Spoiled patterns : how to extend the GoF

Cédric Bouhours, Hervé Leblanc, Christian Percebois

Spoiled patterns: how to extend the GoF

Cédric Bouhours · Hervé Leblanc · Christian Percebois

Abstract Design patterns were popularized by the GoF catalog in 1995. This catalog contains 23 design patterns which concern 23 design problems. Each pattern is detailed with its structure, its intent, and some information including applicability conditions and some possible variations which enable it to be re-used. In 1995, they were the cutting edge thought processes. We consider that design patterns have two major features. First, they are the result of emergent conceptions validated by a community through a well-defined acceptance process. They are a field of expert knowledge. Secondly, they must be as abstract as needed to be able to maximize their reusability. They constitute a compilation of best practices concerning object codes and designs. We propose to extend the GoF with a new concept that we have named "spoiled patterns". They are midway between bad smells in design necessary to go ahead with a refactoring and a necessary learned lesson in order to explain anti-patterns. Each design pattern corresponds to some spoiled patterns. In this paper, we present how we have compiled the first spoiled patterns catalog, by carrying out several experiments with a lot of young inexperienced designers.

Keywords Design patterns · Spoiled patterns · Best practices · Bad smells · Anti-patterns · Pattern teaching

C. Bouhours (✉)
LIMOS, Université d'Auvergne, Clermont-Ferrand, France
e-mail: cedric.bouhours@udamail.fr

H. Leblanc · C. Percebois
IRIT, Université de Toulouse, Toulouse, France
e-mail: leblanc@irit.fr

C. Percebois
e-mail: percebois@irit.fr

1 Introduction

A design pattern represents expert knowledge, validated by the community, and reusable for a type of design problem. For example, the *Composite* design pattern represents a canonical solution for structural and compound problems, such as composing objects, building tree structures, and nesting objects (Kampffmeyer and Zschaler 2007). The GoF catalog (Gamma et al. 1995) presents 23 design patterns which concern 23 types of problems. Each one is detailed with its structure, its intent, and some information which allow its use under the best conditions.

However, two issues arise when designers want to use design patterns. First, designers have to identify the type of problem they encounter, before choosing a suitable design pattern. To facilitate the designer's choice, some works exist for improving the design pattern classification and selection (Kampffmeyer and Zschaler 2007; Albin-Albin-Amiot and Guéhéneuc 2001; Albin-Amiot et al. 2001; Baroni et al. 2003; Dietrich and Elgar 2005; Dong and Zhao 2007; Guennec et al. 2000; Mak et al. 2004). Secondly, designers have to ensure correct integration of the chosen pattern. A design pattern is not simply a design template and requires adaptation in order to be well integrated into an existing context. Some works exist to check correct integration of a pattern (Eden et al. 1997; El-Boussaidi and Mili 2008; France et al. 2003; Mili and El-Boussaidi 2005; O'Cinnéide and Nixon 1999).

For a design problem, several solutions exist. First, there are those using an adequate design pattern correctly, and so recognized as canonical solutions. Secondly, there are the others. For a given problem, an alternative solution is a valid solution, but with a different architecture compared with the canonical solution. These differences may cause degradations, compared with a best practice solution, when solving a design problem adequately. So, an alternative solution reveals a misaligned architecture, which is similar to a misaligned piece perturbs the good work of an entire engine.

Recurrent alternative solutions to a same design problem can be abstracted to a spoiled pattern. We chose the term "spoiled" (Bouhours et al. 2009) to describe this new pattern family, because it introduces structural differences when compared to the structure of the design patterns. A majority of participants of the original pattern are present but not properly connected.

Actually, an alternative solution could come from an improper design pattern instantiation. However, for formalization and reproducibility of the design flaw reasons, we argue that alternative solutions only come from a correct instantiation of a spoiled pattern.

We propose to extend the presentation template of each design pattern with a new section named "spoiled patterns". For each design pattern, there exist a number of corresponding spoiled patterns. Like design patterns, spoiled patterns require adaptation to a current context. Then, the extension of the GoF consists in giving several alternative solutions to the motivation problem, deduced structures and collaborations, and a comparison with the intrinsic qualities of the design pattern. Even if the GoF is the compressed compilation of best object practices, we consider that the given knowledge is complex, reserved for experts, and thus difficult to transmit. To start from alternative solutions, we can go to canonical solutions to explicate objects, object-oriented design, and design patterns themselves. Moreover, students impacted in our experiments have learned the benefits of using patterns by comparing their solutions with the canonical solutions proposed by the GoF.

To obtain solutions to problems without the exploitation of design patterns know-how, we have chosen young inexperienced designers. They produced models according to their

own experience and often with design defects. Distributed over 3 years, our experiments were aimed at students doing a B.Sc. or M.Sc. in Computer Science. From these experiments, we deduced a set of spoiled patterns.

In this paper, we begin with a simple example of spoiled pattern in Sect. 2. Section 3 presents how it is possible to use the base of spoiled patterns we have constituted. Section 4 presents and illustrates the context and the coverage of the experiments we organized to collect spoiled patterns. Next, Sect. 5 aggregates the results of the experiments in order to constitute a first catalog of bad practices revealing the design pattern benefits. Section 6 is devoted to a discussion about the constraints and the limits of our experiments. Lastly, Sect. 7 compares spoiled patterns with structural variations, bad smells, and anti-patterns.

2 Illustration on a simple example

Let us consider a specific problem wording, inspired by the GoF: "Design a system for drawing graphic images: a graphic image is composed of lines, rectangles, texts, and images. An image may be composed of other images, lines, rectangles, and texts". This problem matches the intent of the *Composite* design pattern: "to compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly" (Gamma et al. 1995). To instantiate the *Composite* pattern on this problem, we must identify the objects having the same responsibilities as each participant of the pattern, as seen in Fig. 1a. Figure 1b introduces an alternative solution. 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. Moreover, the *Graphic* class is used to support the factorization of the protocols and to be the unique access point for the client. However, the fact that the classes *Line, Rectangle, Text*, and *Image* itself are attached to *Image* involves code modifications if a new *Leaf* or a new *Composite* is added.

Recurrent alternative solutions to a design problem can be abstracted to a spoiled pattern, in the same manner as a design pattern is the abstraction of canonical solutions. The abstraction process of an alternative solution requires to identify the pattern participants, then carry out a reduction that make it possible to preserve only one class per participant of the pattern. A spoiled pattern is connected to one and only one design pattern. Like alternative solutions, it presents structural differences compared with the design pattern. Figure 2 presents the *Composite* design pattern (a) and the spoiled pattern (b) deduced from the solutions presented in Fig. 1.

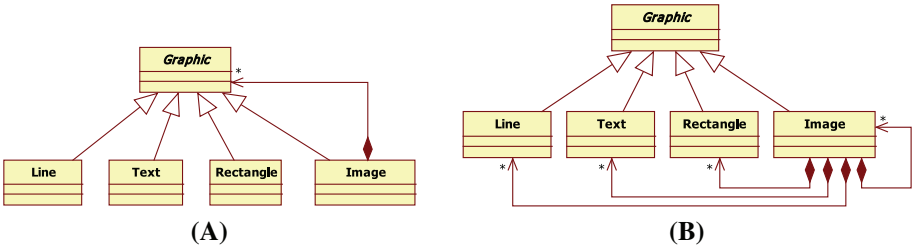


Fig. 1 A canonical solution (**a**) and an alternative solution (**b**)

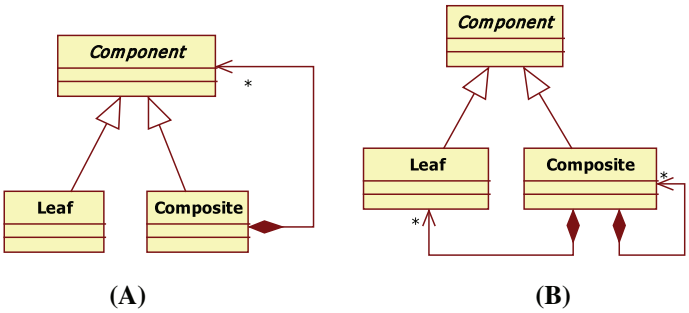


Fig. 2 The *Composite* design pattern (**a**) and one of its spoiled patterns (**b**)

Table 1 The *Composite*'s strong points evaluated on spoiled pattern of Fig. 2

Decoupling and extensibility	
×	Maximal factorization of the composition relationship
×	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

These differences may cause degradations when solving a design problem. Indeed, we make the hypothesis that the structure of a design pattern brings a set of design qualities which emphasize why the design pattern is a canonical solution. Therefore, a spoiled pattern does not bring the same design qualities, which may frequently introduce design defects.

We introduce the strong points of a design pattern which express the expected factors of software quality brought about by its use. In order to quantify the degree of damage from the use of a spoiled pattern, the evaluation of the strong points is realized thanks to sub-features. A sub-feature is a best practice or a quality indicator of the pattern. These criteria are partially deduced from the *consequence* section of the GoF catalog and from the design defects identified in some alternative solutions obtained during our experiments. Table 1 summarizes the degradation of the strong points due to the use of 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 ✔.

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 is expressed with a reflexive connection and with a development on all the leaves, a design defect appears. The factorization is not maximal, and the coupling between *Leaf* and *Composite* imposes code modifications.

Indeed, we postulate that the structure of a design pattern brings a set of design qualities which emphasize why the design pattern is a canonical solution. Therefore, a spoiled pattern does not bring the same design qualities and this may frequently introduce design defects. As it is often useful to correct these local design defects, we suggest to detect (Bouhours et al. 2010) and correct alternative solutions with a tooled design review activity (Bouhours et al. 2009, 2011). The aim is to inspect models to search for fragments which are characteristic of the use of spoiled patterns and to substitute solutions using design patterns for them. We now look for other uses.

3 Two uses of spoiled patterns

The first use is to extend our previous approach, dedicated to *Up-Front design*, toward *Evolutionary design* promoted by *Agile Methods*. In this methodological context, patterns would be injected only when necessary: the code exists and contains some bad smells whose refactoring motivations advocate the use of a pattern. We think our catalog can help developers identify some targets addressed by design patterns during their refactoring processes (Kerievsky 2005). The second use concerns teaching considerations: spoiled patterns are related to best practices thanks to strong points. To start from alternative solutions, we can go to canonical solutions to explicate objects, object-oriented design, and design patterns themselves.

3.1 Refactoring to patterns

3.1.1 Up-Front design

Lot of works that focus on the proper use and good integration of patterns in models make the hypothesis that the designer wants to insert the good design pattern in his model. By contrast, we let the designer model as he wishes, and then, we propose an automatic model inspection devoted to continuous improvement. As there are code review activities (Fagan 2002), we propose, in a previous work (Bouhours et al. 2009), a design review activity that automatically verifies that if there are no known assumed bad design practices in a model. This activity is subdivided into three steps:

1. detection of assumed bad fragments on a model (Bouhours et al. 2010),
2. communication with the designer to check the intent of the detected fragments (Harb et al. 2009), and
3. model repair to integrate the design patterns properly (Bouhours et al. 2009).

So, the concept of spoiled pattern is central to our approach. It represents an abstraction of some classic misconceptions which present structural differences compared with good design practices denoted by the corresponding design pattern. Thanks to a precise structural description, we are able to identify fragments which conform to a spoiled pattern. This comparison is only based on structural features of models, and therefore, the intent of the detected fragment must be validated by the designer himself. These differences may cause degradations in the case to solve known design problems, and we can use the strong points of the design pattern to explain how the designer's solution degrades his software architecture.

According to the GoF's concluding remarks, design patterns capture many of the structures that result from refactoring (Gamma et al. 1995): "One of the problems in developing reusable software is that it often has to be reorganized or refactored. Design patterns help you determine how to reorganize a design, and they can reduce the amount of refactoring you need to do later...Using these patterns early in the life of a design prevents later refactorings". Clearly, *Up-Front* design is promoted by the GoF, and the main aim of our design review activity agrees with the previous quote.

3.1.2 Evolutionary design

However, there is a tension between *Up-Front design* promoted by patterns, and *Evolutionary design* promoted by *Agile Methods*. Agile methodologies advocate the use of a technique when it is needed, or else it can become an anti-pattern: in this case, intensive use of patterns tends to over-engineering.

Refactoring to patterns encourages agile software developers to inject design patterns during their refactoring of code processes (Kerievsky 2005). However, revealing such design evolutions into code during a refactoring requires to discover the problem that the pattern addresses and then matches the correct pattern implementation to the current context. Let us consider our *Composite* pattern red wire example. There are three refactorings to pattern dedicated to this (Kerievsky 2005).

1. *Replace implicit tree with composite* implicit tree is hard coded with primitive types such as string for example. This fact denotes a bad smell named *Primitive Obsession*. The main motivations for the refactoring are communication and simplicity that match with the intent of the *Composite* pattern. The intent of the pattern is present in the code only. So the pattern must be injected from scratch.
2. *Replace one/many distinctions with composite* each business query is encapsulated by a specific client class which manually composes some existing basic queries. The composite queries are hard coded in each composite class. This fact denotes a bad smell named *Shotgun Surgery*. Changes in basic queries must be propagated to many client classes. Then, the composite is proposed, and to our surprise, we would have preferred the *Command* pattern. The *Command* pattern has a composite part structure, but with an intent more in conformity with the problem. Here as well, the pattern must be injected from scratch.
3. *Extract composite* we have clearly two classes having a composite role with the same intent and the same code, as illustrated in Fig. 3. In this variation, composite classes are duplicated which result in duplication of the composite link. We do not agree with the title of this refactoring. For us, it is just an extract class from the two composite classes rather than an Extract Composite, which shows that the granularity looked at is not correct.

The first two situations are detectable only at code level, and the last is clearly a spoiled *Composite* pattern. The misconception appears as trivial as when you see the UML diagram in Fig. 3 and certainly difficult to smell when you stay at code level. It is very surprising to note that bad smells in re-engineered code are not necessarily spoiled patterns. It is very difficult for developers to take a global view of their code.

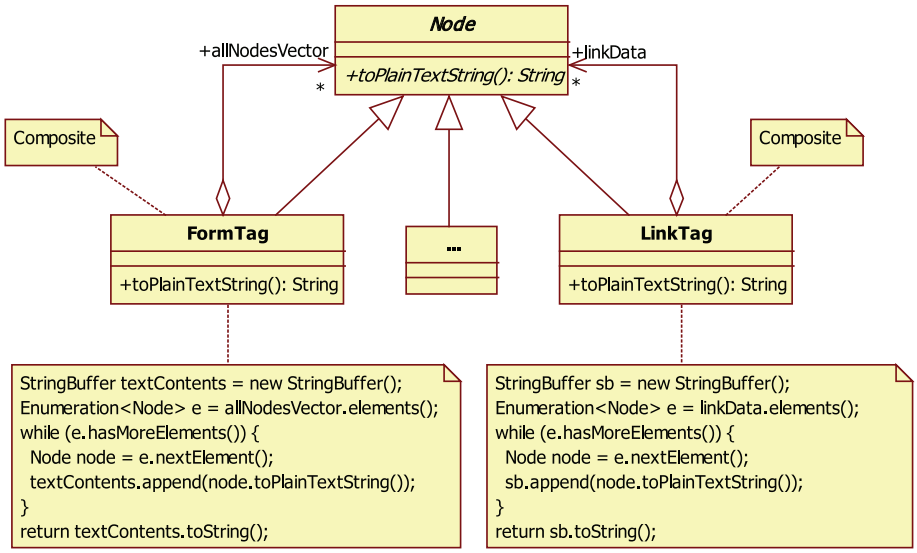


Fig. 3 Re-engineered model for *Extract Composite* refactoring of Kerievsky (2005)

We showed, in these cases, that each refactoring to pattern starts with a description of a situation which can be sometimes expressed as a spoiled pattern. We think our catalog can be extended with some of bad smells pointed out in this work. By contrast, our catalog should help agile developers indicate poor design found in a code and to act accordingly.

3.2 To teach patterns and object-oriented design

The first goal of our catalog is to incite developers to systematically reuse best design practices with respect to design patterns. As spoiled patterns were discovered by experiments with students, a second goal emerged. We reused the experiments, the catalog and the tooled activity to teach patterns. We propose here a brief literature review to teach objects with patterns, teach object-oriented design, and finally teach design patterns themselves.

3.2.1 To teach objects by patterns

There are educational propositions that consider design patterns first versus an objects first approach to teach objects (Pecinovský et al. 2006). Therefore, patterns highlight inter-objects design *via* responsibilities of the participants of a dedicated collaboration which is fundamental in object-oriented design (Dewan 2005). The two approaches presented consider "patterns in the small" i.e., a problem to solve that uses one and only one design pattern. Our catalog is organized in the same manner. Moreover, it enforces the good design properties of a pattern that exhibit their strong points related to alternative solutions.

3.2.2 To teach object-oriented design

After a first course about object-oriented programming, there are educational propositions that consider best practices on object-oriented design. These best practices are given in the

form of informal indicators (Gibbon and Higgins 1996) or in the form of responsibilities patterns (Larman 2002). These informal indicators (or design heuristics Gibbon and Higgins 1996) are not predictive measures about the quality of a design. While design heuristics are guided by structural considerations, the GRASP patterns (Larman 2002) are guided by the interactions of objects and the metaphor to send messages. They are certainly complementary to our approach since they can be used as an explanation of the design defects of alternative solutions.

3.2.3 To teach design patterns

There are two ways to teach patterns. The first deals with "patterns in the small" (Sendall 2002; Jiménez-Díaz et al. 2008), and the second with "patterns in the large" (Siddle 2011). The second approach uses a consequent problem which is solvable by the application of several patterns, each pattern application that solve one part of the overall problem when possible. In this context, interactive stories serve to exhibit different design choices which produce some alternative designs using some different pattern compositions. The aim of this approach is to learn pattern driven design and evaluation design choices related to requirements defined by stories. In the first approach (Jiménez-Díaz et al. 2008), new requirements are used to force the learners to achieve a better solution by means of a pattern driven refactoring. The authors suggest the use of one design pattern to solve one problem, and a course contains some pertinent examples where a specific design pattern can be applied. Then, teaching design patterns can be greatly enhanced if good examples are used. But, what is a good example? Sendall (2002) argues that a good example matches three conditions: where (scope of the problem), when (an efficient solution of the problem), and why (one pattern constitutes the best solution).

A spoiled pattern references one and only one pattern and enforces its strong points. Then, we can easily use it to teach "design patterns in the small". Strong points are related to object-oriented design best practices, and the problems proposed for our experiments respect the three clauses: when, where, and why, as described in the next section.

4 Description of the experiments

We can define two ways to constitute a base of spoiled patterns. The first is to randomly make changes in the structure of the design patterns to distort them. If this solution allows the listing of an exhaustive list of spoiled patterns, many of them would be useless, too artificial or too exotic from classical solutions.

The second possibility is to collect a set of alternative solutions and to deduce a set of spoiled patterns from them. In doing so, we ensure that the base contains spoiled design models which have already been used once. The constraints in this way are how to obtain problem solutions without exploiting design patterns. So, an optimization of the collection consists in questioning designers who do not have the habit of exploiting existing know-how.

In this section, after presenting the website where we put all the results we obtained, we present the experiments we did to fill our base of spoiled patterns. First, we describe the context and the coverage of our experiments. Secondly, we present a subset of the design problems we use in our experiments.

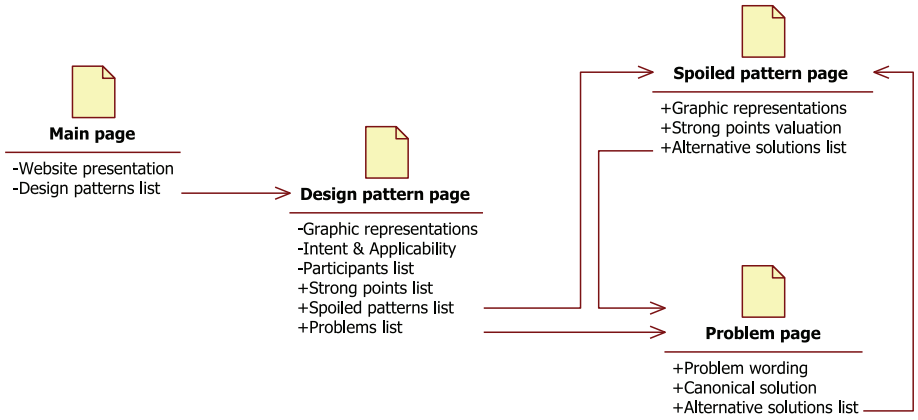


Fig. 4 The navigation map of the website

4.1 Our extension of the GoF: a collaborative online catalog

Even if a lot of websites already exist that present GoF design patterns, we have designed a website containing all GoF patterns extended with our concepts. Classically, this website[1] starts with a design pattern list. Each of them is described, as in the GoF, thanks to its intent, its applicability, its graphic representation, and its participants. We have extended this description with the strong points table, the spoiled patterns list, and the list of problems which are solvable with it. This extra-description allows visitors to choose how they want to use the catalog.

For each design pattern, two methods of consultation are available. The first consists in the consultation of each concrete problem related to the design pattern. This method allows the visitor to understand whether the pattern could be a canonical solution. The second method consists in consulting each spoiled pattern attached to the design pattern. This method allows the visitor to identify some design defects when using the design pattern in a incorrect way. These two ways bring the visitor to dedicated problem pages and spoiled pattern pages. In addition, these pages contain an alternative solutions list. For the problem page, this list contains all alternative solutions for the problem concerned. For the spoiled patterns page, the list contains the alternative solutions instantiated on all problems. Thus, alternative solutions represent the combination when navigating between problems and solutions. The navigation map of the website is illustrated in Fig. 4.

The website manages two user roles. The first concerns simply the visitor. A visitor can display the entire validated catalog and so can use the website as an information source to make his design, to correct his design, or to teach design pattern concepts. In using Sendall (2002) clauses, we can say that the problems list refers to the *where* clause, the spoiled patterns list refers to the *when* clause and the strong point evaluations refers to the *why* clause.

After identification, a visitor becomes a contributor. A contributor can submit new problems, new alternative solutions, or new strong points. For a design pattern, the contributor submits a new problem worded with its canonical solution. For a problem, the contributor submits a new alternative solution. For a design pattern, the contributor submits

[1] Reachable at http://www.goprod.bouhours.net/.

a new strong point. Each submission is sent to the website committee. The committee is an expert group able to validate or invalidate each submission.

Thanks to this system, we present visitors with a permanently correctly updated catalog. Up to now, the website is supplied by the results of our experiments. Moreover, this website would allow the emergence of a community of experts to open a shared zone of best design practices with "smell" variations.

4.2 Context and coverage

To solve a problem without using design patterns, the designer must not instinctively exploit patterns know-how. Young inexperienced designers are good candidates for this task. Generally, students in Computer Science discover object design techniques before design patterns. We question them at this point in their studies.

Distributed over 3 years, our experiments were aimed at students doing a B.Sc. or a M.Sc. degree in Computer Science. Each experiment appeared as ten exercises to be done as homework. Each exercise presented a design problem solvable by the use of a specific design pattern. Thus, we limited the number of non-significant classes so that the students did not stray into designs that were too complex.

We took as a start point the *motivation* section of the GoF patterns or, when they were not appropriate, we worked out our own design problems. In a general way, this *motivation* section presents an example of a problem solvable by the design pattern, which in turn uses classes, sequence, or objects diagrams. The purpose of this example is to help developers understand, with a concrete case, the pattern, and what it brings.

Our experiment procedure has four steps. First, we have put forward a list of design problems solvable with design patterns (see Sect. 4.3). Secondly, we have instantiated design patterns on the problems to obtain canonical solutions (see Sect. 5.1). Thirdly, we have analyzed the students' contributions and taken into consideration the alternative solutions only (see Sect. 5.2). Finally, we have tried to deduce spoiled patterns from alternative solutions, we have elaborated strong points tables, and we have evaluated them on each spoiled pattern (see Sect. 5.2).

Our first experiments primarily concerned structural patterns, as presented in Table 2. The results obtained were sufficient to deduce some structural spoiled patterns presented in the next section. For the last experiments, presented in Tables 3 and 4, we concentrated on behavioral patterns. For these experiments, we imposed the creation of sequence diagrams allowing the illustration of the communication between objects. Actually, over the 3 years, we covered the seven structural patterns, the eleven behavioral patterns and three of the creational patterns. Thus, we obtained 1,357 models from 136 students which it was necessary to analyze in order to eliminate erroneous designs and doubled models. In the 1,357 propositions, we found 647 wrong models, 325 canonical solutions, and 385 alternative solutions.

A first global analysis concludes that 48 % of obtained solutions were wrong, 24 % were canonical, and 28 % were alternative. This distribution is not globally significant because it concerns different design patterns and students with different educational levels. For example, the global results of structural experiments are affected by *Composite* and *Decorator* patterns whose intents are simple and whose structures allow variations according to the problem to solve. On the contrary, problems concerning the *Adapter, Facade,* and *Proxy* patterns give us no alternative solutions, either because the proposed solutions directly used the design pattern or were wrong. It seems that students have

Table 2 Results of our first year experiments

Concerned patterns: Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy, Abstract factory, Builder						
Nos.	Level	Students	Propositions obtained	Wrong propositions	Canonical solutions	Alternative solutions
1	M.Sc.	16	144	61 (42 %)	17 (12 %)	66 (46 %)
2	B.Sc.	30	370	176 (47 %)	24 (7 %)	170 (46 %)
3	M.Sc.	1	9	4 (44 %)	4 (44 %)	1 (11 %)
4	B.Sc.	9	81	18 (22 %)	25 (31 %)	38 (47 %)

Table 3 Results of our second year experiments

Concerned patterns: Mediator, Observer, Singleton, Facade, Interpreter, Iterator, Memento, State, Strategy, Command						
Nos.	Level	Students	Propositions obtained	Wrong propositions	Canonical solutions	Alternative solutions
5	B.Sc.	28	280	143 (51 %)	100 (36 %)	37 (13 %)
6	M.Sc.	5	50	18 (36 %)	22 (44 %)	10 (20 %)

Table 4 Results of our third year experiments

Concerned patterns: Template method, Chain of responsibility, Command, Visitor, Mediator, Bridge, Observer, Facade, Singleton						
Nos.	Level	Students	Propositions obtained	Wrong propositions	Canonical solutions	Alternative solutions
7	B.Sc.	29	261	162 (62 %)	71 (27 %)	28 (11 %)
8	B.Sc.	18	162	65 (40 %)	62 (38 %)	35 (22 %)

unwittingly use design patterns due to their obvious structure. For the *Flyweight* pattern, we did not get any valid solution, probably because of the complexity of the concerned problem type and of the complexity of the solution. Moreover, this pattern does not really solve a design problem but optimizes the size of a set of objects in memory.

Each row in these tables represents an experiment on a particular group of students with the same educational level. We notice that, on average, one-third of the solutions is canonical. Of course, these canonical solutions are the wrong propositions. These ones present some mistakes which invalidate the intent of the problem. In lines 1 and 3, we notice that masters students produced a significant number of wrong models despite their greater experience compared with the B.Sc. students. This fact results from the teaching content they had in previous years. This teaching was changed the year of the experiment. The new teaching is more efficient considering the better results of masters students in line 6. For the B.Sc. students of line 5, the number of wrong models is due to the wording of the experiment. It was the first experiment with behavioral patterns, and we forgot to ask for sequence diagrams. We modified our experiment for the M.Sc.

students thus increasing the number of interesting results. Lastly, in line 7, more than one half of the propositions are wrong. Most of these students did not seriously try to solve the problems.

These experiments allow the students to highlight the interest in using patterns. This constitutes a teaching contribution. Indeed, during their instruction on patterns, we used their models to enhance the design defects corrected by the design patterns.

4.3 A compilation of design problems

The next task is a best of seven problems submitted in our experiments, according to increasing difficulty. The canonical solutions, some of the alternative solutions obtained, and the corresponding spoiled patterns we have abstracted are presented in the next section.

Finding solutions to design problems

This document proposes a set of exercises concerning object 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 colleagues, 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. Update your diagrams consequently

Problem 1

Design a system enabling you 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 you to display visual objects on a screen

A visual object can be composed of one or more texts or images. If needed, the system must allow the addition of a vertical scroll bar, a horizontal scroll bar, an edge and a menu to this object. These additions may be accumulated.

Problem 3

Design a system enabling you to display on a screen some simple 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" (ability to manage applications) and "iconized windows" (with an icon)

Problem 4

Design a customer's bill management at a video store

The video store provides DVDs to its clients with three categories: children, normal, and new. A DVD is new for some weeks, and after changes category. The DVD price depends on the category. It is probable that the system evolves in order to take into account a new category concerning horror movies

Problem 5

Design a help manager for a Java application

Finding solutions to design problems

A help manager allows the display 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 display of the help related to the button or the area where to click. If the button on which one clicks does not contain help, it is the area container which displays its help, and so on. If any object contains help, the help manager displays "No help available for this area". Instantiate your class diagram in a sequence diagram of 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 scenarios: "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 6

Design the communications of a plane approaching an airport

When a plane approaches an airport, it must announce to all the other planes which are around that it intends to land, and await their 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 or destination conflict between several planes. In addition to the class diagram, you must also submit a collaboration (in the form of a diagram of collaboration or a diagram of objects and sequence) that describes the landing of a plane amidst in a context of two demands to land and one wanting to take off

Problem 7

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 for each 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 sequence of operations to obtain the result. It is probable that this tutorial evolves in order to make it possible for the user to remove the last operation of the list and to take into account the operation of modulo

The canonical solutions of each previous problem, respectively, are instantiations of *Composite, Decorator, Bridge, State, Chain of responsibility, Mediator,* and *Command.*

5 Selected extracts of the catalog

In this section, we present some results of the seven problems presented in the previous section. We first introduce our canonical solution for each problem. Next, we present one alternative solution to each problem. We have selected the best alternative solutions for this paper, but as we made clear before, we obtained one or more alternative solutions for each problem. For each alternative solution, the deduced spoiled pattern has a name characterizing its defects and a evaluation of the corresponding design pattern's strong points.

5.1 Canonical solutions

We present here the canonical solutions that are given to the students after their experiments. As mentioned before, these solutions provide a good start to design pattern teaching. Students can compare their solutions with canonical solutions. Then, they can realize the qualities of a design by the use of best practices. The first three problems address structural patterns and the last four behavioral patterns.

5.1.1 Problem 1: the composite instantiation

A canonical solution is illustrated in Fig. 5. This problem is directly inspired by the GoF. Here, the problem is concentrated on compositions between objects.

5.1.2 Problem 2: the decorator instantiation

A canonical solution is illustrated in Fig. 6. This problem is also inspired by the GoF. Here, 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.

5.1.3 Problem 3: the bridge instantiation

A canonical solution is illustrated in Fig. 7. This problem is also inspired by the GoF. Here, the simple delegation between abstractions and implementors is modeled using UML notes. A collaboration diagram can be used in this case.

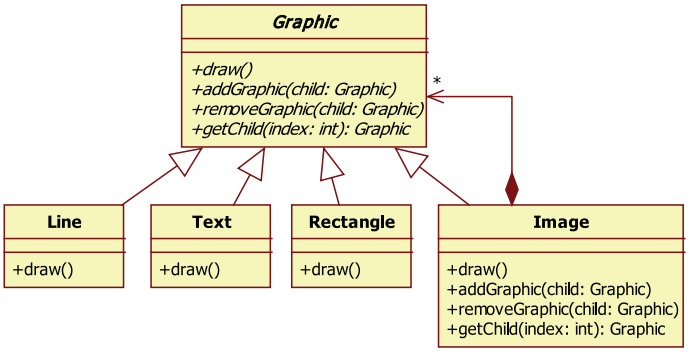


Fig. 5 A canonical solution of the problem 1

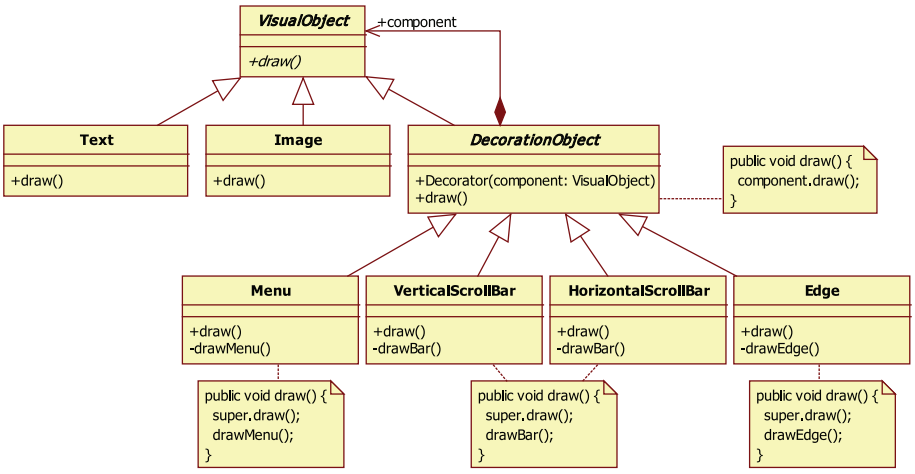


Fig. 6 A canonical solution of the problem 2

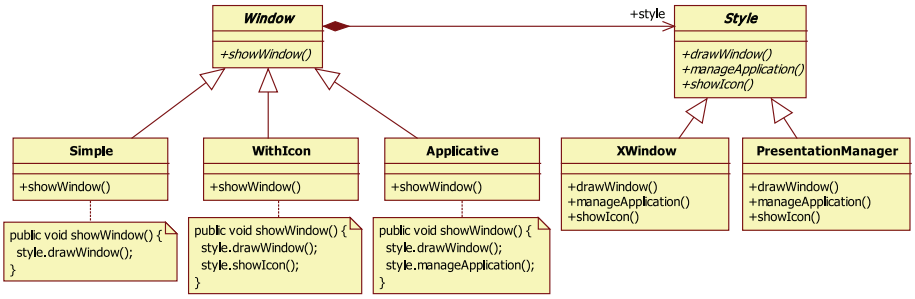


Fig. 7 A canonical solution of the problem 3

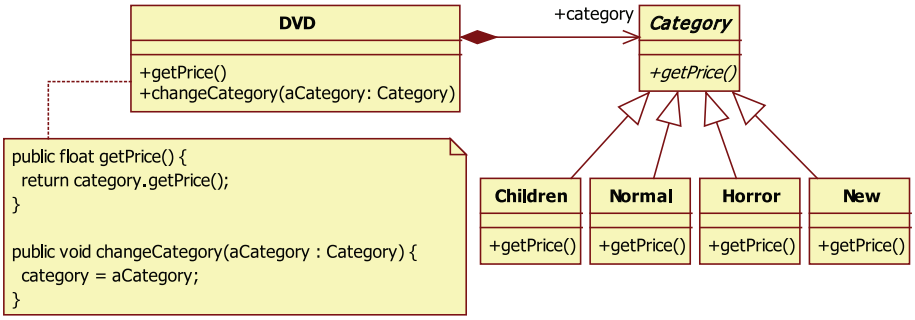


Fig. 8 A canonical solution of the problem 4

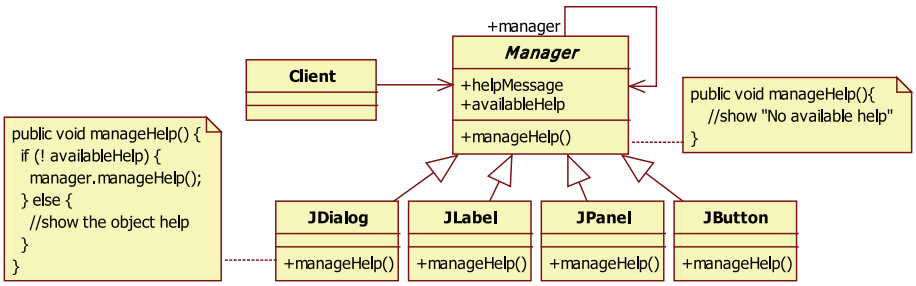


Fig. 9 A canonical solution of the problem 5

5.1.4 Problem 4: the state instantiation

A canonical solution is illustrated in Fig. 8. This problem is inspired by the motivation example of Martin Fowler's refactoring book (Fowler et al. 1999). Although this pattern is labeled as behavioral, it is not necessary to have a collaboration diagram.

5.1.5 Problem 5: the chain of responsibility instantiation

A canonical solution is illustrated in Figs. 9 and 10. 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 of message delegation, and we need the sequence diagram to determine whether an alternative solution is valid.

5.1.6 Problem 6: the mediator instantiation

A canonical solution is illustrated in Fig. 11. This problem comes from Duell et al. (1997). The *Mediator* defines an object that controls how a set of objects interact. Low coupling between colleague objects is obtained by communication with the mediator. The *Control Tower* is a good candidate for playing the role of a concrete mediator, as illustrated in Fig. 12. Pilots of the planes constitute the set of colleagues. It can be noticed that the instantiation is incomplete because we have no need for an abstract mediator.

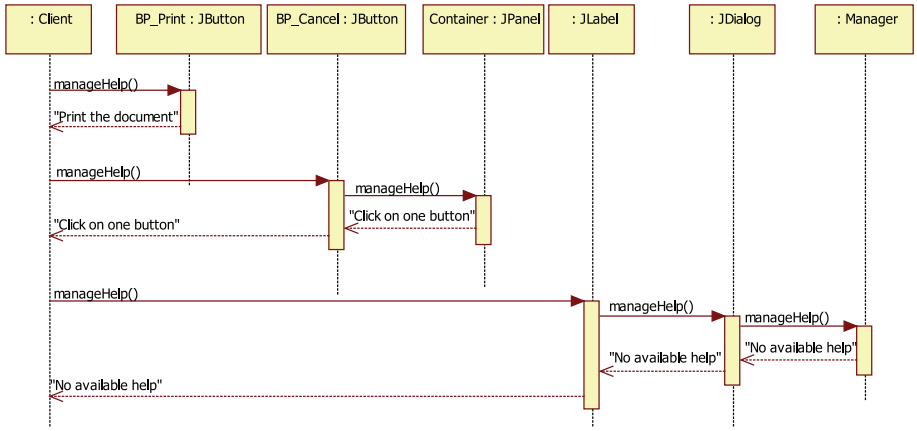


Fig. 10 The sequence diagram of the canonical solution of the problem 5

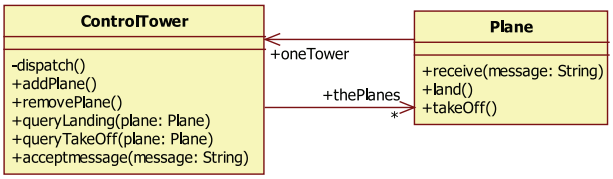


Fig. 11 A canonical solution of the problem 6

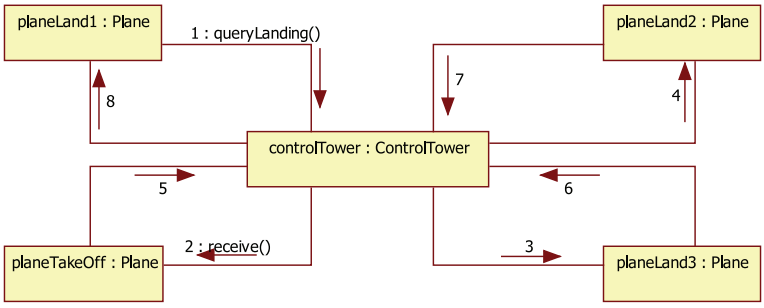


Fig. 12 An illustration of the communication between planes and control tower

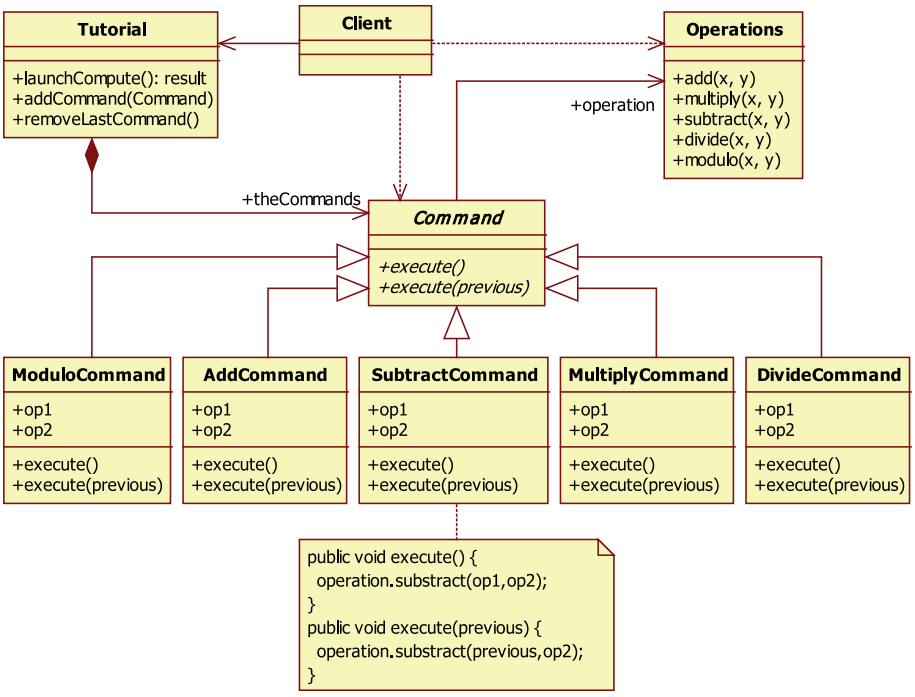


Fig. 13 A canonical solution of the problem 7

5.1.7 Problem 7: the command instantiation

A canonical solution is illustrated in Fig. 13. This problem is inspired by an exercise to manage pointers function in the C language. For this, a collaboration diagram is not necessary. In this case, the *Command* decouples the object that invokes the operation (an instance of the *Tutorial* class) from the one that knows how to perform it (an instance of the *Operations* class). Each concrete command executes a specific arithmetical operation.

5.2 Results

From all alternative solutions suggested by the students, we present one for each problem. Many alternative solutions exist, but are not presented due to space considerations.[2] When needed, we have refined the static diagrams with attributes and methods in order to understand the solutions. For each solution presented, we give the spoiled pattern that corresponds that we have abstracted. We have named spoiled patterns in the same manner as bad smells whose name evokes the noted misconception. For now, we only represent spoiled patterns with static diagrams. We are studying the possibility to add collaboration diagrams.

Starting from several alternative solutions to the same problem, we obtained a set of spoiled patterns, by the application of an abstraction process. This process requires an identification of the pattern participants from the classes of the solution and carries out a graph reduction that preserves one class per participant and relationships between these classes.

[2] For others, see our website http://www.goprod.bouhours.net.

For example, in Fig. 14, the *Graphic* class plays the role of *Component*, the *Image* class plays the role of *Composite*, and the other classes play the role of *Leaf*. Moreover, the connections between participants in Fig. 15 conform to the connections between the classes identified in Fig. 14.

However, some alternative solutions do not use all the participants. This fact implies that some classes have the responsibilities of several participants or that the responsibility of a participant is hard coded in several classes. In these cases, we chose to keep the same abstraction process.

5.2.1 Problem 1: the composite instantiation

An alternative solution to the use of the *Composite* is presented in Fig. 14.

This solution is valid, even if this structure imposes duplications of code for the *Graphic* class. All compositions links are memorized and managed in this class, and this invalidates the strong point "decoupling and extensibility". Moreover, the absence of inheritance link between leaves and component prevents any uniform protocol on operation of composed objects. Table 5 sums up the strong points evaluation. In Fig. 15, we present the deduced spoiled pattern named *Development of composition on component*. Here, composition links should be factorized.

5.2.2 Problem 2: the decorator instantiation

An alternative solution to the use of the *Decorator* is presented in Fig. 16.

This solution is valid, even if the decorations are directly expressed with composition links from the *Object* class which plays the *Component* role. This fact requires a big coding effort to allow the decoration on the fly, because late bindings and calls to the *super* method are not used. In this case, adding a new concrete decorator needs some code

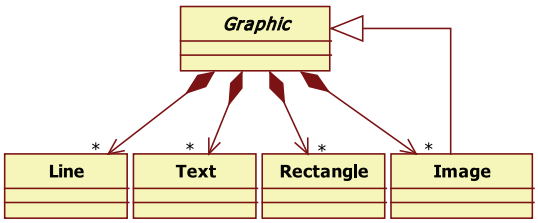


Fig. 14 One alternative solution to the problem 1

Table 5 The strong points evaluation table of the spoiled pattern *Ddevelopment of composition on component*

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

modification, and the objects to decorate are aware of all the decorators. Table 6 presents the strong points of the *Decorator* pattern and sums up the evaluation.

As for the previous example, we notice that this solution contains the same kind of misconception: too much responsibility is supported by the *Object* class (respectively, the *Graphic* class). The *Object* class is responsible for displaying simple data and for decorating itself by use of all the possibilities represented by the *Concrete Decorator* classes. This is denoted by the development of composition relationships that involve the *Component* participant in both cases.

In Fig. 17, 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 and objects to decorate should be added.

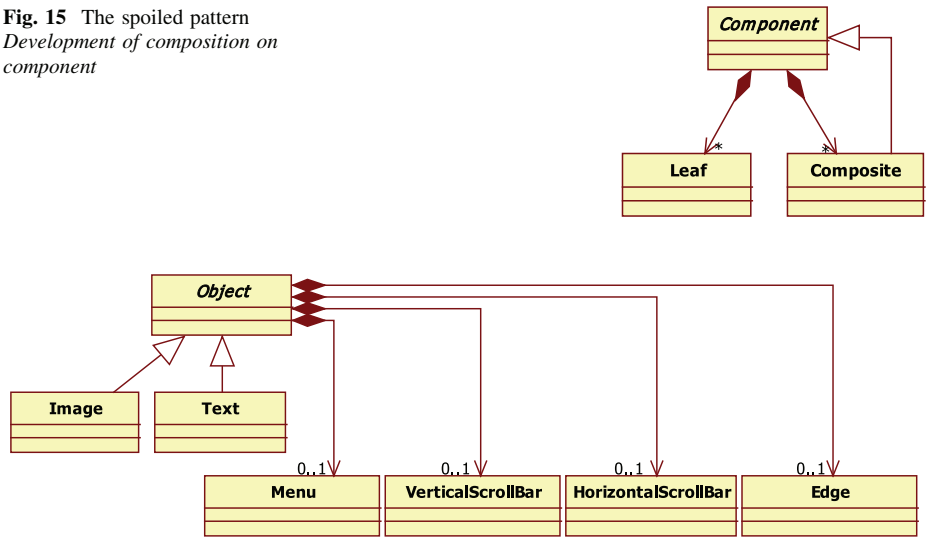


Fig. 15 The spoiled pattern *Development of composition on component*

Fig. 16 One alternative solution to the problem 2

Table 6 The strong points evaluation table of the spoiled pattern *Development of decorations on component*

Extensibility	
×	Addition or removal of a decorator does not need code modification
✓	Addition or removal of an object to decorate does not need code modification
Decoupling between decorator objects and objects to decorate	
✓	Minimal number of ≪ Decorator ≫ classes
×	Maximal factorization between decorators and object to decorate
Decorators managing on program execution	
×	Objects to decorate have any knowledge on decorators
×	A decorator may be decorated by another decorator

5.2.3 Problem 3: the bridge instantiation

An alternative solution to the use of the *Bridge* is presented in Fig. 18. Even if windows are correctly isolated from the GUI environment, the associations between each *Window* and *Style* are duplicated. 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, there might be some window types with different styles. Let us consider the pattern solution in Fig. 7. There is a single association link between *Window* and *Style*, and there is no specialized association between subclasses of *Window* and *Style*. Then, we consider that the style of all window classes is managed by a single piece of code: into a constructor, for example, the *Window* class. In the GoF, the responsibilities of the *Abstraction* are to define the abstraction's interface and to maintain a reference to an object of type *Implementor*. Then, the use of this pattern insures that all application windows have the same style at some point. Table 7 sums up the strong points evaluation.

In Fig. 19, we present the deduced spoiled pattern named *Development of delegation links*. Here, as in previous examples, delegation links are misplaced and should be factorized.

5.2.4 Problem 4: the state instantiation

For this problem, we present the two worst cases that we came across. The first alternative solution for the use of the *State* is presented in Fig. 20. In this case, each category is a subclass of *DVD* which entails the destruction of instances in order to perform a category

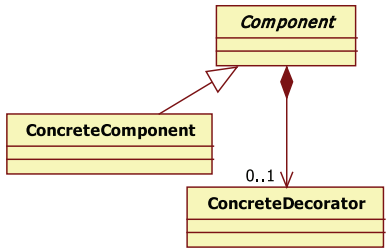


Fig. 17 The spoiled pattern *Development of decorations on component*

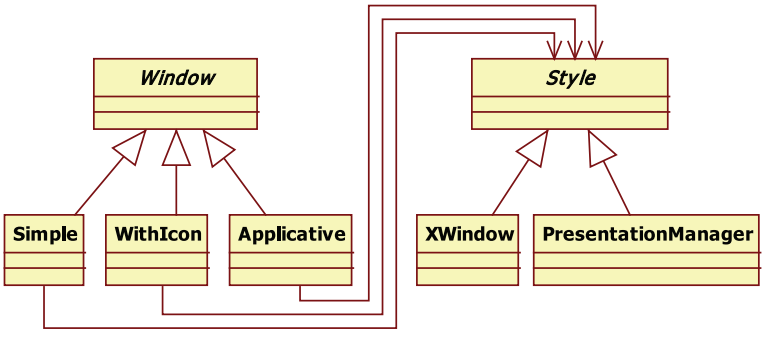


Fig. 18 One alternative solution to the problem 3

Table 7 The strong points evaluation table of the spoiled pattern *Development of delegation links*

Extensibility	
✓	Addition or removal of a concrete implementor does not need code modification
×	Addition or removal of a refined abstraction does not need code modification
Decoupling between abstraction and implementor	
✓	Minimal number of concrete implementor
×	Maximal factorization of the link between abstraction and implementor
×	One concrete implementor for all refined abstraction

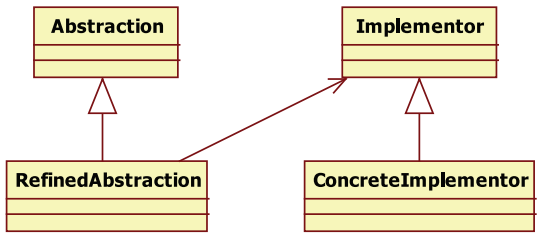


Fig. 19 The spoiled pattern *Development of delegation links*

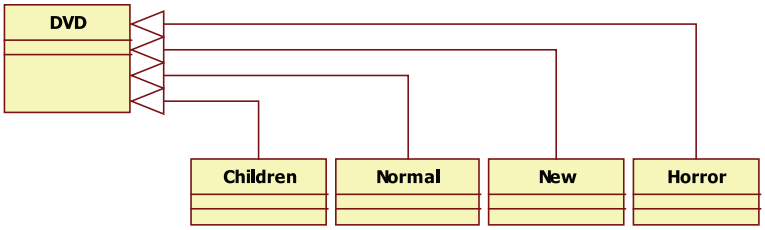


Fig. 20 One alternative solution to the problem 4

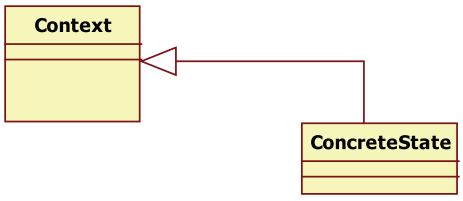


Fig. 21 The spoiled pattern *Spoiled classification*

change. The state management is complicated since there is no possibility of state modification at running time. However, an extensibility feature is available thanks to inheritance links.

In Fig. 21, we present the deduced spoiled pattern named *Spoiled classification*. It needs a *State* class that allows changing category without deleting and recreating a new identical instance (except for state data).

Another alternative solution to the use of the *State* is presented in Fig. 22. Here, the *DVD* class manages its state thanks to an enumeration. In doing so, the solution imposes a *switch* statement, and so, the change of category is possible.

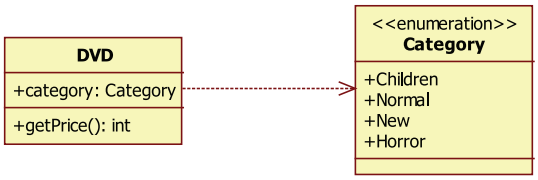


Fig. 22 One alternative solution to the problem 4

Table 8 The strong points evaluation table of the spoiled patterns *Spoiled classification* and *Hidden switch statement*

Extensibility		
✓	×	States' protocol factorization
✓	×	Addition or removal of a state does not need code modification
Simplified management		
×	✓	Possibility of state modification at execution time without deletion
×	×	State behavior decoupling
Figure 21	Figure 23	

Fig. 23 The spoiled pattern *Hidden switch statement*

The problem with this solution concerns extensibility. Indeed, if a new category is added, the *DVD* class must be modified to manage the new type. Table 8 sums up the strong points evaluation for each solution. In Fig. 23, we present the deduced spoiled pattern named *Hidden switch statement*. This is an ideal starting point for big refactoring dedicated in the introduction of the *State* pattern (Fowler et al. 1999).

5.2.5 Problem 5: the chain of responsibility instantiation

An alternative solution to the use of the *Chain of Responsibility* is presented in Fig. 24. Here, there is a separation between containers and contents. We have considered this solution valid even if delegation between content objects is not possible.

However, the main problem is the composition relationship. These links have two intents. The first is a composite/component relationship, and the second is for chaining the management of help messages. Then, we can say that these links have too many responsibilities. For example, it is possible to break the messages chain when reorganizing a part of the hierarchy of the graphical components. But, how should we consider this alternative solution? Is this solution an alternative to the *Chain of Responsibility* using pre-existing composition links or a side effect of a pre-existing alternative solution to the *Composite* between graphical components? Typically for this kind of problem, we would want to listen to the opinion of the community that works on patterns.

The alternative solution in Fig. 24 respects the chain of messages as illustrated in Fig. 25. When a help demand is activated, the object concerned has the possibility either to answer or 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 to deal with error messages, the static architecture between objects can be different. It seems

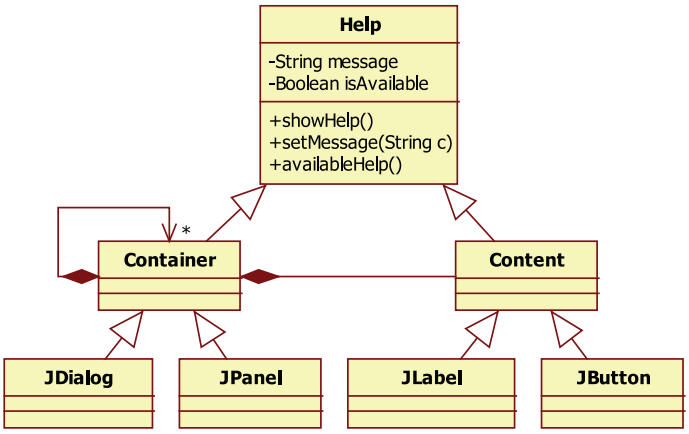


Fig. 24 One alternative solution to the problem 5

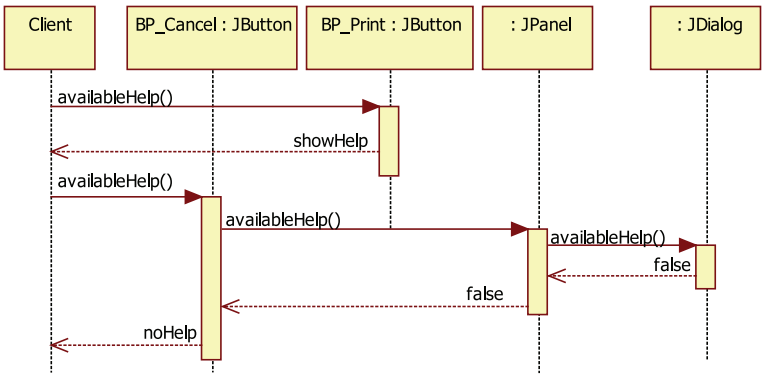


Fig. 25 The sequence diagram of the alternative solution presented in Fig. 24

likely that the study of such a behavioral pattern requires firstly a static diagram and then a complete set of test cases modeled by sequence diagrams. Table 9 sums up the strong points evaluation.

In Fig. 26, we present the deduced spoiled pattern named *Reuse of an association that preexists*. The reflexive association on the *Container* class must be pulled up to the *super* class.

5.2.6 Problem 6: the mediator instantiation

We present an alternative solution to the use of the *Mediator* in Fig. 27. Unfortunately, all the alternatives we have obtained correspond to the worst case presented in the GoF catalog. Students have solved the problem by the instantiation of the pattern itself, or they have solved the problem in accordance with the words of the problem statement. In this last case, message exchanges are in a complete graph form which represents perfectly the design problem to solve that uses the pattern. We explain this particular case in the next section.

Table 9 The strong points evaluation table of the spoiled pattern*Reuse of an association that preexists*

Extensibility	
×	Objects chaining factorization
×	Unique access point for the client class
✓	Addition or removal of a new element in the chain does not need code modification

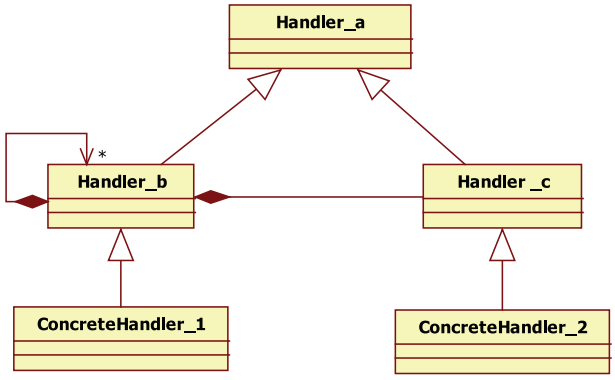
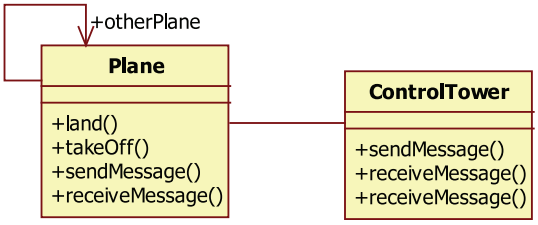


Fig. 26 The spoiled pattern *Reuse of an association that preexists*

Fig. 27 One alternative solution to the problem 6



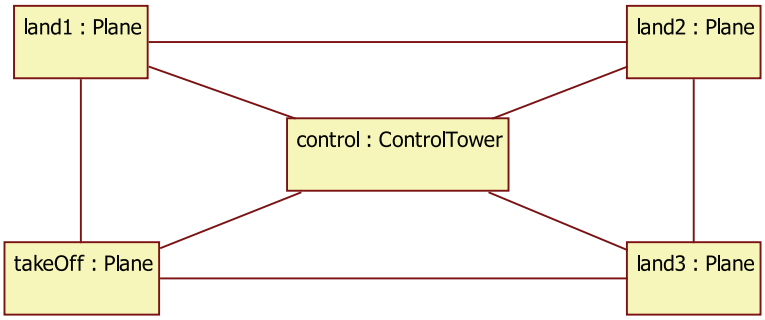


Fig. 28 A collaboration diagram of the alternative solution presented in Fig. 27

In the collaboration diagram in Fig. 28, we show the complete graph structure dedicated to the exchange of messages. We have selected a collaboration diagram to express this fact. Table 10 sums up the strong points evaluation.

Table 10 The strong points evaluation table of the spoiled pattern *Complete collaboration between concrete colleagues*

Extensibility	
×	Decoupled colleagues
×	Mediator's protocol factorization
Simplified management	
×	Simplified object protocols
×	Minimal number of exchanged messages

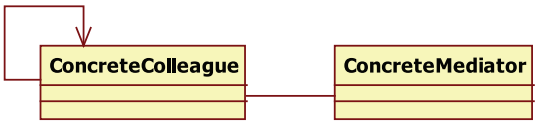


Fig. 29 The spoiled pattern *Complete collaboration between concrete colleagues*

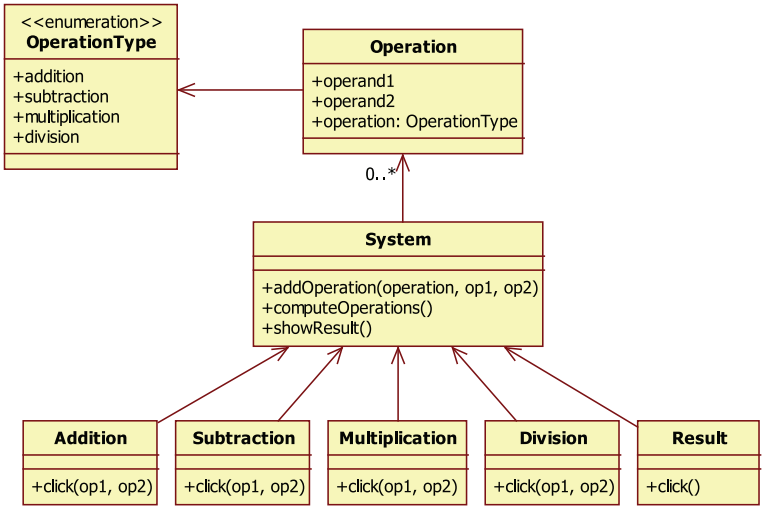


Fig. 30 One alternative solution to the problem 7

In Fig. 29, we present the deduced spoiled pattern named *Complete collaboration between concrete colleagues*.

5.2.7 Problem 7: the command instantiation

An alternative solution to the use of the *Command* is presented in Fig. 30. This solution grants all the management to the *System* class that has the responsibilities to execute the operation code selected into a *switch* statement that is hidden and to manage the history of commands thru links to the *Operation* class. Then, this last class plays the role of a *Memento* that stores the state of an invoked command. In addition, the role of the *Invoker* classes (Addition, Subtraction,...) consists simply in calls of the *addOperation* or the *computeOperations* for the *Result* class. In practice, this kind of human machine interface invoker is managed by a *Contoller* class that plays the role of *Invoker*. This fact denotes an over-engineered design for the problem. The solution is valid, even if an operation is not

Table 11 The strong points evaluation table of the spoiled pattern *No reification of command*

Extensibility	
×	Addition or removal of a command does not need code modification
×	Commands' protocol factorization
Uniform protocol	
✓	Unique access point for the command execution
Recording facilities	
×	Facilities in command recording

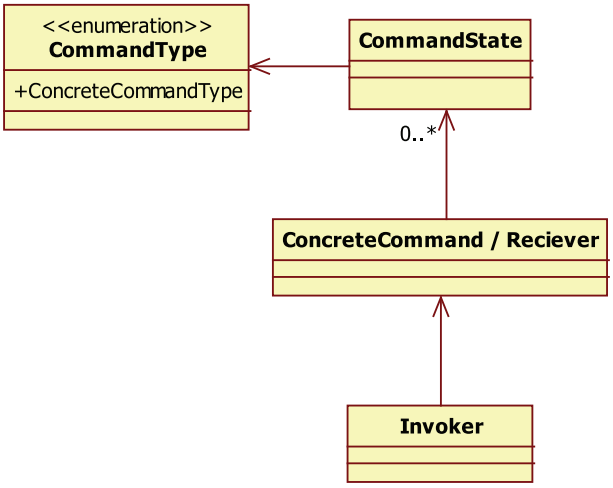


Fig. 31 The spoiled pattern *No reification of the command*

reified in a *Command* object. The *Command* design pattern uses this reification and the polymorphism to dispatch the *switch* statement code into the corresponding objects. Table 11 sums up the strong points evaluation.

In Fig. 31, we present the deduced spoiled pattern named *No reification of the command*. We could name it *Hidden switch statement*, as one spoiled *State* pattern, because it needs the use of dynamic binding for the selection of the appropriate operation. Here, the refactoring consists in work on the reification process which results in the construction of a class hierarchy of commands and creation of a real *Invoker* that stores the commands.

In this case, we can see that the *System* class has the responsibilities of two of the pattern participants: *ConcreteCommand* and *Receiver*. The classes, we have named *CommandType* and *CommandState*, play the part of the *Invoker* participant.

6 Discussion

In these experiments, we have found two major problems: one that concerns the problem statement and more generally the design of a problem, another concerns the specificity of the participants of our experiments. Moreover, even if it would be easier to randomly perturb static and/or dynamic structure of a pattern, we have selected to collect human

generated alternative solutions. This choice requires the analysis of more than one thousand models which constitutes very long and repetitive work.

6.1 The problem designing problems

We have specifically designed problems to collect alternative solutions. Then, the problem statement becomes crucial for our problem. This should not be too open or too directed. Please consider the first words that follow dedicated to the *Mediator*.

Design the communications of a plane that approaches an airport

(Version 1)

When a plane approaches an airport, it must announce *to all the other* p*lanes which are around that* it intends to land, and *await their confirmation* before they carry 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 or destination conflict between several planes

---

This above text is too open and does not conform to the pattern even if the last sentence explicitly implies the use of the *Mediator* and so a reduction of the exchange messages graph. In fact, if we design a system which scrupulously respects the first sentence of the problem, the instantiation of the *Mediator* is not direct and requires that the *Control Tower* class plays the role of a *Mediator*, after studying a collaboration diagram.

Generally, the problems are related to over-specification. For the second word problem, it is very difficult to not instantiate the *Mediator*, and then the problem ceases to be significant.

Design the communications of a plane that approaches an airport.

(Version 2)

When a plane approaches an airport, it must announce *to the control tower that* it intends to land, 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 or destination conflict between several planes.

This kind of problem statement happened with other problems. Let us consider a last example: an instantiation of the *Adapter* pattern on the statement problem that follows. For this exercise, we obtained only correct instantiations of the pattern, as we can see in Fig. 32.

Design a drawing editor

A drawing is composed of graphics (lines, rectangles and roses), positioned 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(): in* that returns the address of the drawing. It is probable that the system evolves in order to draw circles

---

The two first sentences imply the emergence of a new concept that we named *Shape*. This abstract class possesses the Cartesian coordinates of a precise position, and the shared protocol represented by the *draw* method. Then, we impose that the *Rose* complex graphic

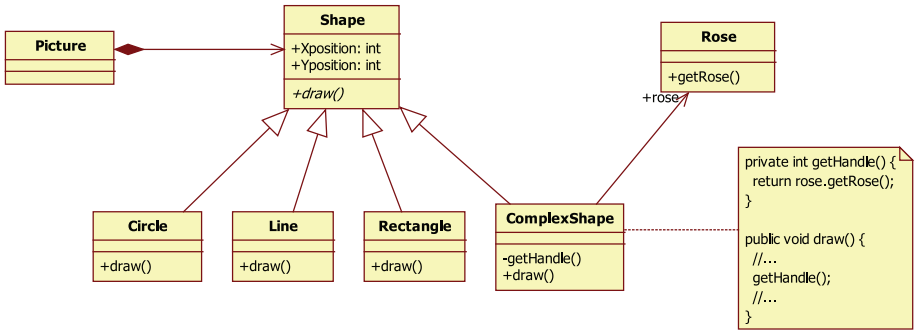


Fig. 32 A canonical solution of the problem of the drawing editor

class must be reused as a black-box component which implies the creation of an adapter class that inherits from the previous abstract class and which reuses the *Rose* class with a delegation link.

It is not easy to put forward a small problem dedicated to a specific design problem that is solvable by a single pattern and then solvable by a micro-architecture. There are several solutions: consider coarser problems and apply composite patterns (Riehle 1997), search problem topics from the experience of designers, ensure that problems are not too solution oriented, ensure that problems are easily solvable by the instantiation of a pattern and more complicated to solve without such a pattern, ensure that problems address other patterns.

6.2 The limits of our experiments

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

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

In order to avoid the multiplication of the same solutions and to increase the diversity of the alternative solutions suggested, it is advisable to encourage a broader scale experimentation by the implication of designers everywhere. The use of our collaborative website would make it possible to identify the most frequent spoiled patterns.

7 Related concepts

We now position spoiled patterns with some related concepts. First, we have a discussion about Riehle's point of view (Riehle 2011): for him some of our spoiled patterns are just

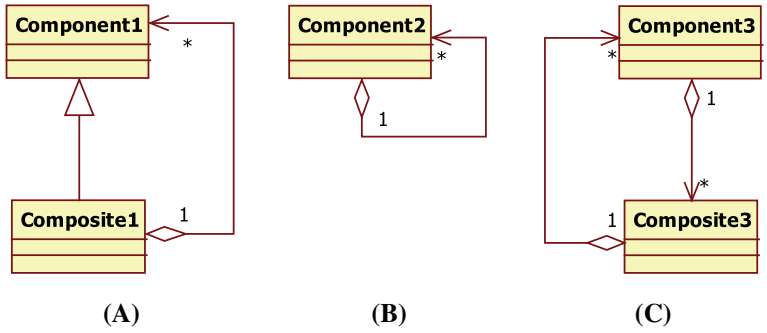


Fig. 33 Three structural variations of the *Composite* pattern (from Riehle)

structural variations of the same design pattern. Secondly, we place this new kind of pattern alongside anti-patterns and bad smells. "Bad smells in code" were introduced by Kent Beck and Martin Fowler to determine when to refactor in term of smells (Fowler et al. 1999). Anti-patterns are very similar: the description of each anti-pattern starts with a problematic solution, which generates a lot of negative consequences and then requires a refactored solution (Brown et al. 1998).

7.1 Structural variations

In his work on application of design patterns in industry projects (Riehle 2011), Riehle says: "... the structure diagram in the Design Patterns book suggests that there is only one particular structure that is to be considered as the pattern. This is an incorrect interpretation. The structure diagram in the Design Patterns book is an illustration of the most common form the pattern may take when it gets applied,..."

A direct application of this concept to the *Composite* pattern can be seen in Fig. 33. For Riehle, the first case (Fig. 33a) is the structure diagram presented in the GoF. The other cases are considered as correct variations. Unfortunately, the arguments given by Riehle are debatable.

First, the description of the *Composite* pattern is incomplete. In the GoF *participants* section, there are three participants: *Component, Composite*, and *Leaf*. To ignore the *Leaf* participant can be problematic. For the first variation (Fig. 33b), even if it is present in the *implementation* section of the GoF (*Should Component implements a list of Components?*), there is however a discussion about memory space considerations. In fact, a leaf inherits from the composite structure declared in the *Component* participant. For us, it is a spoiled pattern, even if we did not find in our experiments.

For the second proposed variation (Fig. 33c), we found it in our experiments. We have named this spoiled pattern *Recursive Composition* which invalidates all the strong points of the *Composite* pattern. Moreover, Riehle said in his work that it is a part of a firm-specific design language.

Structural variations and spoiled patterns can be related to the problem of a correct instantiation of a pattern with respect to a specific context. The UML 2.0 notation suggests the use of a parametrized collaboration to instantiate design patterns (Yacoub and Ammar 2003) (chapter 6). Initially, we agree with this form of instantiation which consists in the match of model classes with pattern participants in using a template collaboration. In this

context, the correctness of the instantiation corresponds to a validation of the collaboration between the objects and the pattern specification.

The main difference between Riehle's work and ours is the appreciation of a correct instantiation. For him, the relationship between pattern and template has a 1-to-*n* cardinality, so a pattern presents some structural variations. For us, we consider the relationship between the pattern and template is 1-to-1; thus, others are spoiled patterns, and the relationship between the template and application is 1-to-*n* as well.

Instead of validating the correctness of patterns instantiations just in time, we want to give designers the freedom to model their application without considerations about patterns, and later to keep track of instantiations of spoiled patterns. In our design review (Bouhours et al. 2009), if our tool discovers an alternative solution, if the designer agrees with the intent that corresponds to the pattern, and if he achieves the model refactoring, then we can say that a misconception was discovered and corrected.

7.2 Bad smells and anti-patterns

Bad smells and anti-patterns are only conjectures about bad design practices that enable to undertake the refactoring process. There are some works that study validity of these conjectures. For example, Khomh et al. (2012) explore the impact of 13 anti-patterns and bad smells on the maintenance of four Java systems. They demonstrate by scrutinizing the change history (the releases) and issue-tracking systems that classes participating in anti-patterns have higher tendency to change or to be subject to fault-fixing than other classes.

7.2.1 Bad smells

Kent Beck and Martin Fowler have introduced the term "bad smells" in Fowler et al. (1999). 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, and the procedures to follow to carry out this reorganization. For example, code duplication is a bad smell which can be corrected by the refactoring "extract method" (Fowler et al. 1999) which consists in the addition of a method in a class that factorizes parts of the code concerned.

An alternative solution, identified in a model, indicates where a defect is able to generate undesirable effects on the micro-architecture level and target a zone which would have to be restructured. Whereas bad smells were defined to target pieces of code, spoiled patterns target fragments of UML models.

7.2.2 Anti-patterns

Whereas a design pattern presents a canonical architecture to be followed to solve a problem, an anti-pattern presents a learned lesson. It describes the effects that result from bad design practices and gives the procedure to follow that tends toward a better software quality. Then, an anti-pattern makes it possible to check or supervise bad practices (Brown et al. 1998). An anti-pattern can also represent best design practices, but if used in an excessive way, eventually produce consequences more harmful than the anticipated results (Dodani 2006). In all these cases, an anti-pattern suggests a refactored solution. As an example, let us quote the anti-pattern "makes an active attempt", in concurrent

programming, which is to test a condition until it is checked. This anti-pattern can be corrected by scrolling events or signals.

Authors of Brown et al. (1998) suggest a software design-level model composed of seven architectural levels: objects and classes, micro-architecture, macro-architecture or frameworks, application, system, enterprise, and global industry. At the same time, they present three kinds of anti-patterns: development anti-patterns that correspond to bad smells at implementation level, architectural anti-patterns that correspond to macro-architecture and application levels, and managerial anti-patterns that correspond to application, enterprise and global levels.

We consider that spoiled patterns are anti-patterns at the micro-architecture level. However, a spoiled pattern does not give information that allows the correction of a bad solution. The set of useful operations to substitute it is much more precise than refactorings suggested by an anti-pattern. Thanks to its description, a spoiled pattern can be automatically detectable (Bouhours et al. 2010) that which is not the case, nor the goal, of anti-patterns.

8 Conclusion

A spoiled pattern is a generic micro-architecture that produces, no matter what the global context, non-optimal solutions to a design problem. Then, spoiled patterns can be considered as bad smells at the modeling stage or as anti-patterns at micro-architecture level. In comparing solutions instantiated by design patterns and solutions instantiated by spoiled patterns, we have inferred some good design properties of patterns. Improvement of the intrinsic quality of the architecture is denoted in terms of strong points. Strong points are regrouped by best practices and enable the evaluation of the degree of damage caused by an alternative solution in a design.

Currently, the GoF description template is composed of an intent, a motivation, an applicability, a structure, some participants, some collaborations, some consequences, an implementation, a sample code, some known uses, and some related patterns (Gamma et al. 1995). Generally, the *motivation* section contains the example of a problem, which is a specific case where the pattern is a good solution. Our first extension of the template consists in adding new problems. For each problem added, we list all the instantiations of the spoiled patterns we have collected. These alternative solutions refer to a new section named “spoiled patterns”. Finally, our strong points complete the *consequence* section by the presentation of a new perspective on the best practices brought by the pattern. Moreover, the evaluation of these strong points determines the degradation of each spoiled pattern.

As major consequence, spoiled patterns can be used for computer science education. If, in an object-oriented design course, we present design patterns solely as best practices, we lose the concept of an emergent and approved solution by a community with its strengths and its weaknesses. On the other hand, if we start with alternative solutions, we can analyze the defects by making explicit the strong points and then present best design practices. We can even take the opportunity to introduce code and pattern refactorings that are crucial for software adaptability.

Moreover, early detection of spoiled patterns can be useful during a weekly meeting of the development team covering the architecture. We have developed a tool that permits a search for alternative solutions at architectural level. Our detection method is based on

structural properties of UML models and allows the identification of all possible instantiations of a generic micro-architecture (Bouhours et al. 2010).

The collection itself is an experiment of interest for several reasons. We can evaluate the degree of maturity of our students in object design. We can point out the comprehension problems linked to some essential object mechanisms used to implement design patterns. We can extrapolate best design practices encapsulated in design patterns. We can construct a base of model fragments from which we can question the relevance of a design and propose a design review activity in the same manner as the code review activity proposed by Fagan (2002). However, experiments that we have conducted consume time and are concentrated on a specific panel. Therefore, we have played too many roles: teacher, analyst, specialist, and committee member. Then, we propose a collaborative website for spoiled patterns dedicated to the community.

Indeed, our website[3] contains all GoF patterns extended with our concepts and offers a contribution system for the submission of new problems, new alternative solutions, new spoiled patterns and new strong points. Each submission is subjected to a committee that examines its validity and its interest as a spoiled pattern. Thus, this site would allow the emergence of a community of experts that opens up a shared zone of best design practices.

References

Albin-Amiot, H., & Guéhéneuc, Y. G. (2001a). Meta-modeling design patterns: Application to pattern detection and code synthesis. In B. Tekinerdogan, P. V. D. Broek, M. Saeki, P. Hruby, & G. Suny (Eds.), *Proceedings of the 1st European conference on object-oriented programming (ECOOP) workshop on automating object-oriented software development methods*. Centre for Telematics and Information Technology, University of Twente. TR-CTIT-01-35.

Albin-Amiot, H., Cointe, P., Guéhéneuc, Y. G., & Jussien, N. (2001). Instantiating and detecting design patterns: Putting bits and pieces together. In D. Richardson, M. Feather, & M. Goedicke (Eds.), *Proceedings of the 16th conference on automated software engineering (ASE)* (pp. 166–173). IEEE Computer Society Press.

Baroni, A. L., Guéhéneuc, Y. G., & Albin-Amiot, H. (2003). *Design patterns formalization*. Research report 03/03/INFO, Computer sciences department, École des Mines de Nantes.

Bouhours, C., Leblanc, H., & Percebois, C. (2011). Sharing bad practices in design to improve the use of patterns (regular paper). In *International conference on pattern languages of programs 2010 (PLoP)*. Reno, Nevada, USA: ACM DL

Bouhours, C., Leblanc, H., Percebois, C., & Millan, T. (2010). Detection of generic micro-architectures on models. In *Proceedings of PATTERNS 2010, the second international conferences on pervasive patterns and applications* (pp. 34–41). Lisbon, Portugal.

Bouhours, C., Leblanc, H., & Percebois, C. (2009). Bad smells in design and design patterns. *Journal of Object Technology*, *8*(3), 43–63.

Brown, W. J., Malveau, R. C., & Mowbray, T. J. (1998). *AntiPatterns: Refactoring software, architectures, and projects in crisis*. London: Wiley.

Dewan, P. (2005). Teaching inter-object design patterns to freshmen. In *Technical symposium on computer science education* (Vol. 37, pp. 482–486).

Dietrich, J., & Elgar, C. (2005). A formal description of design patterns using owl. In *Proceedings of the 16th Australian software engineering conference* (pp. 243–250). Los Alamitos, CA, USA: IEEE Computer Society.

Dodani, M. (2006). Patterns of anti-patterns? *Journal of Object Technology*, *5*(5), 29–33.

Dong, J., & Zhao, Y. (2007). Classification of design pattern traits. In *Proceedings of the 19th international conference on software engineering and knowledge engineering (SEKE)* (pp. 473–477).

Duell, M., Goodsen, J., & Rising, L. (1997). 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)* (pp. 120–124). New York, NY: ACM.

[3] Reachable at http://www.goprod.bouhours.net.

Eden, A. H., Yehudai, A., & Gil, J. (1997). Precise specification and automatic application of design patterns. In *Proceedings of the 12th international conference on automated software engineering (ASE)* (pp. 143–152). Washington, DC, USA: IEEE Computer Society.

El-Boussaidi, G., & Mili, H. (2008). Detecting patterns of poor design solutions using constraint propagation. In *Proceedings of the 11th international conference on model driven engineering languages and systems (MoDELS)* (Vol. 5301/2009, pp. 189–203). Berlin: Springer.

Fagan, M. (2002). Design and code inspections to reduce errors in program development. In M. Broy & E. Denert (Eds.), *Software pioneers* (pp. 575–607). New York, NY: Springer.

Fowler, M., Beck, K., Brant, J., Opdyke, W., & Roberts, D. (1999). *Refactoring: Improving the design of existing code*. Boston, MA: Addison-Wesley.

France, R., Ghosh, S., Song, E., & Kim, D. K. (2003). A metamodeling approach to pattern-based model refactoring. *IEEE Software*, *20*(5), 52–58.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns*. Boston, MA: Addison-Wesley.

Gibbon, C. A., & Higgins, C. A. (1996). Towards a learner-centred approach to teaching object-oriented design. In *Proceedings of the third Asia-Pacific software engineering conference, APSEC '96* (pp. 110–117). Washington, DC: IEEE Computer Society.

Gibbon, C., & Higgins, C. (1996). Teaching object-oriented design with heuristics. *SIGPLAN Notices*, *31*, 12–16.

Guennec, A. L., Sunyé, G., & Jézéquel, J. M. (2000). Precise modeling of design patterns. In *Proceedings of 3rd international conference on the unified modeling language (UML)* (pp. 482–496). Berlin: Springer.

Harb, D., Bouhours, C., & Leblanc, H. (2009). Using an ontology to suggest design patterns integration. In M. Chaudron (Ed.), *Workshops and symposia at models 2008, 5421* (pp. 318–331). Toulouse: Springer (Best paper).

Jiménez-Díaz, G., Gómez-Albarrán, M., & González-Calero, P. A. (2008). Teaching GoF design patterns through refactoring and role-play. *International Journal of Engineering Education*, *24*, 717–728.

Kampffmeyer, H., & Zschaler, S. (2007). 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), lecture notes in computer science* (Vol. 4735/2007, pp. 211–225). Springer.

Kerievsky, J. (2005). *Refactoring to patterns. Addison-Wesley signature series*. Boston, MA: Addison-Wesley.

Khomh, F., Penta, M. D., Guéhéneuc, Y. G., & Antoniol, G. (2012). An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Software Engineering*, *17*(3), 243–275.

Larman, C. (2002). *Applying UML and patterns: An introduction to object-oriented analysis and design and the unified process* (2nd edn). Prentice Hall PTR, Upper Saddle River, NJ, USA. Chapter 16—GRASP: Designing Objects with Responsibilities.

Mak, J. K. H., Choy, C. S. T., & Lun, D. P. K. (2004). Precise modeling of design patterns in UML. In *Proceedings of the 26th international conference on software engineering (ICSE)* (pp. 252–261). Los Alamitos, CA, USA: IEEE Computer Society.

Mili, H., & El-Boussaidi, G. (2005). Representing and applying design patterns: What is the problem? In *Proceedings of the 8th international conference on model driven engineering languages and systems (MoDELS)* (pp. 186–200).

O'Cinnéide, M., & Nixon, P. (1999). A methodology for the automated introduction of design patterns. In *Proceedings of the 15th IEEE international conference on software maintenance (ICSM)* (pp. 463–473). Washington, DC, USA: IEEE Computer Society.

Pecinovský, R., Pavlíčková, J., & Pavlíček, L. (2006). Let's modify the objects-first approach into design-patterns-first. *ACM Sigcse Bulletin*, *38*, 188–192.

Riehle, D. (1997). Composite design patterns. In *Proceedings of the 1997 ACM SIGPLAN conference on object-oriented programming systems, languages and applications* (pp. 218–228). ACM Press.

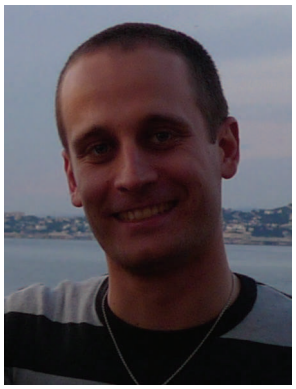
Riehle, D. (2011). Lessons learned from using design patterns in industry projects. In J. Noble, R. Johnson, P. Avgeriou, N. Harrison, & U. Zdun (Eds.), *Transactions on pattern languages of programming II, lecture notes in computer science* (Vol. 6510, pp. 1–15). Berlin, Heidelberg: Springer. doi:10.1007/978-3-642-19432-0_1.

Sendall, S. (2002). Gauging the quality of examples for teaching design patterns. In *Workshop on "Killer Examples" for design patterns and objects first, at the conference on object-oriented programming systems, languages and applications (OOPSLA'2002)*, Seattle, USA, November 4 (2002). Also available as Technical Report IC/2002/83, Ecole Polytechnique Fédérale de Lausanne (EPFL), School of Computer and Communication Sciences.

Siddle, J. (2011). Choose your own architecture—interactive pattern storytelling. In J. Noble, R. Johnson, P. Avgeriou, N. Harrison, & U. Zdun (Eds.), *Transactions on pattern languages of programming II,*

Lecture notes in computer science (Vol. 6510, pp. 16–33). Berlin: Springer. doi:10.1007/978-3-642-19432-0_2.

Yacoub, S., & Ammar, H. (2003). *Pattern-oriented analysis and design: Composing patterns to design software systems*. Boston, MA: Addison-Wesley Longman.



Cédric Bouhours is associate professor in computer sciences at the University of Clermont-Ferrand since 2011. His research interests include design patterns, best practices, Agile methods, and refactoring. He received a Ph.D. in computer science from University of Toulouse. He works on methods and tools promoting the use of design patterns in models. The final aim is to maximize the automation of the misaligned patterns detection in models and to enable their detection directly during coding.

Hervé Leblanc is associate professor in computer sciences at the University of Toulouse since 2002. His research focuses on best practices in the field of software engineering. He worked on automatic restructuring class and interface hierarchies thanks to Galois lattices during his thesis. He received a Ph.D. in computer science from University of Montpellier II. His main research interests are now linked to design patterns (misaligned patterns, correct instantiation in mixing Riehle role diagrams, and Lepus formalization), traceability of models transformations in MDE processes, and more recently Agile methods.

Christian Percebois is professor in computer sciences at the University of Toulouse since 1992. He was always interested in software engineering. In the beginning he worked on Lisp and Prolog interpreters, garbage collecting for symbolic computations, asynchronous backtrackable communications in parallel logic languages, abstract machine construction through operational semantics refinements, typing in object-oriented programming and multiset rewriting techniques in order to coordinate concurrent objects. Today his main research tries to combine formal methods and software engineering, in particular for graph rewriting systems.