven GoF structural patterns, in UML notation. We use the examples presented in the "motivations" section of the GoF catalogue, when they are relevant. As participants, we have chose students in third and fifth year of software engineering course. They were able to make UML models, and to conceive object-oriented software, but they had any pattern knowledge. So, although each problem admits a solution using a pattern, participants have solved these problems without pattern. From three hundred models obtained, we have selected fifteen of them which present significant structural variants with patterns under consideration, the others were either incorrect or duplicated design. We consider these models valid because they permit a solution to the problems under study, namely, they implement functionalities required by this problem. Each model obtained (between two and six for each pattern) constitutes a plausible alternative to just one pattern. To detect these alternatives in a model, we analyze their structural features, in the same manner as in pattern detection in redocumentation at design models.

3.2. Usage

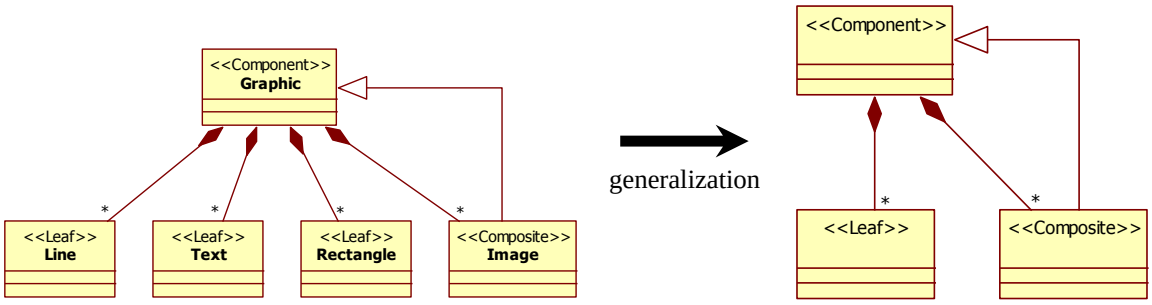
This collection method allows us to constitute a catalogue for each pattern and its associated alternative models that we consider as potentially bad design practices. One entry of the catalogue corresponds to one pattern with its alternative models classified by the strong points of the pattern. A strong point is a key design feature which permits the pattern to resolve a problem most efficiently. For example, the Composite pattern resolves the problem: "How compose and use object hierarchies as simply as possible for a client in keeping extensibility possibilities on components". So the two strong points for this pattern are "uniform protocol" and "decoupling and extensibility".

The strong points are the "essence" of the patterns. They are characterized by criteria of object-oriented architecture and software engineering quality, partially deduced from the "consequences" section of the GoF catalogue and from the study of the design defects of alternative models. As pattern injection may alter some object-oriented metrics (Huston, 2001), they allow us to compute dedicated pattern metrics to classify the alternative models and to help the estimation of the pertinence of pattern injection in a design model.

In order to characterize the design defects, we validate the alternative models in perturbing the strong points of each pattern. A perturbation may either delete a strong point or simply damage it. So to specify the degree of damage, we add sub-features for some strong points. Moreover, thanks to these perturbations, we should build new alternative models not taken from experiments. And, if we reverse these perturbations and if we apply them on alternative models, we would deduce a sequence of structural refactoring operations (Sunyé, 2001) that automatically perform the pattern integration in the models to review.

3.3. Detect

In order to validate our approach, we have applied OCL rules on UML models. So, it has been necessary to implement detection of each alternative model by several rules. When a rule is not validated in the model, the NEPTUNE platform returns the context of the error, which is the model fragment substitutable by a pattern. To do so, we have written OCL rules such that their return value is always false when an alternative model has been identified. In a first attempt, we have applied these rules on industrial models and then on OMG meta-models.



Reference role	«Component» features	«Composite» features	«Leaf» features
«Component»	Classes having at least two aggregations which one is sub-class.	Classes both sub-class and aggregated with «Component»	Classes not sub-class and aggregated with «Component»

Figure 3: Generalization of one Composite alternative model and its structural features

To detect these features in a model, we establish a backtrack method which locates a class able to play a reference role in the pattern to identify. This role is the root of the backtrack stage, chosen from a class playing this role in a model. It permits access directly to all the other classes of the model. Then, the method tries to assign the other roles of the pattern to the other classes of the model. For instance with the Composite pattern, we search for classes being able to play the «Component» role. To finish, we try to assign the «Composite» and «Leaf» roles to the other related classes and whose properties correspond to the remarkable features we wish to locate (Bouhours, 2006). This method has enabled us to deduce the generic search algorithm below.

```
In: UML model And set of remarkable features associated to each role of
      alternative model

Find classes satisfying remarkable features of reference role
For each candidate class do
      Find classes associated in the model
      For each class associated do
            If it satisfies remarkable features of an other role then
                  Affect the role to this class
            Fi
      Done
Done

Out: set of classes may have each role of alternative model
```

Thus, for each UML model, the algorithm finds all the occurrences of the generalization of one alternative model as shown in

Figure 3. Each time one applies this algorithm for each alternative model associated to a given pattern, one retrieves all substitutable model fragments by the pattern itself. Therefore, this method is deterministic and the

result is complete with the set of alternative models dedicated to the pattern. An alternative model may be considered as identified when each remarkable feature returns at least one class. However, we can see in the meta-model given by Figure 2 that some alternative models do not systematically refer all the roles of the pattern.

In the next sections, we exemplify some problems solvable with the Composite, Decorator and Bridge design patterns. We present the alternative models for each pattern, their strong points perturbation, and their design defects. A perturbation may either delete a strong point or simply damage it. So to precise the degree of damage, we add sub-features for some strong points. At the end of each sub-part, we present a table which resumes the perturbation of each strong point for each alternative model. Each model presented is taken from our experiment.

4. Composite alternative models

In the Composite pattern, we find two strong points with their sub-features:

1. Decoupling and extensibility.
   1.1. Maximal factorization of the composition.
   1.2. Addition or removal of a leaf does not need code modification.
   1.3. Addition or removal of a composite does not need code modification.
2. Uniform protocol.
   2.1. Uniform protocol on operations of composed object.
   2.2. Uniform protocol on composition managing.
   2.3. Unique access point for the client.

The problem "*Design a system enabling to draw a graphic image: a graphic image is composed of lines, rectangles, texts and images. An image may be composed of other images, lines, rectangles and texts.*" may be solvable with the Composite design pattern. Figure 4 represents this problem instantiation.

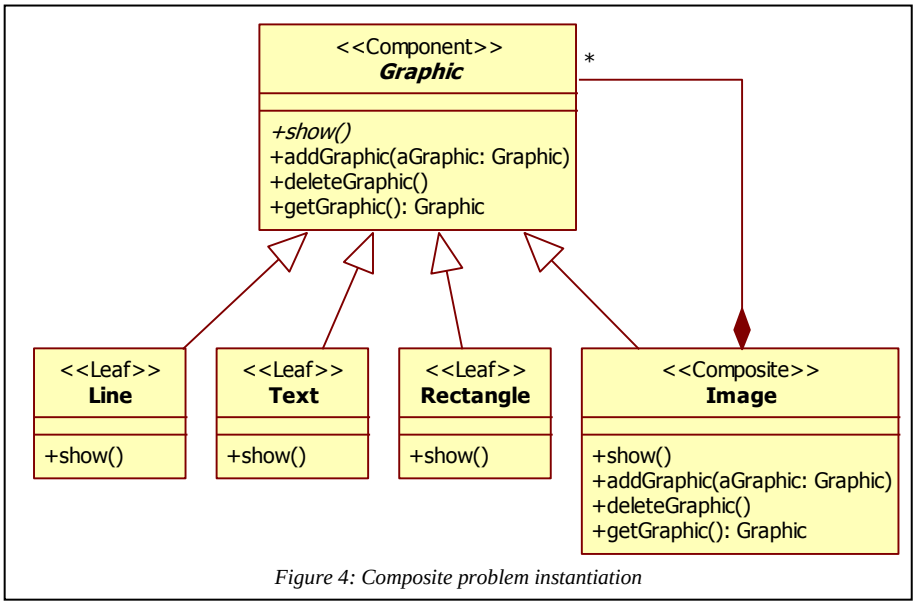


Figure 4: Composite problem instantiation

Figure 5 to Figure 10 represent the six alternative models taken from our experiment, with their strong points perturbations.

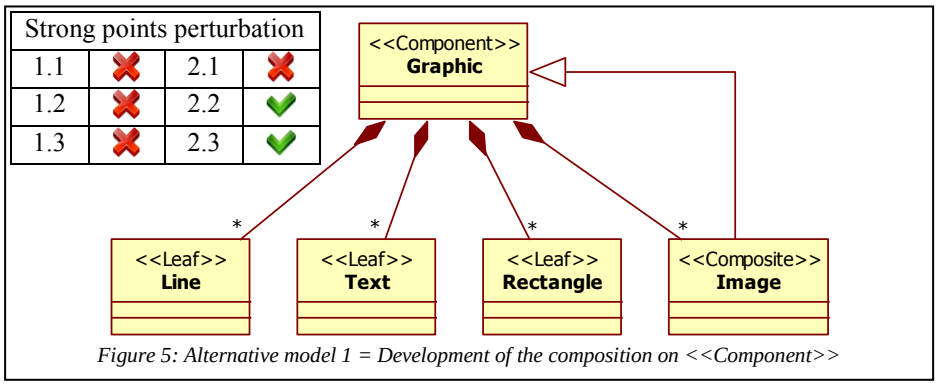


Figure 5: Alternative model 1 = Development of the composition on <<Component>>

In the alternative model 1, the composition is expressed by using directly the aggregation concept. This solution is valid, since the recursive composition is possible, even if through coding, the "Graphic" class is forced to store all the composition information. This causes the multiplication of operations on every type of element of the hierarchy. The use of the pattern would have avoided the redundancy of aggregation links, thus simplifying the structure and code.

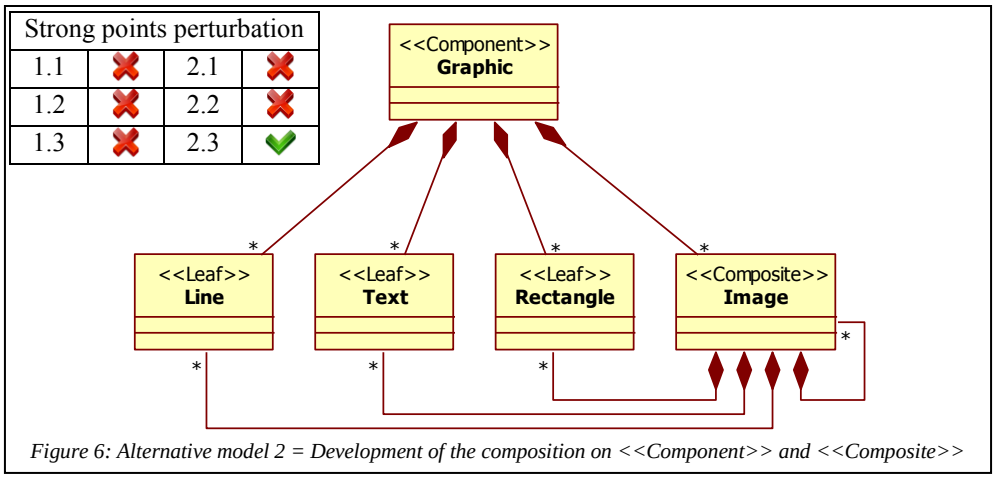


Figure 6: Alternative model 2 = Development of the composition on <<Component>> and <<Composite>>

The second alternative model reproduces the first alternative model structure by ignoring any factorization concept. It clearly appears that "Graphic" contains all the other classes, as well as "Image", by containing itself. This solution is valid, even if the lack of factorization will produce a "bad smelling code" (cf. the expression "Bad smells in code" from Kent Beck in Fowler, 1999, chapter 3).

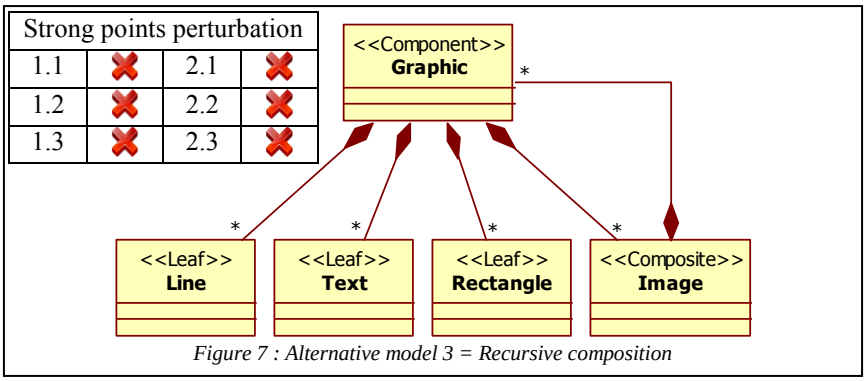


Figure 7 : Alternative model 3 = Recursive composition

The third alternative model looks like the second one, but "Image" is not composed of itself but of the "Graphic" class. So, the code will be less "smelly" than the one obtained by the second model, but the lack of generalization results in code duplication.

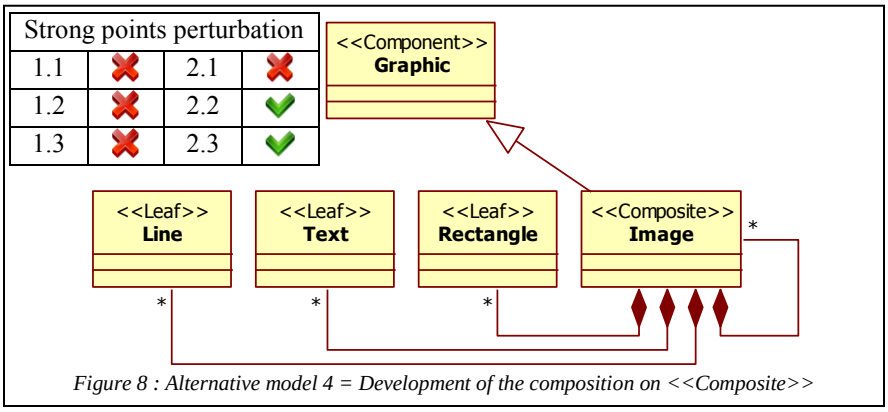


Figure 8 : Alternative model 4 = Development of the composition on <<Composite>>

In the fourth alternative model the composition is expressed such as it is described in the problem. We can locate in this model that an image is composed of other images which could be composed of lines, rectangles or texts. The "Graphic" class is only used as an access point for the client. However, the fact that "Line", "Rectangle" and "Text" do not inherit from "Graphic" will cause some duplication of code with excessive use of delegation.

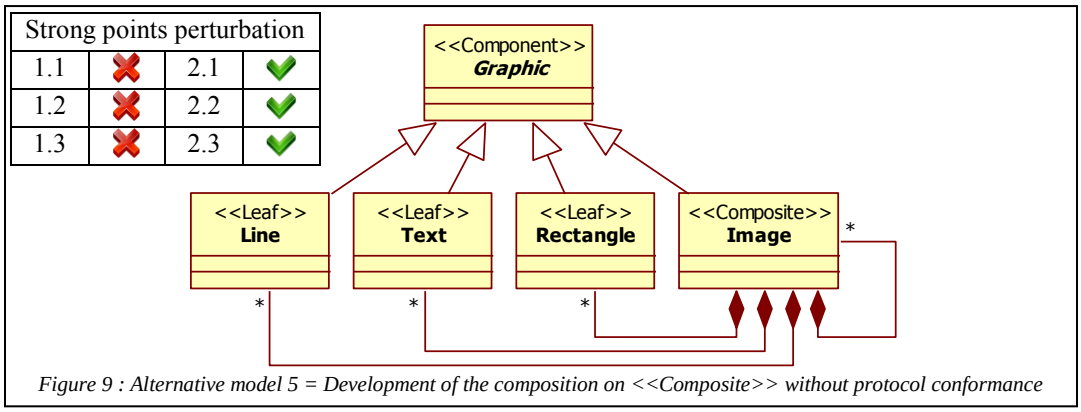


Figure 9 : Alternative model 5 = Development of the composition on <<Composite>> without protocol conformance

The fifth alternative model looks like the fourth, but with a protocol conformance on leaves. So, we can say that this model integrates the modifications avoiding the perturbation of the second strong point.

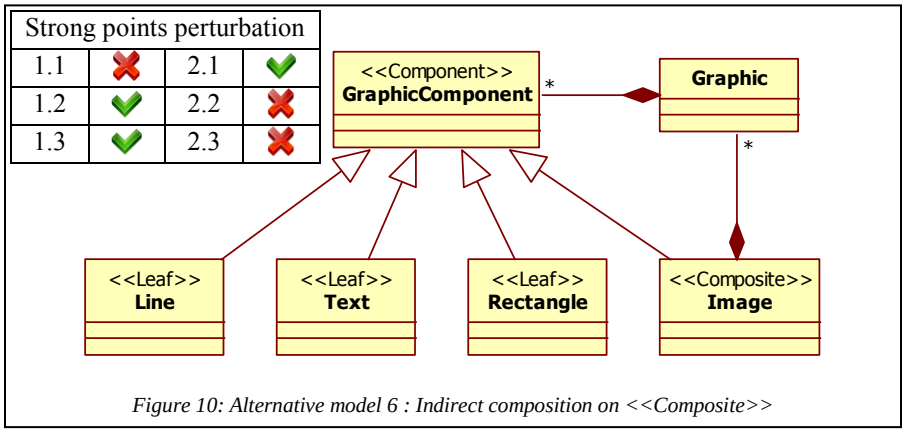


Figure 10: Alternative model 6 : Indirect composition on <<Composite>>

The sixth alternative model looks like the Composite pattern, but the complexity has been increased with one new indirection. This one is completely useless and it will cause multiplication of object references to store the tree structure.

In this experiment, the Composite pattern has a lot of alternative models. There are some interesting ideas to objects composition. The most frequent errors are a lack of factorization of composition relations, and the loss of the «Component» common interface. Composition is too often used with the detriment of inheritance. The structural proximity of alternative models for the Composite pattern may cause multiple detections for the same pattern. Table 1 resumes the state of every strong point for each Composite alternative model. For each strong point, we represent sub-features in this table with a dash if it is present and with a cross if it is deleted. In a first approach, we define "quality metrics" simply based on strong points. We consider the strong points qualitatively equivalent, and compute the metrics with their degree of perturbation.

	Composite alternative model number											
Strong point	**6**		**5**		**1**		**4**		**2**		**3**	
1.1	✗	$\frac{2}{3}$	✗	0	✗	0	✗	0	✗	0	✗	0
1.2	✓		✗		✗		✗		✗		✗	
1.3	✓		✗		✗		✗		✗		✗	
2.1	✓	$\frac{1}{3}$	✓	1	✗	$\frac{2}{3}$	✗	$\frac{2}{3}$	✗	$\frac{1}{3}$	✗	0
2.2	✗		✓		✓		✓		✗		✗	
2.3	✗		✓		✓		✓		✓		✗	
Score :	50%		50%		33.3%		33.3%		16.7%		0%	

Table 1: State of the strong points of the Composite alternative models

5. Decorator alternative models

In the Decorator pattern, we find three strong points with their sub-features:

1. Extensibility
   1.1. Addition or removal of a decorator does not need code modification.
   1.2. Addition or removal of an object to decorate does not need code modification.
2. Decoupling between decorator objects and objects to decorate.
   2.1. Minimal number of «Decorator» classes.
   2.2. Maximal factorization between decorators and object to decorate.
3. Decorators managing on program execution.
   3.1. Objects to decorate have any knowledge on decorators.
   3.2. A decorator may be decorated by another decorator.

The problem "*Design a system enabling to display visual objects on a screen. A visual object can be composed with one or more texts or images. If needed, the system must allow to add to this object a vertical scrollbar, an horizontal scrollbar, an edge and a menu (these additions may be cumulated)*" may be solvable with the Decorator design pattern. Figure 11 represents this problem instantiation.

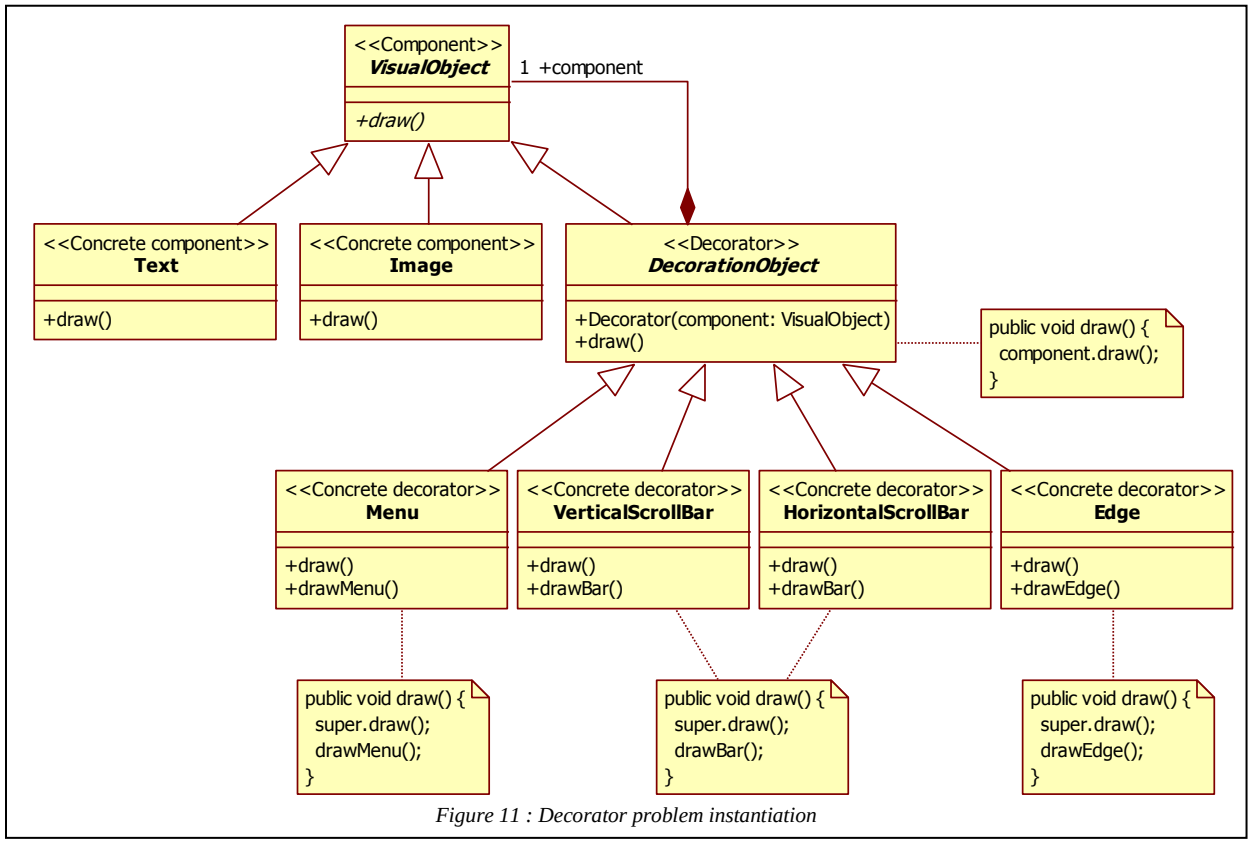


Figure 11 : Decorator problem instantiation

Figure 12 to Figure 17 represent the six alternative models taken from our experiment, with their strong points perturbations.

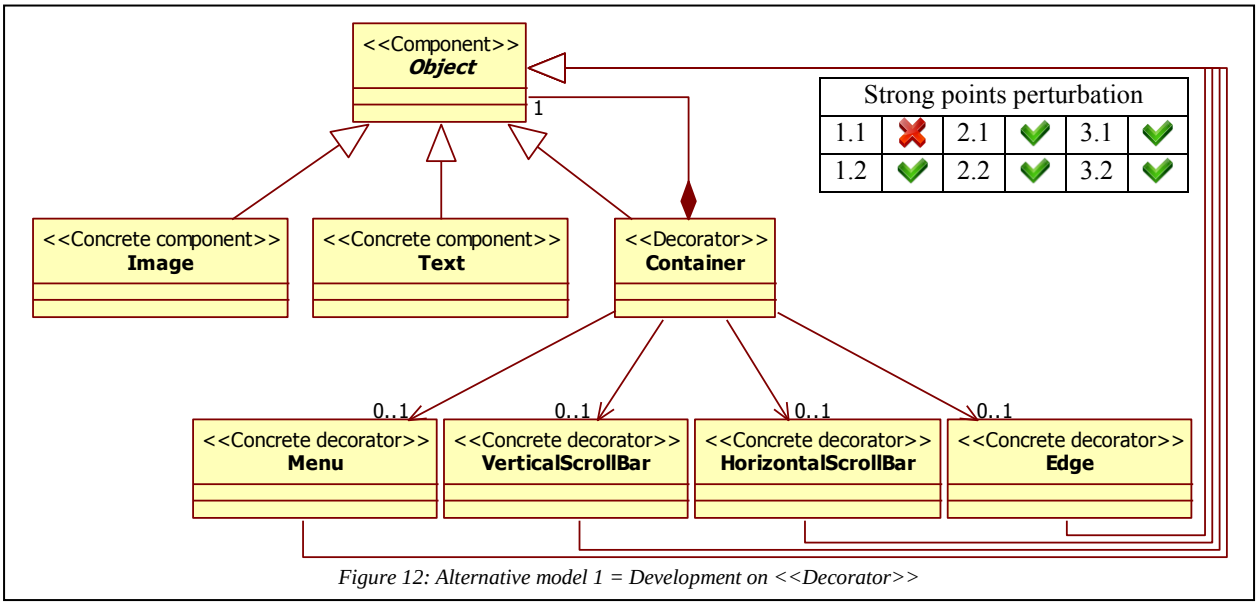


Figure 12: Alternative model 1 = Development on <<Decorator>>

In the first alternative model we recognize the Decorator pattern in which some association links are added to connect different «Concrete decorators» to a unique decorator manager. So, the number of associations depends on the number of decorators. Moreover the structure does not allow easily object decoration on the fly, because all decoration possibilities are bent in the «Decorator» class.

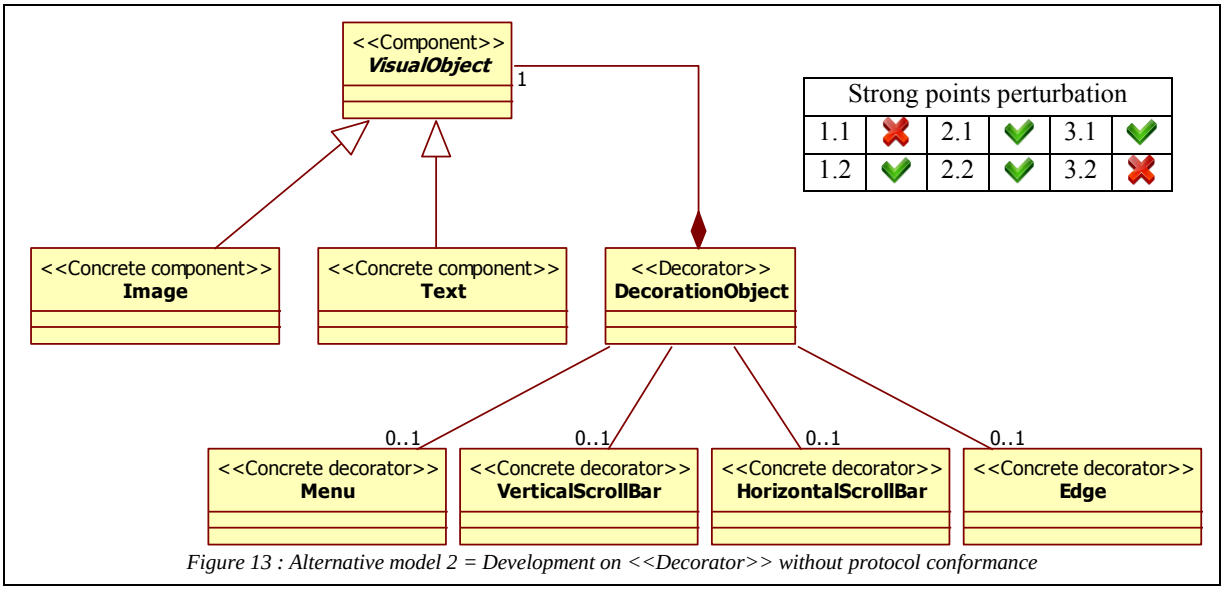


Figure 13 : Alternative model 2 = Development on <<Decorator>> without protocol conformance

The second alternative model does not have any inheritance link on the decoration part. So, there is no possibility to simply expand the model. Indeed, as in the first alternative model, adding or removing a «Concrete decorator» implies code modification in «Decorator» class.

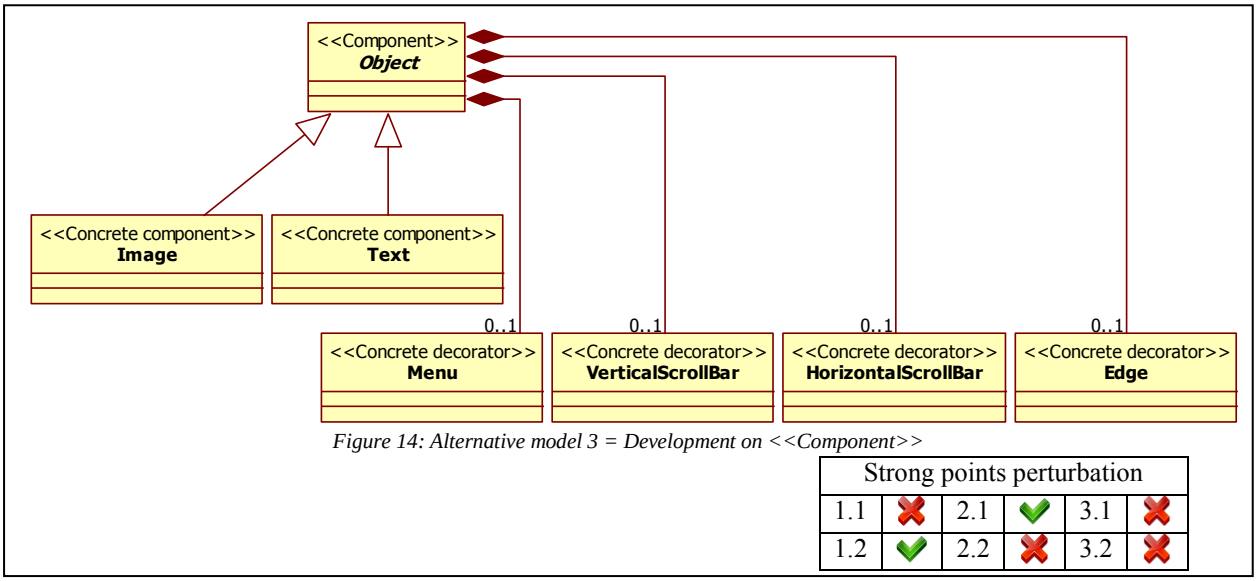


Figure 14: Alternative model 3 = Development on <<Component>>

Strong points perturbation					
1.1	✗	2.1	✓	3.1	✗
1.2	✓	2.2	✗	3.2	✗

In the third alternative model, the decoration of objects is expressed directly with compositions on «Component». This model is valid because it respects the awaited behavior. However, even if we can program the decoration on the fly in one class with much effort, it is not relevant to object-oriented design features. We can see with this alternative model that there is no class to which we can associate the «Decorator» role. To substitute this alternative model by the pattern, the designer must create an abstract super-class of all «Concrete decorator» classes.

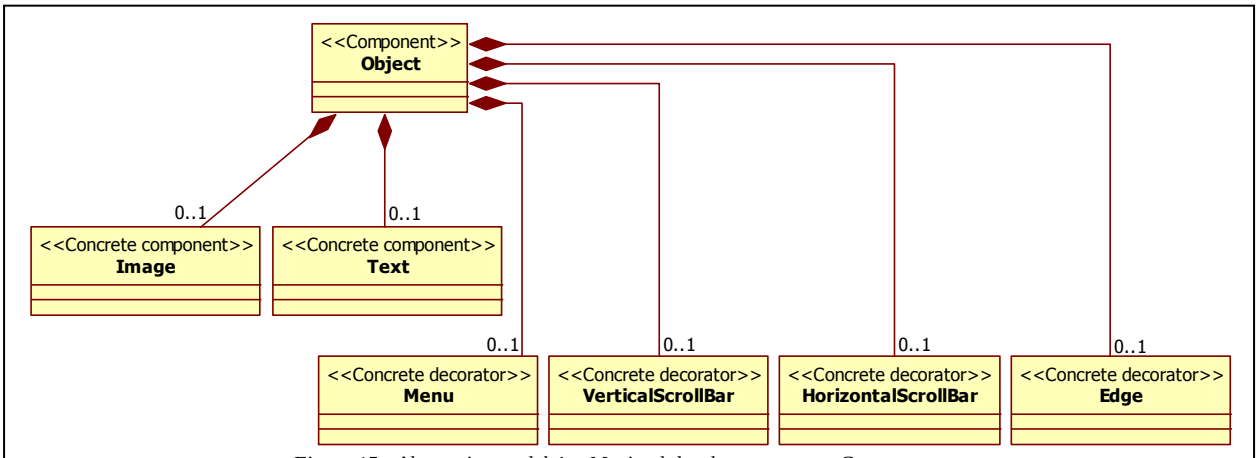


Figure 15 : Alternative model 4 = Maximal development on <<Component>>

Strong points perturbation					
1.1	✗	2.1	✓	3.1	✓
1.2	✗	2.2	✗	3.2	✗

The fourth alternative model is probably the worst case on the decoration managing. The «Component» class has too many responsibilities, that implies a unique access point for everything. There is not any possibility of extension.

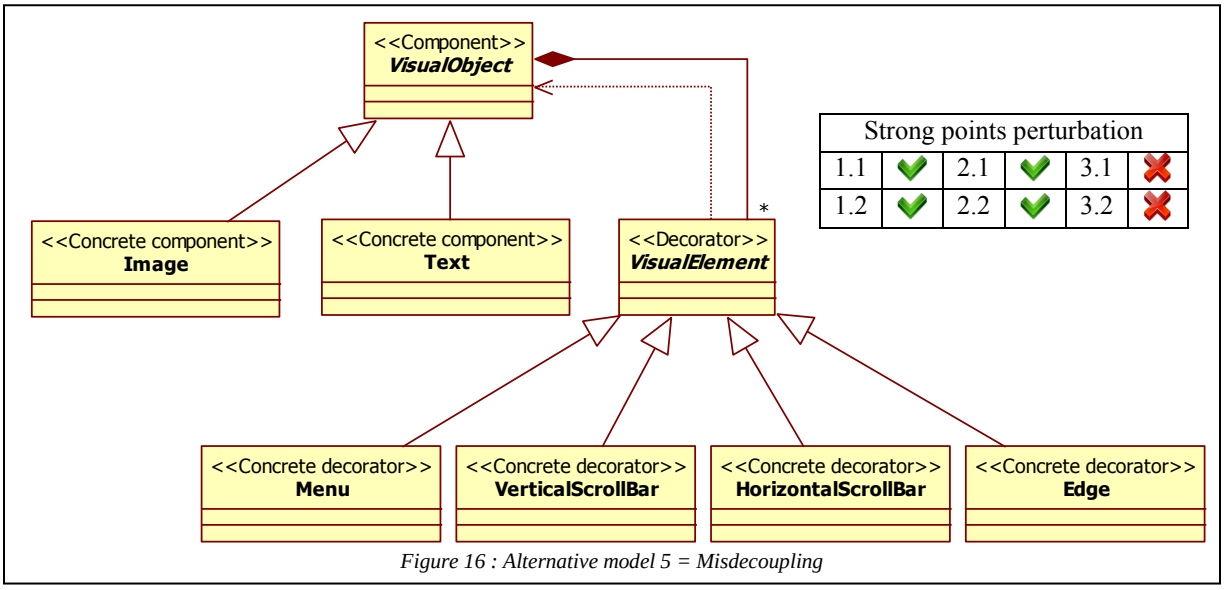


Figure 16 : Alternative model 5 = Misdecoupling

In the fifth alternative model we remark that the composition link between «Decorator» and «Component» is reversed. That implies a non-structural dependency and a more complicated processing to decorate objects. Indeed, as «Concrete decorators» must know the objects they have to decorate, each decorating method has an additional parameter. This processing implies that the cumulation of decorators modifies the state of the objects to decorate, limiting the advantage of the decoupling.

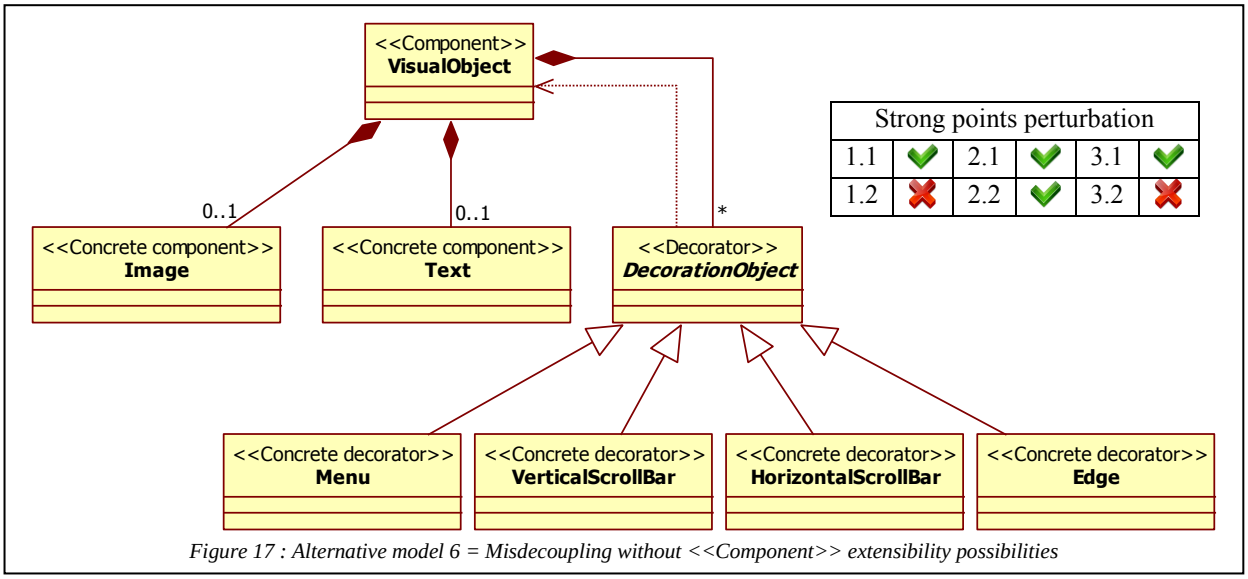


Figure 17 : Alternative model 6 = Misdecoupling without <<Component>> extensibility possibilities

The sixth alternative model is the same as the fifth but with the addition of a decorator that does not modify the state of the objects to decorate. However, there is no extensibility point on the objects to decorate. The «Component» class must know a priori all the «Concrete component» classes.

For this problem, the decoration is implemented thanks to composition links. Except for the decoration on the fly, these alternative models respect the original problem. However, with a structural view, it is difficult to know if the model works. Indeed, could we decorate cumulatively a component with an edge and a vertical scrollbar? The structural proximity of alternative models for the Decorator pattern may cause multiple detections for the same pattern. Table 2 resumes the state of every strong point for each Decorator alternative model.

	Decorator alternative model number											
Strong point	**1**		**2**		**6**		**5**		**3**		**4**	
1.1	✗	1/2	✗	1/2	✓	1/2	✓	1	✗	1/2	✗	0
1.2	✓		✓		✗		✓		✓		✗	
2.1	✓	1	✓	1	✓	1	✓	1	✓	1/2	✓	1/2
2.2	✓		✓		✓		✓		✗		✗	
3.1	✓	1	✓	1/2	✓	1/2	✗	0	✗	0	✓	1/2
3.2	✓		✗		✗		✗		✗		✗	
Score :	83.3%		66.7%		66.7%		66.7%		33.3%		33.3%	

Table 2: State of the strong points of the Decorator alternative models

6. Bridge alternative models

In the Bridge pattern, we find two strong points with their sub-features:

1. Extensibility
   1.1. Addition or removal of a concrete implementor does not need code modification.
   1.2. Addition or removal of a refined abstraction does not need code modification.
2. Decoupling between abstraction and implementor.
   2.1. Minimal number of concrete implementor.
   2.2. Maximal factorization of the link between abstraction and implementor.

The problem "*Design a system enabling to display on a screen some empty windows, applicative windows and windows with icons. A window can have several different styles depending on the platform used. We consider two platforms, XWindow and PresentationManager. The client code must be written independently and without knowledge of the future execution platform.*" may be solvable with the Bridge design pattern. Figure 18 represents the instantiation of this problem.

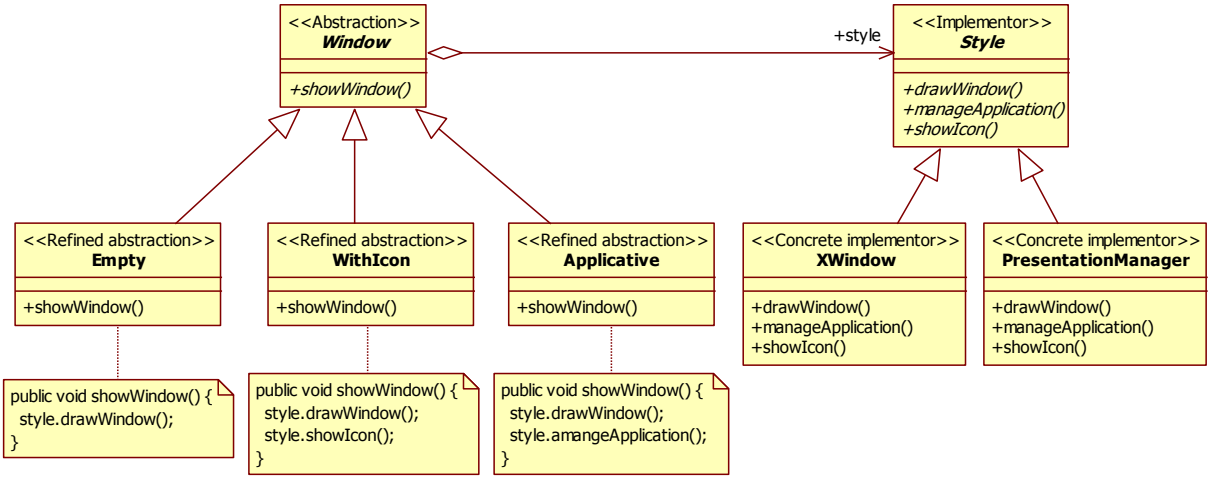


Figure 18 : Bridge problem instantiation

Figure 19 and Figure 20 represent the five alternative models taken from our experiment, with their strong points perturbations.

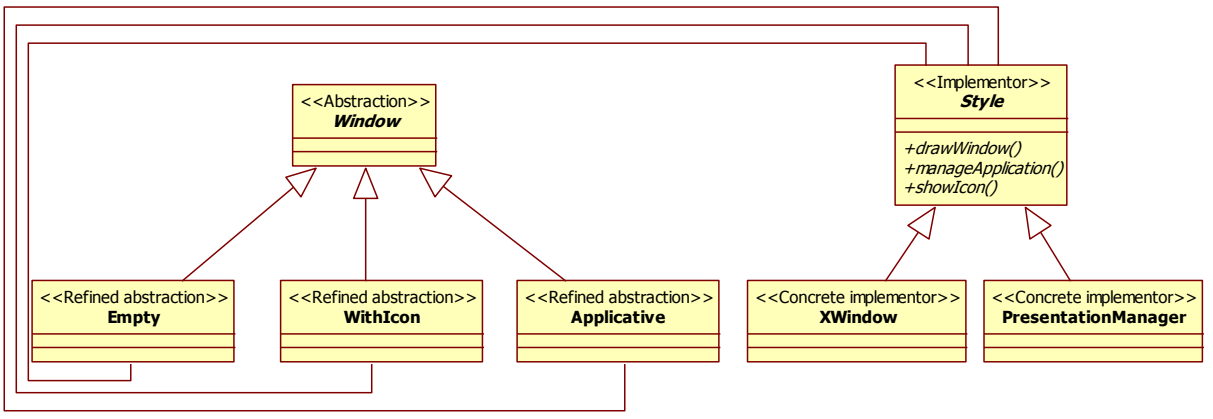


Figure 19 : Alternative model 1 = Bridge development

Strong points perturbation			
1.1	✔	2.1	✔
1.2	✘	2.2	✘

In the first alternative model, the windows are correctly separated from the environment but the associations between «Implementor» and «Refined abstraction» are not factorized. Therefore, there will be no problem if we add a new platform, but if we add a new window, a new association link will be added to the «Implementor» class. However, with this model, it is possible to have some windows types with different styles. If the problem imposed a unique style, the designer should implements this integrity constraint.

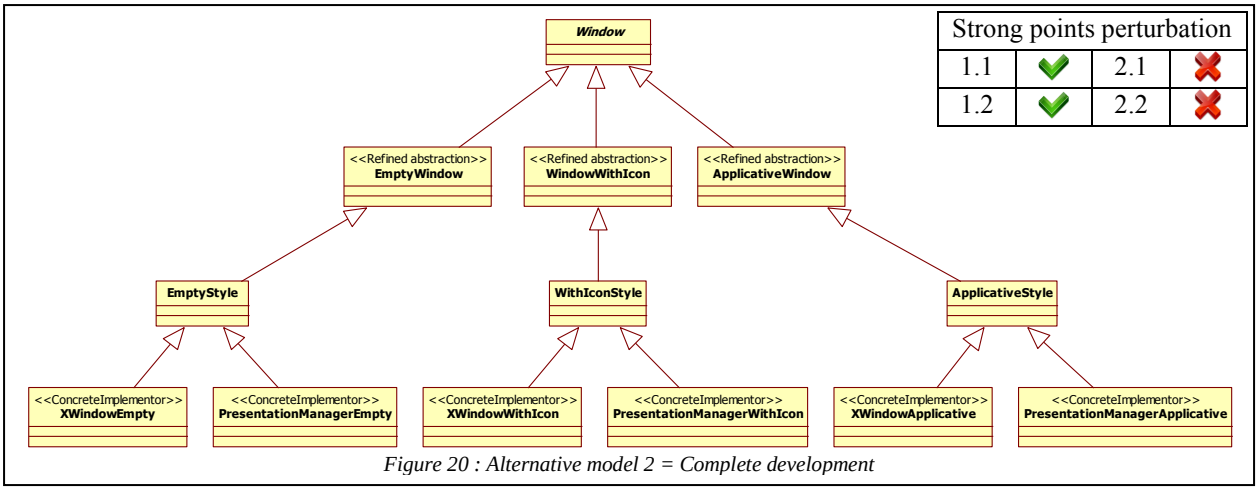


Figure 20 : Alternative model 2 = Complete development

The second alternative model represents typically a very bad design, because there is a class multiplication between «Refined abstraction» and «Concrete implementor». This case is presented in the GoF catalogue (Gamma et al., 1995) to illustrate the interest of the *Bridge* pattern.

Table 3 resumes the state of every strong point for each Bridge alternative model.

	Bridge alternative model number			
Strong point	**1**		**2**	
1.1	✗	$\frac{1}{2}$	✓	1
1.2	✓		✓	
2.1	✗	$\frac{1}{2}$	✗	0
2.2	✓		✗	
Score :	**50%**		**50%**	

Table 3: State of the strong points of the Bridge alternative models

7. Conclusion

Our experiment concerns all structural GoF patterns. For patterns not presented in this paper, we are forced to adapt the method. To detect Adapter, we need to add new visibility stereotypes in UML, for example: «Black box» or «Glass box». The remarkable features of the Facade pattern integrate object-oriented metrics: the coupling between packages, the multiplicities between classes, the number of associations for a class… However, Flyweight and Proxy patterns cannot be detected with our method. These patterns increase the structural complexity of the models, which does not allow the detection of remarkable structural properties.

We have done a new experiment with new participants. The problems have been modified and some creational and behavioral patterns have been integrated. We are currently studying these experiments and we hope that we will obtain new alternative models, which variously perturb the strong points. Moreover, in order to upgrade our alternative models detection method, we hope define behavioral features.

8. References

Arcelli Fontana F., Raibulet C., Tisato F., "Design Pattern Recognition", in *Proceedings of the ISCA 13th IASSE*, Nice, France, July 1-3, 2004, pages 290-295.

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.

Buschmann F., Meunier R., Rohnert H., Sommerlad P., Stal M., "Pattern-Oriented Software Architecture", John Wiley & Sons, August 1996.

Chikofsky E. J., Cross J. H., "Reverse engineering and design recovery: A taxonomy", in *IEEE Software*, 7(1), page 13-17, January 1990.

O'Cinnéide M., Nixon P., "A Methodology for the Automated Introduction of Design Patterns", in *ICSM '99: Proceedings of the IEEE International Conference on Software Maintenance*, IEEE Computer Society, 1999, 463.

Dunsmore A.P., "Comprehension and Visualisation of Object-Oriented code for Inspections", *Technical Report*, EFoCS-33-98, Computer Science Department, University of Strathclyde, 1998.

Eden A. H., Yehudai A., Gil J., "Precise specification and automatic application of design patterns", in *ASE '97: Proceedings of the 12th international conference on Automated software engineering (formerly: KBSE)*, IEEE Computer Society, 1997, 143.

Fowler M., "Analysis patterns: reusable objects models", Addison Wesley Longman Publishing Co, Inc., 1997.

Fowler M., Beck K., Brant J., Opdyke W., Roberts D., "Refactoring: Improving the Design of Existing Code", Addison-Wesley Professional, 1999.

France R. B., Kim D., Ghosh S., Song E., "A UML-Based Pattern Specification Technique", in *IEEE Trans. Softw. Eng.*, IEEE Press, 2004, 30, 193-206.

Gamma E., Helm R., Johnson R., Vlissides J., "Design Patterns: Elements of Reusable Object-Oriented Software", Addison Wesley Professional, 1995.

Guennec A. L., Sunyé G., Jézéquel, J., "Precise Modeling of Design Patterns", in UML, 2000, 482-496.

Guéhéneuc Y. G., Albin-Amiot. H., "Using Design Patterns and Constraints to Automate the Detection and Correction of Inter-Class Design Defects", in *Proceedings conference TOOLS*, July 2001, pages 296-305.

Huston B., "The effects of design pattern application on metric scores", in *Journal of Systems and Software*, 58(3), Elsevier Science, September 15, 2001, pages 261-269.

Larman C., "Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development", (3rd Edition), Prentice Hall PTR, 2004.

Mak J. K. H., Choy C. S. T., Lun D. P. K., "Precise Modeling of Design Patterns in UML", in *ICSE*, IEEE Computer Society, 2004, 252-261.

Meyer B., "What is an Object-Oriented Environment? Five Principles and their Application", in *Journal of Object-Oriented Programming*, Volume 6, Number 4, July-August 1993, pages 75-81.

Neptune, Nice Environment with a Process and Tools using Norms - UML, XML and XMI - and Example, [w] http://neptune.irit.fr, 2003.

Sunyé G., Pollet D., Le Traon Y., Jézéquel J.M., "Refactoring UML Models", in *Proceedings of UML 2001*, pages 134-148.