Sharing Bad Practices in Design to Improve the Use of Patterns

Cédric BOUHOURS, Hervé LEBLANC, Christian PERCEBOIS
IRIT – MACAO team – University of Paul Sabatier
118 Route de Narbonne
31062 TOULOUSE CEDEX 9 FRANCE
{bouhours, leblanc, percebois}@irit.fr

ABSTRACT

In order to guarantee the use of good analysis and design practices and an easier maintenance of software, analysts and designers may use patterns. To help them, we propose models inspection in order to detect instantiations of "spoiled pattern" and models reworking through the use of the design patterns. As a design pattern allows the instantiation of the best known solution for a given problem, a "spoiled pattern" allows the instantiation of alternative solutions for the same problem: requirements are respected, but architecture is improvable. We have collected a set of alternative solutions and deduced the corresponding spoiled patterns. We have defined a first catalog of these improvable practices from several experiments with students. To overcome the limits imposed by this method (restricted public, limited problems and tiresome validation process), we would like open this problematic to the expert community. To achieve this, we propose a collaborative website sharing bad practices in object oriented design to improve the use of patterns.

Categories and Subject Descriptors

D.2.2 [**Software Engineering]**: Design Tools and Techniques; D.2.13 [**Software Engineering]**: Reusable Software - Reuse models

General Terms

Design

Keywords

Design patterns; Spoiled pattern

1. INTRODUCTION

In order to guarantee the use of good analysis and design practices and an easier maintenance of software, analysts and designers may use patterns. A pattern is a consensus on the most efficient solution to solve a given problem [1]. The use of a pattern is the guarantee to re-use the most adequate solution and thus, to maintain a consensual quality with analysis and design.

To assist designers, the design patterns catalog of Gang of Four [2] provides a set of solutions. If a designer uses the GoF on his design, we consider that he is ensured to select the best known solution to solve his problems. However, if some errors persist, or if the designer is not accustomed to use design patterns, design defects may remain. To limit or avoid this risk, some works help the use of the patterns. In particular, patterns were classified and described in several manners to help their selection [3] [4] [1] [5] [6] [7] [8] [9], for example, in classifying the patterns according to their intent or in formalizing the problem they solve. Another way is to check how a pattern can be well-integrated in an existing solution [10] [11] [12] [13] [14].

Besides these approaches, we defined the concept of spoiled pattern [15]. Its main interest is to identify a bad practice with respect to design patterns. A spoiled pattern is a pattern which corresponds to a deterioration of the intrinsic qualities of a design pattern. The structural differences between a design pattern and a spoiled pattern cause an efficiency damage to solve a problem in an adequate way.

For the same design problem, we consider that several solutions exist: the ones recognized as the most powerful and the most efficient, i.e., using the adequate design pattern correctly, and the others, certainly less powerful and less efficient using spoiled patterns. We suggest detecting and correcting these others solutions by a tooled design review activity. The aim is to inspect models to search fragments characteristic of typical bad design practices and to substitute them by design patterns, after communication with the designer.

In this paper, we present in Section 2 the concept of spoiled pattern. We show how a spoiled pattern can solve the same problem as a design pattern, but in a different way: the problem is solved but some intrinsic properties of the design pattern are damaged. These properties called strong points can be valuated to define the level of degradation of the pattern. This leads to alternative fragments we define as spoiled patterns instantiations whose intent conforms to the corresponding design pattern. Section 3 gives an overview of a design review activity that we have defined in order to detect such alternative fragments within a model. This detection uses a spoiled pattern catalog we present in Section 4. We discuss on the way we abstract spoiled patterns from experiments with design problems addressed to students. The last section is devoted to a collaborative Web site we currently elaborate. By submitting new problems and their alternative solutions, its main objective is to complete the catalog with new spoiled patterns, and so to share bad practices in design.

2. THE SPOILED PATTERNS

Since a design pattern was approved, tested and validated by an expert community, we estimate that it provides the best known solution to a given problem. This problem is introduced in a generic and adaptable form. Thus, the design pattern is a reusable and adaptable architecture to a problem in a context. Moreover,

as it is a proof of development facility and time-saver during the design, thanks to the best design practices which it brings, we make the hypothesis that it represents the optimal architecture (classes and messages arrangement) to solve a specific problem type.

Axiom 1: *"A design pattern" is the optimal reusable micro-architecture for one and only one problem type.*

By micro-architecture, we gather the classes fitting, the attributes and methods distribution, and the structure of the messages exchanged between the classes. To adapt a pattern on a problem, the problem must conform to the problem type solvable by the pattern.

Corollary 1: *For each design problem solvable by a design pattern, "the best solution" is the adaptation of the design pattern to the context of the problem.*

As design patterns are generic and describe a general context, it is necessary to adapt them to the context of the problem we want to solve. So we define the processes allowing the use of design patterns. The instantiation process consists in adapting a design pattern to the particular context of a problem. The abstraction process is the inverse.

2.1 Definitions

In the following definitions, we admit that a given problem is solvable by the instantiation of a design pattern.

Definition 1: *"An alternative solution" is a valid solution for a given problem, but with a different architecture compared to the best solution.*

Thus, the requirements of the design are respected but the relations inter-classes are different or/and there is not the whole pattern participants. According to our first axiom, we consider that if the designer were confronted to a design problem solvable by a design pattern, and if he did not use it, he has solved the problem with an alternative solution.

We can deduce the following corollary from it:

Corollary 2: *An alternative solution is not the best solution for a given problem and therefore is substitutable with the instantiation of the design pattern corresponding to the problem.*

Since an alternative solution is valid for a given context, it is possible to abstract it in order to obtain a generic model allowing the solving of a certain problem type, but in an inadequate way.

Definition 2: *"A spoiled pattern" is the abstraction of an alternative solution, in the same manner as a design pattern is the abstraction of the best solution. A spoiled pattern is connected to one and only one design pattern.*

A spoiled pattern is comparable to a design pattern. Structurally, it is represented at the same level of granularity. It is reusable to produce models which solve problems. Thus, for a problem type, there is a set of spoiled patterns allowing the production of a set of non optimal solutions.

We could say that the instantiations of spoiled patterns produce the same results as incomplete or failing design patterns instantiations. Thanks to their structural descriptions, we are able to identify the fragments structurally comparable with the spoiled patterns. This comparison is only structural, and therefore the intent of the fragment detected must be validated by the designer himself. Indeed, the structural concordance does not guarantee that the fragment intent conforms to the spoiled pattern.

Definition 3: *"An alternative fragment" is a model fragment such as its structure corresponds to the structure of a spoiled pattern instantiation and whose intent conforms to the corresponding design pattern.*

Each alternative fragment detected in a model represents a potential fragment. A fragment becomes effective when the designer confirms his intent during a review activity detailed in section 3.

We chose the term "spoiled" to describe this new type of pattern, because it corresponds to a deterioration of the intrinsic qualities of the design patterns. Thus, spoiled patterns are substitutable by the corresponding design patterns. The structural differences between a design pattern and a spoiled pattern cause an efficiency to solve a problem type in an adequate way.

Definition 4: *"The strong points" of a design pattern express the criteria of architecture and the factors of software quality brought by its use. These criteria are partially deduced from the "consequence" section of the GoF catalog and from the design defects noted during the use of the spoiled patterns. They valorize why the design pattern is the best known solution for a problem.*

The alternative solutions are in fact more or less effective to solve a problem. It is possible to quantify a degree of damage by considering the valuation of the strong points of a pattern. As the strong points of the pattern characterize the effectiveness and the quality of the solution, we can say that the substitution of an alternative fragment by an optimal fragment corrects the design defects generated by the use of the spoiled pattern.

The detection of alternative fragments in a model can evoke bad smells and the explanation about their defects in referring to design patterns can evoke anti-patterns.

2.2 Illustration

We illustrate our concepts by the *Composite* design pattern, described in Figure 1. We deliberately chose to represent the design patterns by class diagrams only, inspired by the "structure" section of GoF, with the pattern participants and their relations only (associations and inheritance). We omitted the methods of each participant, on the class diagram, when they were indicated in the GoF.

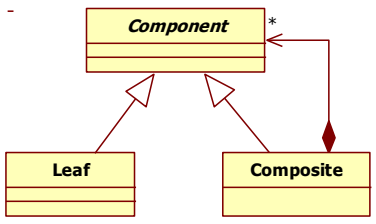

Figure 1: The *Composite* design pattern

The intent of the *Composite* pattern is "*compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly*" [2]. Applying the axiom 1, this pattern is the best known solution to solve the following global problem:

composition of objects, building tree structures and nesting objects [8]. In the ontology proposed by [8], each problem is derived from the intent item of the corresponding pattern and decomposed into sub-problems. Then several patterns can share one or more sub-problems, but only one pattern is the best candidate to solve a global problem.

The *Composite* pattern introduces three participants: an abstract *Component*, a *Composite*, and a *Leaf*. The abstract *Component* defines a common interface to the composed objects and to composition management, and offers a unique access point for the client. This entity allows the factorization of the composition on the composites and the leaves. The *Composite* participant manages the relation of composition and recursively delegates the operations along the tree structure. The *Leaves* represent the terminal elements of the tree structure.

Now let us consider a specific problem statement, inspired by the GoF: *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.*

This statement implies that the problem type relates to a hierarchical composition of objects, the hierarchy being articulated around the concept of *Image*. To instantiate the *Composite* pattern on this problem, we must identify the problem elements having the same responsibilities as each participant of the pattern. The concept of *Image* has the same responsibilities as the *Composite* participant. The classes *Line*, *Text* and *Rectangle* constitute the terminal elements of the hierarchy and thus have the same responsibilities as the *Leaf* participant. Lastly, we can consider that *Graphic* constitutes the generic element of the hierarchy of composition, which brings it closer to the responsibilities for the *Component*. We obtain an instantiation of the *Composite* pattern, and, in agreement with our hypothesis, we can say that Figure 2 represents the best solution to the problem introduced above.

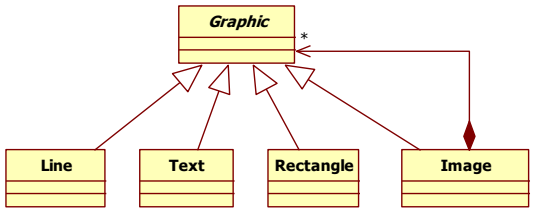

Figure 2: The best solution for the problem

Figure 3 introduces an alternative solution of the preceding problem. In this solution, we can identify that an image is composed of other images which can be composed of lines, texts and rectangles. So, the requirements of the problem are respected. The *Graphic* class is used to support the factorization of the protocols and to be the unique access point to the client. However, the fact that the classes *Line*, *Rectangle* and *Text* are attached to *Image* involves code modifications if new classes are added, with the responsibilities of *Leaf* or *Composite*. Thus, if a new *Circle* class is added as *Leaf*, the *Image* class will have to manage this new reference, which will involve a code modification of the *Image* class.

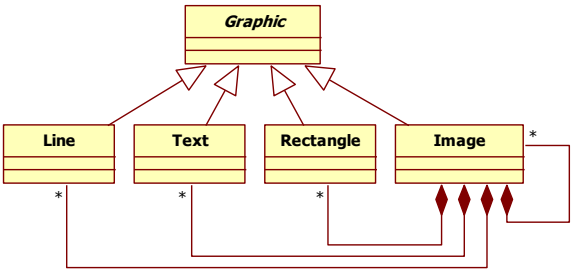

Figure 3: An alternative solution for the problem

In order to detect an alternative solution in a model whatever the context of the problem, it is necessary to abstract it. This abstraction enables us to obtain a "generic" spoiled pattern, able to be adapted to any context of problem. This abstraction enables us to consider a spoiled pattern as a generating base of alternative fragments.

The abstraction process of an alternative solution requires to identify the pattern participants, then to carry out a "reduction" making it possible to preserve only one class per participant of the pattern. However, some alternative solutions do not use the totality of the participants, which implies that some of the classes have the responsibilities of several participants.

The first step of this abstraction process consists in marking each class with the name of one of the participants of the pattern having the same responsibilities. The abstract *Graphic* class offers a common interface to all the other classes and a unique access point for the client. Thus, it has the responsibilities of the *Component* participant. The *Image* class manages the composition and represents, by its recursive connection the *Composite*. Finally the classes *Line*, *Text* and *Rectangle* are clearly the terminal elements of the tree structure and thus have the same responsibilities as the *Leaf* participant.

This class marking of an alternative solution is done manually, since it requires an analysis of the semantics of the classes. The result, summarized by Figure 4, shows the marking of the classes of the alternative solution by the participants of the *Composite* pattern.

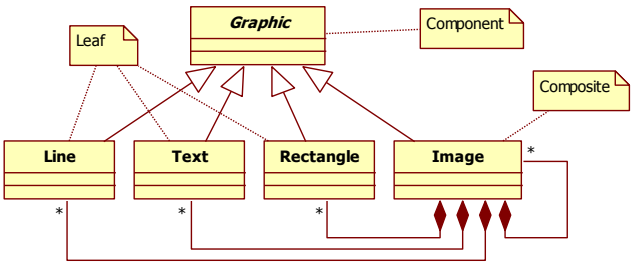

Figure 4: The marked alternative solution

After the marking, the second step of the abstraction process consists in preserving, only one times, each participant in the same way as in the alternative solution. This reduction can be complex on some participants when several classes have the same responsibilities.

In our case, we deduce a model with three classes *Component*, *Composite* and *Leaf*, substituting respectively the *Graphic* class, the *Image* class and one of the classes *Text*, *Line* or *Rectangle*. Then, we obtain the structure of a spoiled pattern of

the *Composite* where the composition is developed on the *Composite* class. Figure 5 presents a spoiled pattern for the *Composite* design pattern, named "*development of the composition on <<Composite>>*".

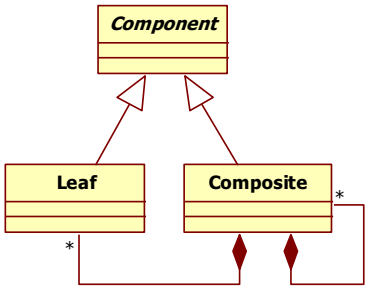

Figure 5: The spoiled pattern *development of the composition on <<Composite>>*

Starting from several alternative solutions of the same type of problem, we obtained a set of spoiled patterns. To classify the spoiled patterns, we quantified their degree of damage thanks to the strong points of the design pattern concerned. Indeed, each spoiled pattern has only a part of the strong points of the pattern. It is what explains its damage.

For the *Composite* pattern, the maximal factorization of the composition and the standardization of the protocol, thanks to inheritance links, enable us to say that the strong points of the pattern are "*decoupling and extensibility*" and "*uniform protocol*". As the composition of the spoiled pattern of Figure 5 is expressed with a reflexive connection and with a development on all the leaves, a design defect appears, consequence of the damage of the strong point "*decoupling and extensibility*". Factorization is not maximal and the coupling between *Leaf* and *Composite* imposes code modifications. However, as there are always inheritance links, the spoiled pattern does not degrade the strong point "*uniform protocol*".

This characterization of the spoiled patterns enables us to present to the designer the advantage of the substitution of the fragment detected by the corresponding design pattern.

Table 1 summarizes the degradation of the strong points by the spoiled pattern. The strong points of the *Composite* pattern damaged by the spoiled pattern are described preceded by the symbol ✘ contrary to preserved strong points which are preceded by ✔.

Table 1: The strong points valuation of the spoiled pattern

Decoupling and extensibility	
✘	Maximal factorization of the composition.
✘	Addition or removal of a leaf does not need code modification.
✘	Addition or removal of a composite does not need code modification.
Uniform protocol	
✔	Uniform protocol on operations of composed object.
✔	Uniform protocol on composition management.
✔	Unique access point for the client.

2.3 Bad smells and antipatterns

We now position "spoiled pattern" term compared to "bad smells" and "antipatterns".

2.3.1 Bad smells

Kent Beck and Martin Fowler have introduced the term "bad smells" in [16]. These bad smells are a set of clues in the code suggesting bad design practices. They allow the identification of the parts of the code to restructure in order to correct the problems, and the procedures to follow to carry out this reorganization. For example, the code duplication in a program is a bad smell which can be corrected by the refactoring "extract method" [16]. It consists in adding a method in a class so that it factorizes the parts of code concerned.

An alternative fragment indicates where a defect is being able to generate undesirable effects on the model and target a zone which would have to be restructured. Whereas the bad smells were defined to target pieces of code, the spoiled patterns target fragments of model. Thus, identifying alternative fragments can be comparable with a search of bad smells in designs. As an alternative fragment comes from the instantiation of a spoiled pattern, we consider that the spoiled patterns are bases generating bad smells in designs.

2.3.2 Antipatterns

There exist two manners to define an antipattern. Whereas a design pattern presents the best solution to be followed to solve a problem, the antipattern presents a learned lesson. It describes the effects resulting from bad design practices and gives the procedure to follow for tending towards a better software quality. Then, an antipattern makes it possible to check or supervise bad practices [17]. An antipattern can also represent good design practices, but which used in an excessive way produce, at last, consequences more harmful than the anticipated results [18]. In all the cases, an antipattern suggests a suite of refactorings. An antipattern is described by the explanations of the defects and by a reorganization process which explains how to pass from the bad to a good solution. As example, let us quote the antipattern "makes an active attempt", in concurrent programming, which consists in testing a condition until it is checked. This antipattern can be corrected by scrolling events or signals.

Let us consider that spoiled patterns are antipatterns, but with a finer precision. The spoiled pattern does not give information allowing the correction of the bad solution. Thanks to the fine description of the bad solution, the spoiled pattern can be detectable automatically, which is not the case, nor the goal, of the antipatterns. A spoiled pattern permits to verify if a "bad manner to make" was not used, and it is directly related to a design pattern. The set of useful operations of reorganization to substitute it is much more precise than a refactoring suggested by an antipattern.

3. A DESIGN REVIEW ACTIVITY

To concretize the concepts presented and in order to be able to integrate it in a development process, we conceived and implemented a design review activity, as well as it exists code inspections for improving programming quality and productivity [19]. This activity is decomposed in three steps [23]: detection of alternative fragments on a model expressed in XMI format [20], communication with the designer to check the intent of the

detected fragments, and model refactorings to integrate the design patterns.

3.1 A case study

Figure 6 presents the model to analyze. It was found in a subject of an object-oriented programming supervised practical work and constitutes the model in input of our activity.

Initially, we can say that this UML class diagram represents a basic architecture of a files system management. The authors of this model took care that the good design practices are respected:

- *Inheritance between classes.* A uniform protocol for every *FileSystemElement* is encapsulated by a corresponding abstract class. *Directories* and *Files* must respect this protocol via inheritance relationships. We can note that all concrete classes are derived directly or indirectly from an abstract class. This rule enforces the emergence of reusable protocols.
- *Management of reference and delegation.* There are composition links between container and components. A *directory* object manages some references to *files* and *directories* objects. A *directory* object delegates some actions to sub-directories and files, for example, the *getSize*() method.

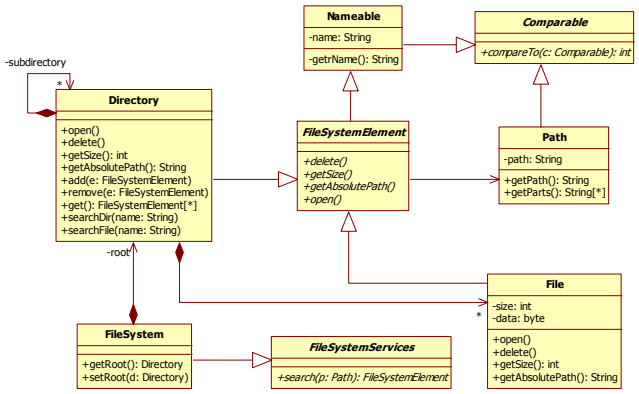

Figure 6: The model to analyze

A good effort on design was carried out, producing a design in good quality. However, this model presents some design defects. Although there is a uniform protocol offered by the *FileSystemElement* class, the management of composition relationships towards the other types of data present in the hierarchy is duplicated. Indeed, the *Directory* class manages independently connections on *Files* and those on itself.

It is enough to consider the two following evolution scenarios to discredit our first opinion on the quality of the design:

- The first is the addition of new terminal types in the tree structure, for example, symbolic links in UNIX file systems. This evolution requires the management of this new type of link by the *Directory* class and then requires code modification and code duplication in this class.
- The second is the addition of new non terminal types in the tree structure, for example archive files in UNIX or in Java environment. We can consider that an archive file has the same functionalities as a *Directory*. This evolution requires a reflexive link on an archive file class and the duplication of all links that represent composition links in the tree structure. Moreover directories can contain archive files too, then duplication of management of composition and code modification is required for the *Directory* class.

These two scenarios show a decoupling problem (each container manages a part of the composite structure) and an extensibility limitation (every modification will require existing code modification for the addition of a new type of terminal or non terminal element of the composition hierarchy). Therefore, this model can be improved. Furthermore, when the authors have implemented this model, they realized that there were defects. They adapted their code to correct them, without changing the design model.

3.2 An activity execution

To be able to execute the activity, we developed the Triton software, whose screenshot is presented on the Figure 8 [23]. It reaches the whole of a catalog of the spoiled patterns and uses the Neptune platform [21] to carry out research with OCL queries [22]. The constitution of the catalog is presented in Section 4.

The first step of the activity consists in seeking fragments which correspond structurally to possible instantiations of spoiled patterns. After the loading of the model to analyze in Triton, the OCL queries deduced from the structure of each spoiled pattern are carried out on the model, according to the selection done by the designer in the principal window of Triton.

In the case of our model, Triton has identified the fragment {*FileSystemElement*, *File*, *Repository*}, illustrated in Figure 7.

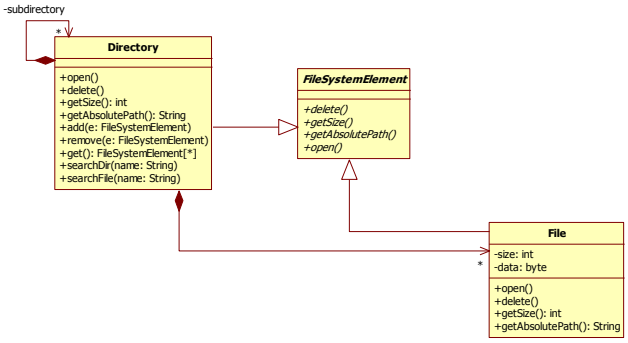

Figure 7: The identified fragment

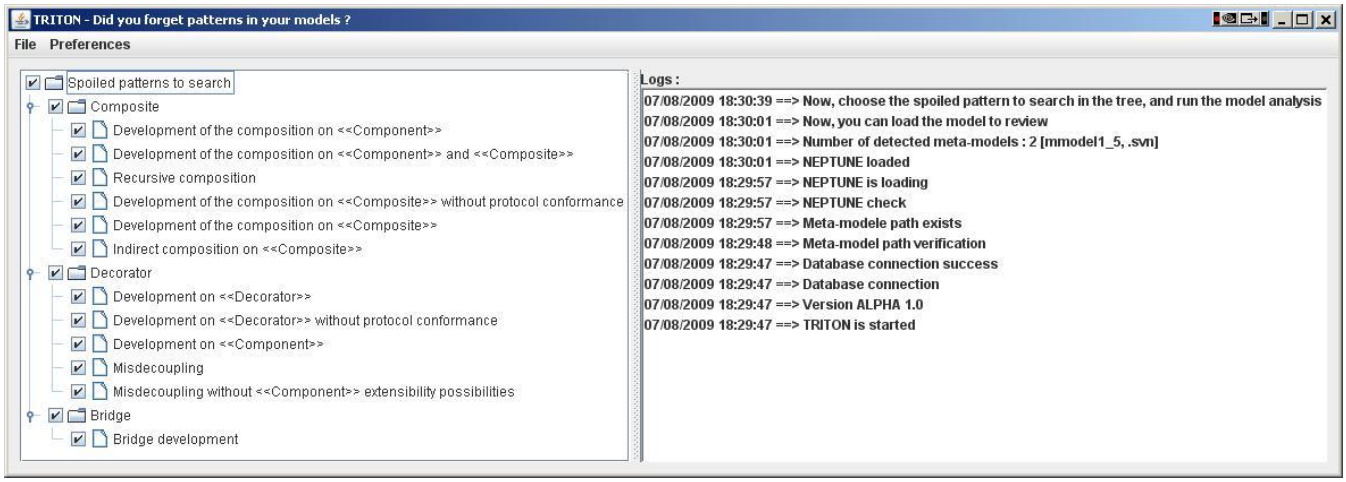

Figure 8: Triton – the tool allowing the activity execution

At the end of the detection step, the identified fragments are not regarded as alternative yet because we do not know their intent. The designer can check more in detail each identified fragment and thus pass to the following step: the checking of the intent and the presentation of the advantages of substitution. Figure 9 introduces the dialog box emitted by Triton to check the intent with the designer. To do so, we use an ontology defined in OWL [24], containing information relating to the intents of the design patterns, as well as the strong points degraded by the spoiled patterns [23].

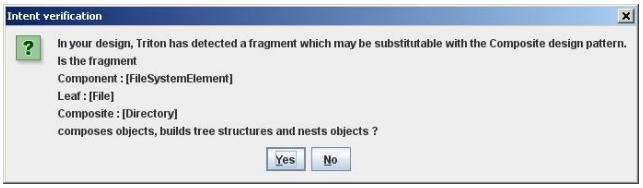

Figure 9: Intent verification

In our case, since it is a fragment corresponding to the spoiled pattern of the *Composite*, it is the intent of the *Composite* pattern which is introduced. If the designer validates the intent conformity, Triton presents the strong points of the pattern whose model will benefit after the injection of the pattern. In our example, we can say that our fragment composes hierarchically of the objects. Thus, since we accept the intent, Triton shows the dialog box illustrated by Figure 10.

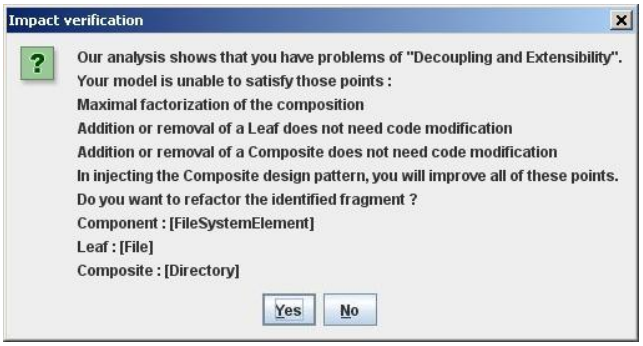

Figure 10: Advantages of the refactoring

By injecting the *Composite* design pattern, the designer gains in decoupling and in extensibility, which corresponds to the defects that we had identified during the previous analysis. For our example, we accept the transformation, and so Triton carries out the transformation of the model into memory. The model refactoring is done automatically: each class of the alternative fragments is marked, according to its responsibility. This marking facilitates the injection of the pattern [23].

After the transformation, the designer is invited to execute the detection again in order to check if other fragments appeared, or disappeared, if several fragments would have been identified during the first analysis. Finally, when the designer estimates that its model is in a sufficient quality, or if Triton does not identify any more fragment, a models serialization system allows the generation of a new XMI file containing the transformed model.

At the end of the review, the model presented in Figure 6 is transformed to integrate the *Composite* design pattern. The result is presented on the Figure 11.

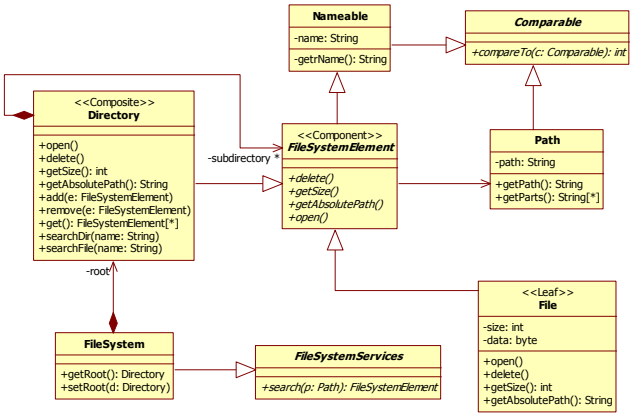

Figure 11: The improved model

We can see that the transformation implies the factorization of the composition by removing the developed compositions. The consequences of this transformation are found in the simplification of the management of the tree structure and by the fact that the two evolution scenarios do not require any more

modification of the existing code. At the end of the activity, we can thus consider that the model was improved.

4. A FIRST CATALOG

We can define two manners to constitute a spoiled patterns base. The first is to analyze the design patterns and to carry out transformations in order to denature them. Indeed, too artificial or too distant of designs by people, they would not be found in standard models. The second possibility is to collect a set of alternative solutions solving problems solvable by a design pattern, and to deduce a set of spoiled patterns from them. We choose the second possibility to obtain the most relevant spoiled patterns.

In doing so, we are sure that it is possible to apply a context on the spoiled pattern since they are deduced from models which had a context. Thanks to this method, the entire base contains spoiled patterns which have already been used once. As the constraints of this way are to obtain problem solutions without exploiting design patterns, a heuristic to optimize the collect consists in making experiments with designers not having the reflex to exploit existing know-how.

4.1 Experiments building

First, we have proposed a list of design problems solvable with design patterns. Second, we have instantiated design patterns on the problems. These best solutions had been presented to the students after their contributions, and it was a good start for a course dedicated to advanced object oriented programming and reusable micro-architectures. Third, we have analyzed the contributions and take into consideration valid solutions to a problem: the alternative solutions. Four, we have tried to deduce spoiled patterns from alternative solutions by an abstraction process. In the same time, spoiled patterns permit us to enforce software qualities in using design patterns by a fine comparison between different design solutions to a generic design problem.

4.1.1 The public

Generally, students in computer science discover initially the design techniques, and then the design patterns. At this precise time, students produce models solving problems, without using the design patterns. It is at this stage of their formation we asked them to solve design problems. Thus, they produced models according to their own experience and often with design defects. Moreover, these experiments made it possible to the students to put forward the interest to use the patterns, which constitutes a considerable teaching contribution. Indeed, during their formation on the patterns, we confronted them with their models, putting thus ahead the design defects corrected by the design patterns.

Distributed on three years, our experiments aimed at students in third and fifth year of studies in computer science. Each experiment appeared as a personal work composed of about ten exercises. Each exercise raised a design problem solvable by the use of a design pattern. We worked the statement of each problem so that the solutions correspond directly to the use of the design pattern. Thus, we limited the number of non-significant classes so that the students do not disperse in too complex designs.

4.1.2 The process

To do so, we took as a starting point the "motivation" section of the patterns of GoF or, when they were not appropriate to us, we worked out our own design problems. In a general way, this section presents a problem solvable by the design pattern, in classes, sequence or objects diagrams. The purpose of this example is to help to understand, on a concrete case, the pattern and what it brings.

Our first experiments concerned the structural patterns primarily. The results obtained were sufficient to deduce structural spoiled patterns. For the following experiments, we concentrated on the behavioral patterns. It is for these last experiments that we imposed, in the statement distributed to the students, the creation of sequence diagrams allowing the illustration of the communication between objects.

At last, over the three years, we covered the seven structural patterns, the eleven behavioral patterns and three of the creative patterns. Thus, we obtained thousand three hundred models which it was necessary to analyze in order to eliminate the erroneous designs and the doubled models.

4.2 A complete example

The next example of our experiment is a compilation of problems submitted and results obtained. Problems, optimal solutions (i.e. instantiation of the dedicated design pattern), alternative solutions and spoiled patterns are presented according to increasing difficulty. Progressively, the problems are more difficult to solve, alternative solutions more difficult to obtain and spoiled patterns more difficult to abstract.

Finding solutions to design problems

This document proposes a set of exercises concerning objects modeling. You must produce a UML class diagram **and a UML sequence or collaboration diagram illustrating each exercise**. Each diagram should contain sufficient information to demonstrate that the problem is solved (attributes, methods, relationships, stereotypes).

The purpose of these exercises is that you use your own knowledge. These designs can be envisaged in several ways. Do not look for shared solutions with your collegues, or solutions on the Internet or in design books.

Some problems are presented with probable evolutions. Your designs should be structured so that these changes are easily integrated. Make these changes occur in your diagrams.

Problem 1:

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.

Problem 2:

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, a horizontal scrollbar, an edge and a menu (these additions may be cumulated).

Problem 3:

Design a system enabling to display on a screen some empty windows (no button, no menu...).

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. It is probable that the system evolves in order to display specialized windows by "application windows" (able to manage applications) and "iconised windows" (with an icon).

Problem 4

Design a drawing editor.

A design is composed of graphics (lines, rectangles and roses), positionned at precise positions. Each graphic form must be modeled by a class that provides a method *draw(): void*. A rose is a complex graphic designed by a "black-box" class component. This component performs this drawing in memory, and provides access through a method *getRose(): int* that returns the address of the drawing. It is probable that the system evolves in order to draw circles.

Problem 5:

Design a DVD market place work.

The DVD market place provides DVD to its clients with three categories: children, normal et new. A DVD is new during some weeks, and after change category. The DVD price depends on the category. It is probable that the system evolves in order to take into account the horror category.

Problem 6:

Design a help manager of a Java application.

A help manager allows the show of a help message depending on the objects on which a client has clicked. For example, the "?", sometimes located near the contextual menu of a Windows dialog box, allows the show of the help of the button or the area where we click. If the button on which one clicks does not contain help, it is the area containing which displays its help, and so on. If no object contains help, with final, the manager displays "Not help available for this area". Instantiate your class diagram in a sequence diagram of on the example of a printing window. This window (JDialog) consists in an explanatory text (JLabel), and in a container (JPanel). This last contains a Print button (JButton) and a Cancel button (JButton). The Print button contains help "Launches the impression of the document". The Cancel button, the text as well as the window do not contain help. Lastly, the container contains help "Click on one of the buttons". In the sequence diagram, reveal the scénarii: "The user asks for the help of the Print button", "the user asks for the help of the Cancel button", and "the user asks for the help of the text".

Problem 7:

Design the communications of one plane to the approach of an airport.

When a plane is in approach of the airport, it must announce to all the other planes which are around that it intends to be posed, and await their confirmation with all before carrying out the operation. It is the control tower of the airport which guarantees the regulation of the air traffic, by making sure that there is no trajectory conflict or destination between several planes. Besides the class diagram, represent by a collaboration (diagram of collaboration or diagram of objects and sequence) the landing of a plane among two wanting to land and one wanting to take off.

Problem 8:

Design a tutorial to learn how to program a calculator.

This calculator executes the four basic arithmetic operations. The goal of this tutorial is to make it possible to take a set of operations to be executed sequentially. The tutorial presents a button by arithmetic operation, and two input fields for the operands. After each click on a button of an operation, the user has then the choice to start again or execute the suite of operations to obtain the result. It is probable that this teachware evolves in order to make it possible to the user to remove the last operation of the list and to take into account the operation of modulo.

4.2.1 Best solutions

We present here the best solutions that are given to the students after their experiments. As mentioned before, these solutions provide a good start to a design pattern formation. Students can compare their solutions with best solutions. Then they can realize the qualities of a design by the use of best practices.

The first four problems address structural patterns, the last four behavioral patterns. The proportion between problems type is respected. We have trying some problems addressing creational patterns unsuccessfully. We consider that creational patterns can be used after the use of others patterns in the development process.

Problem 1 refers to the *Composite* pattern and its best solution is described in Figure 12. This problem is directly inspired by the GoF. Here, the problem is concentrated about compositions between objects and there is no need to precise methods.

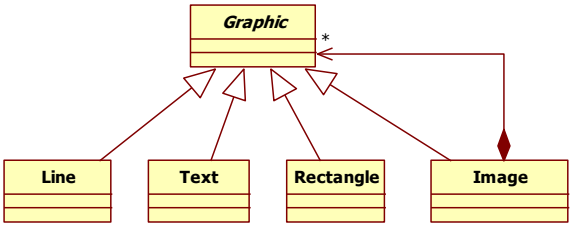

Figure 12: The best solution of the problem 1

Problem 2 refers to the *Decorator* pattern and its best solution is described in Figure 13. This problem is also inspired by the GoF. Here, we precise methods in classes and we add notes to show the collaboration between concrete and abstract decorators. The fact that this pattern uses an explicit call to the *super* method is difficult to see in a UML collaboration diagram.

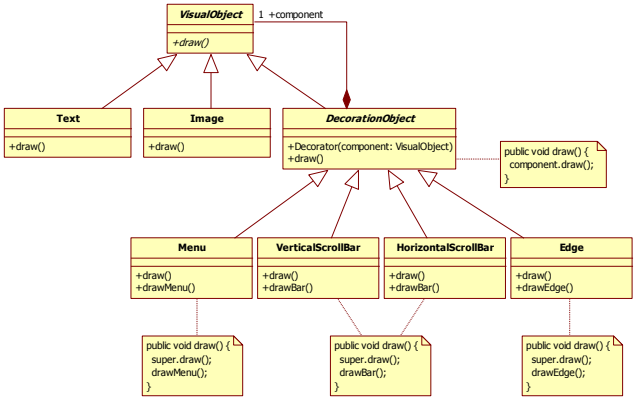

Figure 13: The best solution of the problem 2

Problem 3 refers to the *Decorator* pattern and its best solution is described in Figure 14. This problem is also inspired by the GoF. Here, the simple delegation between abstractions and implementors are modeled using UML notes. A collaboration diagram can be used in this case.

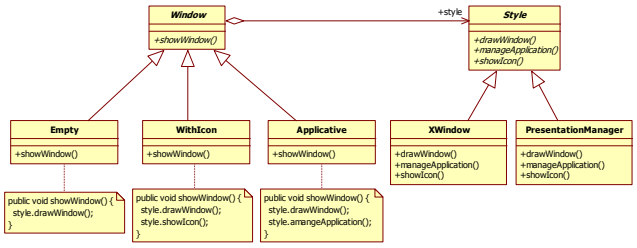

Figure 14: The best solution of the problem 3

Problem 4 refers to the *Adapter* pattern and its best solution is described in Figure 15. We have chosen to use uniquely the object instantiation because in their formation, students program in Java only. As the previous problem, the simple delegation is modeled using UML notes, but a collaboration diagram can be used too.

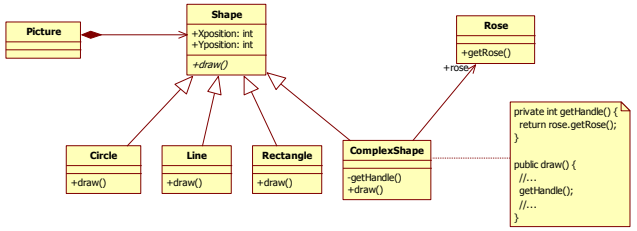

Figure 15: The best solution of the problem 4

Problem 5 refers to the *State* pattern and its best solution is described in Figure 16. This problem is inspired by the motivation example in the Martin Fowler refactoring book [16]. Although this pattern is labeled as behavioral, it is not necessary to have a collaboration diagram.

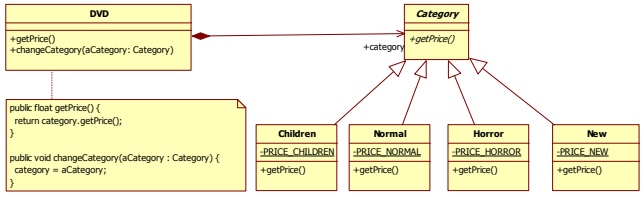

Figure 16: The best solution of the problem 5

Problem 6 refers to the *Chain of Responsibility* pattern and its best solution is described in Figures 17 and 18. This problem is inspired by the GoF. Here, we ask students to give us a collaboration diagram. We consider that the structure is not sufficient to show the chain, and we need the sequence diagram to determine if an alternative solution is valid.

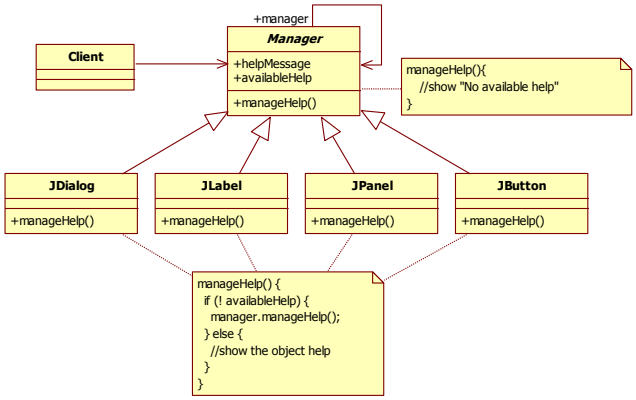

Figure 17: The best solution of the problem 6

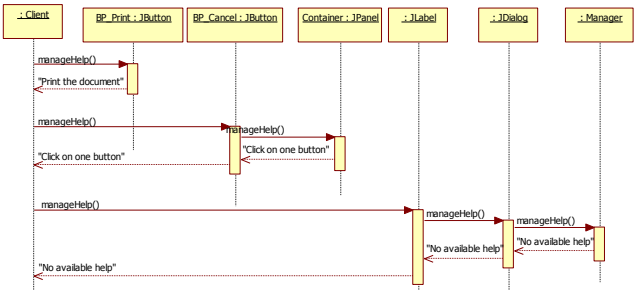

Figure 18: The sequence diagram of the best solution of the problem 6

Problem 7 refers to the *Mediator* pattern and its best solution is described in Figures 19 and 20. This problem is issued from [25]. Even if this statement is attractive to present a metaphor of the pattern, students have resolved the problem by the instantiation of a mediator; or they have resolved the problem in accordance with the statement and messages exchanges are in a complete graph form. Here, alternative solutions become the design problem to resolve by the use of the pattern.

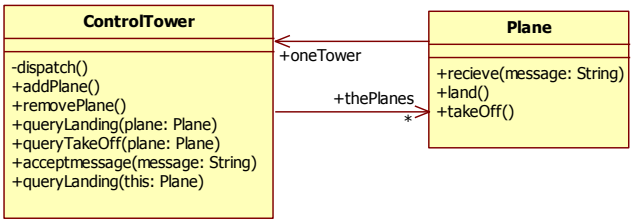

Figure 19: The best solution of the problem 7

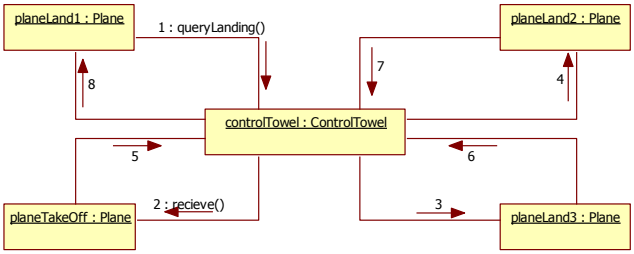

Figure 20: The collaboration diagram of the best solution of the problem 7

Problem 8 refers to the *Command* pattern and its best solution is described in Figure 21. This problem is inspired from an exercise to manage pointers function in the C language. For this, a collaboration diagram is not necessary. The important fact is to detect the presence of switch statement into the code. However, we have made the choice to think at design level, and we did not find a simple way to model the kinematics of a program in the UML notation. Then we have inferred switch statements from UML designs.

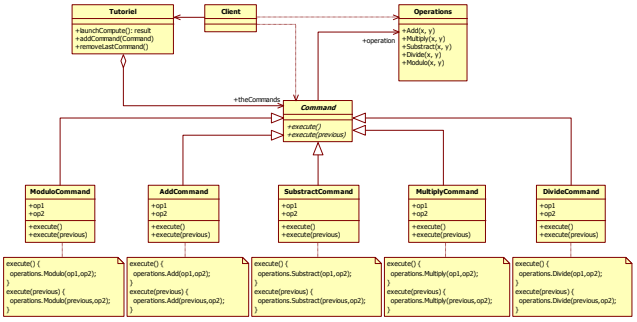

Figure 21: The best solution of the problem 8

4.2.2 The problem of the design of a problem

We have specifically designed problems for the collect of alternative solutions. Then, the statement of a problem should not be too open or too directed. Consider the two following statements for the *Mediator* problem.

"*When a plane is in approach of the airport, it must announce to all the other planes which are around that it intends to be posed, and await their confirmation with all before carrying out the operation. It is the control tower of the airport which guarantees the regulation of the air traffic, by making sure that there is no trajectory conflict or destination between several planes.*"

"*When a plane is in approach of the airport, it must announce to the control tower that it intends to be posed, and await the confirmation before carrying out the operation. It is the control tower of the airport which guarantees the regulation of the air traffic, by making sure that there is no trajectory conflict or destination between several planes.*"

The first statement is too open and does not conform to the pattern. In fact, if we design a system which scrupulously respects the problem, it is very difficult to instantiate the *Mediator*. For the second statement, it is very difficult to not instantiate the mediator, and then the problem is not significant.

The problem of the design of a problem happened to other problem statements. It is not easy to propose a small problem dedicated to a specific design problem solvable by a unique pattern and then solvable by a minimal architecture. There are several solutions: consider problems coarser and apply composite patterns, search topics of problems from the experience of designers, ensure that problems are not too didactic, ensure that problems are easily solvable by the instantiation of a pattern and more complicated to solve without, ensure that problems address other patterns…

4.2.3 Results

From all the solutions suggested by the students, we present here one for each problem. Others alternative solutions exist but are not presented due to space considerations. When needed, we have refined the static diagrams with attributes and methods necessary to the solutions understanding. For each alternative solution presented, we propose the corresponding spoiled pattern that we have abstracted from some alternative solution. We have named spoiled patterns in the same manner as bad smells. Their names evoke the noted misconception. For now, we have uniquely represented spoiled patterns by static diagrams. We study the possibility of adding collaboration diagrams.

An alternative solution to the use of *Composite* is presented in Figure 22. This solution is valid, even if this structure imposes duplications of code for the *Graphic* class. All compositions are memorized and managed in this class and this fact invalidates the strong point "*decoupling and extensibility*".

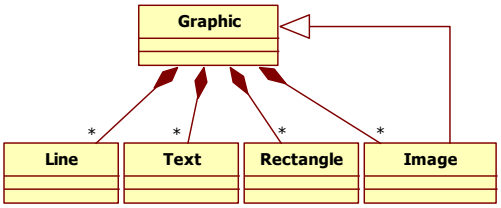

Figure 22: One alternative solution for the problem 1

In the Figure 23, we present the deduced spoiled pattern named: *Development of composition on component*. Here, composition links should be factorized.

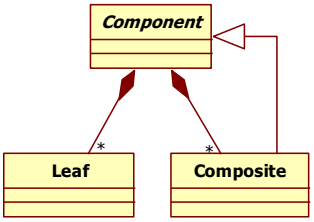

Figure 23: The spoiled pattern *Development of composition on component*

An alternative solution to the use of *Decorator* is presented in Figure 24. This solution is valid, even if the decorations are directly expressed with composition links on the class object that plays the *Component* role. This fact requires a big programming effort to permit the decoration on the fly, because late binding and calls to the *super* method are not used. In this case, the adding of a new concrete decorator needs some code modification, and there is a decoupling problem between objects to decorate and decorators.

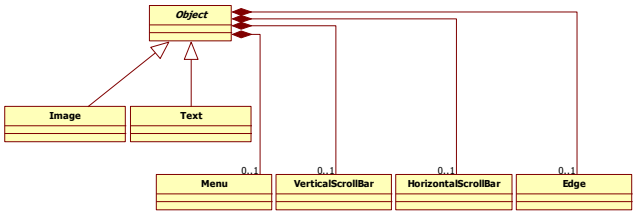

Figure 24: One alternative solution for the problem 2

In the Figure 25, we present the deduced spoiled pattern named: *Development of decorations on component*. Here, decoration links should be factorized and a class dedicated to the delegation between concrete decorators should be added.

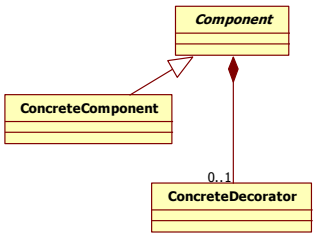

Figure 25: The spoiled pattern *Development of decorations on component*

An alternative solution to the use of *Bridge* is presented in Figure 26. Even if windows are correctly separated from the environment, the associations between each window and *Style* are not factorized. There will be no problem if a new platform is added, but for a new window, a new association link will be added to the *Style* class. This model is valid. However, it is possible to have some window types with different styles.

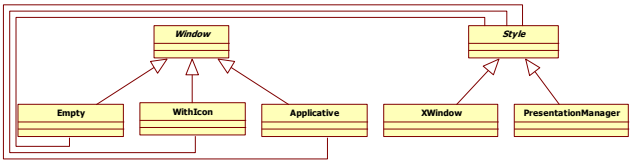

Figure 26: One alternative solution for the problem 3

In the Figure 27, we present the deduced spoiled pattern named: *Development of delegation links*. Here, delegation links are misplaced and should be factorized.

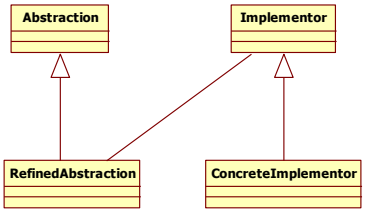

Figure 27: The spoiled pattern *Development of delegation links*

We do not present an alternative solution for the problem 4 (*Adapter* pattern), because all the solutions we have obtained were instantiations of the design pattern

An alternative solution to the use of *State* is presented in the Figure 31. For the problem 5, we obtain two worst cases. In the first worst case, the category is a subclass of *DVD* imposing instances destruction to change of category. The question of the validity of this solution is open in regard of the proposed exercise. However, we have considered this solution valid in using a prototype creational pattern with a category as parameter.

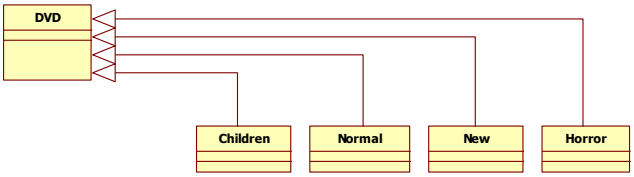

Figure 28: One alternative solution for the problem 5

In the Figure 29, we present the deduced spoiled pattern named: *Bad classification*. It needs a *State* class that allows the category changing without destroying and recreating a new instance globally identical.

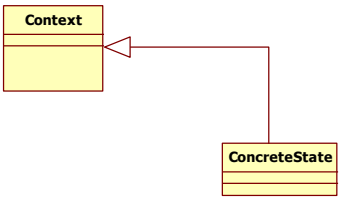

Figure 29: The spoiled pattern *Bad classification*

Another alternative solution to the use of *State* is presented in the Figure 33. Here, the *DVD* class manages its state thanks to an enumeration. In doing so, the solution imposes a "switch" statement, and so, the category changing is possible. The problem of this solution concerns the extensibility. Indeed, if a new category is added, the *DVD* class must be modified to manage the new type.

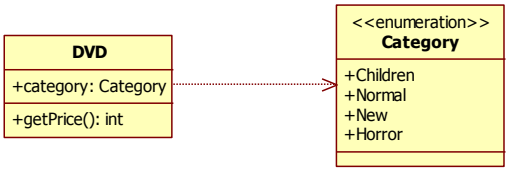

Figure 30: One alternative solution for the problem 5

In the Figure 31, we present the deduced spoiled pattern named: *Hidden switch statement*. This is an ideal start point of a big refactoring dedicated to introduce the *State* pattern. It is given in the chapter example of the refactoring book by Martin Fowler [16].

Figure 31: The spoiled pattern *Hidden switch statement*

An alternative solution to the use of *Chain of Responsibility* is presented in the Figure 32. Here, there is a separation between containers and contents. Two issues arise. The first concerns the validity of the solution and the second concerns the interaction with another spoiled pattern presents in the design. We have considered this solution valid even if delegation between content objects is not possible. The problem can be solved by adding a reflexive association on the class *Content*.

But the main problem is the composition relationship between *Container* and *Content*. We have inferred that this composition expresses another thing about containers and contents, and there is a reuse of this link for chaining the management of help messages. Then, we can say that this composition link have too many responsibilities as the same manner that we say on a class. But, what should we consider about this solution? Is this solution is an alternative solution of the *Chain of Responsibility* using a preexisting composition link or a side effect of a preexisting alternative solution to the composite between graphical components? It is typical for this kind of problem we want to hear the opinion of the community working on patterns.

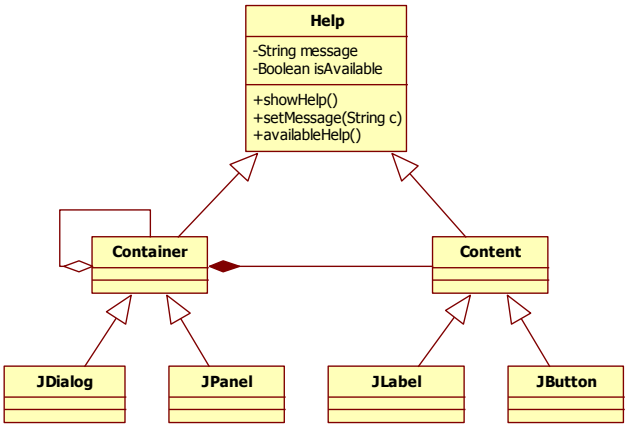

Figure 32: One alternative solution for the problem 6

In the Figure 33, we present the deduced spoiled pattern named: *Excessive reuse of a preexisting association*. The reflexive association on *Container* class must be pulled up to the super class.

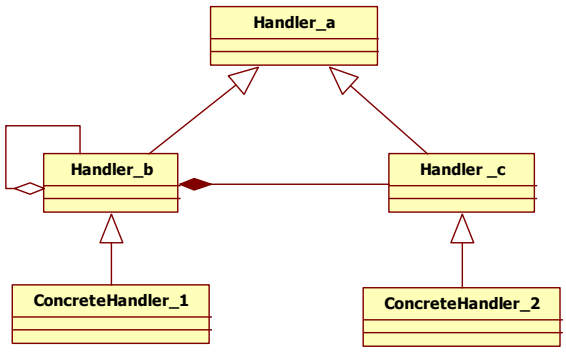

Figure 33: The spoiled pattern *Excessive reuse of a preexisting association*

The alternative solution of the Figure 32 respects the messages chaining, as illustrated in the sequence diagram in the Figure 34. When a help demand is activated, the object concerned has the possibility either of answering or to communicate it to another object. However, we do not show that different associations are used in this collaboration. So even if a particular scenario unfolds a chain of responsibility for dealing with error messages, the static architecture between objects can be different. It seems likely that the study of such a behavioral pattern requires firstly a static diagram and the other a complete set of test cases modeled by sequence diagrams.

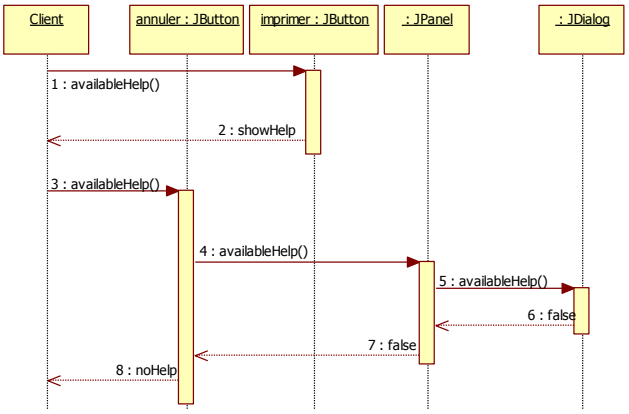

Figure 34: The sequence diagram of the previous alternative solution

We present an alternative solution to the use of *Mediator* in Figure 35. Unfortunately, all the alternatives we have obtained corresponded to the worst case ever presented in the GoF catalog. The concrete mediator that is represented by the control tower is not used. As mentioned before, it is due to the difficulty to propose an adequate exercise.

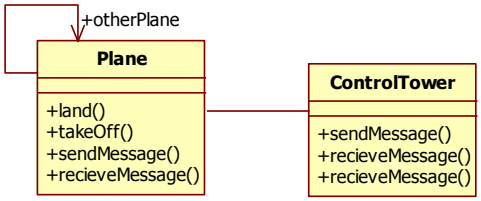

Figure 35: One alternative solution for the problem 7

In the Figure 36, we present the deduced spoiled pattern named: *Complete collaboration between concrete colleagues*. Here the refactoring consists to move the association-end from planes to the *Control Tower*.

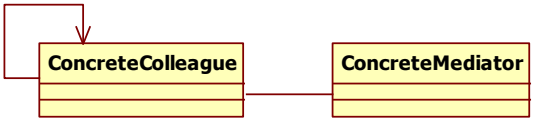

Figure 36: The spoiled pattern *Complete collaboration between concrete colleagues*

In the next collaboration diagram of the Figure 37, we show the complete graph structure dedicated to exchange of messages. We have chosen a collaboration diagram to express this fact.

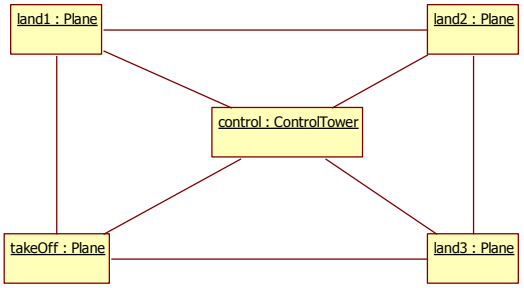

Figure 37: The collaboration diagram for the previous alternative solution

An alternative solution to the use of *Command* is presented in the Figure 38. This solution grants all the management to the *System* class but separates the real operation in different classes. So, the solution is valid, but imposes a lot of communications between the *System* class and the operations classes. Moreover, *System* does not memorize the operation but an identifier from *OperationType*. So, the *System* class must test all the identifiers during the *computeOperation* that is problematic if there are a lot of operations.

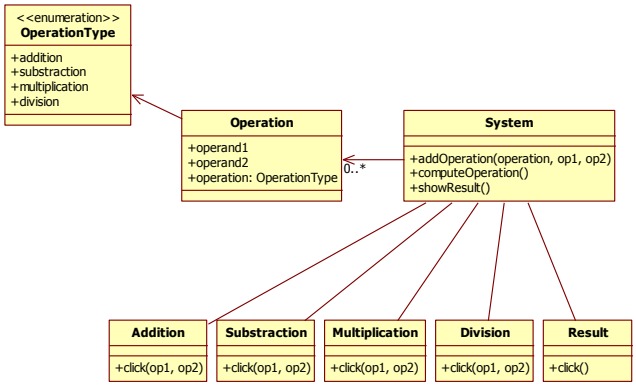

Figure 38: One alternative solution of the problem 8

In the Figure 39, we present the deduced spoiled pattern named: *Partially reification of command*. As one of the spoiled pattern concerning the state, we can name it the hidden switch statement because it needs the use of dynamic binding for the selection of the appropriate operation. Here the refactoring consists in terminating the reification process by transforming each association link between the invoker and a concrete command by an inheritance link with the command.

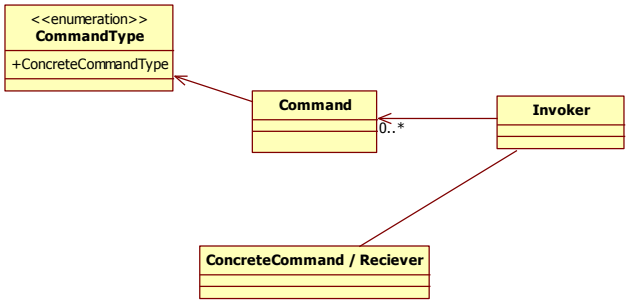

Figure 39: The spoiled pattern *Partially reification of command*

4.3 Limits

Our collection method of spoiled patterns presents, in its current form, two limits. The first relates to the collection with experiments, the second to the manual analysis of the alternative solutions.

- To collect alternative solutions with designers corresponds to an approach which enables us to exploit a large number of solutions. However, the participation of students of the same course produces very close results when the problems become more and more complex. Having the same formal and technical training, the same design defects are found in their models, thus limiting the number of different alternative solutions.
- To constitute our spoiled pattern base, we manually analyzed each solution suggested. Such an analysis remains manual, because it seems difficult to automate the examination of a model starting from a simple class diagram. For the structural patterns, the effort is not very consequent since only the structure of the solution is significant, contrary to behavioral patterns which bring into play the kinematics of the messages exchanges between the objects.

In order to avoid the multiplication of the same solutions and to increase the diversity of the alternatives solutions suggested, it is advisable to urge an experiment on broader scale by touching designers of any horizon. The use of a collaborative sharing website of problems and alternative solutions would make it possible to identify the most frequent spoiled patterns, and more largely, the bad practices of designs. Moreover, this website would allow the emergence of an experts community opening a sharing zone of "bad practices".

5. A WEBSITE TO SHARE SPOILED PATTERNS

We have designed a collaborative website giving an access to the whole catalog of the spoiled patterns. This site, reachable at http://www.irit.fr/GOPROD/, introduces each design pattern with a list of its spoiled patterns and a problem list solvable with the pattern. Each spoiled pattern is described with a justification of its damage. The site proposes a contribution system allowing the submission of new problems, new alternative solutions, new spoiled patterns and new strong points. Each submission is subjected at a committee examining its validity and its interest as a spoiled pattern. Thus, this site makes it possible to create a community of experts to urge the use of the design patterns.

The website manages three roles. The first concerns simply the *visitor*. A visitor can show the entire catalog already validated and so can use the website as an information source to do its design, to correct its design, or to teach design patterns concepts. After identification, a visitor becomes a *contributor*. With this role, the contributor can submit new problems, new alternative solutions or new strong points. Each submission is sent to the *committee* of the website. This committee is an expert group able to validate or invalidate each submission. While the submission is not validated, the visitors cannot see it. Thanks to this system, we present to the visitors a catalog always validated.

The submission process of the website is conceived in the same logic as the conduction of the experiments presented in the section 4. All the user stories of the website are presented is the Table 2.

Finally, the entire website is articulated around a specific business model presented in the Figure 40. We can see in this model all the concepts previously presented. A pattern has some strong points decomposed in subfeatures, and solves a problem. So, this problem has a best solution which instantiates the concerned design pattern, and some alternative solutions. These alternative solutions solve the problem but with a different architecture compared to the best solution. A spoiled pattern may be deduced from each alternative solution, and the difference between the alternative and the best solution produces a strong points degradation. Except for the strong points and the sub-features, each entity has one or more representations allowing the

illustration of each concept, like static diagrams or sequence diagrams as shown in the previous sections of this paper.

Table 2: The user stories

As visitor, I can	
	show design patterns description (structure, strong points, intent and applicability).
	show spoiled patterns for one design pattern.
	show alternative solutions allowing the deduction of a spoiled pattern.
	show problems allowing to obtain alternative solutions.
	show the best solution for a problem.
	register as contributor.
As contributor, I can	
	submit a new problem and its best solution.
	submit a new alternative solution for a problem.
	submit a new spoiled pattern for an alternative solution.
	submit a new strong point or a new subfeature.
As committee member, I can	
	validate or invalidate the submissions, in motivating my choice.
	show all the submissions.
	submit a new design pattern.

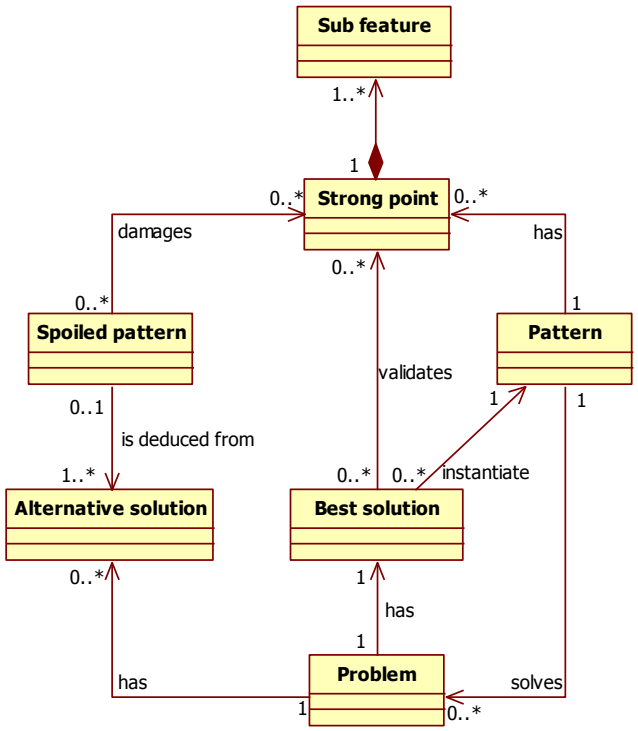

Figure 40: The business model

6. CONCLUSION

A spoiled pattern is a generic micro-architecture that produces non-optimal solutions to a design problem. Therefore, by comparisons with best solutions instantiated with design patterns, spoiled patterns allow to enforce the good properties of design patterns. We think that spoiled patterns can be used for others purposes too. First, a didactic purpose: spoiled patterns can be considered as bad smells at design time or as "small" anti-patterns. Then early detection of them can be useful during a weekly meeting of the development team covering architecture. Second, a dissemination purpose as we propose in the collaborative website. Spoiled patterns can consolidate the proof of the pertinence of the pattern concept. Therefore, having an extensional definition of design problems covered by the patterns can help designers to detect more easily misconceptions on their designs.

However experiments that we have driven are expensive in time and are concentrated on a specific panel. Therefore we have played too much roles: teacher, analyst, specialist, and committee member. Then, we propose a collaborative web site to open spoiled patterns to the community.

We encounter some difficulties in the process of abstraction concerning spoiled behavioral patterns. Here, structure is not sufficient and interactions diagrams represent specific collaborations between objects.

REFERENCES

1. A.L. Baroni, Y.-G. Guéhéneuc, H. Albin-Amiot, "**Design patterns formalization**", rapport de recherche, Département d'informatique, École des Mines de Nantes, *number* 03/03/INFO, 2003.
2. E. Gamma, R. Helm, R. Johnson, J. Vlissides, "**Design Patterns: Elements of Reusable Object-Oriented Software**", Addison Wesley Professional, 1995.
3. H. Albin-Amiot, P. Cointe, Y.-G. Guéhéneuc, N. Jussien, "**Instantiating and Detecting Design Patterns: Putting Bits and Pieces Together**", *in: proceedings of the 16th conference on Automated Software Engineering (ASE)*, IEEE Computer Society Press, pages 166-173, 2001.
4. H. Albin-Amiot, Y.-G. Guéhéneuc, "**Meta-Modeling Design Patterns: Application to Pattern Detection and Code Synthesis**", *in: proceedings of the 1st ECOOP workshop on Automating Object-Oriented Software Development Methods*, University of Twente, 2001.
5. J. Dietrich, C. Elgar, "**A Formal Description of Design Patterns Using OWL**", *in: proceedings of the 16th Australian Software Engineering Conference*, IEEE Computer Society, pages 243-250, 2005.
6. J. Dong, Y. Zhao, "**Classification of Design Pattern Traits**", *in: proceedings of the 19th International Conference on Software Engineering & Knowledge Engineering (SEKE)*, pages 473-477, 2007.
7. A.L. Guennec, G. Sunyé, J.-M. Jézéquel, "**Precise Modeling of Design Patterns**", *in: proceedings of 3rd International Conference on the Unified Modeling Language (UML)*, Springer Verlag, pages 482-496, 2000.
8. H. Kampffmeyer, S. Zschaler, "**Finding the Pattern You Need: The Design Pattern Intent Ontology.**", *in: proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, Springer, volume 4735, pages 211-225, 2007.

9. J.K.H. Mak, C.S.T. Choy, D.P.K. Lun, "**Precise Modeling of Design Patterns in UML**", *in: proceedings of the 26th International Conference on Software Engineering (ICSE)*, IEEE Computer Society, pages 252-261, 2004.
10. A.H. Eden, A. Yehudai, J. Gil, "**Precise specification and automatic application of design patterns**", *in: proceedings of the 12th international conference on Automated Software Engineering (ASE)*, IEEE Computer Society, pages 143-152, 1997.
11. G. El-Boussaidi, H. Mili, "**Detecting Patterns of Poor Design Solutions Using Constraint Propagation**", *in: proceedings of the 11th international conference on Model Driven Engineering Languages and Systems (MoDELS)*, Springer-Verlag, volume 5301, pages 189-203, 2008.
12. R. France, S. Ghosh, E. Song, D.-K. Kim, "**A Metamodeling Approach to Pattern-Based Model Refactoring**", *in: IEEE Software*, IEEE Computer Society Press, volume 20, *number* 5, pages 52-58, 2003.
13. H. Mili, G. El-Boussaidi, "**Representing and Applying Design Patterns: What Is the Problem?**", *in: proceedings of the 8th international conference on Model Driven Engineering Languages and Systems (MoDELS)*, pages 186-200, 2005.
14. M. O'Cinnéide, P. Nixon, "**A Methodology for the Automated Introduction of Design Patterns**", *in: proceedings of the 15th IEEE International Conference on Software Maintenance (ICSM)*, IEEE Computer Society, pages 463-473, 1999.
15. C. Bouhours, H. Leblanc, C. Percebois, "**Bad smells in design and design patterns**", *in: Journal of Object Technology*, ETH Swiss Federal Institute of Technology, volume 8, *number* 3, pages 43-63, 2009.
16. M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts, "**Refactoring: Improving the Design of Existing Code**", Addison-Wesley Professional, 1999.
17. M. Dodani, "**Patterns of Anti-Patterns ?**", *in: Journal of Object Technology*, volume 5, *number* 5, pages 29-33, 2006.
18. W.J. Brown, R. C. Malveau, T.J. Mowbray, "**AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis**", Wiley, 1998.
19. M. Fagan, "**Design and code inspections to reduce errors in program development**", *in: Software pioneers: contributions to software engineering*, Springer-Verlag New York, pages 575-607, 2002.
20. Object Management Group., "XML Metadata Interchange", http://www.omg.org/technology/xml/index.htm, 2007.
21. T. Millan, L. Sabatier, T. T. Le Thi, P. Bazex, C. Percebois, "**An OCL extension for checking and transforming UML Models**", in : *proceedings of the 8th International Conference on Software Engineering, Parallel and Distributed Systems (SEPADS)*, WSEAS Press, pages 144-150, 2009.
22. Object Management Group., "Object Constraint Language", http://www.omg.org/cgi-bin/apps/doc?formal/06-05-01.pdf, 2006.
23. C. Bouhours, "**Detection, Explications et Restructuration de défauts de conception : les patrons abîmés**", PhD, IRIT, 2010.
24. D.L. McGuinness, F.V. Harmelen, "**OWL Web Ontology Language Overview**", http://www.w3c.org/TR/owl-features/, 2004.
25. M. Duell, J. Goodsen, L. Rising, "**Non-software examples of software design patterns**", *in:* OOPSLA '97: Addendum to the 1997 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (Addendum), ACM, pages 120-124, 1997.