View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Scientific Publications of the University of Toulouse II Le Mirail

Bad smells in design and design patterns

Cédric Bouhours, Hervé Leblanc, Christian Percebois

To cite this version:

Cédric Bouhours, Hervé Leblanc, Christian Percebois. Bad smells in design and design patterns. The Journal of Object Technology, Chair of Software Engineering, 2009, 8 (3), pp.43–63. <hal-00522587>

HAL Id: hal-00522587

https://hal.archives-ouvertes.fr/hal-00522587

Submitted on 24 Nov 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2009**

Vol. 8, No. 3, May-June 2009

Bad smells in design and design patterns

Cédric Bouhours, **Hervé Leblanc**, and **Christian Percebois**

Abstract

To give a consistent and more valuable property on models, model-driven processes should be able to reuse the expert knowledge generally expressed in terms of patterns. We focus our work on the design stage and on the systematically use of design patterns. Choose a good design pattern and ensure the correct integration of the chosen pattern are non trivial for a designer who wants to use them. To help designers, we propose design inspection in order to detect "bad smells in design" and models reworking through use of design patterns. The automatic detection and the explanation of the misconceptions are performed thanks to spoiled patterns. A "spoiled pattern" is a pattern which allows to instantiate inadequate solutions for a given problem: requirements are respected, but architecture is improvable.

1 INTRODUCTION

The MDE community, aiming at giving a productive property on models, has proposed a framework for model-driven processes development. However, to obtain guarantees on model relevance at the end of each activity, these processes should promote the reuse of analysis [Fowler97], design [Gamma95], or architectural [Buschmann96] patterns. After considering that the use of analysis patterns is business domain specific, and that the use of architectural patterns must be planned before any design stage, we have chosen to focus our work on the design stage and on design patterns. Design patterns have some advantages concerning genericity, reusability, and integrability. They are business model independent, and promote only architectural qualities. Their use must not be necessarily planned, and thus allow a designer to integrate a design pattern into its object architecture, at any moment.

A design pattern represents an expert knowledge, validated by the community, reusable for a type of design problems. For example, the *Composite* design pattern represents the optimal solution for structural and compound problems, as to compose objects, to build tree structures, and to nest objects. This pattern is contained in the GOF catalog [Gamma95] with its structure, its intent, and some information which allows to use it under the best condition. This catalog regroups twenty three design patterns which

Cédric Bouhours, Hervé Leblanc, Christian Percebois: "Bad smells in design and design patterns", in *Journal of Object Technology*, vol. 8, no. 3, May-June 2009, pp. …..
http://www.jot.fm/issues/issue_2009_03/column4

concern twenty three types of problem. So, during the design stage, through use of this catalog, designers may be able to use the optimal solution for the problems they want to design.

However, two troubles arise when the designers want to use design patterns:

- The designers have to identify what type of problem they design, to choose a good design pattern. Indeed, the designers have to abstract the problem they design before identify the type of problem. To facilitate the designer choice, some works exists to improve design pattern classification [Hasso04] [Diaz87], and to propose a dedicated pattern knowledge base [Kampffmeyer07]. These works are related to design problem formalization.
- The designers have to ensure a correct integration of the chosen pattern in a model. To do so, some works extend the capability of UML notation [France04] or propose an automatic transformation process which preserves behavior of the model [Sunyé01]. These works are related to the design pattern formalization.

We have a complementary approach which permits the designers to avoid this choice by automatically inspecting models. As there are code review activities [Fagan02], we propose a design review activity, situated after the design stage, which automatically verifies if there is no known bad design practices in a model. To find model fragments substitutable with design patterns, our activity automatically parse the model to identify fragments that are similar computed similarities with a knowledge base of bad design practices. After a dialog with the designer to verify the intent of the fragment identified, a refactoring transformation is applied on the model to inject the needed design patterns. So, the designers do not need to identify design problem; it is the activity which identifies inadequacies in their model and thus suggests design patterns integrations instead.

The initial part of this paper defines our concepts which initially suppose that a design pattern is the optimal reusable micro-architecture for a type of problem. In section 2 some specific definitions are introduced about others architectures which resolve the same problem leads to the so-called “spoiled pattern”. In section 3, we focus on a design review activity directed by design patterns. Its originality consists in an automatic research of the instantiation of spoiled patterns which determines the fragments that may be substitutable with a design pattern. The whole activity is finally illustrated in section 4 on an example where the aim is to implement a file management system. Our detection algorithm, based on comparisons of structural similarities, identifies in the corresponding UML model an alternative fragment of the *Composite* spoiled pattern which may be replaced by the suggested design pattern.

2 SPOILED PATTERNS AND ALTERNATIVE FRAGMENTS

In order to encourage the reuse of design patterns in models, we want to identify model fragments substitutable with patterns in a design. Indeed, we identify the part of the design where the designer has used another way to solve a design problem which would

be better designed with a design pattern. This other way concerns the "spoiled pattern" concept that underpins our activity.

After the presentation of our work hypotheses, we give the definitions of the concepts we use. Then we present them on an illustration. We terminate this section with a comparison of works closely related to ours in particular bad smells and anti-patterns.

Hypotheses

> **Axiom 1:** *"A design pattern" is the optimal reusable micro-architecture for a type of problem.*

> **Corollary 1:** *Then, for each design problem that is solvable with a design pattern, "the optimal solution" is the instantiation of the design pattern.*

Design patterns are approved solutions, validated by an expert community specialized in object-oriented design. They are the result of know-how accumulation. Design patterns are context-free, generic, and thus it is necessary to adapt them to the context of the problem we want to solve. We name this process an "instantiation". For us, the optimal solution comes from the instantiation of the pattern.

We have wittingly chosen to represent a design pattern as a class diagram (inspired from the structure section of the GOF catalog), only with pattern participants as class name and inter-classes relations (association and inheritance links). Indeed, since we are at the design level, we are interested in the architecture of the pattern (classes and relations between them). As there are too many variants of a pattern that depend on the problem to solve, a combinatorial explosion of the different possible forms of instantiations may exist. In this context, our approach is to identify structural similarities using model comparisons during the preliminary design stage, without dealing with a detailed implementation which may complicates the identification of the patterns instantiation.

Some definitions

> **Definition 1:** *"An alternative solution" is a valid solution but with a different architecture compared to the optimal solution.*

So, the requirements of the design are respected but the relations inter-classes are different or/and there is not the whole pattern participants.

> **Corollary 2:** *Then, an alternative solution is an inadequate solution for a given problem, and is substitutable with the instantiation of the concerned pattern.*

> **Definition 2:** *"A spoiled pattern" is the abstraction of one alternative solution, in the same manner as a design pattern is the abstraction of optimal solutions. A spoiled pattern is linked to one and only one design pattern.*

A spoiled pattern is comparable to a design pattern in the sense that it is reusable to produce models which solve problems. There are one or more spoiled patterns for one design pattern. Structurally, a spoiled pattern is represented at the same level of

granularity as a design pattern allowing us to identify them as design patterns. The abstraction is the inversion of the instantiation.

Definition 3: *"The remarkable properties" of a spoiled pattern are its significant UML architectural characteristics (associations, generalizations, and composition links, but neither interfaces nor class semantics).*

We have decomposed the remarkable properties into two subsets: the local properties that characterize individually each class and the global properties which characterize the classes against each other depending on their inter-relations. This separation allows us to constitute different levels of filters during the detection, through use of structural similarity comparisons. The result of the search is a set of fragments identified in the model analyzed.

Definition 4: *"An alternative fragment" is a model fragment such as its remarkable properties match with the remarkable properties of a spoiled pattern and whose intent conforms to the corresponding pattern.*

This match indicates that there is a structural concordance between an alternative fragment and the spoiled pattern. This means that classes and inter-class relations defined by the spoiled pattern have been identified within the alternative fragment. Consequently, an instantiation of the spoiled pattern on the design problem leads to an alternative fragment.

A spoiled pattern does not have the entire pattern properties, and is thus spoiled. To note this degradation, we consider the non achieved properties of the design pattern that come from the instantiation of the spoiled pattern.

Definition 5: *"The pattern properties" of a design pattern are criteria of object-oriented architecture or software quality factors, partially deduced from the consequences section of the GoF catalogue and from our study on the design defects of spoiled patterns. They valorize why the design pattern is the optimal solution for a problem.*

Therefore, an alternative fragment is substitutable with the optimal fragment issued from the design pattern instantiation.

Definition 6: *"An optimal fragment" is the substitution of an alternative fragment by the instantiation of the corresponding design pattern.*

Figure 1 links all our concepts together. This synopsis presents two symmetrical notions between design patterns and spoiled patterns: an optimal fragment and an optimal solution share the same concepts in the same design pattern; in the same way, one can observe that an alternative fragment and an alternative solution have similar aspects with respect to the same spoiled pattern.

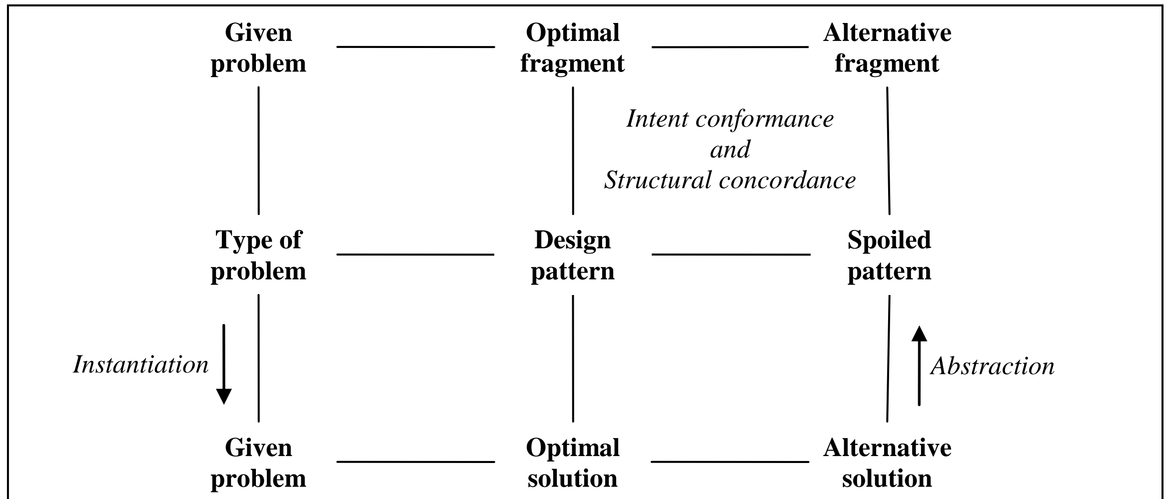


Figure 1: Synopsis of our concepts

Illustration

To illustrate the concepts presented in the previous section, we use the *Composite* pattern described by Figure 2. According to [Gamma95], the intent of this pattern is "*Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly*". In following our work hypothesis, this pattern is the optimal solution to solve structural and compound problems, more precisely to compose objects, to build tree structures, and to nest objects. The *Composite* pattern introduces three participants: an abstract *component*, a *composite*, and a *leaf*. The abstract *component* defines the interface for objects in the composition, defines an interface to manage the composition, and offers a unique access point for the client; this entity allows the factorization of the composition on *composites* and *leaves*. The *composite* participant manages the composition relationship and delegates operations along the tree structure.

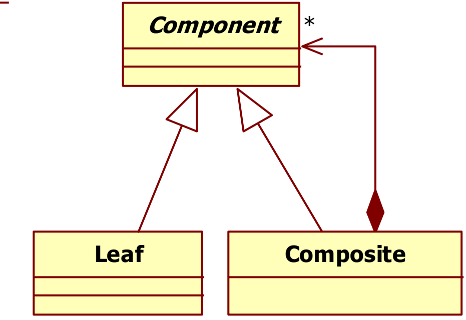


Figure 2: Structure of the *Composite* design pattern

Let us consider the following problem extracted from [Gamma95]: "*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*". To instantiate the pattern to this problem, we must identify the classes in the

problem with identical responsibilities as the pattern participants. Thus, we obtain an instantiation of the *Composite* pattern, and according to our hypothesis, we consider that Figure 3 represents the optimal solution for the problem.

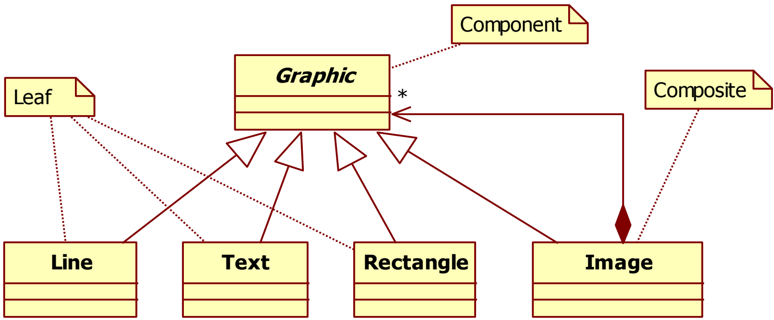


Figure 3: The optimal solution

Figure 4 presents an alternative solution of the previous problem. In this model, we can locate that an image is composed of other images which could be composed of lines, rectangles or texts. So, the requirements are respected in terms of objects composition. The *Graphic* class is used to assume the factorization of the protocols and to be the unique access point for the client. However, the fact that *Line*, *Rectangle*, and *Text* are linked to *Image* will cause some code modifications in *Image* if new classes are added with leaf or composite responsibilities. So, if a new *Circle* class is added as leaf, the *Image* class has to reference the new class. This is not the case in using the design pattern presented in Figure 3.

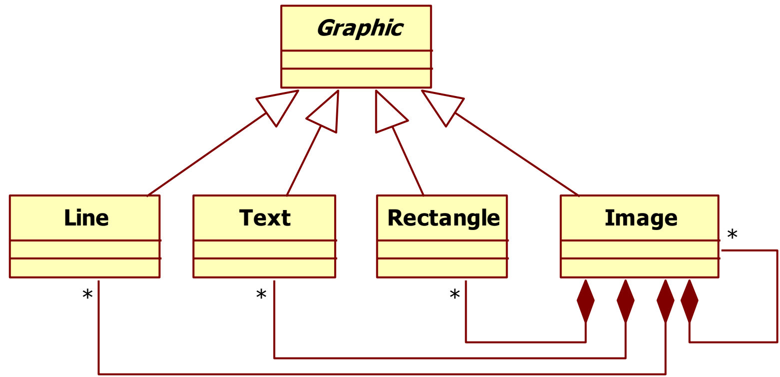
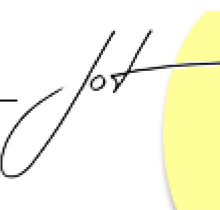


Figure 4: One alternative solution

To detect the alternative solutions in models with its structural variants, an abstraction must be done. Indeed, an alternative solution is associated to a problem context, and so must be abstracted in order to obtain a spoiled pattern. To do so, we will try to identify the pattern participants in the alternative solution, in doing a model reduction after marking the responsibility of each class.

The first step of the abstraction process consists in marking each class of the alternative solution with a name of a pattern participant corresponding to the responsibilities of the class. The result, resumed by Figure 5, explains a match between alternative solution and pattern participants.



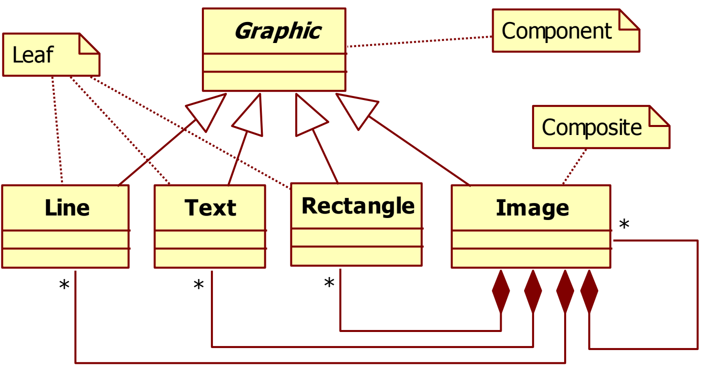


Figure 5: The alternative solution after the mark step

After the marking, the second step of the abstraction consists in keeping one of each participant and in transforming them in classes linked in the same manner as in the alternative solution. So, we obtain a spoiled *Composite* pattern where the composition is developed on the *Composite* class. Figure 6 resumes this step of an abstraction process.

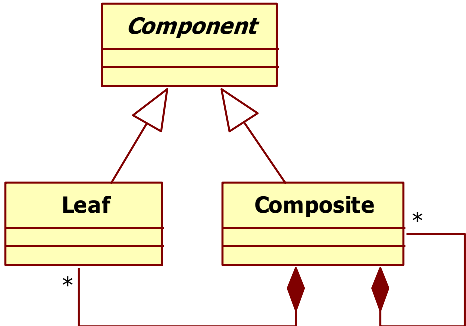


Figure 6: A spoiled *Composite* pattern

To classify the spoiled patterns, we have quantified their degree of degradation through use of the patterns properties. Indeed, each spoiled pattern does not use all the whole of the patterns properties. For the *Composite* pattern of Figure 2, the maximal factorization of the composition and the standardization of the protocol, thanks to the inheritance links, allow us to say that the pattern properties are "*decoupling and extensibility*" and "*uniform protocol*". As the composition is now recursive and developed on leaf participants, the main problem for the spoiled *Composite* pattern presented in Figure 6 is the lost of the advantages of the "d*ecoupling and extensibility*". However, as there are always the inheritances links, this spoiled pattern keeps the "*uniform protocol*".

Table 1 summarizes the properties with respects to the *Composite* pattern. The presence of the corresponding property in the spoiled pattern is represented with ✔ marks, the absence with ✖ marks.

Composite pattern properties	Characterization
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.

Table 1: Qualification of the spoiled *Composite* design pattern

To detect the spoiled patterns, we use their remarkable properties. The remarkable properties allow us to do the detection in the same way as a structurally conformant relationship or as a sub-graph matching that is known to be NP-complete. Due to the reification of association and generalization links in the UML meta-model, we consider that we have a specific sub-graph matching problem, since nodes that are not classes are strongly typed. Moreover our detection algorithm template retrieves a family of sub-graphs that have the same core sub-graph and treats the case of extraneous links that change desired semantics. Each spoiled pattern has a specific detection query automatically completed by an inspection of a dedicated structural property model. Local remarkable properties serve to mark classes with their possible spoiled pattern participants in the model to review. Global remarkable properties serve to filter marked classes and to regroup them into alternative fragments. The reference participant serves to limit the time complexity of queries. It is chosen according to its local remarkable property complexity and the other participants use it for the global remarkable properties specification.

If we consider the spoiled *Composite* pattern represented in Figure 6, we can see that the *composite* participant is the most constrained. So, we define it as the reference participant. Structurally, we can say that the *composite* participant has at least two composition relationships, one to *leaf* and one that is recursive. Concerning the *component*, we notice that it is characterized with two inheritance links from *leaf* and from *composite*. Lastly, the *leaf* participant is a component of *composite*. Table 2 presents the structural properties of the *Composite* spoiled pattern presented in Figure 6.

<table>
<tr><td>Reference participant</td><td colspan="2">Composite</td></tr>
<tr><td rowspan="3">Local properties</td><td>Composite</td><td>Class with at least two compositions (0..*) with one recursive and one generalization.</td></tr>
<tr><td>Leaf</td><td>Class with at least one end of composition (0..*) and one generalization.</td></tr>
<tr><td>Component</td><td>Class with at least two specializations.</td></tr>
<tr><td rowspan="2">Global properties</td><td>Leaf</td><td>Subclass of Component linked to the reference role with a composition (0..*).</td></tr>
<tr><td>Component</td><td>Super-class of the reference role and of Leaf.</td></tr>
</table>

Table 2: The structural features of the *Composite* spoiled pattern

Our activity uses the spoiled patterns to search for substitutable fragments. To be effective, we must to be sure that the spoiled patterns large enough entities that people may grasp. Indeed, as the models analyzed by the activity are designed by people, the spoiled patterns must be designed by people, too. To do so, we have suggested to some students without special pattern knowledge to solve some specific design problems. We have written each problem in using the intent paragraph or the motivation example of the GOF catalog. Thus, the optimal solution of each problem was the instantiation of the concerned design pattern, but the students have proposed alternative solutions without the use of pattern instantiation.

We have realized some different experiments which concern structural and behavioral design patterns. Our collection hypothesis is that problems we have made for our experiments are efficiently accurate to imply that the optimal solution is a unique design pattern. Therefore, when a model result is composed with a pattern that is different to the pattern solution of the problem, we consider that it is not valid. For example, if the problem concerns the *Composite* pattern, a *Decorator* pattern is not a solution. This process can take into consideration pattern composition in collecting spoiled patterns for each problem solved by a precise composition of patterns.

For the structural patterns, we have collected more than three hundred models. One hundred and fifty of them constituted a real alternative solution of the pattern. Others did not solve the problem or did not respect the requirements. The validation process is currently done manually. After the removal of duplications, we have conserved fifteen models only. They constitute the alternative solution to the pattern for the problems (six for the *Composite* pattern, six for the *Decorator* pattern, and three for the *Bridge* pattern) [Bouhours07].

Thus, after the experiments, we have obtained a set of alternative solutions that we have abstracted to deduce the spoiled patterns. Thus, we have constituted a catalog of bad design practices classified by design patterns, which contain the identified spoiled patterns and their valuations that belong to the design pattern's properties.

Related concepts

Now, we give a comparison between our main concepts and the software engineering terms consecrated to patterns. 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.

Alternative fragments and bad smells in design*:* bad smells in code [Fowler99] are any symptom in the source code of a program that possibly indicates a deeper design problem. Especially more, code smells are heuristics to indicate where to refactor, and what specific refactoring techniques to use. To refactor models is equivalent to refactor programs but at the model level. In the same manner, bad smells in design is the equivalent of bad code smells at model level. We consider that alternative fragments are consolidated design smells. They denote some design problems via the valuation of pattern properties and are substituable via model refactorings by an optimal solution which is the instantiation of the associated design pattern.

Spoiled patterns and anti-patterns*:* there are several definitions of the anti-pattern concept. For [Dodani06], they are descriptions of bad design practices with properties and with a refactoring suite. For [Brown98], they are some repeated patterns of action, process, or structure that initially appear to be beneficial, but produce *in fine* more bad consequences than beneficial results, and a refactored solution that is clearly documented, proven and repeatable. We consider that spoiled patterns are precise design-anti-patterns, follow the first definition, they are slimmer, more precise, detectable, substitutable and directly linked to design patterns. In following the second definition, we do not make hypothesis about their uses in models if not ignorance or oversight about patterns from the designer.

3 A DESIGN REVIEW ACTIVITY DIRECTED BY DESIGN PATTERNS

Our activity is decomposed into three steps. The first step consists in an automatic research of instantiation of spoiled patterns. This research is based on structural concordance with spoiled patterns and determines the model fragments which may be substitutable with a design pattern.

To detect each alternative fragment in a model, we use a detection query that conforms to Figure 7. This query is a set of OCL rules [OMG06] based on remarkable properties. These rules are automatically generated with a plug-in we have developed for the Neptune platform [Neptune03]. This plug-in analyzes the remarkable properties of the spoiled patterns.

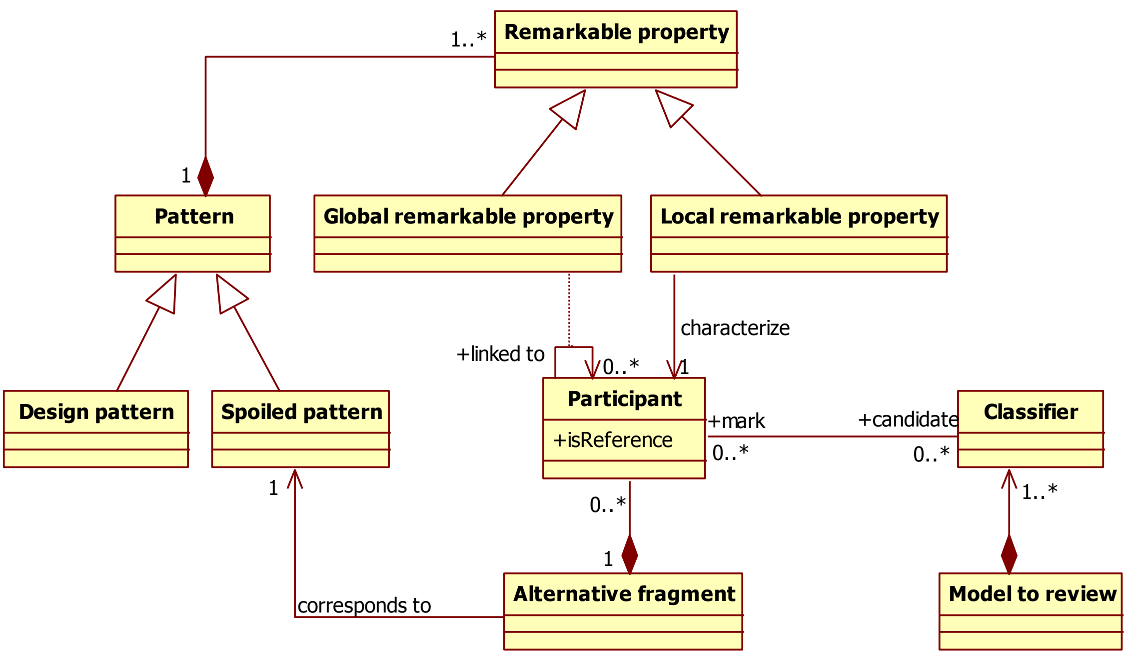


Figure 7: Meta-model of alternative fragment detection

For each UML model, the detection query finds all the possible instantiation of one spoiled pattern. Each time one applies the detection query of one spoiled pattern associated to a given GOF pattern, one retrieves all alternative fragments potentially substitutable with the pattern itself. Therefore, this method is deterministic and the result is complete for any set of spoiled patterns. According to the taxonomy proposed by Chikofsky and Cross [Chikofsky90], our detection technique can be connected to a redocumentation technique so as to permit model restructuring.

Each alternative fragment detected in the model represents propositions of fragments substitutable with a design pattern. The second step consists in the validation of pattern integration propositions. These propositions may be large where some fragments may not be relevant for a substitution. So, to help the designer to filter the fragments, we use an ontology that formalizes the design patterns and the information we have collected on the spoiled patterns. This ontology will help the designer to determine if his intent matches with the intent of the suggested pattern and whether the propositions are needed in the model to review. The intent of a pattern is represented by the set of design pattern problems that the pattern solves. To do so, we have extended an existing OWL [MCGuinness04] ontology [Kampffmeyer07] by addition of our knowledge on spoiled patterns and their relations with each design pattern along with their intents [Harb09]. Connections with the existing ontology are described in Figure 8.

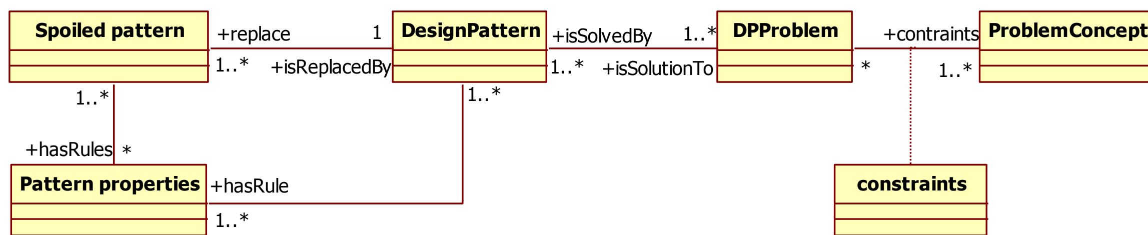


Figure 8: Structure of the extended ontology

If the designer confirms that the intent of the alternative fragment detected is conforms with the suggested design pattern, and if he considers that the transformation is suitable, the last step consists in the integration of the validated propositions into the model. This integration is done thanks to an automatic model refactoring.

The Figure 9 illustrates our activity.

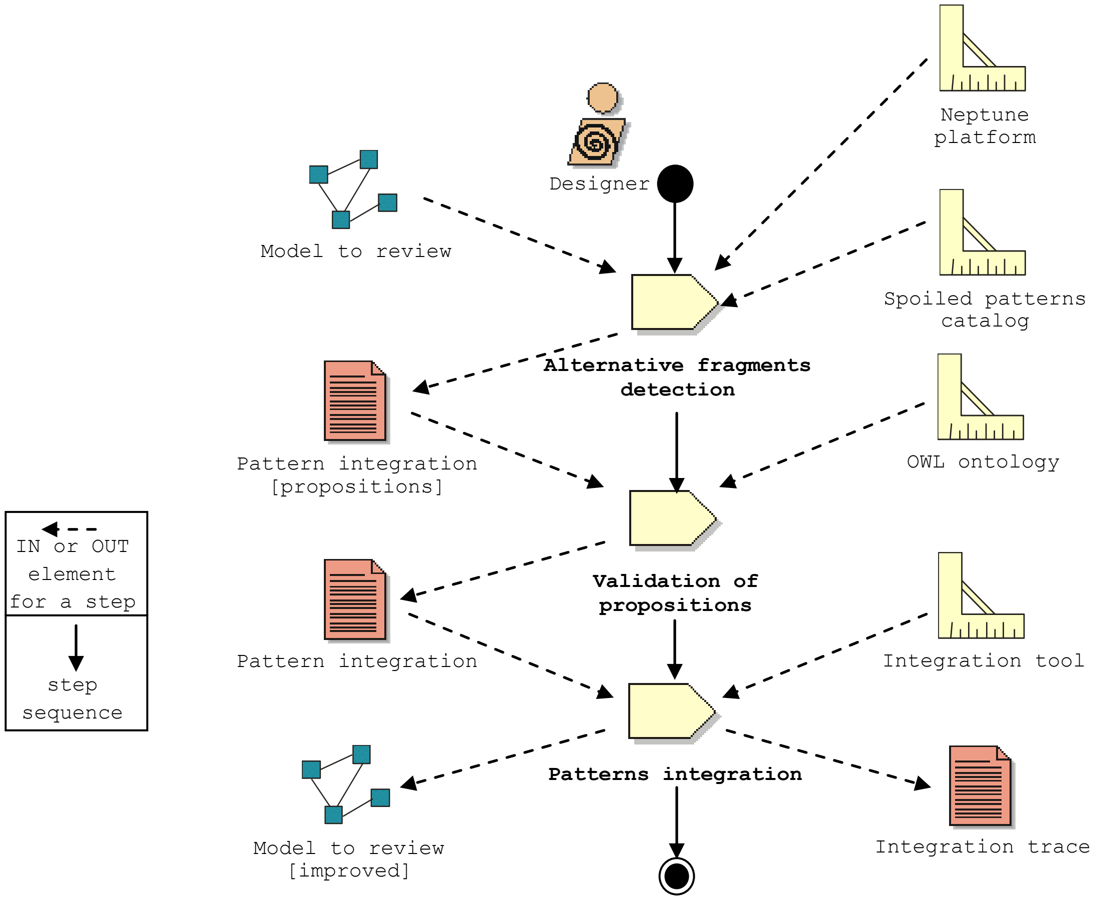


Figure 9: Our design review activity

In reference to the Fagan inspection process, we have named our activity a design review activity. "*Software inspection is a method of static testing to verify that software meets its requirements. It engages the developers and others in a formal process of investigation that usually detects more defects in the product - and at lower coast - than does machine testing*" [Fagan86]. All products of a software development process can be

inspected. An inspection consists of some participants assigned to a specific role (Moderator, Author, and Readers/Testers), and a six-step process (planning, overview, preparation, group inspection, rework, and follow-up). Our activity is executed by the designer of the system to review but can be improved by others participants: other designers and experts on design patterns. The core of the process is guided by a prototype that tools our activity. Planning and overview are general steps. The preparation step consists of a study of the bad practices catalog. The group inspection consists of an analysis followed by answers to each question proposed by our system. The rework step is taken into account by an automatic refactoring. The to-do list is implicitly produced by all occurrences of alternative fragments founded during the process.

4 ILLUSTRATION ON A "FILE SYSTEM MANAGEMENT" DESIGN

In order to illustrate our approach, we execute the whole activity on an example. It was found in a subject of an object-oriented programming supervised practical work. It aims to implement a file management system represented in Figure 10.

The model to review

This static UML model represents a basic architecture for a file system management. Authors of this model are interested in the presentation of some object concepts:

- *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.

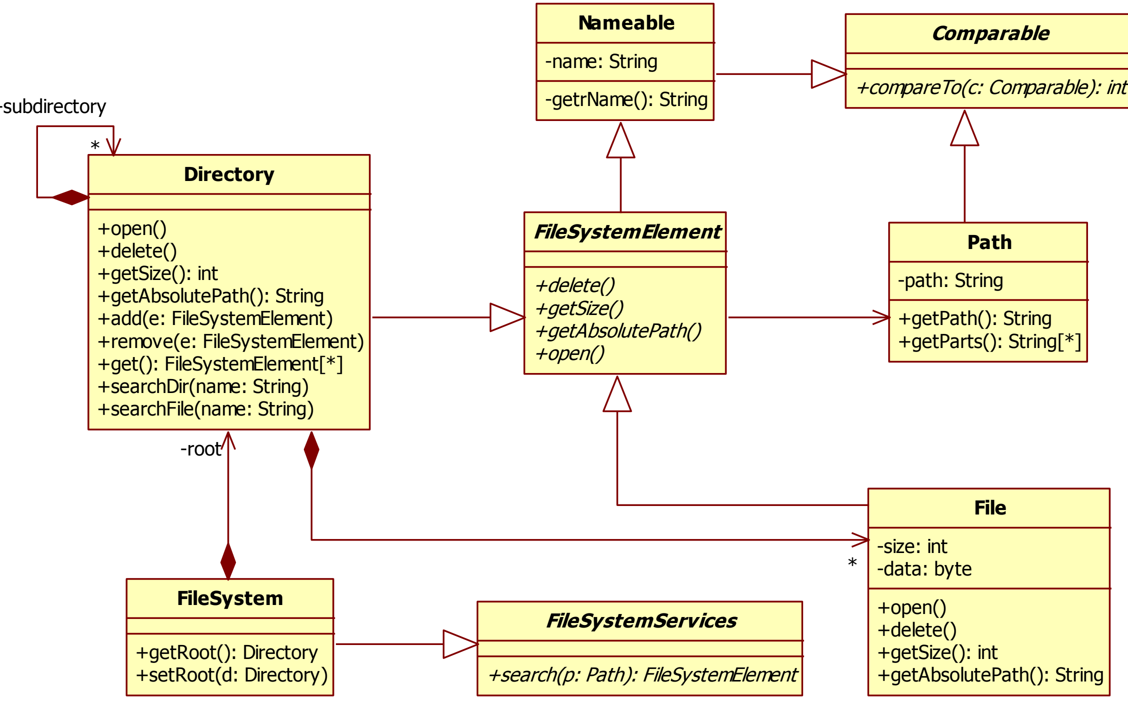


Figure 10: The model to review

Nevertheless, this model contains a misconception. Although there is a uniform protocol owned by the class *FileSystemElement*, the composite links management along a hierarchical structure is duplicated. Indeed, the *Directory* class manages independently links on *Files* and *Directories*. Now, we consider two evolution scenarios.

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.

Alternative fragments detection

This step consists in the execution of all queries which correspond at each spoiled pattern of the catalog. In this example, the query of the *Composite* spoiled model returns these match classes:

1. The *Directory* class is markable with the *Composite* participant.
2. The *File* class is markable with the *Leaf* participant.
3. The *FileSystemElement* is markable with the *Component* participant.

This means that we detected an alternative fragment for the *Composite* pattern because they have the same structural properties as illustrated by Figure 11.

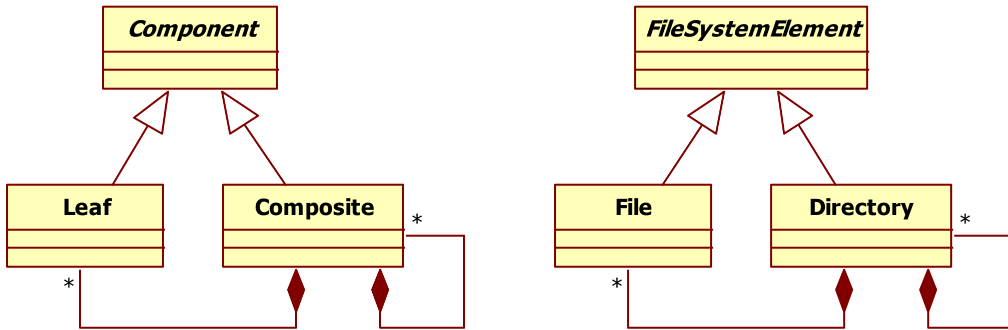


Figure 11: Alternative fragment compared to the *Composite* spoiled pattern

Validation of propositions

At this step, the designer must verify if the detected fragment has the same intent as the considered alternative fragment. If this is the case, we have a bad smell in design. Then the designer must validate or not the substitution of the detected fragment by the instantiation of the design pattern. To do so, we build questions thanks to SPARQL [Prud'hommeaux08] queries we have coded. These questions permit to initiate a dialog with the designer.

The first query retrieves the intent of the design pattern in using the alternative fragment detected and deals with the designer's intent according to recursive composition of *FileSystemElement*, *File,* and *Directory* within the model. The second query retrieves the pattern properties not present in the spoiled pattern and checks the interest to replace the alternative fragment {*FileSystemElement*, *File*, *Directory*} by an instantiation of the *Composite* pattern.

We can ask the designer the first question: "We have detected in your design an alternative fragment of the Composite design pattern. Is the fragment {FileSystemElement, File, Directory} used to compose objects, build a tree structure, and nest objects?"

We can note that the intent of {*FileSystemElement*, *File*, *Directory*} is a recursive composition: "*Directories are composed with Files or Directories which are composed with*...". So the answer to the previous question is positive.

Then we continue the dialog with the designer: "Our analysis shows that you have problems of "decoupling and extensibility"; your model is unable to satisfy these points:

1. Maximal factorization of the composition.
2. Addition or removal of a leaf does not need code modification.
3. Addition or removal of a composite does not need code modification.

In the injection of the Composite design pattern, you will improve all of these points. Do you want to refactor the identified fragment {FileSystemElement, File, Directory}?"

As we consider that the model may evolve, it is useful to guarantee that there are extensibility and decoupling possibilities. Therefore, the fragment must be substituted with the pattern.

Pattern integration

In this step, the identified fragment is replaced by the suggested design pattern as shown in the Figure 12 below.

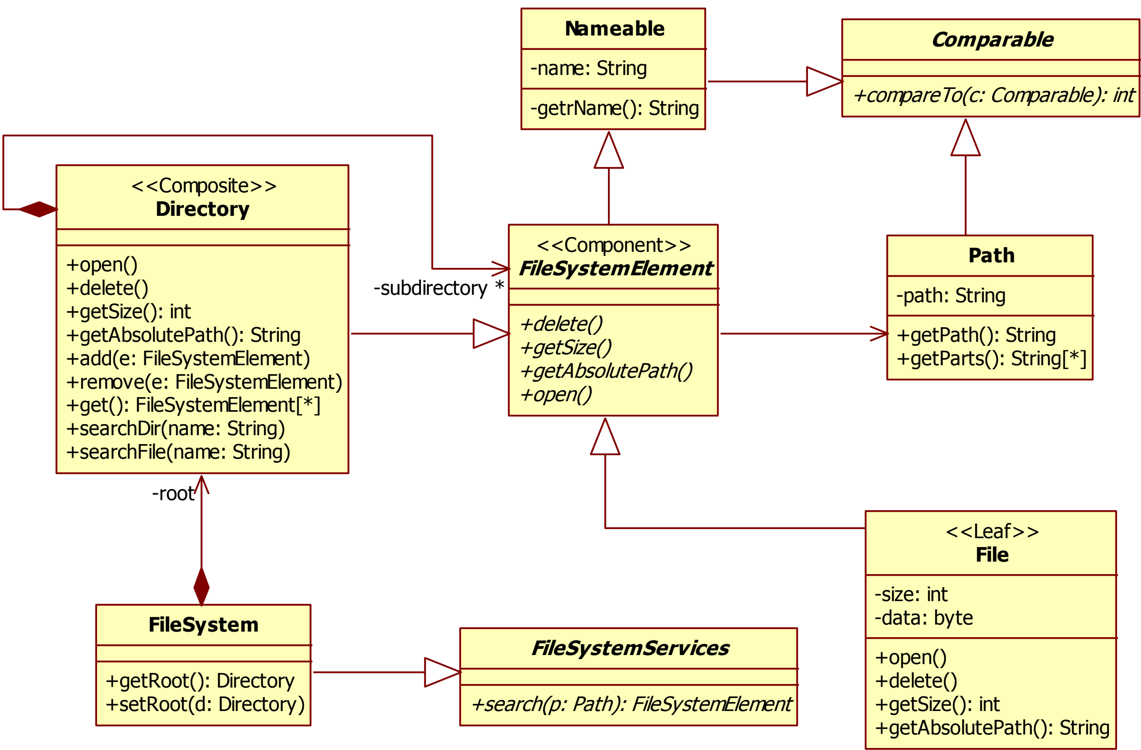


Figure 12: The model to review improved

To do so, a suite of simple model refactoring suffices to integrate the pattern.

Here, it consists of:

1. Remove composition link between *Directory* and *File*.
2. Move the end of the recursive composition link from *Directory* to *FileSystemElement*.

These inter-classes refactorings can automatically be deduced with an operation of differentiation between the alternative model and the pattern structure.

If at a first sight, this transformation may appear as non fundamental in the model, the implications are substantial at the code level. Every hierarchical traversal method is simpler to implement, and there is less code to write. Moreover, in case of extensions, there is no code modification of existing classes. Then, at the design level this transformation brings about substantial benefits. At the end of the activity, we can consider that this model is improved. We are in the process to test all activity on more consistent and valuable models.

5 RELATED WORKS

Currently, a lot of works which concern design patterns exist but at the coding stage. These works concern code generation to integrate patterns into applications and pattern reverse engineering to supply miss-traceability design choices. We are centered in patterns and models driven engineering at the model level. We can identify two topics in pattern engineering. The first concerns the validation of a good integration of a pattern within a model. The second concerns the promotion of the use of patterns in models.

The process of a model transformation through use of a design pattern is called pattern-based refactoring thanks to a "pattern problem specification", a "pattern solution specification" and a "transformation specification". Robert France, et al. [France04], proposes to validate the pattern integration with a conformance relation between specified models to their meta-models. Our design review activity contains a pattern integration, but with less constrained requirements because we modify inter-class relations only.

We experiment Fujaba Tool Suite RE [FUJABA04], which contains a Design-Pattern Recognition component. This tool allows the detection of design patterns in reverse engineered design, thanks to a hierarchical sub-pattern catalog. These sub-patterns are concepts like "isAstractionOf", "redefineMethod", etc., which once associated represent the pattern concept. They specify patterns as graph transformation rules, with respect to the abstract syntax graph of a source code system, and they use bottom-up/top-down analysis. Our spoiled patterns are described on inter-class relations only, and are shown on static class diagrams. We do not need to be as accurate in our description as in their pattern description. In our design activity, we choose to verify the intent correspondence with the designer and we do not need the source code. However, we search model fragments whose inter-classes relations are exactly the same as those of a spoiled pattern. So, our detection is more constrained than a graph homomorphism used in Fujaba.

Sven Wenzel [Wenzel05] uses a fuzzy-like evaluation mechanism so that it is able to recognize not only entire patterns but also incomplete instantiations. He researches all approximate pattern solutions in using proximity percentages. Therefore, we can say that our catalog is an extension definition of the set of approximate patterns. Moreover, as our alternative solutions had been built by human, there is a guarantee that these models are retrievable in a human model.

Kampffmeyer, et al. [Kampffmeyer07] have developed a wizard that enables designers to efficiently find design patterns applicable for their design problems, during the design stage. Thanks to an ontology, they classify some problem concepts that allow to retrieve concerned patterns. Our approach is different especially because we do not impose on the designer to identify design problems.

El-Boussaidi, et al. [ElBoussaidi08] consider that design problems solvable by design patterns are sometimes badly-designed, provoking poor solutions to modeling requirements. To limit that, they propose a semi-automatic tool for marking models using constraint satisfaction techniques. This tool aims at providing a framework for the recognition of design problems solved by design patterns and rewrites them according to the appropriate solutions. Their purpose is close to ours, except that the key element of their approach is the explicit representation of design problems solved by design patterns. This representation limits their detection range to precise design and does not allow to identify all fragments potentially substitutable. Moreover, thanks to our experiments that allow to constitute a spoiled patterns base that is designable by people, the fragments we search have been thought once.

6 CONCLUSION AND PERSPECTIVES

In this paper, we propose to detect bad smells in design and to rework models with adequate design patterns thanks to “spoiled patterns” which are the main concept of our approach. A spoiled pattern can be used according two viewpoints. The first is an architectural or macro-structural view composed by links between participant classes of the spoiled pattern. This view permits us to generate automatically OCL queries that find possible alternative fragments into a model. The second is a knowledge view composed by the intent and a valuation of the defects of the spoiled pattern compared to a corresponding design pattern. This view permits us to validate a fragment as an instantiation of a design pattern and to explain the misconceptions and the advantages of using the design pattern with an ontology base. Moreover, spoiled patterns are obtained by experiments and so designed by people. Thus alternative fragments are more likely to exist in models designed by people.

We have encapsulated spoiled patterns into a design review activity directed by design patterns. For prototyping this activity, we have developed an extension of a pattern oriented ontology and we have implemented a generator of OCL queries using a specific profile that encode structural particularities of spoiled patterns. This generator can be reused for retrieving patterns in a model. Finally, we consider that our catalog of spoiled patterns is a natural extension of the GoF catalog in justifying the choice of the design pattern as the optimal solution for a given problem and in explaining the misconceptions entailed by the associated spoiled patterns.

To keep the tooling of our activity, we work on some transformation rules to obtain an automatic pattern integration. A structural comparison between a spoiled pattern and the corresponding design pattern can be used to obtain a set of structural differences for

each participant. These differences constitute operations once applied on classes to do structural transformations. To extend the field of our approach, we propose to adapt remarkable properties on spoiled pattern to behavioral properties.

ACKNOLEDGEMENT

We are grateful to Dr. Ralph Sobek for his precious comments during this paper reading, and to Dr. Guillaume Cabanac for his precious example.

REFERENCES

[Bouhours07] Bouhours C., Leblanc H., Percebois C., "Alternative Models for Structural Design Patterns", research report, IRIT/RR--2007-1--FR, IRIT, december 2007, [w] http://www.irit.fr/recherches/DCL/MACAO/docs/AlternativeModelsForStructuralDesignPatterns.pdf.

[Brown98] Brown William J., Malveau Raphael C., Mowbray Thomas J. "AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis", 1998.

[Buschmann96] Buschmann F., Meunier R., Rohnert H., Sommerlad P., Stal M., "Pattern-Oriented Software Architecture", John Wiley & Sons, August 1996.

[Chikofsky90] Chikofsky E. J., Cross J. H., "Reverse engineering and design recovery: A taxonomy", in *IEEE Software*, 7(1), page 13-17, January 1990.

[Diaz87] Diaz P., Classification of reusable modules, IEEE Software 4(1), pp.6-16, 1987.

[Dodani06] Dodani M.H., "Patterns of Anti-Patterns", in *Journal of Object Technology*, vol. 5, no. 6, July-August 2006, pp. 29-33

[ElBoussaidi08] El-Boussaisi G., Hafedh M., "Detecting Patterns of Poor Design Solutions Using Constraint Propagation", in *MoDELS, Springer*, 2008 , volume 5301, pages 189-203.

[Fagan02] Fagan M., "Design and code inspections to reduce errors in program development", Springer-Verlag, New York, Inc., 2002, pages 575-607.

[Fowler97] Fowler M., "Analysis patterns: reusable objects models", Addison Wesley Longman Publishing Co, Inc., 1997.

[Fowler99] Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D. "Refactoring: Improving the Design of Existing Code", Addison-Wesley Professional, 1999.

[France04] France R. B., Kim D., Ghosh S., Song E., "A UML-Based Pattern Specification Technique", in *TSE,* IEEE Press, 2004, 30, 193-206.

[FUJABA04] FUJABA, From UML to Java and Back Again, [w] http://wwwcs.uni-paderborn.de/cs/fujaba/projects/reengineering/index.html

[Gamma95] Gamma E., Helm R., Johnson R., Vlissides J., "Design Patterns: Elements of Reusable Object-Oriented Software", Addison Wesley Professional, 1995.

[Harb09] Harb D., Bouhours C., Leblanc H.. "Using an ontology to suggest software design patterns integration". in *First international workshop on TWOMDE*, Toulouse, France, Oct. 2008. **Selected as one of best papers of the workshop**. Workshops and Symposia at MoDELS 2008, M.R.V. Chaudron ed., LNCS, vol. 5421, Springer, 2009.

[Hasso04] Hasso S., Carlson C. R., Linguistics-based Software Design Patterns Classification. In *HICCS*, Honolulu, HI, Jan 2004.

[Kampffmeyer07] Kampffmeyer H., Zschaler S., "Finding the Pattern You Need: The Design Pattern Intent Ontology", in *MoDELS*, Springer, 2007, volume 4735, pages 211-225.

[MCGuinness04] McGuinness D.L. and Van Harmelen F., OWL Web Ontology Language Overview, 2004.http://www.w3c.org/TR/owl-features/

[Neptune03] NEPTUNE, Nice Environment with a Process and Tools using Norms - UML, XML and XMI - and Example, [w] http://neptune.irit.fr, 2003

[OMG06] Object Management Group., "Object Constraint Language", http://www.omg.org/cgi-bin/apps/doc?formal/06-05-01.pdf, 2006.

[Prud'hommeaux08] Prud'hommeaux E., Seaborne: SPARQL Query Language for RDF, January 2008. http://www.w3.org/TR/rdf-sparql-query/

[Sunyé01] Sunyé G., Pollet D., Traon Y. L., Jézéquel J., Refactoring UML models in *UML*, Springer, 2001, 134-148.

[Wenzel05] Wenzel S., "Detection of Incomplete Patterns Using FUJABA Principles", in *3rd International Fujaba Days : MDD in practice*, 2005.

About the authors

Cédric BOUHOURS is a PhD student in computer science since 2006 at the University of Toulouse, France. He works on the reuse of the expert knowledge in MDE processes models. He conceives and tools a design review activity that allows the verification if design patterns were forgotten by the designers of a model, and to automatically correct this situation, with the designers approval. His research interest includes best practices, development processes and methods, design patterns, and refactoring. He can be reached at bouhours@irit.fr.

Hervé LEBLANC is associate professor of computer science at the University of Toulouse since 2002. His research interest includes Object Oriented language and design. He worked on automatic restructuring class and interface hierarchies thanks to the Galois lattice. He received a PhD in computer science from University of Montpellier II. His main research interests are now linked to patterns, refactoring, agile methods and traceability in MDE processes. He can be reached at leblanc@irit.fr.

Christian PERCEBOIS is professor of computer science at the University of Toulouse since 1992. 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. His main research interests are now linked to design methodologies and object-oriented modeling approaches. He can be reached at percebois@irit.fr