

éfaut de patron2.113

# Catalog of Design Defects

version 1.0

Naouel Moha

GEODES Laboratory  
DIRO - University of Montreal  
[mohanaou@iro.umontreal.ca](mailto:mohanaou@iro.umontreal.ca)

October 31, 2005

This catalog aims to provide a specification of some design defects. By design defects, we mean the two following categories of defects : design pattern defects [MOH 05] and antipatterns [BRO 98]. Code smells are not considered as design defects but we include them in this catalog because they represent indicators or symptoms of the presence of antipatterns.

Design pattern defects are the distorted or degraded shapes of design patterns. We distinguish two types of defects related to design patterns, or design pattern defects : (1) distorted patterns and (2) degraded patterns. We define distorted patterns as micro-architectures similar but non-equal to these proposed by design patterns [GUÉ 01]. These patterns, following a design choice, have been altered in their structure and their class organization in order to response to specific constraints required by the environment. The degraded patterns are in the contrary bad design choices.

An antipattern is a literary form that describes a common solution to a design pattern that rather leads to negative effects on quality [BRO 98]. Antipatterns are bad solutions to recurring design problems. The idea behind antipatterns is to show what not to do. Antipatterns have been described in different domains : J2EE [DUD 03 ; TAT 02], XML [TAT 02], multi-threaded applications [BOR 05], performance [SMI 02] and in general in [BRO 98].

Code smells are not considered as design defects because they are situated in the implementation level. However, we have noticed that code smells are good indicators of the presence of antipatterns. A code smell is a term that refers to a symptom or a problem at the code level. Beck and Fowler describe code smells as “*certain structures in code that suggest the possibility of refactoring*” [FOW 99]. Refactoring is defined as “*a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior*” [FOW 99]. Duplicated code, long method, large class, long parameter list, data class are some examples of code smells. The presence of code smells suggests the possible presence of an antipattern as well as of a design defect. Code smells are generally related to the inner workings of classes while design defects include the relationships among classes and are more situated on a micro-architectural level.

As a specification, we provide the model and the rule card of the design defects Naouel ► *A prciser* ◀ .

A rule card describes a set of rules and compositions or associations of rules. These rules are defined by a set of internal attributes that specify the key concepts with eventually threshold levels (high, medium, low).

Note that it is only the first version of our catalog : this one is continually completed and enhanced. Moreover, the description given in this catalog come from the literature [BRO 98 ; DUD 03 ; FOW 99]. For more information, please refer to the literature.

We provide the specification of the antipatterns in section 1, of the design pattern defects in section 2, and of the code smells in section 3.

# Contents

<b>1</b>	<b>Antipatterns</b>	<b>3</b>
1.1	Blob . . . . .	3
1.2	LavaFlow . . . . .	5
1.3	Poltergeists . . . . .	6
1.4	Spaghetti Code . . . . .	7
1.5	Functional Decomposition . . . . .	8
1.6	Cut-And-Paste Programming . . . . .	9
1.7	Swiss Army Knife . . . . .	10
<b>2</b>	<b>Design Pattern Defects</b>	<b>13</b>
2.1	Observer pattern . . . . .	13
<b>3</b>	<b>Code smells</b>	<b>14</b>
3.1	Shotgun Surgery . . . . .	14
3.2	Divergent Change . . . . .	14
<b>4</b>	<b>J2EE antipatterns</b>	<b>16</b>
4.1	Multiservice . . . . .	16
4.2	Stovepipe . . . . .	17

# Chapter 1

## Antipatterns

### 1.1 Blob

The Blob [BRO 98] (called also God class [RIE 96]) corresponds to a large complex class that monopolizes all the treatment realized in a program and that is surrounded by simple data classes (*cf.* Figure 1.1). The large class is made up of fields and methods where the cohesion between these entities are relatively low. It is also characterized by some death code. An example of Blob is the interface user classes.

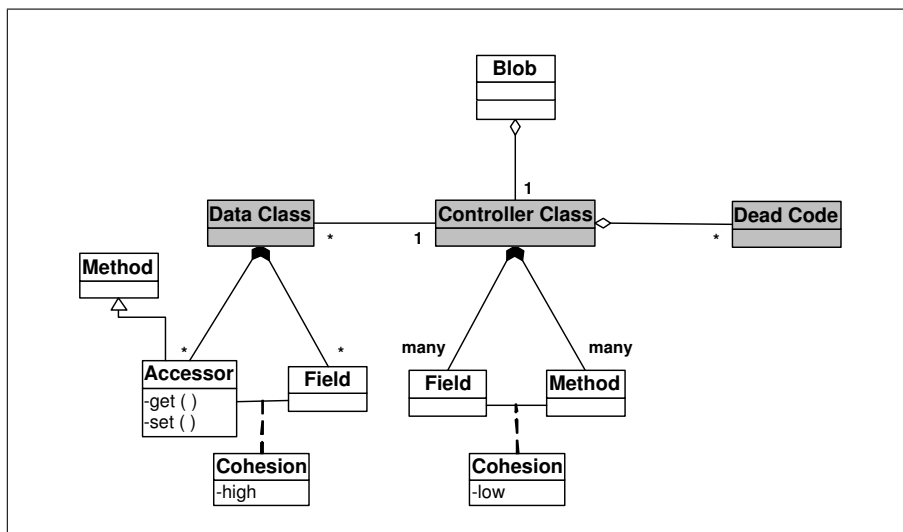


Figure 1.1: The model of Blob

The Blob is characterized by a controller class. A controller makes decisions and closely directs others' actions [WIR 02]. In the case of the Blob, the controller is dependent upon information holders' contents i.e. data classes.

The figure 1.2 gives the rule card of the Blob.

<u>Rule Card of the Blob</u>			
<b>Rule for «Blob» role:</b>			
<b>Aggregation:</b>	one	«ControllerClass»	
<b>Rule for « ControllerClass » role:</b>			
<b>Association:</b>	«ControllerClass»	to many	«DataClass»
<b>Agregation:</b>	many	Dead_Code	
<b>ClassName:</b>	{	System; Subsystem; Manager; Driver; Controller	}
<b>Complexity:</b>	Number of methods		High
	Number of fields		High
<b>Cohesion:</b>	between methods and fields		Low
<b>Rule for «DataClass»:</b>			
<b>Methods:</b>	Accessors		
<b>Cohesion:</b>	between methods and fields		High

Figure 1.2: Rule Card of the Blob

## 1.2 LavaFlow

The Lava Flow [BRO 98] is a controller class that includes death code (or at least unrelated code). This term has been used to refer to the fluid nature of the lava flow in the beginning and that over the years turn into basalt, hard material difficult to remove once solidified by comparison with not well-documented code that no one can remember much and that we cannot dare to remove.

The Lava-Flows are characterized sometimes by a lack of comment and if there are comments they precise the keywords: InFlux, TODO, or to be replaced. To resume, in the Lava Flow, we have a lot of incomplete, unused or unjustified code.

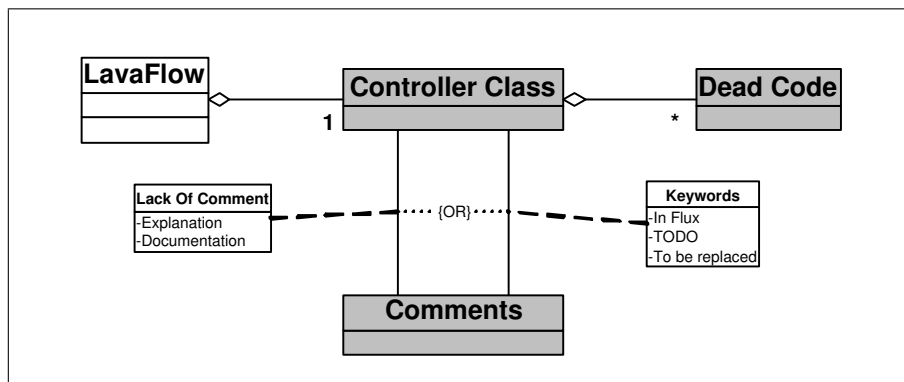


Figure 1.3: The model of Lava Flow

The figure 1.4 gives the rule card of the Lava Flow.

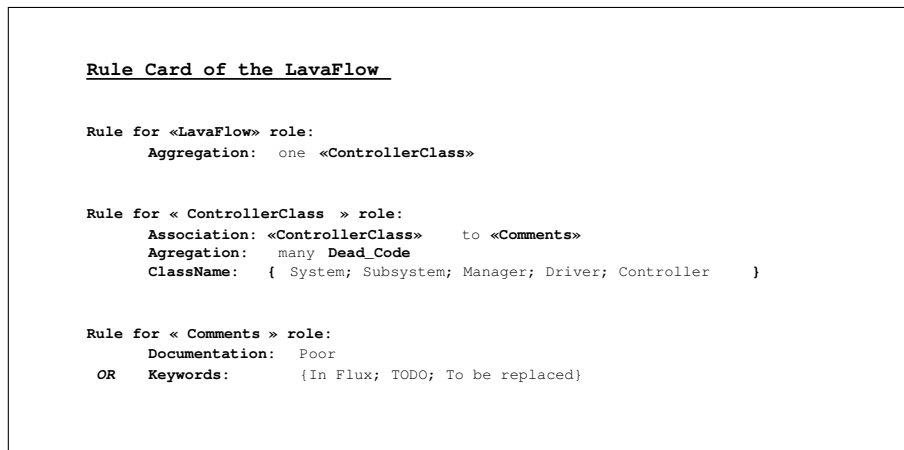


Figure 1.4: Rule Card of the Lava Flow

## 1.3 Poltergeists

The Poltergeist appear and disappear all along the execution of a program. Usually, Poltergeist are classes that launch the execution of threads or process. So they have very brief life-cycles.

The symptoms of the presence of the Poltergeist are the following :

- Unnecessary and redundant navigation paths;
- Highly transient associations of a particular class with another one;
- Presence of stateless classes;
- Occurrence of temporary and short duration objects/classes;
- Classes that exist only to invoke other classes through temporary associations. These classes have usually “control-like” operation names such as start\_process.

Poltergeists are one of the most unwanted antipatterns because they consume unnecessary resources every time they appear, they waste useful energy by traversing redundant navigation paths and they clutter object models introducing unnecessary confusion in the course of system analysis.

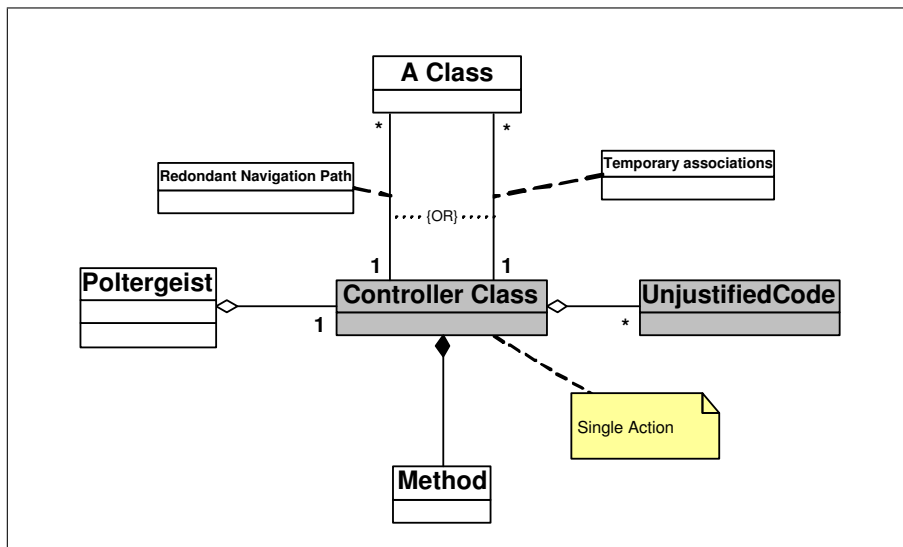


Figure 1.5: The model of Poltergeist

The figure 1.6 gives the rule card of the Poltergeist.

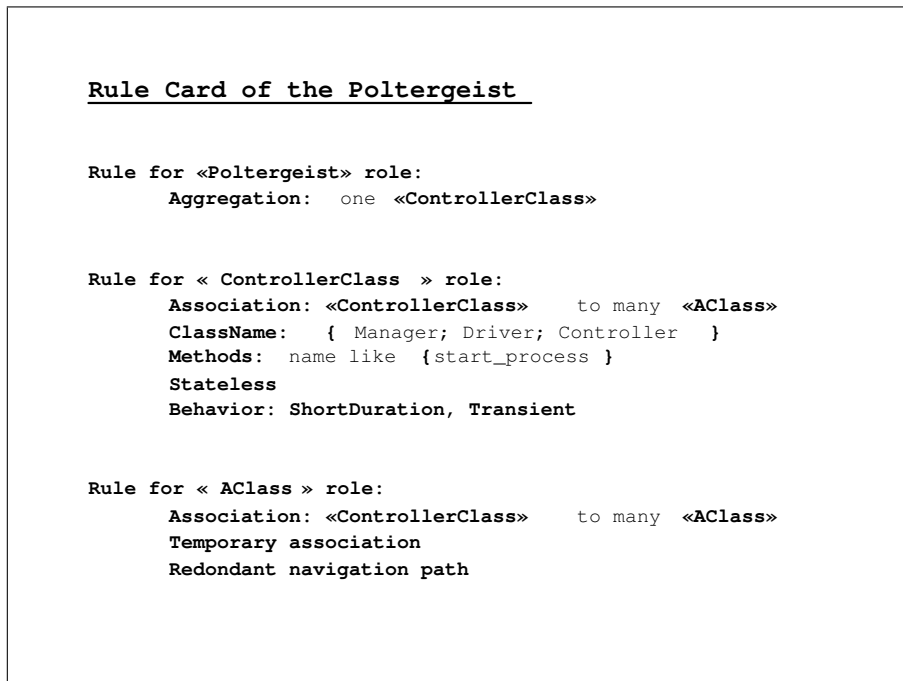


Figure 1.6: Rule Card of the Poltergeist

## 1.4 Spaghetti Code

Spaghetti code is an antipattern that defines a program or a system that lacks of structure. It is hard to reuse the objects and modules because of the lack of clarity and the negligible degree of dynamic interaction between objects [BRO 98].

The symptoms of the presence of Spaghetti code antipattern are the following :

- Inheritance and polymorphism are unused;
- No reuse of the program or system;
- Minimal relationship exist between objects;
- Composed of long methods process oriented with no parameters and low cohesion;
- These methods utilize class and global variables for processing.

The figure ?? gives the rule card of the Spaghetti Code.

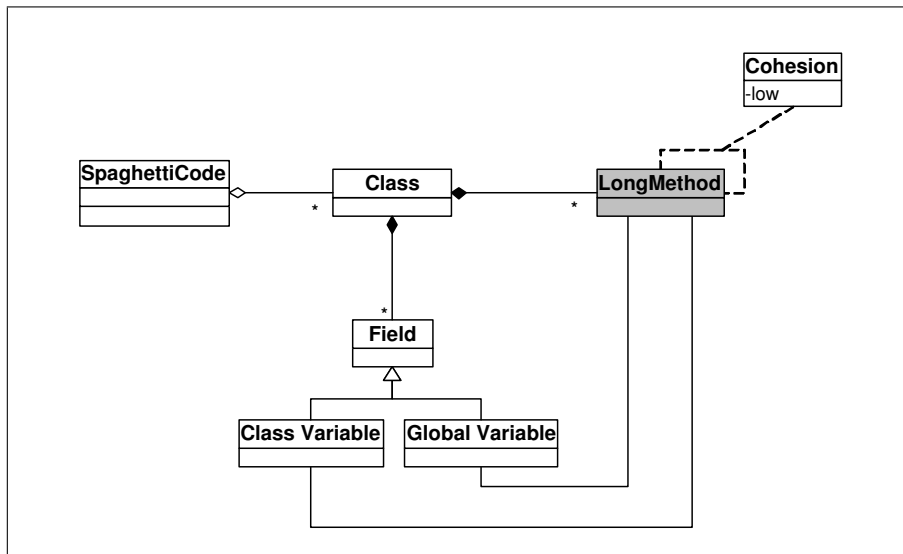


Figure 1.7: The model of Spaghetti Code

## 1.5 Functional Decomposition

This antipattern occurs when experienced developers with a little knowledge of object-oriented concepts implement an application in an object-oriented language. Browns describes this antipattern as “*When developers are comfortable with a “main” routine that calls numerous subroutines, they may tend to make every subroutine a class, ignoring class hierarchy altogether (and pretty much ignoring object orientation entirely). The resulting code resembles a structural language such as Pascal or FORTRAN in class structure.*”

The symptoms of the presence of Functional decomposition antipattern are the following :

- Classes with “function” names such as *Calculate*, *Display*;
- Inheritance and polymorphism are unused;
- No reuse of the program or system;
- All class attributes are private and used only inside the class;
- Poor documentation on how the system works.

The figure 1.9 gives the rule card of the Functional Decomposition.

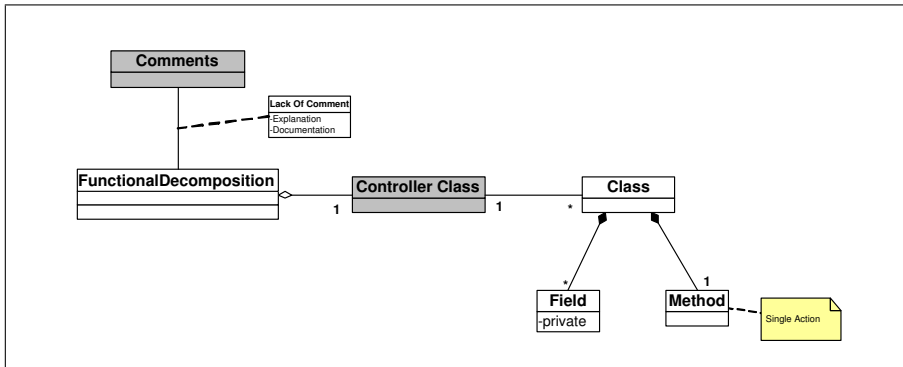


Figure 1.8: The model of Functional Decomposition

## 1.6 Cut-And-Paste Programming

This antipattern is due to the reuse of code with a minimum of effort. The programmer tends to modify, reuse, and customize code that has been proven to work in similar situations. This leads to several similar segments of code along the program i.e. duplicated code.

The figure 1.9 gives the rule card of the Functional Decomposition.

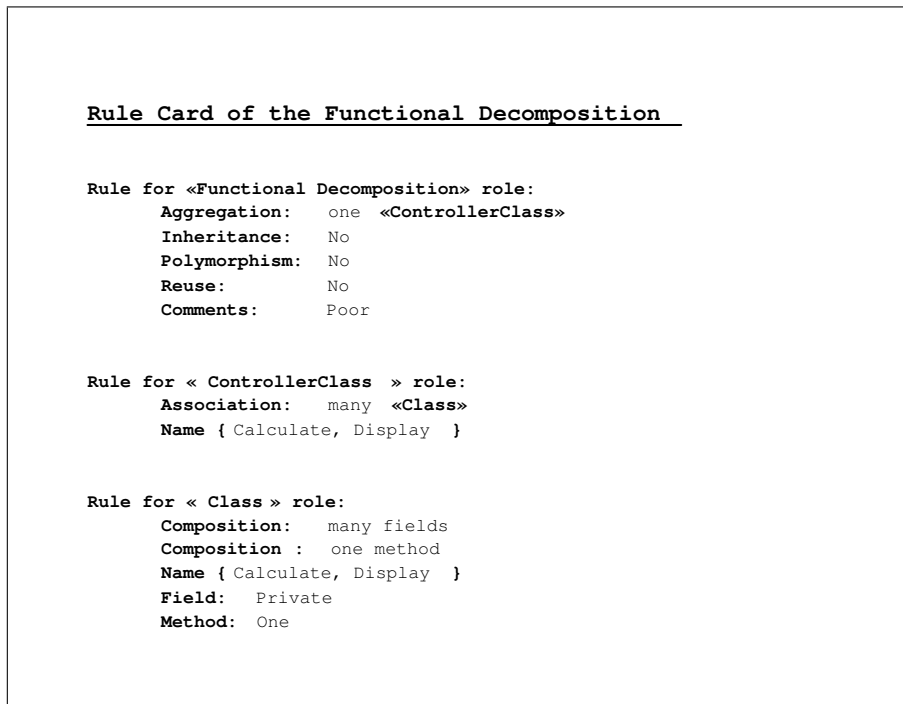


Figure 1.9: Rule Card of the Functional Decomposition

## 1.7 Swiss Army Knife

The Swiss army knife<sup>1</sup> [BRO 98] is a complex interface class that includes a high number of signatures in order to provide all possible uses of the class (*cf.* figure ??).

A Swiss army knife differs from the Blob in the sense that the Swiss army knife exposes a high complexity in order to address all foreseeable needs of the class, whereas the Blob is a singleton object that monopolizes all the treatment and the system data. So there can be several Swiss army knives in a design [BRO 98]. An exemple that illustrates the swiss army knife could be the utility classes in the object programs.

The symptoms of the presence of Swiss Army Knife antipattern are the following :

- Complex interfaces;
- No clear abstraction or purpose for the class, which is represented by the lack of focus in the interface.

The figure 1.13 gives the rule card of the Swiss Army Knife.

---

<sup>1</sup>This term swiss army knife is used to make allusion to the set of little tools provided to response to any kind of needs.

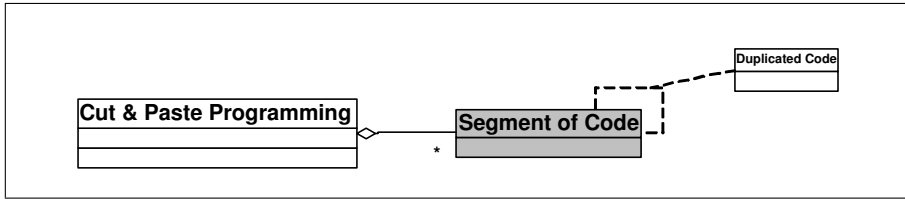


Figure 1.10: The model of Cut-And-Paste Programming

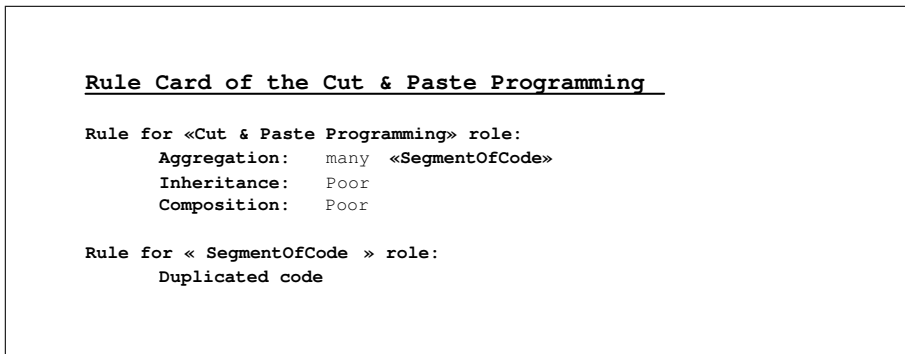


Figure 1.11: Rule Card of the Cut-And-Paste Programming

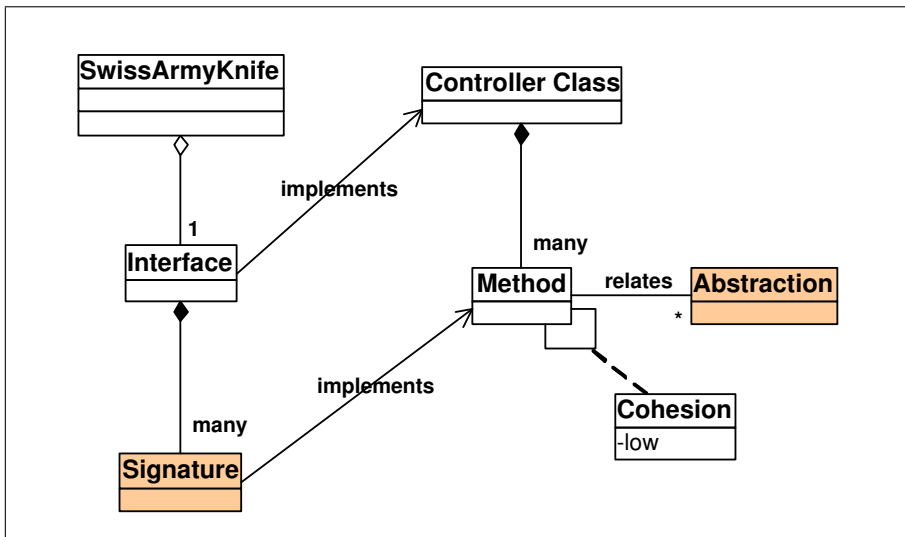


Figure 1.12: The model of Swiss Army Knife

### Rule Card of the Swiss Army Knife

Rule for « SwissArmyKnife » role:

**Aggregation:** one «Interface »

Rule for «Interface» role:

**Complexity:** Number of signatures High

**Implements:** «ControllerClass»

Rule for «ControllerClass»:

**Complexity:** Number of methods High

**Cohesion:** between methods Low

**Association:** methods with to many **Abstraction**

Figure 1.13: Rule Card of the Swiss Army Knife

## Chapter 2

# Design Pattern Defects

Design pattern defects are represented as the design patterns but with constraints on relations between entities and elements.

### 2.1 Observer pattern

One defect of the Observer pattern can be a fusion between the `ConcreteSubject` and the `Subject`.

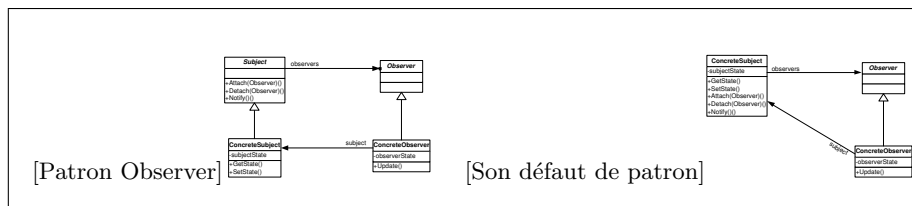


Figure 2.1: Le patron Observer et son défaut de patron

## Chapter 3

# Code smells

As the antipatterns and the design pattern defects, we formalize complex code smells as the “Shotgun Surgery” and the “Divergent Change” (*cf.* figure ??).

### 3.1 Shotgun Surgery

The “Shotgun Surgery” describes a class that a change affect other classes, specially when the coupling between classes is strong.

### 3.2 Divergent Change

A “Divergent Change” appears when a class is modified in different manners for different reasons [FOW 99]. For example, the migration from one database to another involves the modification of a class in multiple places.

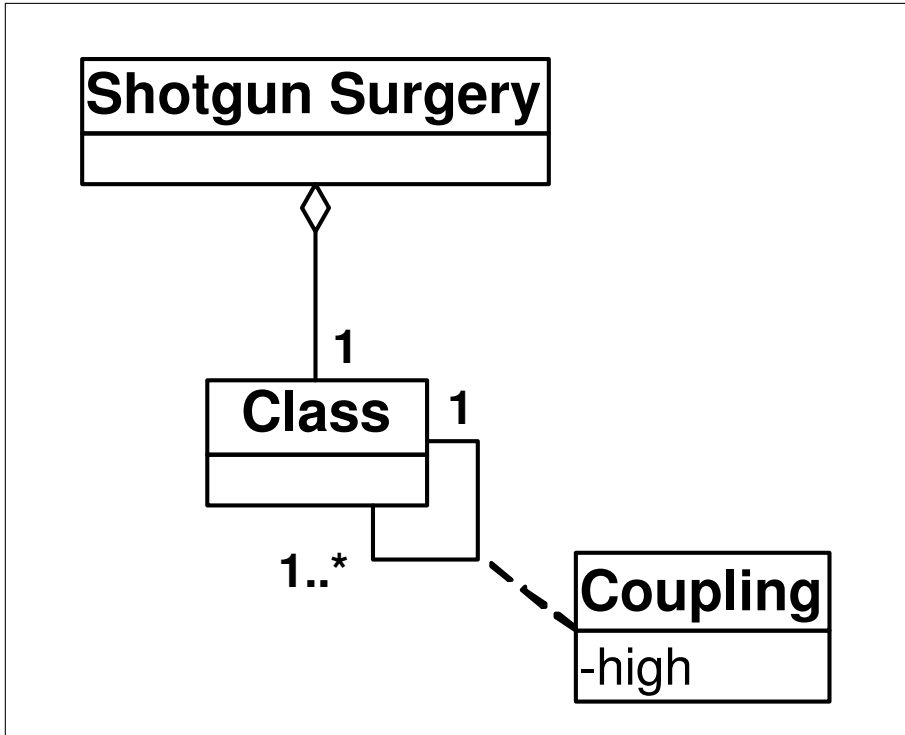


Figure 3.1: The model of the Shotgun Surgery

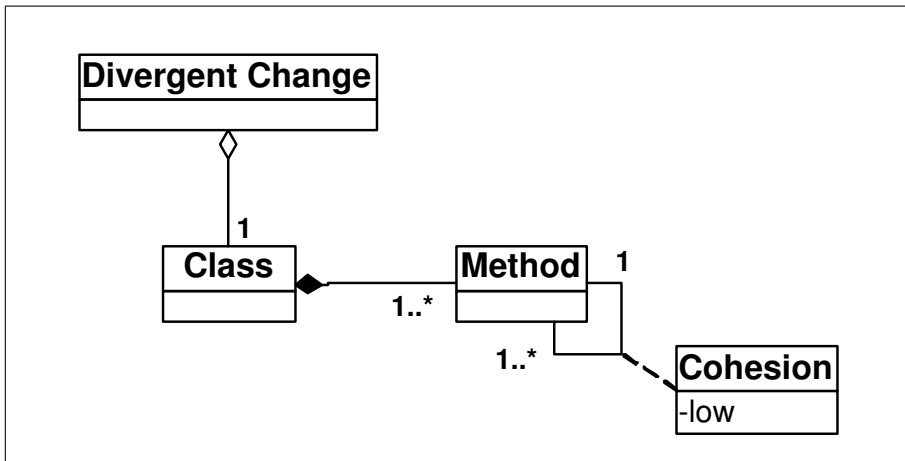


Figure 3.2: The model of the Divergent Change

# Chapter 4

## J2EE antipatterns

The J2EE antipatterns are characterized by the notion of web services. A web service is a software entity, providing one or several functionalities, that can be exposed, discovered and accessed over the network. All the descriptions of the antipatterns are extracted from the book [DUD 03].

### 4.1 Multiservice

A service that does too much, implementing multiple business or technical abstractions. This aggregates too much into a single service, reducing usability.

The antipattern “Multiservice” is very similar to the antipattern “SwissArmyKnife” (*cf.* 1.7).

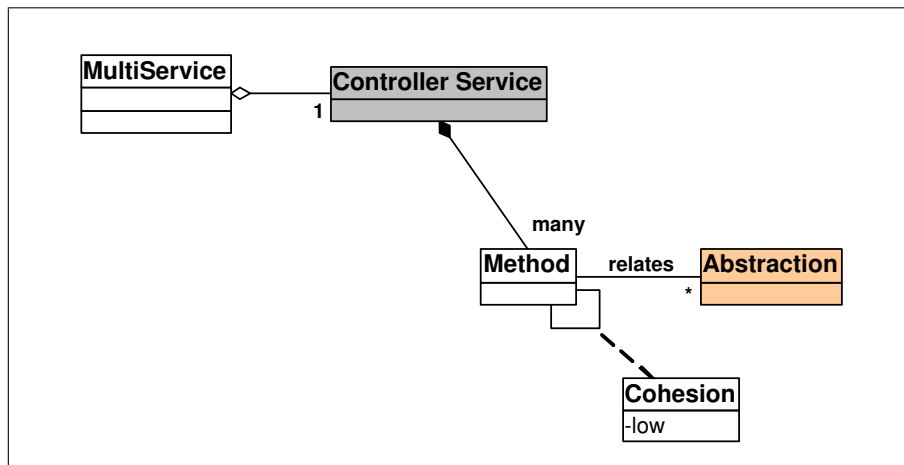


Figure 4.1: The model of the Multiservice

## 4.2 Tinyservice

A service that only implements part of an abstraction, with other sibling services implementing other processes within abstraction. This requires several, coupled services to be used together, resulting in more development complexity and reduced usability.

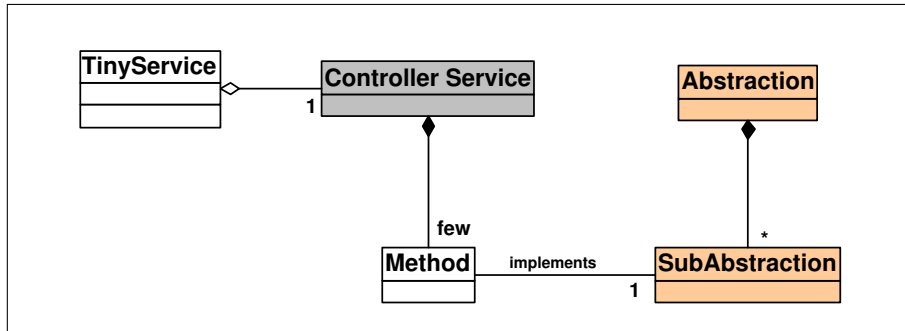


Figure 4.2: The model of the Tinyservice

## 4.3 Stovepippe

The antipattern “Stovepippe” is very similar to the antipattern “Cut & Paste programming” (*cf.* 1.6).

# Bibliography

- [BOR 05] BORODAY S., PETRENKO A., SINGH J.HALLAL H., Dynamic analysis of java applications for multithreaded antipatterns, *WODA '05: Proceedings of the third international workshop on Dynamic analysis*, New York, NY, USA, 2005, ACM Press, 1-7.
- [BRO 98] BROWN W. J., MALVEAU R. C., BROWN W. H., III H. W. M.MOWBRAY T. J., *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*, John Wiley and Sons, 1<sup>st</sup>, March 1998.
- [DUD 03] DUDNEY B., ASBURY S., KROZAK J.WITTKOPF K., *J2EE AntiPatterns*, Wiley, 2003.
- [FOW 99] FOWLER M., *Refactoring – Improving the Design of Existing Code*, Addison-Wesley, 1<sup>st</sup>, June 1999.
- [GUÉ 01] GUÉHÉNEUC Y.-G.ALBIN-AMIOT H., Using Design Patterns and Constraints to Automate the Detection and Correction of Inter-Class Design Defects, LI Q., RIEHLE R., POUR G.MEYER B., , *proceedings of the 39<sup>th</sup> conference on the Technology of Object-Oriented Languages and Systems*, IEEE Computer Society Press, July 2001, 296–305.
- [MOH 05] MOHA N., HUYNH D.-L.GUÉHÉNEUC Y.-G., A Taxonomy and a First Study of Design Pattern Defects, ANTONIOL G.GUÉHÉNEUC Y.-G., , *Proceedings of the STEP International Workshop on Design Pattern Theory and Practice (IWDPTP05)*, September 2005.
- [RIE 96] RIEL A. J., *Object-Oriented Design Heuristics*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.
- [SMI 02] SMITH C. U.WILLIAMS L. G., *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*, Addison-Wesley Professional, Boston, MA, USA, 2002.
- [TAT 02] TATE B. A.FLOWERS B. R., *Bitter Java*, Manning Publications, 2002.
- [WIR 02] WIRFS-BROCK R.McKEAN A., *Object Design: Roles, Responsibilities and Collaborations*, Addison-Wesley Professional, 2002.