A Metric-Based Approach for Anti-pattern Detection in UML Designs

Rahma Fourati, Nadia Bouassida, and Hanêne Ben Abdallah

Abstract. Anti-patterns are poor solutions of recurring design problems, which decrease software quality. Numerous anti-patterns have been outlined in the literature as violations of various quality rules. Most of these anti-patterns have been defined in terms of code quality metrics. However, identifying anti-patterns at the design level would improve considerably the code quality and substantially reduce the cost of correcting their effects during the coding and maintenance phases. Within this context, we propose an approach that identifies anti-patterns in UML designs through the use of existing and newly defined quality metrics. Operating at the design level, our approach examines structural and behavioral information through the class and sequence diagrams. It is illustrated through five, well-known anti-patterns: Blob, Lava Flow, Functional Decomposition, Poltergeists, and Swiss Army Knife.

1 Introduction

Design patterns [11] propose "good" solutions to recurring design problems. To benefit from design patterns, a developer must have a thorough understanding of and a good practice with design patterns in order to identify the appropriate patterns to instantiate for his/her application. The absence of such high-level of expertise often results in *anti-patterns*. In fact, anti-patterns [19] describe commonly occurring solutions to problems but generate negative consequences on the quality of object-oriented software in terms of quality factor (so called in the ISO 9126 norm) as complexity, reusability. Evidently, anti-patterns detection and correction improves substantially the software quality. This benefit motivated several researchers to propose assistance for inexperienced designers through the detection of anti-patterns, *cf.*, [13] [14], [7], [4] [15] and [2]. Existing propositions differ mainly in: *i)* the level they consider (code *vs.* design level); and *ii)* their objectives

Rahma Fourati · Nadia Bouassida · Hanêne Ben Abdallah
Miracl Laboratory, University of Sfax, Tunisia
e-mail: rahma.fourati10@gmail.com, nadia.bouassida@isimsf.rnu.tn, hanene.benabdallah@fsegs.rnu.tn

R. Lee (Ed.): Computer and Information Science 2011, SCI 364, pp. 17–33.
springerlink.com

(code improvement through re-engineering for maintenance purposes, *cf.*, [2], [15], [4], [13] and [14] *vs.* design improvement, *cf.*, [17], and [3]).

In reality, the code is rich with information useful for the detection of anti-patterns, *e.g.,* variable instances used by a method, number of lines in the code, comments and global variable used. However, the detection of anti-patterns at the code level is considered too late and may not reduce the correction cost. Hence, in our work, we are interested in anti-pattern detection at the design level. During the design phase, a designer (inexperienced with design patterns) may inadvertently specify a design fragment that "resembles" a design pattern. The resemblance can be manifested either as an anti pattern, or as a "poor/bad" design solution. Adopting our anti-pattern detection approach at this phase helps the designer in improving the quality of his/her design, henceforth the code.

In this paper, we propose a metric-based approach for anti-patterns detection in UML designs, *i.e.,* the class and sequence diagrams. In our detection approach, we have selected a set of most pertinent metrics from the works of [16] and [9]. To these metrics, we added a set of specific metrics useful for anti-patterns detection.

The remainder of this paper is organized as follows: Section 2 overviews currently proposed approaches for anti-pattern identification. Section 3 first presents the OO software metrics used and, secondly, introduces our approach. Section 4 illustrates our approach through an example of a Functional Decomposition anti-pattern [19]. Section 5 summarizes the paper and outlines our future work.

2 Related Works

Several works have tackled software (code) quality through various techniques. Amongst these works, design patterns and heuristics have been considered as the most promising approaches. In fact, Gamma [11] introduced *design patterns* as "good" solutions that can be instantiated and composed to produce software faster; being agreed up-on solutions, design patterns guarantee the good-quality of the produced software. In addition, Riel [1] presented more than sixty guidelines as object oriented *heuristics* to evaluate manually existing software and to improve its quality. On one hand, design patterns and heuristics are considered as good design practices. On the other hand, *anti-patterns* and *bad smells* are considered as results of bad design practices. Bad smells are code-level symptoms indicating the possible presence of an anti-pattern (also called '*Design Flaw'* [12]). Anti-patterns and bad smells are sometimes merged into one term: *design defects* [13] [14]. However, bad smells are fine-grained and strongly linked to the code level; on the other hand, anti-patterns are coarse-grained and can be represented at the design level.

Since we are treating quality at the design level, in this paper, we are interesting in detecting only anti-patterns. We next overview the definitions of anti-patterns and then discuss current identification approaches.

2.1 Anti-pattern Definition

Anti-patterns are *bad solutions* to recurring design problems. For example, the Blob, also called God class [1], corresponds to one single complex controller class that monopolizes the processing and that is surrounded by simple data classes [19]. Brown [19] defines the relation between patterns and anti-patterns. He indicates that, patterns are applicable in a well-known context, while anti-patterns are structures that appear to be beneficial, but produce more bad consequences than good ones. Furthermore, Dodani [11] added that developing patterns is a bottom-up process, whereas developing anti-patterns is a top-down process. He insisted that we should learn from our 'mistakes' which are anti-patterns.

On the other hand, while design patterns are well-described/documented thanks to class diagrams [8], anti-patterns presented in the literature are only informally described in natural language. This hinders the development of CASE tools to assist in their detection. Nevertheless, several detection approaches have been proposed. We next discuss the most complete approaches.

2.2 Anti-patterns Detection Approaches

Current approaches for the identification of anti-patterns operate either at the code level (for software re-engineering purposes), or at the design level (for design quality improvement purposes).

2.2.1 Code Level

Marinescu [15] [4] proposed a semi-automatic approach that detects design flaws [12] through a set of metric-based rules which called them 'detection strategy'. The rules first analyze the code to separate correlated symptoms (e.g., excessive method complexity, high coupling) that can be measured by a single metric. Secondly, for each symptom, the rules apply filters (*e.g.,* "HigherThan") proposed to detect various flaws. This approach was shown experimentally capable to detect ten design flaws with the tool named iPlasma; the evaluation is presented through precision, while the recall rate was ignored. One advantage of this work is that it uses correlated metrics instead of separate metrics that produce individual measurements whose interpretation does not reflect the cause of a flaw. In addition, Marinescu specified a process to organize the detection strategy. This process facilitates the passage from the description of a design flaw to a detection strategy.

Trifu et al. [2] also proposed an approach to detect design flaws. They used a tool called "jGoose" that creates a design database with relevant information about the system's source code. The design DB contains information about all structural design entities of the system such as classes, methods and attributes along with their relations. Moreover, the design DB contains the values of some basic metrics such as control flow, complexity of methods ([18]) and lines of code. This design DB is stored as an XMI model accessed through the standard query language XQuery to find design flaws. An important aspect in the approach of Trifu is that he related the quality factor to the design flaw. However, this work showed how to detect only design flaws that have a well known form expressible through the

handled metrics. Furthermore, expressing the rule into a query is complex and the authors do not provide for an assistance means.

Alikacem et al. [7] proposed an approach to detect violations of quality rules in object-oriented programs. For this, they classified quality rules into three categories: metric-based rules, structural information-based rules and rules expressing abstract notions. They defined a meta-model for representing the source code and a language to express a quality rule and its metrics independent of the programming language (java, C...). Moreover, they used fuzzy thresholds during the application of the quality rule to decide up-on the quality of the source code. The advantages of this proposition are pending a validation.

Moha [13] [14] combined the idea of fuzzy thresholds proposed by Alikacem et al. [7] and the idea behind the detection strategy presented by Marinescu [15]. She proposed to specify each design defect by a rule card. The rule card is a specification of a defect in terms of their measurable, structural and lexical properties. Given the rule card of a given design defect, the approach of Moha generates automatically the algorithm for detecting it. One advantage of this Moha's work is that the passage from the specification of a rule card to the algorithm is transparent. A second advantage is that Moha's work is the first complete approach containing both detection and correction of the detected design defects. Moreover, the efficiency of this approach depends on the designer capacity in defining manually the rule card necessary to detect a particular design defect. While correcting in the code source, the traceability between design and implementation will be lost (i.e. the code source does not reflect the design since it is modified).

We remark that all these previous works do not operate directly on the code source but they define a language or a model to represent the source code in order to make its manipulation easier. Note that, this supplementary step is not necessary when working at the design level.

2.2.2 Design Level

Besides the source code level, other works detect anti-patterns at the design level, *cf.,* [17], [5] and [6]. For example, Grotehen *et al.* [17] proposed an approach named METHOOD that checks the conformity of a set of heuristics ([1]) using measures like: size, hiding, coupling and cohesion. In addition to their detection, heuristics deviations can be corrected in METHOOD by a transformation rule which proposes an alternative fragments that respect the set of metrics used by the violated heuristic. However, in general an anti-pattern violates more than one heuristic, thus its detection is more complex. Finally, note that while METHOOD has been implemented in a tool named MEX (MEthood eXpert), this approach has not been evaluated experimentally.

Ballis et al. [5] [6] detect both patterns and anti-patterns using a rule-based matching method to identify all instances of a pattern/anti-pattern in the graph which underlies the designer's diagram [5]. A pattern/anti-pattern is defined either textually or in a graphical language that extends UML by adding few graphical primitives [6]. The strong point of this approach is that it allows the designer either to redefine easily the descriptions of canonical patterns (anti-patterns) to specify his/her customizations, or to define new ones from scratch. Evidently, the

graphical notation can specify only patterns/anti-patterns with a well-defined structural form.

Our contribution consists in: i) detecting anti-patterns at the design level, and ii) modeling both the structure and the behavior of five anti-patterns. Detecting anti-patterns at the design level allows the designer to anticipate the problems that could be generated by an implementation. It is true that, when moving from the code to the design, we lose information. But we compensate this loss by using the sequence diagrams (which highly reflects the dynamic information in the code) and by defining new metrics. Another distinction of our approach is that it exploits the semantics carried by the names; this information can easily characterize several aspects that are not captured through design metrics. Finally, similar to Moha's approach, our detection approach can be extended to assist in the correction step since it extracts a clear and significant report of the detected anti-pattern.

3 A New Anti-pattern Identification Approach

Our anti-pattern detection approach is based on OO software metrics used to measure quantifiable properties. In this section, we begin by listing a set of the most pertinent, existing metrics used in the detection of anti-patterns. Afterwards, we describe the metrics we added. Finally, we explain our detection rules which use the metrics for the class and sequence diagrams and rely on structural, behavioral and semantic information.

3.1 Useful Existing and New Metrics

Several metrics have been defined in the OO software engineering field. We selected the most important metrics and we classified them according to four categories: Coupling, cohesion, complexity and inheritance. Next, we explain each category and we present its associated metrics.

3.1.1 Coupling

Coupling measures the degree of interdependency between classes/objects. Two objects X and Y are coupled if at least one of them acts upon the other, *i.e.,* there is at least one method in X that calls methods or uses instance variables in Y and vice versa [16]. Coupling could, essentially, be measured with the following two metrics:

- **CBO** (Coupling Between Objects): *"The CBO for a class is a count of the number of other classes to which it is coupled"* [16]. The CBO value should be minimized. In fact, when it increases, the sensibility to changes is higher and therefore maintenance is more difficult [16].
- **RFC** (Response For Call) is the cardinality of the response set containing the methods called by each method in the class and the set of methods defined in the class. The larger the number of methods that can be invoked by a class, the greater the complexity of the class is [16].

3.1.2 Cohesion

Cohesion is a measure of how strongly-related and focused the various responsibilities of a class. A cohesive class is a class all of whose methods are tightly related to the attributes defined locally. The cohesion should be maximized to get a design with a good quality. The essential metrics measuring cohesion are:

- **LCOM** (Lack Of Cohesion in Methods)**:** It counts the number of method pairs whose similarity is zero, minus the count of method pairs whose similarity is not zero; the similarity of a pair of methods is the number of joint instance variables used by both methods. The larger the number of similar methods, the more cohesive the class is [16].
- **TCC** (Tight Class Cohesion)**:** It measures the cohesion of a class as the relative number of directly connected methods, where methods are considered to be connected when they use at least one common instance variable [9].
- **LCC** (Loose class Cohesion)**:** It counts, for each class, the percentage of method pairs related either directly or indirectly [9].
- **Coh** for a class with N methods M(1)…M(N) with N sets of parameters I(1)…I(N) and M is the number of the disjoint sets of parameters formed by the intersection of these N sets [10], then

$$Coh=M/N * 100\%$$

Note that Coh is insufficient to quantify the cohesion (useless), when most of the methods do not have parameters. In addition, LCOM, TCC and LCC are based on "instance variables used by methods" while this information is not available during the design phase. For this reason, we use the 'Coh' metric and we rely on semantic information.

3.1.3 Complexity

Complexity measures the simplicity and understandability of a design. Many complexity measures have been proposed in the literature, amongst which we find:

- **WMC** (Weighted Methods per Class)**:** This metric determines the complexity of a class by summing the complexities of its methods. Note that, WMC cannot be applied at the design level since the code of the method is not available.
- **NAtt:** the Number of the Attributes of a class [13].
- **NPrAtt:** the Number of Private Attributes [13].
- **NOM:** the Number Of Methods of a class including the constructor [13].
- **NII:** the Number of Imported Interfaces [13].

3.1.4 Inheritance

Inheritance measures the tree of inheritance and the number of children. In this category, we find:

- **DIT** (Depth of Inheritance of a class) is the depth in the inheritance tree. If multiple inheritances are involved, then the DIT will be the maximum length from the node to the root of the tree [16].
- **NOC** (Number Of Children) is the number of immediate subclasses subordinated to a class in the class hierarchy [16].

In addition to the above existing metrics, our detection approach uses the following new metrics for a class:

- **NAcc:** the Number of the Accessors in a class.
- **NAss:** the Number of Associations (association link, agregation, composition, dependency link) of a class.
- **NInvoc:** the Number of the Invoked methods (Call Action in the sequence diagram) of a class.
- **NReceive:** the Number of the Received messages that invoke methods of this class.

3.2 The Metric Threshold Issue

Choosing useful metrics is not enough to ensure an efficient detection; it is necessary to fix the threshold values which highly influence the efficiency of the detection process.We should caution that, even in the software engineering field, in general, there is not yet a precise guideline for how to fix thresholds and good interpretation of the proposed quality metrics.

Marinescu [15] proposed three means to fix metric thresholds. The first mean consists in using metrics from the literature with already predefined thresholds; this might require the adaptation of the thresholds to the system size [15]. In our case, there are several defined threshold metrics, but a few of them are validated; for instance, DIT is fixed to six according to [1]. In addition, to adapt such thresholds to the system size, the detection process would take more time since it needs to calculate the maximum size of each used metric (the maximum NAtt, the maximum NOM, etc). The second means to choose thresholds is to define a tuning machine which tries to find automatically the proper threshold values. In order to determine correct threshold values, this means requires a large repository of design fragments containing anti-patterns. Finally, the third means is to enhance the detection process by combining detection strategies applied on a single version with additional information about the history of the system. This means would be appropriate for evolving systems. However, when the design is being constructed for the first time, this means is not applicable. Marinescu [15] used in his detection strategy the boxplot technique which is a statistical method by which the abnormal values in a data set can be detected (http://en.wikipedia.org/wiki/Box_plot). On the other hand, Alikacem [7] resolved this problem by introducing fuzzy thresholds. In this case, the main goal is to encompass the exact value, for example a class with 19 or 21 methods will not be considered as a large class when the NOM is fixed to 20. As for Moha whose work combines the above two approaches, she used the boxplot and fuzzy thresholds at the same time.

In our approach, we try, on the one hand, to use already predefined thresholds and to let the designer parameterize the threshold value when necessary. Determining appropriate thresholds empirically is our ongoing work. In the next section, we assume that we have four thresholds delimiting each metric: very low, low, high and very high.

3.3 Detection

In this section, we present the detection of five anti-patterns: Blob, Lava Flow, Functional Decomposition, Poltergeists and Swiss Army knife. Table 1 lists the relationship between the different metric categories and these five anti-patterns. Note that, not all the anti-patterns defined by Brown [19] can be detected at the design level. For example, Spaghetti Code is a complex class containing methods without parameters and *using global variables*. The last symptom cannot be detected at the design level.

Table 1 Classification of anti-patterns

Anti-Patterns	Metrics Category			
	Coupling	Cohesion	Complexity	Inheritance
Blob	high	low	High	low
Lava Flow	low		High	
Functional Decomposition		high	Low	Very low
Poltergeists	high		low	
Swiss Army Knife	high		high	

The structural detection of an anti-pattern is insufficient. Behavioral information extracted from the sequence diagram and semantic information, in terms of anti-pattern class names and method names within the classes, are also required to confirm the presence of the anti-pattern and, hence, the necessity of a correction. Thus, we divide our anti-pattern detection process into three steps:

Structural anti-pattern detection: This step relies on the transformation of the class diagram into an XML document and a straightforward calculation of metrics such as NOM, NAtt. In fact, the transformation of UML diagrams into XML documents is rather trivial and can be handled by all existing UML editors.

Behavioral anti-pattern detection: Similar to the structural detection, this step also relies on the transformation of the sequence diagram into an XML document and metric calculation such as method calling, the sender, the receiver…

Semantic anti-patterns detection: After examining anti-pattern symptoms described in the literature, we found out that, in some cases the anti-pattern has not only quantifiable symptoms (structural and behavioral) but also semantic symptoms. It relies on linguistic information to identify the classes of the anti-pattern

problem. In fact, the determination of the semantic characteristics of anti-patterns can be handled through lexical dictionaries such as WordNet (http://fr.wikipedia.org/wiki/WordNet).

For the semantic identification, we use the following semantic information:

- **IsController(ClassName):** It determines if a class has a name as 'Controller', 'Main', or a synonym.
- **IsAccessor(MethodName):** It tests if a method is named as 'Get', 'Set' or their synonym.
- **Coh1(Att, Arg):** It determines if the argument 'Arg' is a synonym of the attribute 'Att'.
- **Coh2(Att, Meth):** It indicates if the name of the method 'Meth' contains the name of the attribute 'Att'.
- **IsFunctional(ClassName):** It tests if a class has a name as a function i.e. a verb or a noun action like 'Creation', 'Making',…
- **IsStart(MethodName):** It indicates if a method is named as 'Start','Initiate' or their synonym.

Table 2 shows the correlation between the symptoms and the category of metrics. It helps us to choose the convenient metrics for each anti-pattern.

Table 2 Correlation symtoms/metrics

Metrics	Symptoms Nature		
Category	Structural	Behavioral	Semantic
Coupling		CBO, RFC, NInvoc, NReceive	
Cohesion	Coh		Coh1, Coh2
Complexity	NPrAtt, NAtt, NOM, NAcc, NII, NAss		
Inheritance	DIT, NOC		

In the following sub-sections, we present our detection approach for the five anti-patterns in two categories: structurally by applying metrics on the class diagram, and both structurally and behaviorally through metrics applied on the class and sequence diagrams. We also present 'Correction proposition' which are useful later at the correction phase.

3.3.1 Structural Detection

Swiss Army Knife

Swiss army knife implements a *large number of interfaces* to expose the maximum possible functionalities. In addition, the classes associated to Swiss Army knife offer public interfaces. The Swiss Army Knife structure is illustrated in Fig. 1. The detection rule of this anti-pattern is: NII high.

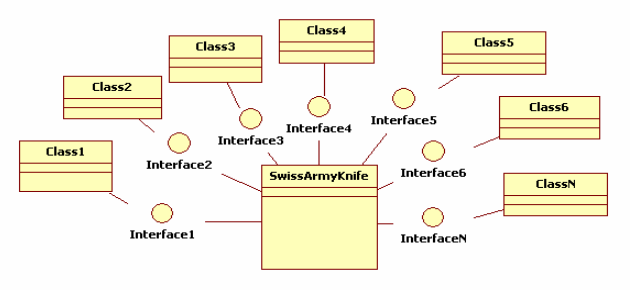

Fig. 1 Swiss Army Knife structure

3.3.2 Structural and Behavioral Detection

BLOB

The Blob anti-pattern consists in a class containing a large number of attributes and methods which makes it *complex*. This *large class* depends on the classes that surround it, and which are called *data classes*. A data class is a simple class that has only attributes and their method accessors (Fig. 2). The methods in the large class use attributes of the data classes (Fig. 3). Thus, the blob class is *highly coupled* with its data classes. In addition, the Blob anti-pattern contains methods that do not operate on local data, this makes it *non cohesive.*

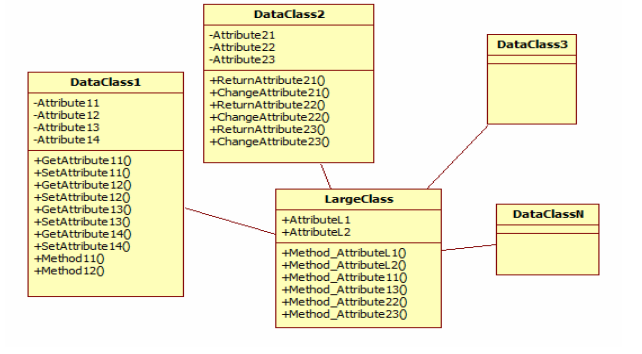

Fig. 2 Blob structure

We noticed that cohesion is necessary for detecting especially the blob; we overcome the insufficiency of the Coh metric by relying on the sequence diagram to find methods using attributes (by the accessors Fig. 3). In addition, we use the semantics for cohesion: we search sets of methods and attributes that have a common word or a term such as Title_Book and Borrow_Book().

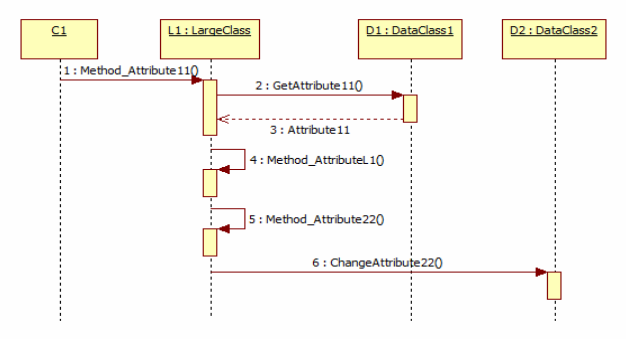

Fig. 3 A sample of a sequence diagram showing the low cohesion of a Blob

- **Large class:** NAtt high and NOM high and Coh low and Coh1 is true and Coh2 is true and DIT low and NOC low and RFC high and CBO high and IsController is true.
- **Data class:** NAcc high and NOM low and DIT low and NOC low and IsAccessor is true.

Correction proposition

- Methods of the large class should be moved to the data class.
- Methods and attributes of the large class should be moved to a new class.

Lava Flow

Lava Flow is known as a dead code. In terms of design, it consists in an isolated class which makes it uncoupled with others classes. Furthermore, on one hand, it is characterized by a large number of attributes and it is complex, so as a result its code is not clear and often developers cannot understand the main functionality of this class. On the other hand, it has no interactions, thus in general this class is isolated (i.e., it does not figure as an interacting object in any sequence diagram).

This anti-pattern can be detected through the following rule:

NAtt high and NOM high and NAss low and DIT zero and NOC zero and absence of interaction in the sequence diagram.

Functional Decomposition

This anti-pattern is materialized by classes having functional names (e.g., *Calculate_Interest*) and associations with cardinality "1". Moreover, all class attributes are private and used only inside the class; hence, there is a *high cohesion*. In general, it has classes with a *single action* such as a function, which makes them *simple* classes. Fig. 4 and Fig. 5 sketch the typical structure and the behavior of this anti-pattern, respectively. Note that, in the sequence diagram, when following the sequence of method calls, we observe that there is an invocation of one method on each class ensuring a chronological order.

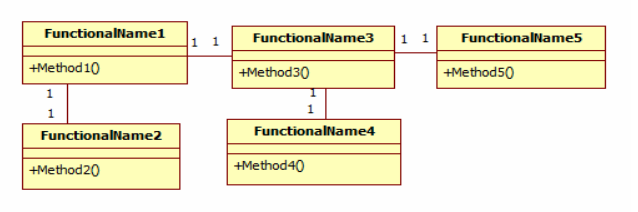

Fig. 4 Functional Decomposition structure

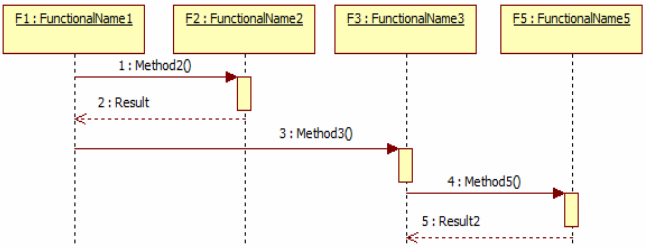

Fig. 5 A sample of a sequence diagram showing typical behavior of the Functional Decomposition anti-pattern

The Functional Decomposition can be identified through the following rule:

IsFunctional is true and NPrAtt low and NOM low and NAcc zero and DIT zero and NOC zero, some classes having NInvoc zero and other classes have successive invocations. Also the cardinality between classes has a value of "1".

Correction Proposition

- Classes having NInvoc zero should be merged with the class which invokes her method.
- Classes having successive invocations form the correct modelisation.

Poltergeists

This anti-pattern is characterized by a class containing a *single method* having *redundant navigation paths* to all other classes to 'seed' or '*invoke'* other classes through temporary associations (see Fig. 7). As a result, classes surrounding Poltergeists are *coupled* with it. Often, a Poltergeists is a class with 'control–like' method names such as *start_process* since it just initiates the system (Fig. 6).

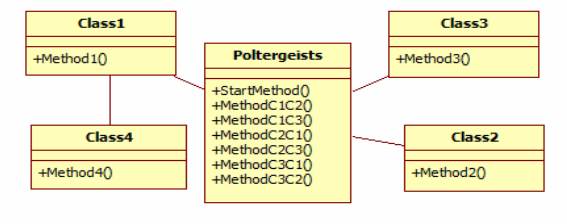

Fig. 6 Poltergeists Structure

When observing the sequence diagram, we find that all classes communicate with one another always through a central class which is the poltergeists (Fig. 7). In other word, many classes invoke method of the poltergeists which, in turn, invokes method of the destination class.

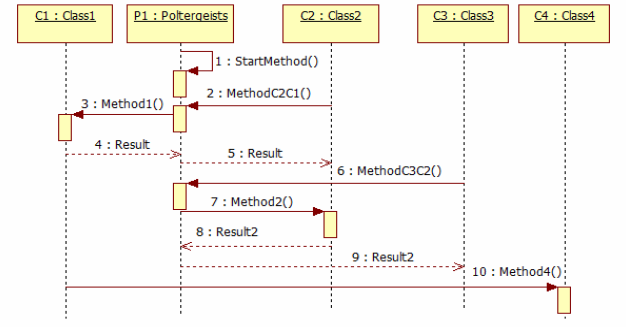

Fig. 7 A sequence diagram showing typical behavior and Centralized invocations of Poltergeists

We can identify this anti-pattern according to the following rule:

NAss high and NOM low and NInvoc=NReceive+1(the start method + methods for transient communication.) and IsStart is true.

Correction Proposition

- Class Poltergeist should be redesigned.
- Classes communicating through Poltergeist should be associated.

4 Illustration of Our Approach

In this section, we illustrate our anti-pattern detection approach through an example of a Functional Decomposition anti-pattern, adapted from the literature [19]. In fact, we added attributes and methods, essentially for clarity and understandability purposes. Fig. 8 presents the diagram class of the **Calculate Loan** example.

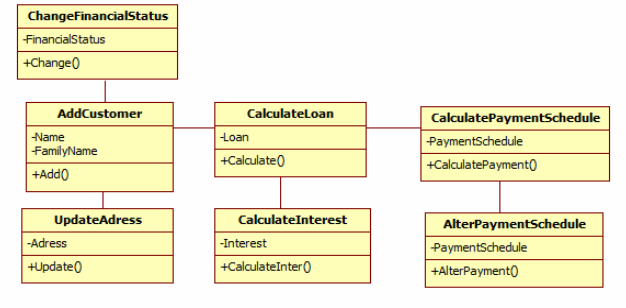

Fig. 8 Class diagram of the Calculate loan example

In addition, we specified the sequence diagram of the Calculate loan example by examining the documentation and scenario presented by Brown et al., [19]. Fig. 9 presents the sequence diagram illustrating the object interactions of the Calculate loan example. Next, we illustrate this scenario:

1. Adding a new Customer
2. Updating a Customer Address
3. Calculating a Loan to Customer
4. Calculating the Interest on a Loan
5. Calculating a Payment Schedule for a Customer Loan
6. Altering a Payment Schedule.

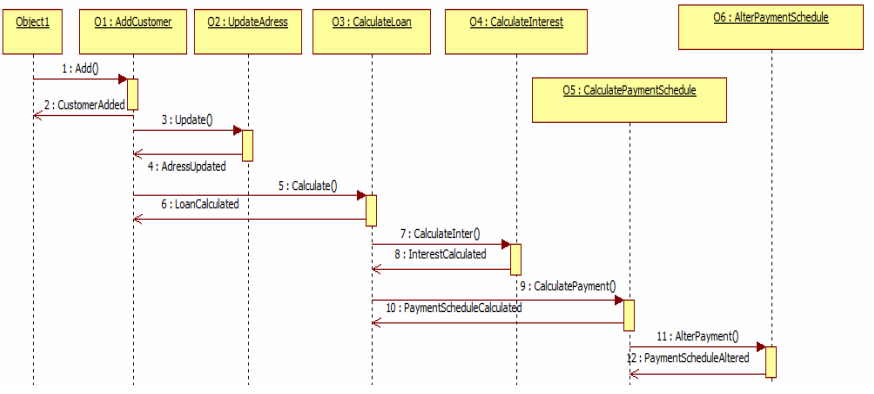

Fig. 9 Sequence diagram of the **Calculate loan** example

In order to detect the functional decomposition anti-pattern, the following 6 steps are applied:

Anti-pattern symptoms detection on the Class diagram

Step 1: Find the first class, having 'functional name' (IsFunctional is true).

Step 2: Follow the associated classes to this class having cardinality '1'.

Step 3: Repeat Step1 and Step2 to determine sets of related classes having 'functional name' (IsFunctional is true) and cardinality '1'.

Step 4: Calculate metrics NAtt, NOM, DIT, NOC and NAcc.

Anti-pattern symptoms detection on the sequence diagram

Step 5: Extract Classes that exist only to serve just one class. These classes are highly coupled and have to be merged together (NInvoc=0).

Step 6: Extract Classes that invoke methods of another class, just after receiving a message invoking one of its methods. Note that, these classes are related by association link.

The Results of the detection steps

Step 1, 2 and 3 : These steps detect the following sets of classes which have associations with cardinality "1" and which have functional names:

{ChangeFinancialStatus, AddCustomer}, {AddCustomer, UpdateAdress, CalculateLoan}, {CalculateLoan, CalculateInterest, AddCustomer, CalculatePaymentSchedule}, {CalculatePaymentSchedule, AlterPaymentSchedule, CalculateLoan}

Step 4: Table 3 shows the metric values relative to the set of classes already determined in the previous step.

Table 3 Calculated metrics on each class of the suspious Functional Decompossition

Class Name	Metrics				
	NPrAtt	NOM	DIT	NOC	NAcc
ChangeFinancialStatus	1	1	0	0	0
AddCustomer	2	1	0	0	0
UpdateAdress	1	1	0	0	0
CalculateLoan	1	1	0	0	0
CalculateInterest	1	1	0	0	0
CalculatePaymentSchedule	1	1	0	0	0
AlterPaymentSchedule	1	1	0	0	0

Step 5: Table 4 lists the sets of classes that interact only together.

Table 4 Result of Step 5

Class Name	Properties	
	NInvoc	ClassInvocator
UpdateAdress	0	AddCustomer
CalculateInterest	0	CalculateLoan
AlterPaymentSchedule	0	CalculatePaymentSchedule

Step 6: Now, we extract the sequence of ordered invocation methods, as illustrated in Fig.9. Add()→ Calculate()→ CalculatePayment()→ AlterPayment()

{AddCustomer, CalculateLoan, CalculatePaymentSchedule}

Finally, Fig. 10 shows the correction of the functional decomposition anti-pattern detected in the example of Fig. 8. Note that, we added the attributes and methods defined in each class.

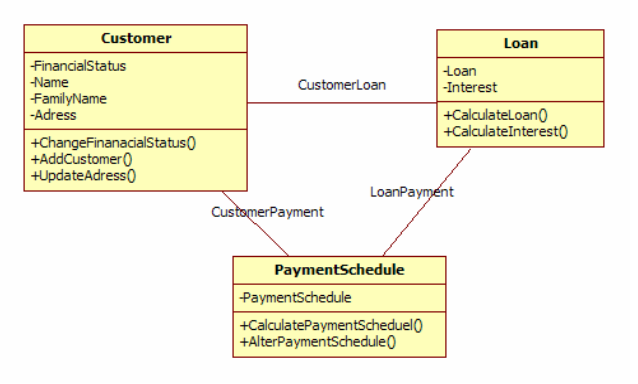

Fig. 10 The Correction of the **Calculate loan** example

5 Conclusion and Future Works

In this paper, we proposed an approach for detecting five anti-patterns described by Brown [19]. Working at the design level, our approach has the merit of anticipating anti-patterns at the code level and thus reduces their correction cost. Similar to existing approaches, ours relies on a set of existing and new design metrics defined for the class and sequence diagrams. It uses these metrics in a set of rules that, in addition, exploit the form of the anti-pattern such as the successive invocation in the Functional Decomposition anti-pattern. On the other hand, one main distinction of our approach from others is that it considers three types of information: structural and *semantic* from the class diagram, and behavioral from the sequence diagram.

Evaluated on several designs collected from the literature, our design approach produces very satisfactory levels of recall and precision. However, a thorough evaluation is underway to better place our approach amongst the existing approaches.

Furthermore, we are currently working on how to complement our detection approach with a correction phase. Here, we will exploit the report produced during the detection phase in terms of metrics, problematic classes, relationships and naming choices. Such detailed information will be used in a set of correction rules to produce alternative fragments. We will adapt the algorithm generation approach proposed by Moha [13].

References

1. Riel, J.: Object-Oriented Design Heuristics. Addison Wesley, Reading (1996)
2. Trifu, A., Seng, A., Genssler, T.: Automated design flaw correction in object-oriented systems. In: Proceedings of the 8th Conference on Software Maintenance and Reingineering (CSMR 2004), pp. 174–183. IEEE Computer Society Press, Los Alamitos (2004)

3. Bouhours, C., Leblanc, H.: Christian Percebois. Bad smells in design and design patterns. Journal of Object Technology 8, 43–63 (2009)
4. Marinescu, C., Marinescu, R., Mihancea, P.F., Wettel, R.: iPlasma: An integrated platform for quality assesment of object-oriented design. Loose Research group. Politechiea University of Tinoisoora (ICSM 2005), pp. 77–80. Society Press (2005)
5. Ballis, D., Baruzzo, A., Comini, M.: A rule-based method to match Software Patterns against UML Models. In: Proceedings of International Workshop on Rule Based Programming (RULE 2007), vol. 219, pp. 51–66. Theoretical Computer Science Press (2008)
6. Ballis, D., Baruzzo, A., Comini, M.: A Minimalist Visual Notation for Design Patterns and Antipatterns. In: Proceedings of the 5th International Conference on Information Technology: New Generations (ITNG 2008), April 2009, pp. 51–66 (2009) (in press)
7. Alikacem, E., Sahraoui, H.A.: Détection d'anomalies utilisant un langage de description de règle de qualité. In: Rousseau, R., Urtado, C., Vauttier, S.(eds.) 12 Conference on Languages and Models with Objects (LMO 2006), March 2006, pp. 185–200 (2006)
8. Gamma, E., Helm, R., Johnson, R., Vlissides, J.M.: Design patterns: Elements of reusable Object Oriented Software. Addisson-Wesley, Reading (1995)
9. Bieman, J.M., Kang, B.K.: Cohesion and reuse in an object oriented system. In: Proceedings ACM Symp., On Software Reusability, pp. 259–262 (1995)
10. Chen, J.Y., Lu, J.F.: A new metric for object-oriented design. Information and Software Technology 35, 232–240 (1993)
11. Dodani, M.: Patterns of antipatterns. Journal Of Object Technology 5, 29–33 (2006)
12. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring-Improving the Design of Existing Code, 1st edn. Addison-Wesley, Reading (1999)
13. Moha, N., Guéheneuc, Y.G., Leduc, P.: Automatic generation of detection algorithms for design defects. In: Proceedings of the 21st Conference on Automated Software Engi-neering (ASE 2006), September 2006, pp. 297–300. IEEE Computer Society Press, Los Alamitos (2006)
14. Moha, N., Guéhéneuc, Y.-G., Le Meur, A.-F., Duchien, L.: A domain analysis to specify design defects and generate detection algorithms. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 276–291. Springer, Heidelberg (2008)
15. Marinescu, R.: Detection strategies: Metrics-based rules for detecting design flaws. In: Proc. of the 20th International Conference on Software Maintenance (ICSM 2004), pp. 350–359. IEEE Computer Society Press, Los Alamitos (2004)
16. Chidamber, S.R., Kemerer, C.F.: A metric suite for object oriented design. IEEE Transactions on Software Engineering 20, 476–493 (1994)
17. Grotehen, T., Dittrich, K.R.: The MeTHOOD Approach: Measures, Transformation Rules, and Heuristics for Object-Oriented Design. Technical Report: ifi-97.09 (1997)
18. McCabe, T.J.: A Complexity Measure. IEEE Transactions on Software Engineering 4, 308–320 (1976)
19. Brown, W.J., Malveau, R.C., McCormick, H.W., Mowbray, T.J.: Antipatterns: Refactorin Software, Architectures, and Projects in Crisis, 1st edn. John Wily and Sons, West Sussex (1998)