Anti-Pattern Detection: Methods, Challenges, and Open Issues

FABIO PALOMBA, GABRIELE BAVOTA,
ROCCO OLIVETO, ANDREA DE LUCIA

Abstract
Anti-patterns are poor solutions to recurring design problems. They occur in object-oriented systems when developers unwillingly introduce them while designing and implementing the classes of their systems. Several empirical studies have highlighted that anti-patterns have a negative impact on the comprehension and maintainability of a software systems. Consequently, their identification has received recently more attention from both researchers and practitioners who have proposed various approaches to detect them. This chapter discusses on the approaches proposed in the literature. In addition, from the analysis of the state of the art, we will (i) derive a set of guidelines for building and evaluating recommendation systems supporting the detection of anti-patterns; and (ii) discuss some problems that are still open, to trace future research directions in the field. For this reason, the chapter provides a support to both researchers, who are interested in comprehending the results achieved so far in the identification of anti-patterns, and practitioner, who are interested in adopting a tool to identify anti-patterns in their software systems.

Keywords
Anti-pattern, Code Bad Smells, Linguistic Anti-pattern, Software Metrics.

Fabio Palomba • Andrea De Lucia
Department of Management and Information Technology
University of Salerno, Fisciano (SA) - Italy

Fabio Palomba
e-mail: fabio.palomba.89@gmail.com

Andrea De Lucia
e-mail: adelucia@unisa.it
URL: http://www.unisa.it/docenti/deluciaa/index

Gabriele Bavota
Department of Engineering
University of Sannio, Benevento – Italy
e-mail: gbavota@unisannio.it
URL: http://www.dmi.unisa.it/people/bavota/www/index.html

Rocco Oliveto
Department of Bioscience and Territory
University of Molise, Pesche (IS) - Italy
e-mail: rocco.oliveto@unimol.it
URL: http://distat.unimol.it/people/oliveto

1. Anti-pattern: Definitions and Motivations

During lifecycle, a software system undergoes continuous changes aiming at maintaining high its business level [36]. Unfortunately, these changes are often made by developers in a rush due to market/customers constraints. As a consequence, source code quality is often neglected with the risk of introducing *code bad smell* (or simply *code smells* or *smells*) [24], *i.e.*, symptoms of possible design problems in source code. For example, developers might add several responsibilities to a class feeling that it is not required to include them in separate classes. As a result, the class grows rapidly and when the added responsibilities grow and breed, the class becomes too complex and its quality deteriorates. Such classes are know as *Large Class* [24] and could be the cause of a design problem in source code, *i.e.*, *anti-pattern*[1]. Fowler [24] and Brown *et al*. [13] defined a catalogue of more than 30 anti-patterns. For each anti-pattern, they reported the definition of the anti-pattern as well as specific refactoring operations aimed at removing it in order to improve the quality of source code.

Even if there is anecdotal evidence that design problems (such as anti-patterns) negatively impacts software comprehension and maintenance, in the last decade, anti-patterns have been the subject of empirical studies aiming at empirically analyzing their impact on software maintainability. We now have empirical evidence that code containing code smells or participating in anti-patterns is significantly more change-prone than "clean" code [31]. Also, code participating in anti-patterns has a higher fault-proneness than the rest of the system code [31][37].

Other studies aimed at understanding the impact of anti-patterns on program comprehension [1] showed that the presence of an anti-pattern in the source code does not decrease the developers' performance, while a combination of anti-patterns results in a significant decrease of their performance [1][61]. While the results of this study indicates that single anti-patterns are not harmful, they also reveals that anti-patterns are quite diffuse in software systems and very often code components are affected by more than one anti-pattern. In addition, we have empirical evidence that the number of anti-patterns in software systems increases over time and only in few cases they are removed through refactoring operations [3][17].

All these findings suggest that code smells and anti-pattern need to be carefully detected and monitored and, whenever necessary, refactoring operations should be planned and performed to deal with them. Unfortunately, the identification and the correction of design flaws in large and non-trivial software systems can be very challenging. These observations call for recommendation systems supporting the software engineer in (i) identifying anti-patterns and (ii) designing and applying a refactoring solution to remove them.

In this chapter, we will focus on the analysis of different techniques aimed at detecting anti-patterns in source code[2]. By analyzing these approaches we will derive a set of guidelines for building and evaluating recommendation systems supporting the detection of anti-patterns. In addition, we will also discuss some problems that are still open, to trace future research directions in the field.

2. Methods for the Detection of Anti-Patterns

Among the 30+ anti-patterns defined in the literature [13][24], only for a subset of them we have approaches and tools for their automatic identification. Table 1 reports the list of such anti-patterns,

[1] Very often code bad smells and anti-pattern are used as synonym. However, the scenario we report allows to understand the difference between code smells and anti-patterns. Specifically, code smell represents something "probably wrong" in the code, while an anti-pattern is certainly a design problem in source code. In other words, a code smell might indicate an anti-pattern.

[2] The interested reader can find a survey on recommendation systems supporting refactoring operations in the Chapter 15 of the book "Recommendation Systems in Software Engineering" [51].

while in the next sections we discuss in details each of them. Specifically, for each anti-pattern we present (i) its definition; (ii) the approaches proposed for its identification; and (iii) the results of the empirical evaluation conducted to assess the detection accuracy of the approaches proposed in the literature.

2.1 Blob

Definition. The *Blob*, also named *God Class*, is a class implementing different responsibilities, generally characterized by the presence of a high number of attributes and methods, which implement different functionalities, and by many dependencies with data classes (*i.e.*, classes implementing only getter and setter methods) [24].

Anti-pattern	Description	References
Blob (God Class)	A class having huge size and implementing different responsibilities.	[32][41][44][47][48]
Feature Envy	A method making too many calls to methods of another class to obtain data and/or functionality.	[5][8][48][57]
Duplicate Code	Classes that show the same code structure in more than one place.	[11][12][28][30][59]
Refused Bequest	A class inheriting functionalities that it never uses.	[39]
Divergent Change	A class commonly changed in different ways for different reasons.	[48][52][53]
Shotgun Surgery	A class where a change implies cascading changes in several related classes.	[48][52]
Parallel Inheritance	Pair of classes where the addition of a subclass in a hierarchy implies the addition of a subclass in another hierarchy.	[48]
Functional Decomposition	A class implemented following a procedural-style.	[44]
Spaghetti Code	A class without structure that declare long methods without parameters	[44]
Swiss Army Knife	A class that exhibits high complexity and offers a large number of different services.	[44]
Type Checking	A class that shows complicated conditional statements.	[58]

Table 1 List of anti-patterns for which a detection strategy has been proposed in the literature

Detection strategies. The problem to identify classes affected by the Blob anti-pattern has been analyzed under three perspectives. First, researchers focused their attention on the definition of heuristic-based approaches that exploit several software quality metrics (*e.g.*, cohesion and coupling [19]). For instance, DECOR (DEtection and CORrection of Design Flaws) [44], use a set of rules, called "rule card"[3], describing the intrinsic characteristics of a class affected by Blob (see Figure 1). As described in the rule card, DECOR detects a Blob when the class has an LCOM5 (Lack of Cohesion Of Methods) [19] higher than 20, a number of methods and attributes higher than 20, a name that contains a suffix in the set {*Process, Control, Command, Manage, Drive, System*}, and it has an one-to-many association with data classes.

Beside DECOR, other approaches rely on metrics to identify Blobs in source code. For example, Marinescu [41] presented a detection strategy able to identify Blobs looking at deviations from good design principles (see Figure 2). Specifically, for each class a combination of cohesion and coupling metrics is computed and Blobs are identified as classes that exhibit an overall quality

3 http://www.ptidej.net/research/designsmells/

lower than other classes (absolute and relative thresholds are used in order to discriminate between Blobs and "clean" classes).

```
RULE_CARD: Blob {
  RULE: Blob {ASSOC: associated FROM: mainClass ONE
                     TO: DataClass MANY}
  RULE: mainClass {UNION LargeClassLowCohesion ControllerClass}
  RULE: LargeClassLowCohesion {UNION LargeClass LowCohesion}
  RULE: LargeClass {(METRIC: NMD + NAD, VERY_HIGH, 20)}
  RULE: LowCohesion {(METRIC: LCOM5, VERY_HIGH, 20)}
  RULE: ControllerClass {UNION (SEMANTIC: METHODNAME,
        {Process, Control, Command, Manage, Drive, System}),
        (SEMANTIC: CLASSNAME,  {Process, Control, Command,
        Manage, Drive, System}}
  RULE: DataClass {(STRUCT: METHOD_ACCESSOR, 90)}
};
```

Figure 1 Rule Card for the Blob identification in DECOR [44]

$$GodClass(S) = S' \Big|_{(WMC(C), TopValues(25\%)) \wedge (ATFD(C), HigherThan(1)) \wedge (TCC, BottomValues(25\%))}^{S' \subseteq S, \forall C \in S'}$$

$$GodClass(S) = S' \Big|_{(ATFD(C), HigherThan(1)) \wedge ((WMC(C), TopValues(25\%)) \vee (TCC, BottomValues(25\%)))}^{S' \subseteq S, \forall C \in S'}$$

Figure 2: Detection strategy to identify Blobs proposed by Marinescu [41]

Although Blob can be detected solely using structural properties, historical analysis can aid the identification of complementary, additional useful information. In fact, as the Blob is a class that centralizes most of the system's behavior, is possible to think that despite the kind of change a developer has to perform in a software system, if a Blob class is present, it is very likely that something will need to be changed in it [48]. This conjecture is on the basis of HIST (**H**istorical **I**nformation for **S**mell de**T**ection) [48]. In HIST, Blobs are detected as classes modified in more than α% of commits involving at least one another class. The parameter α has been empirically evaluated, and a value that provides good detection accuracy is α = 8.

The approaches reported above classify classes strictly as being or not anti-patterns, while an accurate analysis for the borderline classes is missing [32]. To mitigate such a problem, it is possible to build an identification model able to assign a probability that a class is affected by a Blob. Specifically, the DECOR identification rule card can be translated into a Bayesian Network using a discretization of the metric values. Then, a probability that a class is affected by a Blob is computed [32]. On the same line, Oliveto *et al.* [47] proposed the identification of Blobs by building the signature of Blobs. Given a set of Blobs is possible to derive their signature (represented by a curve) that synthetize the quality of the class. Specifically, each point of the curve is the value of a specific quality metrics (e.g., the CK metric suite). Then, the identification of Blob is simply obtained by comparing the curve (signature) of a class given in input with the (curves) signatures of the previous identified Blobs. The higher the similarity, the higher the likelihood that the new class is a Blob as well.

Blob classes could be also detected indirectly. There are approaches used to recommend Extract Class Refactoring (ECR) operations, which are operations specialized for removing Blob from a software system [47]. For instance, Fokaefs *et al.* [23] proposed an approach that takes in input a software system and suggests a set of ECR operations. In other words, the tool suggests to split a set of classes in several classes in order to have a better distribution of the responsibility. Clearly, the original classes are candidate Blob. The approach proposed by Fokaefs *et al.* [23] formulated the detection of ECR operations (and thus, indirectly, of Blobs) as a cluster analysis problem, where it

is necessary to identify the optimal partitioning of methods in different classes. In particular, for each class they analyze the structural dependencies between the entities of a class, *i.e.*, attributes and methods, in order to build, for each entity, the entity set, *i.e.*, the set of methods using it. The Jaccard distance between all couples of entity sets of the class is computed in order to cluster together cohesive groups of entities that can be extracted as separate classes. The Jaccard distance is computed as follows:

$$Jaccard(E_i, E_j) = 1 - \frac{|E_i \cap E_j|}{|E_i \cup E_j|}$$

where E_i and E_j are two entity sets, the numerator is the number of common entities between the two sets and the denominator is the total number of unique entities in the two sets. If splitting the class in separate classes the overall quality (in terms of cohesion and coupling) of the system improves, the approach proposed the splitting as a candidate ECR operation. In other words, a Blob has been identified. The approach proposed by Fokaefs *et al.* [23] has been implemented as an Eclipse plug-in, called JDeodorant[4].

Analysis of the detection accuracy. All the approaches described above have been empirically evaluated in order to understand the accuracy of the suggestions provided to the software engineer. DECOR has been empirically validated first on an open source system, called Xerces[5], and then on other 8 systems [44]. Overall, the precision of the approach is 88.6%, while the recall reaches 100% [44]. Regarding the detection strategy proposed by Marinescu, the empirical study revealed 60% of accuracy [41]. As for HIST, the accuracy has been evaluated in terms of precision and recall on 8 open source systems. The study showed that the overall precision of HIST is 76%, while the recall is 61% [48]. More importantly, HIST has been compared with DECOR showing that the historical analysis can support the software engineer better than the structural ones. Oliveto *et al.* [47] provided a comparison of their approach based on the signature of Blobs with DECOR [44] and the approach based on Bayesian Belief Network proposed by Kholm *et al.* [32]. The study revealed that the accuracy of their approach generally outperform the other ones. Finally, the benefits of JDeodorant have also been empirically analyzed. The empirical evaluation indicated that the refactoring operations provided by JDeodorant are meaningful and they approximate the refactoring operations previously performed by three developers with 67% precision and 82% recall [23]. Clearly, the accuracy of the refactoring operations identified represents a good proxy for the accuracy of the identification of Blobs.

2.2 Feature Envy

Definition: A method suffers of the *Feature Envy* anti-pattern if it is more interested in another class (also named envied class with respect the one it actually is in). It is often characterized by a large number of dependencies with the envied class [24]. Usually, this negatively influences the cohesion and the coupling of the class in which the method is implemented. In fact, the method suffering of Feature Envy reduces the cohesion of the class because it likely implements different responsibilities with respect to those implemented by the other methods of the class and increases the coupling, due to the many dependencies with methods of the envied class.

Detection strategies. A first simple way to detect Feature Envy in source code is to traverse the Abstract Syntax Tree (AST) of a software system in order to identify, for each field, the set of the referencing classes [5]. So, using a threshold it is possible to discriminate the fields having too many references with other classes.

[4] http://www.jdeodorant.com

[5] http://xerces.apache.org

Although the Feature Envy is one of the most studied anti-patterns in literature, the only automatic approach defined to detect it is the one proposed by Palomba *et al.* [48], namely HIST. The conjecture is that a method affected by Feature Envy changes more often with the envied class than with the class it is actually in. Given this conjecture, HIST identifies methods affected by this anti-patterns as those involved in commits with methods of another class of the system β% more than in commits with methods of their class. The parameter β underwent an empirical calibration that showed how better performance can be obtained with β = 70.

Despite the lacking of automatic techniques to detect this kind of anti-pattern, there are several approaches able to identify *Move Method Refactoring* (MMR), i.e., operations aimed at removing the Feature Envy anti-pattern [8][57]. In some way, these approaches can aid the software engineer also in the identification of the Feature Envy: if the suggestion proposed by the refactoring tool is correct, then an instance of the Feature Envy anti-pattern is present in the source code.

Besides ECR operations, JDeodorant also support MMR operations. The underlined approach uses a clustering analysis algorithm (see Figure 3).

```
extractMoveMethodRefactoringSuggestions(Method m)
        T = {}
        S = entity set of m
        for i = 1 to size of S
                entity = S[i]
                T = T U {entity.ownerClass}
        sort(T)
        suggestions = {}
        for i = 1 to size of T
                if(T[i] ≠ m.ownerClass ∧ modifiesDataStructureInTargetClass(m, T[i]) ∧
                preconditionsSatisfied(m, T[i]))
                        suggestions = suggestions U
                        {moveMethodSuggestions(m ⟶ T[i])}
                if(suggestions ≠ ∅)
                        return suggestions
                else
                        for i = 1 to size of T
                                if(T[i] = m.ownerClass
                                        return {}
                                else if preconditionsSatisfied(m, T[i])
                                        return moveMethodSuggestions(m ⟶ T[i])
                return {}
```

Figure 3 - Algorithm used by JDeodorant for the identification of Move Method Refactoring operations [57]

Given a method *m*, the approach forms a set of candidate target classes where *m* should be moved (set *T* in Figure 3). This set is obtained by examining the entities (i.e., attributes and methods) that *m* accesses from the other classes (entity set *S* in Figure 3). In particular, each class in the system containing at least one of the entities accessed by *m* is added to *T*. Then, the candidate target classes in *T* are sorted in descending order according to the number of entities that *m* accesses from each of them (*sort(T)* in Figure 3). In the following steps each target class T_i is analyzed to verify its suitability to be the recommended class. In particular, T_i must satisfy three conditions to be considered in the set of candidate suggestions: (i) T_i is not the class *m* currently belongs to, (ii) *m* modifies at least one data structure in T_i, and (iii) moving *m* in T_i satisfies a set of behavior preserving preconditions [57]. The set of classes in *T* satisfying all the conditions above are put in the *suggestions* set (see Figure 3). If *suggestions* is not empty, the approach suggests to move *m* in the first candidate target class following the order of the sorted set *T*. On the other side, if *suggestions* is empty, the classes in the sorted set *T* are again analyzed by applying milder constraints than before. In particular, if a class T_i is the *m* owner class, then no refactoring suggestion is performed and the algorithm stops. Otherwise, the approach checks if moving the method *m* into T_i satisfies the behavior preserving preconditions. If so, the approach suggests to move *m* into T_i. Thus, an instance of the Feature Envy anti-pattern is identified.

This technique uses structural information to suggest MMR operations. However, there are cases where the Feature Envy and the envied class are related by a conceptual linkage rather than a structural one. Here the lexical properties of source code can aid in the identification of the right refactoring to perform. This is the reason why Bavota *et al.* presented MethodBook [8], in which methods and classes play the same role of the people and groups, respectively, in Facebook[6]. In particular, methods represent people, and so they have their own information as, for example, method calls or conceptual relationships with the other methods in the same class as well as the methods in the other classes. To identify the envied class, MethodBook use Relational Topic Model (RTM). Following the Facebook metaphor, the use of RTM is able to identify "friends" of the method under analysis. If the class having the highest number of "friends" of the considered method is not the current owner class, a refactoring operation is suggested (*i.e.*, a Feature Envy is detected).

Analysis of the detection accuracy. The evaluation on the detection algorithm provided by HIST has been conducted on 8 open source systems in terms of precision and recall [48]. The results show that HIST is able to suggest candidate Feature Envy anti-patterns with a precision of 71% and a recall of 81%. HIST has also been compared with the tool JDeodorant and the results have shown that the precision of the two approaches are comparable (71% against 68%), while the recall of HIST is quite higher than the one achieved by JDeodorant (81% against 60%) [48]. Also MethodBook has been compared with JDeodorant on 6 open source systems. The comparison performed with JDeodorant highlighted that Methodbook is generally more precise than JDeodorant, providing less suggestions to the developers of an average higher quality. However, the results also clearly highlighted that JDeodorant is able to identify good refactoring operations (and thus correct instances of Feature Envy) that are missed by Methodbook [8].

2.3 Duplicate Code

Definition: Classes that show the same code structure in more than one place in the system are affected by *Duplicate Code* anti-pattern. Code duplication is a potentially serious problem that affects the maintainability of a software system, but also its comprehensibility. The problem of the identification of code duplication is very challenging simply because, during the evolution, different copies of a feature suffer different changes and this affect the possibility of the identification of the common functionality provided by different copied features. Moreover, having duplicate components in source code implies the increase of the effort in the maintenance of a system because when a change request involving such duplicate components is received, developer needs to modify several times the same feature because several copies are disseminated in different places. For all these reasons the problem of finding duplicate code, has been recognized as a potentially serious problem affecting the stability of an application, but also its fault-proneness [29].

Detection strategies. In the literature several approaches for clone detection have been proposed. It is worth noting that a semantic detection of clones could be very hard to perform and, in general, this is an undecidable problem. This is the reason why most of the proposed techniques focus their attention on the detection of syntactic or structural similarity of source code. For example, the AST of the given program is built in order to find matches of sub-trees in [11][12][59]. Alternatively, Kamiya *et al.* [30] introduced the tool *CCFinder*, where a program is divided in lexemes and the token sequences are compared in order to find matches between two subsequences.

However, such approaches appear to be ineffective in cases where duplicated code suffers several modifications during its evolution. To mitigate such a problem, Jiang *et al.* [28] introduced DECKARD, a technique able to identify clones using a mix of tree-based and syntactic-based approaches. The process they follow can be summarized as follow:

[6] https://www.facebook.com

1. Given a program, a parser translates source code into parse tree;
2. Syntactic trees are processed in order to produce a set of vectors capturing the syntactic information of parse tree;
3. The Euclidean distances are performed. Thus, the vectors are clustered;
4. Heuristics are applied to detect clones.

Analysis of the detection accuracy. The detection methods based on AST [11][12] have been applied on one process-control system written in C, having 400 KLOC. The authors found that most of detected clones were of small size and they were functions performing the same operations in different places. The main problem of this approach is that it is able to determine only exact tree match. The empirical evaluation conducted on DECKARD involved large systems as JDK and the Linux kernel, and showed that the tool performs significantly better than the other state-of-art technique to detect clones [28].

2.4 Refused Bequest

Definition: In Object Oriented development, one of the key features aiming at reducing the effort and the cost in software maintenance is inheritance [24]. For example, if there is something wrong in the definition of an attribute inherited by some children classes, such attribute needs to be changed only in the parent of such classes. However, it is not uncommon that developers make improper use of the concept of inheritance, especially in the cases where other kind of relationships would be more correct. The *Refused Bequest* anti-pattern arises when a subclass does not support the interface of the superclass [24]. On one hand, this happens when a subclass overrides a lot of methods inherited by its parent, on the other hand the relationship of inheritance can be wrong also if the subclass does not override the methods inherited from the parent, but never uses them or such methods are never called by the clients of the subclass. In some cases, this anti-pattern simply means that the relationship of inheritance is wrong, namely the subclass is not a specialization of the parent.

Detection strategies. A simple naive-method to estimate the presence of Refused Bequest anti-pattern in a software system is by looking for classes having the following characteristics: (i) the class is in a hierarchy; and (ii) the class overrides more than γ% of the methods defined by the parent. However, such a method does not take into account the semantics established by an "is-a" relationship between two classes of the system. For this reason, Ligu *et al.* [39] introduced the identification of Refused Bequest using a combination of static source code analysis and dynamic unit test execution. In particular, the approach identifies classes affected by this anti-pattern by looking at the classes that really *"wants to support the interface of the superclass"* [24]. If a class does not support the behavior of its parent, a Refused Bequest is detected. In order to understand if a method of a superclass is actually called on subclass instances by other classes of the system, Ligu *et al.* [39] intentionally override these methods introducing an error in the new implementation (e.g., division by zero). If there are classes in the system invoking the method, then a failure will occurs. Otherwise, if the execution of all the test cases does not lead to a failure, the inherited superclass methods are never used by the other classes of the system and, thus, an instance of Refused Bequest is found. The approach proposed by Ligu *et al.* [39] has been implemented as an extension of the JDeodorant Eclipse plugin [57].

Analysis of the detection accuracy. Unfortunately, the approach proposed by Ligu *et al.* [39] for the identification of Refused Bequest has not been empirically evaluated yet. The authors only applied the approach on one medium-size open source system in order to have a preliminary idea of its accuracy.

2.5 Divergent Change

Definition: Fowler describes a *Divergent Change* as a class that is *"commonly changed in different ways for different reasons"* [24]. Classes affected by this anti-pattern generally have low cohesion.

Detection strategies. The definition of Divergent Change provided by Fowler suggests that structural techniques are not completely suitable to detect instances of such an anti-pattern. The reason is that in order to identify Divergent Change anti-patterns it is necessary to analyze how the system evolves over time. Only one approach provides an attempt to exploit structural information (i.e., coupling) to identify this anti-pattern [52]. In particular, using coupling information it is possible to build a Design Change Propagation Probability (DCPP) matrix. The DCPP is an n x n matrix where the generic entry A_{ij} is the probability that a design change on the artifact i requires a change also to the artifact j. Such probability is given by the *cdegree* [53], i.e., an indicator of the number of dependencies between two artifacts. Once the matrix is built, a Divergent Change instance is detected if a column in the matrix contains high values for a particular artifact. In other words, high values on the columns of the matrix correspond to have an high number of artifacts related to the one under analysis and so the probability to have a Divergent Change increase.

However, by looking at the definition, it is reasonable to think to detect this kind of anti-pattern using the historical information that a system can have (i.e., change log). The conjecture is that classes affected by this anti-pattern present different sets of methods each one containing methods changing together but independently from methods in the other sets. Such a conjecture has been implemented in HIST, that uses association rules discovery to detect a subsets of methods in the same class that often change together [48]. Once HIST detects these change rules between methods of the same class, the approach identifies Divergent Change as those containing at least two or more sets of methods with the following characteristics:

1. The cardinality of the set is at least γ;
2. All methods in the set change together, as detected by the association rules;
3. Each method in the set does not change with methods in other sets as detected by the association rules.

Analysis of the detection accuracy. Unfortunately, there is not a real empirical evaluation of the approach based on structural information to detect Divergent Change anti-pattern. The authors of this approach only proposed two example scenarios in which the defined approach is able to detect instances of Divergent Change [52][53]. They planned to make an empirical study in order to validate their technique in a real context. Regarding HIST, the evaluation was focused on the accuracy of the detection algorithm, but also on its behavior when compared with a static approach using solely structural information (i.e., methods' calls). The results indicated that HIST has higher precision (73% *vs.* 20%) and recall (79% *vs.* 7%) than the technique that exploits only structural information [48].

2.6 Shotgun Surgery

Definition. This anti-pattern appears when *"every time you make a kind of change, you have to make a lot of little changes to a lot of different classes"* [24]. As for Divergent Change, also in this case finding a purely structural technique able to provide an accurate detection of this anti-pattern is rather challenging.

Detection strategies. There are only two approaches able to identify this anti-pattern in the source code: the first is proposed by Rao and Raddy [52] and it is based on the DCPP matrix, the second is included in HIST and relies on historical information [48]. As for the former, after the construction of the DCPP matrix (as for the identification of Divergent Change) the approach detects instances of Shotgun Surgery in a way complementary to that for the Divergent Change detection. In fact, if a

row of the matrix contains high values for an artifact, it means that there is high probability that a change involving the artifact impact on more than one artifact. Regarding the latter, HIST implements the following conjecture: "*a class affected by Shotgun Surgery contains at least one method changing together with several other methods contained in other classes*" [48]. As for the Divergent Change, association rule discovery allows to identify a set of methods belonging to different classes often changing together. Thus, a class is affected by Shotgun Surgery if it contains at least one method that changes with methods contained in more than δ different classes. The parameter δ has been empirically calibrated and the empirical analysis revealed that the best performance are achieved with $\delta = 3$.

Analysis of the detection accuracy. The empirical evaluation of the approach based on the DCCP matrix was only planned and not available yet [52]. Instead, the detection algorithm of HIST has been validated on 8 open source systems in terms of accuracy. The results showed that on one hand, the first point highlighted is that this anti-pattern is not quite common (only 4 instances on the 8 systems), on the other hand HIST was able to detect all the instances found, reaching a precision and a recall of 100% [48].

2.7 Parallel Inheritance Hierarchies

Definition. Fowler defined the *Parallel Inheritance* as a special case of Shotgun Surgery, where *"every time you make a subclass of one class, you also have to make a subclass of another"* [24].

Detection strategies. Since this anti-pattern is considered a special case of Shotgun Surgery, it is reasonable to think that, also in this case, detecting it using structural properties of source code might be very challenging. The only technique able to identify instances of Parallel Inheritance Hierarchies is the one included in HIST [48] that relies on historical information. Following the definition provided by Fowler, HIST detects the Parallel Inheritance Hierarchies as pairs of classes for which the addition of a subclass for one class implies the addition of a subclass for the other class. Also in this case, association rule discovery is used to mine pairs of classes that respect this rule [48].

Analysis of the detection accuracy. HIST has been evaluated in terms of precision and recall and compared with a static technique constructed by the authors of HIST for providing some insights about the usefulness of historical analysis when detecting this anti-pattern. Such a technique exploits lexical properties of source code and it is based on the heuristic provided by Fowler for the identification of this anti-pattern: *"You can recognize this smell because the prefixes of the class names in one hierarchy are the same as the prefixes in another hierarchy."* [24]. The results showed that HIST reaches 61% of precision and recall, while the lexical-based technique identifies instances of Parallel Inheritance Hierarchies with a recall of 45%. Also, only 17% of the correct instances are detected by both the approaches, while 43% of instances are identified only by HIST, the remaining 40% only by the lexical-based technique. This result highlights that the combination of structural/lexical and historical analysis could be worthwhile for detecting this kind of anti-pattern.

2.8 Functional Decomposition

Definition. A class in which inheritance and polymorphism are poorly used, declaring many private fields and implementing few methods [13]. It can be symptom of procedural-style programming. The concepts of Object Oriented development are not always clear to developers working for the first time using such a paradigm. Indeed, a developer with high experience in functional paradigm tends to apply procedural rules in the development of Object Oriented software, producing errors in

the design of the application. The anti-pattern coined as *Functional Decomposition* is the most common anti-pattern appearing in these cases. Brown *et al.* [13] define Functional Decomposition as "*a main routine that calls many subroutines*".

Detection strategies. In order to detect this anti-pattern, some heuristics have been proposed. For example, knowing that usually a routine is called with a name invoking its function, it is not surprising to find an instance of Functional Decomposition in classes called with prefix as Make or Execute. At the same time, also heuristics based on the structure of a class can support the identification of this anti-pattern. For instance, a functional class can have many dependencies with classes composed by a very few number of methods addressing a single function.

```
RULE_CARD: FunctionalDecomposition {
  RULE: FunctionalDecomposition {ASSOC: associated FROM:
              mainClass ONE TO: aClass MANY}
  RULE: mainClass {UNION NoInheritPoly FunctionClass}
  RULE: NoInheritPoly {INTER NoInheritance NoPolymorphism}
  RULE: NoInheritance {(METRIC: DIT, SUP_EQ, 1, 0)}
  RULE: NoPolymorphism {(STRUCT: DIFFERENT_PARAMETER)}
  RULE: FunctionClass {(SEMANTIC: CLASSNAME, {Make, Create,
        Creator, Execute, Exec, Compute, Display, Calculate}}
  RULE: aClass {INTER ClassOneMethod FieldPrivate}
  RULE: ClassOneMethod {(STRUCT: ONE_METHOD)}
  RULE: FieldPrivate {(STRUCT: PRIVATE_FIELD, 100)}
};
```

Figure 4 - Rule Card for the Functional Decomposition identification in DECOR [44]

The only approach able to identify this anti-pattern is DECOR [44]. To infer the presence of the anti-pattern, DECOR uses a set of structural properties together to lexical analysis of the name of a class (see the rule card in Figure 4). Given a class, such class is affected by Functional Decomposition if it is a main class (a class generally characterized by a procedural name, e.g., Display, in which inheritance and polymorphism are poorly used) having many dependencies with small classes (classes with a very few number of methods and many private fields) [44].

Analysis of the detection accuracy. The accuracy of the detection algorithm implemented in DECOR has been preliminary validated on Xerces showing that the precision of the approach reaches 52%, while it has a recall of 100%. Then, the study was replicated on other 8 systems and the results confirmed those achieved in the preliminary evaluation [44].

2.9 Spaghetti Code

Definition. Classes affected by this anti-pattern are characterized by complex methods, with no parameters, interacting between them using instance variables. As for the Functional Decomposition anti-pattern, this is also a symptom of procedural-style programming [13].

Detection strategies. The Spaghetti Code anti-pattern describes source code difficult to comprehend by a developer, often without a well-defined structure and with several long methods without any parameter. From a lexical point of view, classes affected by this anti-pattern have usually procedural names.

DECOR [44] is also able to identify Spaghetti Code, once again by using a specific rule card that describes the anti-pattern through structural properties. As you can see in Figure 5 DECOR classifies the anti-pattern using only software metrics able to identify the specific characteristic of classes affected by this anti-pattern. This is possible simply because the Spaghetti Code does not involve any relationships with other classes, but it is a design problem concentrated in a single class

having procedural characteristics. Specifically, in DECOR instances of Spaghetti Code are found looking for classes having at least one long method, namely a method composed by a large number of LOC and declaring no parameters. At the same time, the class does not present characteristics of Object Oriented design. For example, the class does not use the concept of inheritance and should use many global variables.

```
RULE_CARD: SpaghettiCode {
  RULE: SpaghettiCode {INTER: NoInheritanceClassGlobalVariable
                       LongMethodMethodNoParameter}
  RULE: LongMethodMethodNoParameter {INTER LongMethod
                       MethodNoParameter}
  RULE: LongMethod { (METRIC: METHOD_LOC, VERY_HIGH, 0) }
  RULE: MethodNoParameter { (STRUCT: METHOD_NO_PARAM) }
  RULE: NoInheritanceClassGlobalVariable {INTER NoInheritance
                       ClassGlobalVariable}
  RULE: NoInheritance { (METRIC: DIT, INF_EQ, 2, 0) }
  RULE: ClassGlobalVariable {INTER ClassOneMethod FieldPrivate}
  RULE: ClassOneMethod { (STRUCT: GLOBAL_VARIABLE, 1) }
};
```

Figure 5 - Rule Card for the Spaghetti Code identification in DECOR [44]

Analysis of the detection accuracy. The empirical evaluation of DECOR indicated that the number of instances correctly detected by DECOR reaches 61%. However, a better detection of this kind of anti-pattern could be possible through the use of some lexical rules. Since a class affected by Spaghetti Code is a class built through procedural thinking, the use of lexical rules can be complementary to the structural rules used by DECOR. For example, the same strategy applied for Functional Decomposition can be applied here: checking the prefix of the class name can aid the identification of this anti-pattern.

2.10 Swiss Army Knife

Definition. A *Swiss Army Knife* is a class that exhibits high complexity and offers a large number of different services. This type of anti-pattern is slightly different from a Blob, because in order to address the different responsibilities, it exposes high complexity, while a Blob is a class that monopolizes processing and data of the system.

Detection strategies. A class affected by the Swiss Army Knife is a class that provides answer to a large range of needs. Generally, this anti-pattern arises when a class has many methods with high complexity and the class has a high number of interfaces. For example, a utility class exposing high complexity and addressing many services is a good candidate to be a Swiss Army Knife. The definition of this anti-pattern could seem very similar to the one given for the Blob. However, there is a slight difference between the two anti-patterns: the Blob is a class that monopolize most of processing and data of the system, having a “selfish behavior” because it works for itself, while a Swiss Army Knife is a class that provides services to other classes.

The characteristics of this anti-pattern suggest that structural information can be useful for its detection. In particular, a mix of software complexity metrics and semantic checks in order to identify the different services provided by the class could be used for the detection. Once again, DECOR is able to detect this anti-pattern through a rule card [44]. In the case of Swiss Army Knife, the rule card characterizes the anti-pattern on the Number of Interfaces metric, which is able to identify the number of services provided by a class. If the metric exceeds a given threshold, a Swiss Army Knife is detected.

Analysis of the detection accuracy. The accuracy of DECOR has been empirically validated on 9 open source systems in terms of precision and recall. Results showed that the recall reaches 100%, while the precision obtained was not so high (i.e., 41%) [44]. This result suggests that the use of lexical properties together with the structural ones could produce a more accurate detection tool.

2.11 Type Checking

Definition. A class that shows complicated conditional statements making the code difficult to understand and maintain. One of the most common situation in which a programmer linked to procedural languages can fall down using Object Oriented programming is the misunderstanding or lack of knowledge on how to use OO mechanisms such as polymorphism. This problem manifests itself especially when a developer uses conditional statements to dynamically dispatch the behavior of the system instead of polymorphism. Such a problem is known as *Type Checking* anti-pattern and, as for all the anti-patterns, in the maintenance phase this problem can grow up, creating problems to the developers.

Detection strategies. For the identification of symptoms of such anti-pattern, a simple heuristic can be used: when you see long and intricate conditional statements, then you have found a Type Checking candidate. Tsantalis *et al.* [58] proposed a technique to identify and refactor instances of this anti-pattern implemented in the JDeodorant Eclipse plugin. In particular, their detection strategy takes into account two different cases: in the first case, an attribute of a class represents a state and, depending on its value, different branches of the conditional statements are executed. Thus, if in the analyzed class, there is more than one condition involving the attribute, a candidate of Type Checking is found. In the second case, a conditional statement involves *RunTime Type Identification* (RTTI) in order to cast the type of a class in another to invoke methods on the last one. For example, this is the case of two classes involved in a hierarchy where the relationship is not exploited by using polymorphism. In such case, RTTI often arises in the form of an if/else if statement in which there is one conditional expression for each control on the type of a class. However, if the RTTI involves more than one conditional expression, a candidate of Type Checking is found.

Analysis of the detection accuracy. The Type Checking detection technique proposed by Tsantalis *et al.* [58] has been evaluated on three teaching examples in the textbooks by Demeyer *et al.* [21] and Fowler *et al.* [24] and it is able to correctly detect the instances reported by the authors of these books.

3. A New Frontier of Anti-patterns: Linguistic Anti-Patterns

There are several empirical studies in literature showing how linguistic aspects of source code can cause misunderstanding during the development of a software system. Indeed, the lack of comments, ambiguous selected identifiers and poor coding standards increase the risk to have problems during maintenance (see *e.g.*, [16][22][33][34][42][55]). Based on these observations, Arnaudova *et al.* [4] defined the linguistic anti-patterns (LAs), a new family of source code design problems that, as for the other anti-patterns in literature, could be described in terms of symptoms and consequences. Whereas “design” anti-patterns represent recurring, poor design choices, linguistic anti-patterns represent recurring, poor naming and commenting choices [4]. The first catalogue of this new family of anti-pattern contains six categories of linguistic design problems. Their detection has been done using linguistic heuristics, which depend on the type of anti-pattern defined. The section reports a brief introduction for each category of this kind of anti-patterns.

3.1 Does more than it says

This linguistic anti-pattern arises when there is a contradiction between the signature of the method and what the method really does. In particular, the method does more with respect to what the signature suggests. One of the examples provided by Arnaudova *et al.* [4] regards the use of the verb *is* in the name of a method. It suggests that the method performs an action returning a Boolean value. Thus, having an "is" method performing more than returning a Boolean value could be destabilizing for a developers. An example of such anti-pattern is provided in the Listing 1.

```
public int isValid () {
      final long currentTime = System.currentTimeMillis();
      if ( currentTime <= this.expires ) {
      // The delay has not passed yet
      // assuming source is valid.
            return SourceValidity.VALID;
      }

      // The delay has passed,
      // prepare for the next interval.
      this.expires = currentTime + this.delay;
      return this.delegate.isValid();
}
```

Listing 1 - An example of *Does more than it says* linguistic anti-pattern [4]

The method *isValid()* of the class *DelayedValidity* from the system *Cocoon 2.2.0* should return a Boolean value, but it performs more than checking if a component of that class is valid or not. In such cases, the rational behind the choice to design the method in this way should be included in the method summary or in the low-level documentation. In addition, a rename of the method is highly recommended.

The identification of such anti-pattern is based on the identification of all the methods with a signature not conforming to the standard guidelines provided in the catalogue defined by Arnaudova *et al.* [4].

```
protected void getMethodBodies
      (CompilationUnitDeclaration unit, int place) {
      // fill the methods bodies in order
      // for the code to be generated
      if ( unit.ignoreMethodBodies ) {
            unit.ignoreFurtherInvestigation = true;
            return;
            // if initial diet parse did not
            // work, no need to dig into method bodies.
      }

      if ( place < parseThreshold )
            return ; // work already done...

      // real parse of the method...
      parser.scanner.setSourceBuffer(unit.compilationResult
                  .compilationUnit.getContents());
      if (unit.types != null) {
            for (int i = unit.types.length; ,ài,àíi >= 0;)
                  unit.types[i].parseMethod(parser , unit);
      }
}
```

Listing 2 - An example of *Says more than it does* linguistic anti-pattern [4]

3.2 Says More than it Does

This anti-pattern occurs when a method has a signature evoking a liar behavior and the method body does not what the signature says [4]. For example, when a method has a name composed by the verb *get*, a developer might think that this method returns the value of an object instance variable. Listing 2 is provided a case where a "get" method does return nothing. Also in this case, method summary should contain the rationale of this method. However, a renaming should be applied. Also in this case, the identification of methods affected by this kind of anti-pattern could be done exploiting the definition provided in the catalogue.

3.3 Does the Opposite

This anti-pattern has been defined as "*the intent of the method suggested by its name is in contradiction with what it returns*" [4]. To understand this kind of anti-pattern, an example is provided in Listing 3. The name and return type of the method *disable()* are inconsistent. The reason is that the method *disable* returns an "enable" state. The problem with anti-pattern like this is that, if the documentation is missing, developers involved in maintenance operations can infer that the return type is a control state that can be enabled or disabled, while the method behavior is different.

```
/* Saves the current enable/disable state of
 * the given control and its descendents in
 * the returned object;
 * the controls are all disabled.
 * @param w the control
 * @return an object capturing the enable/disable state */

      public static ControlEnableState disable (Control w){
            return new ControlEnableState (w);
      }
```

Listing 3 - An example of *Does the opposite* linguistic anti-pattern [4]

In order to detect such anti-pattern, a set of heuristics can be applied. For instance, in the case of the example reported above, a contradiction is found looking at the method name and a part of its returned type. Using a thesaurus it is easy to derive that the two terms (i.e., disable and enable) are antonyms [4].

3.4 Contains More than it Says

The *Contains more than it says* anti-pattern appears when a variable of a class is declared as a single object, while its type is a collection of elements.

```
      int[] isReached;
```

Listing 4 - An example of *Contains more than it says* linguistic anti-pattern [4]

Listing 4 shows an example in which the variable *isReached* is an array of integer values; while a developer expects that its type is a Boolean.

A possible consequence is that a developer does not know that changing this variable, he changes multiple objects. Wrong uses in the declaration of variables in a class can be detected using a terms vocabulary, which contains information about the number of a name.

3.5 Says More than it Contains

This anti-pattern is exactly the opposite of the *Contains more than it says* [4]. In this case, a variable declared as multi-objects, contains only one object (Listing 5). Unlike what happens before, a developer performing maintenance activities on the class affected by this anti-pattern could think to change a collection of objects, while the real meaning of the variable is different. Also for this anti-pattern, wrong uses in the declaration of variables in a class can be detected using a thesaurus.

```
private static boolean stats = true;
```

Listing 5 - An example of *Says more than it contains* linguistic anti-pattern

3.5 Contains the Opposite

The goal of this anti-pattern is to find improper use of a variable in a class [4]. In particular, the anti-pattern aims at representing situations where a variable and its type are in contradiction, as shown in Listing 6.

```
MAssociationEnd start = null;
```

Listing 6 - An example of *Contains the Opposite* linguistic anti-pattern [4]

Once again, this anti-pattern can be detected by checking a vocabulary in order to find when the name of a variable is exactly the opposite of one of the terms composing its type.

4. Key Ingredients for Building an Anti-pattern Detection Tool

This section describes some of the challenges that need to be addressed when building a tool able to detect anti-pattern in software systems. Specifically, there are two critical issues to deal with:

- *Extracting information from source code components in order to identify symptoms of poor design choices*. This is a critical issue because there are different sources of information that can be exploited (e.g., structural or semantic relationship) and in several cases the choice of the type of information to be exploited depends on the kind of anti-pattern to be detected.
- *Define the algorithm to identify candidate anti-patterns*. Each algorithm has strengths and weaknesses. On the basis of the anti-pattern under analysis, an algorithm ensuring a fair compromise of strengths and weaknesses must be chosen.

In the following sections we present some guidelines on how to deal with these two issues.

4.1 Identifying and extracting the characteristics of anti-patterns

The first step in the identification of a specific anti-pattern regards the definition of its characteristics. Such characteristics should help in discriminating between components that are affected by the anti-pattern and “clean” components. As said before, there are different sources of information that can be used to extract source code properties that characterize a specific anti-pattern. Most of the techniques existing in the literature exploit structural information extracted by statically analyzing the source code to capture characteristics that could help in the identification of an anti-pattern, such as, the number of calls between two source code entities, variable accesses, or inheritance relations. A second option is dynamic information, which takes into account call relationships between entities occurring during program execution. Finally, historical data (e.g., co-changes) can be exploited to identify anti-patterns that are intrinsically characterized by how source code changes over time. Thus, in the following, we discuss all these sources of information.

4.1.1 Extraction of Structural Properties using Code Analysis Techniques

In this section we discuss the sources of information capturing structural properties (e.g., structural coupling) between code components.

Method calls. The most obvious source of information that can be exploited to capture structural relationships between code components is the calls interaction. A measure capturing method call interactions is the Call-based Dependence between Methods (CDM) [9]. CDM has values in [0, 1]; the higher the number of calls between two methods, the higher the CDM value and thus, the coupling between methods. This source of information can be useful for identifying anti-patterns acting at both method (e.g., Feature Envy) and class (e.g., Swiss Army Knife) level. Method calls have been exploited for the detection of many anti-patterns. In particular, different researchers have used them to investigate the presence of Blob instances [23] [32] [47], but also to find instances of Feature Envy [8] [57], Refused Bequest [39], Type Checking [58], Divergent Change [52], and Shotgun Surgery [52].

The calls between methods belonging to different classes also represent a particularly useful property for the detection of anti-patterns. Indeed, classes having many call interactions co-operate to implement the same (or strongly related) responsibilities and thus, are highly coupled. This information is particularly important for detecting anti-patterns aimed at causing harms to the modularization of Object-Oriented systems, e.g., Inappropriate Intimacy [24]. There are many metrics available in literature to measure the coupling between classes based on their call interactions. Examples are the Information-Flow-based Coupling (ICP) [35], the Message Passing Coupling (MPC) [38] and the Coupling Between Object Classes (CBO) [38].

Shared Instance Variables. The instance variables shared by two methods are an important source of information for detecting anti-pattern (e.g., Blob) inherent to the cohesion of a class. This source of information has been successfully used for the detection of Blob [9] [44], Spaghetti Code [44], and Type Checking [58] instances. In fact, they also represent a form of communications between methods (performed through shared data). Thus, methods sharing instance variables are more coupled than methods not sharing any data. A measure to capture this form of coupling between methods is the Structural Similarity between Methods (SSM) [26], used to compute the cohesion metric ClassCoh [26]. SSM has values in [0, 1]; the higher the number of instance variables the two methods share, the higher the SSM value and thus the coupling between methods. Another example of metric measuring the extent to which the methods of a class share instance is the Lack of Cohesion of Methods (LCOM) [19].

Inheritance Relationships. Inheritance dependencies among classes is another source of structural information to capture relationships between classes, and thus useful to identify anti-patterns involved in a hierarchy. Generally, the measurement of inheritance relationships between two classes is performed through a simple Boolean value: *true* if two classes have inheritance relationships or *false* otherwise. In the context of anti-pattern detection, inheritance relationships have been exploited to support the detection of Parallel Inheritance [48] and Refused Bequest [39].

4.1.2 Extraction of Lexical Properties through Natural Language Processing

A source code component contains text, e.g., identifiers, methods' names, and comments. Thus it can be considered as a textual document. Such a likeness has induced researchers to apply Natural Language Processing (NLP) techniques on source code in order to extract lexical properties. For example, Information Retrieval (IR) [6] techniques have been used to analyze the conceptual cohesion or coupling of classes. For instance, two methods are conceptually related if their (domain) semantics are similar, i.e., they perform conceptually similar actions. In order to compute

conceptual cohesion (or coupling), Marcus *et al.* [40] [50] proposed the use of Latent Semantic Indexing (LSI) [20], an advanced IR method that can be used to compute the textual similarity between the two methods. The higher the similarity between the methods of a class, the higher the conceptual similarity and thus the conceptual cohesion of the class. On the other hand, the higher the similarity between two classes, the higher the conceptual coupling between them. Empirical studies have indicated that conceptual metrics are orthogonal to structural ones. In other words, using NLP technique it is possible to identify specific properties of source code that are missed by looking only at structural information.

In the context of anti-pattern detection, lexical properties play a crucial role in the identification of linguistic anti-patterns. For instance, a part of speech analysis is required to (i) identify whether a term is a noun, an adjective, an adverb or other parts-of-speech; (ii) distinguish singular from plural nouns; and (iii) identify dependencies between words, e.g., between subjects and predicates, as well negative form. In addition, thesauruses (e.g., Wordnet [43]) are required to identify synonymous and antonymous relations.

Lexical properties are not only useful for the identification of linguistic anti-pattern. The analysis of the textual similarity between methods or classes has been used to characterize "design" anti-patterns, such as Feature Envy [8] and Blob [9]. Empirical studies have indicated that using lexical properties it is possible to identify instances of anti-patterns, which are missed by using only structural properties [8].

4.1.3 Extraction of the History Properties through Mining of Software Repositories

In order to find interesting and/or complementary properties with respect to the structural ones, it would be important to take into account the history of a system. In particular, it is possible to consider the way the code elements changes over time. This is really important when the goal is to detect anti-patterns generally characterized by how code elements evolve during the evolution of the system. For example, in the case of Divergent Change, we have a candidate anti-pattern when a class is commonly changed over time in different ways for different reasons [24]. In the following, we discuss which kinds of historical information could be extracted for the identification of anti-patterns.

Co-changes at File-level. Configuration Management Tools (CMTs) allow to manage different versions of a system. In particular, developers can manage the changes of the artifacts (configuration items) and keep track of changes occurring to configuration items. In addition, by analyzing the log file developers can get information about what was changed, who did the change, when the change was made and the message left by the developer who made the change. Thus, the log file contains a lot of information that can be mined aiming at finding interesting properties related to the development of a software system. Mining of software repositories is a quite new approach in the sphere of anti-pattern detection, while it has been widely used to support other software engineering tasks, such as [7, 14, 15, 18, 25, 45, 56, 60, 62]. In the context of anti-pattern detection, mining log files is worthwhile to detect instances of Parallel Inheritance by simply looking at the way subclasses are added during the evolution of the system.

Co-changes at Method-level. Using the standard APIs of CMTs it is only possible to mine co-changes at file level. Such a granularity level could be not sufficient to detect some of the anti-patterns defined in the literature. In fact, many of them describe method-level behavior (see, for instance, Feature Envy). This requires the use of an ad-hoc tool that is able to identify co-changes at method level. A tool supporting such a task has been recently developed in the context of a European project, namely the Markos project[7]. Specifically, the tool is able to identify (i) classes added, removed, moved, or renamed, (ii) class attributes added, removed, moved, or renamed, (iii)

[7] http://markosproject.berlios.de

methods added, removed, moved, or renamed, (iv) changes applied to all the method signatures (i.e., visibility change, return type change, parameter added, parameter removed, parameter type change, method rename), and (v) changes applied to all the method bodies. This information is particularly useful in the detection of anti-patterns involve methods changing often together during the history of the system, such as Divergent Change.

4.2 Defining the Detection Algorithm

The second step for the definition of an approach for the detection of anti-patterns is related to the choice of the algorithm to use in order to identify candidate instances of the specific anti-pattern. Such a choice depends on two factors:

- The kind of anti-patterns we are interested in detecting.
- The kind of information we want to exploit to characterize the anti-pattern.

Looking at the literature, we can observe that the algorithms proposed so far fall in two categories: (i) heuristic-based and (ii) machine learning-based.

A lot of anti-patterns can be identified using heuristic-based algorithms. A simple analysis of the dependencies between code components can be sufficient to correctly identify anti-patterns in source code. For example, in order to detect the Feature Envy anti-pattern, two kinds of heuristics can be used:

- A structural heuristic, counting the dependencies (e.g., calls) existing between a method m_i and a class C_j;
- A historical heuristic, counting the number of commits in which a method changes together with methods of the same class (*internal changes*) with respect to the number of commits in which the method change with methods of another class of the system (*external changes*).

The problem with these kinds of algorithms is related to the definition of a set of thresholds to use to identify an anti-pattern. For example, to detect the Feature Envy using a historical heuristic, a threshold is needed to determine the minimum difference between *internal* and *external changes*. In order to mitigate this problem, an empirical calibration of the threshold can be performed. However, the value identified empirically could be not sufficient to correctly detect anti-patterns in all the software systems due to the heterogeneity of software systems. Thus, specific tuning should be required on each system to improve the detection accuracy.

Machine learning techniques have also been widely used to detect different kinds of anti-patterns. For instance, rule cards defined by DECOR have been successfully translated into a Bayesian Network using a discretization of the metric values to detect Blobs. In addition, association rule discovery, an unsupervised learning technique, has been used to detect three different anti-patterns (i.e., Divergent Change, Shotgun Surgery, and Parallel Inheritance). Such a technique is able to identify local patterns showing attribute value conditions that occur together in a given dataset [2]. In the context of anti-pattern detection, the dataset is composed of a sequence of change sets, e.g., methods, that have been committed (changed) together in a version control repository [63]. An association rule, $M_{left} \rightarrow M_{right}$, between two disjoint method sets implies that if a change occurs in each $m_i \in M_{left}$, then another change should happen in each $m_j \in M_{right}$ within the same change set. The strength of an association rule is determined by its support and confidence [2], defined as:

$$Support = \frac{|M_{left} \cap M_{right}|}{T}$$

$$Confidence = \frac{|M_{left} \cap M_{right}|}{M_{left}}$$

where T is the total number of change sets extracted from the repository. In the context of anti-pattern identification, association rule mining has been performed using a well-known algorithm,

namely *Apriori* [2]. Note that, minimum *Support* and *Confidence* to consider an association rule as valid can be set in the *Apriori* algorithm. Once again such values can be determined empirically and could be necessary to re-tune such values when applying the approach on other systems.

4.3 Evaluating the Accuracy of a Detection Tool

The evaluation of an anti-pattern detection tool can consist of different steps and can be done following different strategies. In most of cases, a first evaluation has been done analyzing the accuracy in terms of metrics able to evaluate the goodness of an approach (i.e., *precision* and *recall* [6]). In other cases, the accuracy of the tool can be evaluated directly involving developers who express their opinion regarding the suggestions provided by the tool. In the following we report a set of evaluation strategies that can be used to evaluate an anti-pattern detection tool.

4.3.1 Evaluation based on an Automatic Oracle

To evaluate the accuracy of a detection tool, an oracle is required that reports the anti-pattern instances contained in a system. Unfortunately, very few software systems with annotated anti-patterns are available. This means, that to have a larger data set for experimenting anti-pattern detection tools a manual identification of anti-pattern instances in the object systems is required to build the oracle. In particular, starting from the definition of the anti-patterns reported in literature, the analysis of each class of a software system should be performed in order to identify instances of those anti-patterns. Once the oracle is defined, the detection tool is executed to extract the set of candidate anti-patterns. Finally, the two sets of anti-patterns (i.e., the manually identified and the candidate sets) can be compared and two widely-adopted Information Retrieval (IR) metrics, namely recall and precision [6] can be used to estimate the accuracy of the tool:

$$recall = \frac{|Correct \cap Detected|}{|Correct|}$$

$$precision = \frac{|Correct \cap Detected|}{|Detected|}$$

where *Correct* and *Detected* represent the set of true positive anti-patterns (those manually identified) and the set of anti-patterns detected by the approach, respectively. As an aggregate indicator of precision and recall, F-measure, i.e., the harmonic mean of precision and recall, can be used:

$$F - measure = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

4.3.2 Evaluation based on Developer's Judgment

When an oracle reporting the list of anti-pattern instances present in the system is not available, it is possible to involve developers in order to evaluate the performance of a detection tool. It is worth noting that this is not an optimal solution, because the developers can judge the goodness of a suggestion provided by the tool, while they cannot evaluate the performance of the tool with respect the totality of the anti-pattern instances present in the system (*i.e.*, they can judge the precision of the tool, but they can not aid in the evaluation of the recall). However, this kind of evaluation is still useful in two cases:

- As complement of the analysis based on the metrics, in order to understand to what extent the tool supports the developer.

- When you have to compare the support provided by two (or more) tools. In this case, a partial oracle can be built as the union of the suggestions provided by the tools. Thus, both the precision and recall can be evaluated.

We distinguish two kinds of studies that could be performed: (i) with the original developers of a system and (ii) with external developers. The evaluations performed with original developers are preferred since external developers do not have a deep knowledge of the design of the subject system under analysis and thus may not be aware of some of the design choices that could appear as suboptimal, but that are the results of a rational choice. However, studies with external developers can complement studies performed with original developers. In fact, even if the original developers have deep knowledge of the system's design, they could be the authors of some bad design choices and consequently could not recognize good suggestions by the tool. This means that the two studies complement each other mitigating the specific threats they have.

5. Conclusion and Open Issues

Anti-patterns represent symptoms of poor design and implementation choices [24]. Having classes affected by anti-patterns cause time-consuming maintenance operations due to their lower comprehensibility and maintainability. Thus, detecting anti-patterns in the source code is an important activity in order to improve the quality of a software system during its evolution. Some anti-patterns can be detected by using simple heuristics, while others are really complex to identify. This is the reason why in the last decade a lot of effort has been devoted to the definition of approaches able to recommend to developers problematic components that need to be refactored in order to improve their comprehensibility and maintainability.

In this chapter, we have described the state of the art regarding the detection of "design" and "linguistic" anti-pattern. We have also identified and described the challenges that need to be addressed for building a detection tool. Thus, this chapter provides a support to both researcher, who are interested in comprehending the results achieved so far in the identification of anti-patterns, and practitioner, who are interested in adopting a tool to identify anti-patterns in their software systems.

Even if the analysis of the literature reveals that the anti-pattern identification is a quite mature field, there are still some open issues that need to be addressed:

- *Are anti-patterns really harmful?* Despite the existing evidence about the negative effects of anti-patterns (*e.g.,* [1] [31] [61]) and the effort devoted to the definition of approaches for detecting and removing them, it is still unclear whether developers would actually consider all anti-patterns as actual symptoms of wrong design/implementation choices, or whether some of them are simply a manifestation of the intrinsic complexity of the designed solution. In other words, there seem to be a gap between the theory and the practice. For example, a recent study found that some source code files of the Linux Kernel intrinsically have high cyclomatic complexity. However, this is not considered a design or implementation problem by developers [27]. Also, empirical studies indicated that (i) developers perceived different instances of Blob as not particularly dangerous for the system maintainability, especially because they change these classes sporadically [54]; and (ii) some developers, in particular junior programmers, work better on a version of a system having some classes that centralized the control, i.e., Blob [46]. These results suggest that the presence of anti-patterns in source code is sometimes tolerable, and part of developers' design choices.
- *Providing a complete support for the identification of anti-patterns*. Fowler [24] and Brown *et al.* [13] defined more than 30 anti-patterns. In the last years, researchers concentrate their attention only on a small subset of anti-patterns defined in the literature. The world of anti-

patterns is still full of interesting points to pick in order to construct more useful and usable tools in a real context. To succeed in this goal the research community should focus the attention on other kinds of anti-patterns, but also on novel strategies for improving the accuracy of approaches defined in the literature.

- *On the use of historical and lexical analysis to detect anti-pattern*. A recent study indicates that historical properties of code components are able to complement structural properties in the process of anti-pattern detection [48]. This finding suggests that a better identification of anti-patterns can be achieved with a combined technique using a mix of historical and structural properties. Moreover, together with the structural and historical information, lexical properties could be exploited to better capture the "semantics" of design decisions, in order to have recommendations that are more focused on how developers designed the system under analysis.
- *Analyzing the usability of detection tools*. An important threat to the success of detection tools is related to their usability. Detection tools might require the definition of several parameters. Thus, they might be hard to understand and to work with, making developers more reluctant to use such tools. In addition, it is necessary to define a good strategy for the visualization and the analysis of the candidate anti-patterns. This issue is particular important since the anti-patterns identified by any detection tool need to be validated by the user. Thus, a good graphic metaphor is required to highlight problems to the developer's eye, allowing her to decide which of the code components suggested by the tool really represent design problems.

REFERENCES

[1] M. Abbes, F. Khomh, Y.-G. Guéhéneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," in Proceedings of the 15th European Conference on Software Maintenance and Reengineering, Oldenburg, Germany. IEEE Computer Society, 2011, pp. 181–190.

[2] R. Agrawal, T. Imielinski, and A. N. Swami, "Mining association rules between sets of items in large databases," in Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, 1993, pp. 207– 216.

[3] R. Arcoverde, A. Garcia, and E. Figueiredo, "Understanding the longevity of code smells: preliminary results of an explanatory survey," in Proceedings of the International Workshop on Refactoring Tools, 2011, pp. 33– 36.

[4] V. Arnaoudova, M. D. Penta, G. Antoniol, and Y.-G. Guéhéneuc, "A new family of software anti-patterns: linguistic anti-patterns," in Proceedings of the European Conference on Maintenance and Reengineering, Genova, Italy, 2013, pp. 187–196.

[5] D. C. Atkinson and T. King, "Lightweight detection of program refactorings," in Proceedings of 12th Asia-Pacific Software Engineering Conference. Taipei, Taiwan: IEEE CS Press, 2005, pp. 663–670.

[6] R. Baeza-Yates and B. Ribeiro-Neto, Modern Information Retrieval. Addison-Wesley, 1999.

[7] S. Bajracharya and C. Lopes, "Mining search topics from a code search engine usage log," in Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories, Washington, DC, USA. IEEE Computer Society, 2009, pp. 111–120.

[8] G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, A. De Lucia. "Methodbook: Recommending Move Method Refactorings via Relational Topic Models". In Transactions on Software Engineering, To appear.

[9] G. Bavota, A. De Lucia, and R. Oliveto, "Identifying extract class refactoring opportunities using structural and semantic cohesion measures," Journal of Systems and Software, vol. 84, pp. 397–414, 2011.

[10] G. Bavota, A. De Lucia, A. Marcus, R. Oliveto. "Automating Extract Class Refactoring: an Improved Method and its Evaluation," Empirical Software Engineering, To appear.

[11] I. D. Baxter, C. Pidgeon, and M. Mehlich, "Program transformations for practical scalable software evolution," in Proceedings of the International Conference on Software Engineering, 2004, pp. 625–634.
[12] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in Proceedings of the International Conference on Software Maintenance, 1998, pp. 368–377.
[13] W. J. Brown, R. C. Malveau, W. H. Brown, H. W. McCormick III, and T. J. Mowbray, "Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis", 1st ed. John Wiley and Sons, 1998.
[14] G. Canfora, L. Cerulo, and M. Di Penta, "Identifying changed source code lines from version repositories," in Proceedings of 4th International Workshop on Mining Software Repositories. Minneapolis, Minnesota, USA: IEEE CS Press, 2007, pp. 14–21.
[15] G. Canfora and L. Cerulo, "Impact analysis by mining software and change request repositories," in Proceedings of 11th IEEE International Symposium on Software Metrics. Como, Italy: IEEE CS Press, 2005, pp. 20–29.
[16] B. Caprile and P. Tonella, "Restructuring program identifier names," in Proceedings of the International Conference on Software Maintenance, 2000, pp. 97–107.
[17] A. Chatzigeorgiou and A. Manakos, "Investigating the evolution of bad smells in object-oriented code." in Proceedings of the 7th International Conference on the Quality of Information and Communications Technology, Porto, Portugal, 2010.
[18] T. Chen, S. W. Thomas, M. Nagappan, and A. E. Hassan, "Explaining software defects using topic models," in Proceedings of the 9th Working Conference on Mining Software Repositories, 2012.
[19] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," IEEE Transactions on Software Engineering, 20(6): 476–493, 1994.
[20] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer,R. Harshman, Indexing by latent semantic analysis. Journal of the American Society for Information Science, 41(6): 391–407, 1990.
[21] S. Demeyer, S. Ducasse, and O. Nierstrasz, Object-Oriented Reengineering Patterns. Morgan Kaufmann Publishers Inc., 2002.
[22] F. Deissenbock and M. Pizka, "Concise and consistent naming," in Proceedings of the International Workshop on Program Comprehension, 2005.
[23] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "Identification and Application of Extract Class Refactorings in Object Oriented Systems," Journal of Systems and Software, vol. 85, no. 10, pp. 2241-2260, 2012.
[24] M. Fowler, "Refactoring: improving the design of existing code". Addison- Wesley, 1999.
[25] H. Gall, M. Jazayeri, and J. Krajewski, "CVS release history data for detecting logical couplings," in Proceedings of the 6th International Workshop on Principles of Software Evolution, 2003, pp. 13–23.
[26] G. Gui and P. Scott, "Coupling and cohesion measures for evaluation of component reusability," in Proceedings of the 5th International Workshop on Mining Software Repositories, 2006, pp. 18–21.
[27] A. Jbara, A. Matan, and D. G. Feitelson, "High-MCC functions in the linux kernel," in Proceedings of the IEEE 20th International Conference on Program Comprehension, Passau, Germany, 2012, pp. 83–92.
[28] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and accurate tree-based detection of code clones," in Proceedings of the International Conference on Software Engineering, 2010.
[29] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in Proceedings of the 31st International Conference on Software Engineering, 2009, pp. 485–495.

[30] T. Kamiya, S. Kusumoto, and K. Inoue, "CCfinder: a multilinguistic token-based code clone detection system for large scale source code," Transactions on Software Engineering, no. 4, 2002.
[31] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, G. Antoniol, "An exploratory study of the impact of antipatterns on class change- and fault-proneness". Empirical Software Engineering 17(3): 243-275, 2012.
[32] F. Khomh, S. Vaucher, Y.G. Guéhéneuc, and H. Sahraoui, "A Bayesian Approach for the Detection of Code and Design Smells", in Proceedings of the 9th International Conference on Quality Software, 2009.
[33] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "Effective identifier names for comprehension and memory," Innovations in Systems and Software Engineering, 3(4): 303–318, 2007.
[34] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "What's in a name? A study of identifiers," in Proceedings of the International Conference on Program Comprehension, pp. 3–12, 2006.
[35] Y. Lee, B. Liang, S. Wu, and F. Wang, "Measuring the coupling and cohesion of an object-oriented program based on information flow," in Proceedings of the International Conference on Software Quality, 1995, pp. 81–90.
[36] M. M. Lehman, "On understanding laws, evolution, and conservation in the large-program life cycle," Journal of Systems and Software, vol. 1, pp. 213–221, 1980.
[37] W. Li and R. Shatnawi, "An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution," Journal of Systems and Software, pp. 1120–1128, 2007.
[38] W. Li and S. Henry, "Maintenance metrics for the object oriented paradigm," in Proceedings of International Symposium on Software Metrics, 1993, pp. 52–60.
[39] E. Ligu, A. Chatzigeorgiou, T. Chaikalis, and N. Ygeionomakis, "Identification of Refused Bequest Code Smells," in Proceedings of the 29th IEEE International Conference on Software Maintenance, 2013.
[40] A. Marcus, D. Poshyvanyk, R. Ferenc, "Using the conceptual cohesion of classes for fault prediction in object-oriented systems". IEEE Transaction on Software Engineering, 34(2), 287– 300, 2008.
[41] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in Proceedings of the 20th International Conference on Software Maintenance, Chicago, IL, USA. IEEE Computer Society, 2004, pp. 350–359.
[42] E. Merlo, I. McAdam, and R. De Mori, "Feed-forward and re-current neural networks for source code informal information analysis," Journal of Software Maintenance, vol. 15, no. 4, pp. 205–244, 2003.
[43] G. A. Miller, "WordNet: A lexical data base for English," Communications of the ACM, 38(11): 39-41, 1995.
[44] N. Moha, Y.G. Guéhéneuc, L. Duchien, and A.F. L. Meur, "DECOR: A method for the specification and detection of code and design smells," IEEE Transactions on Software Engineering, 36(1): 20–36, 2010.
[45] M. Ohba and K. Gondow, "Toward mining concept keywords from identifiers in large software projects," in Proceedings of International Workshop on Mining Software Repositories, St. Louis, Missouri, USA, 2005, pp. 1–5.
[46] S. M. Olbrich, D. S. Cruzes, and D. I. K. Sjøberg, "Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems," in Proceedings of the International Conference on Software Maintenance. IEEE, 2010, pp. 1–10.
[47] R. Oliveto, F. Khomh, G. Antoniol, and Y.-G. Guéhéneuc, "Numerical signatures of antipatterns: An approach based on b-splines," in Proceedings of the 14th Conference on Software Maintenance and Reengineering, IEEE Computer Society Press, 2010.

[48] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "Detecting bad smells in source code using change history information," in Proceedings of the 11th ACM/IEEE International Conference on Automated Software Engineering (ASE 2013), Silicon Valley, CA, USA. IEEE, 2013.

[49] R. Peters and A. Zaidman, "Evaluating the lifespan of code smells using software repository mining," in Proceedings of the European Conference on Software Maintenance and Reengineering, 2012, pp. 411–416.

[50] D. Poshyvanyk, A. Marcus, R. Ferenc, T. Gyimothy, "Using information retrieval based coupling measures for impact analysis. Empirical Software Engineering, 14(1), 5–32, 2009.

[51] M. P. Robillard, W. Maalej, R. J. Walker, T. Zimmermann, Recommendation Systems in Software Engineering. Springer, 2014.

[52] A. Rao and K. Reddy, "Detecting Bad Smells in Object Oriented Design Using Design Change Propagation Probability Matrix", in International MultiConference of Engineers and Computer Scientists, 2008.

[53] A. Rao and D. Ram, "Software Design Versioning using Propagation Probability Matrix", in Proceedings of Third International Conference on Computer Applications, Yangon, Myanmar, 2005.

[54] D. Ratiu, S. Ducasse, T. Gîrba, and R. Marinescu, "Using history information to improve design flaws detection," in Proceedings of the 8th European Conference on Software Maintenance and Reengineering, Tampere, Finland. IEEE Computer Society, 2004, pp. 223–232.

[55] A. Takang, P. A. Grubb, and R. D. Macredie, "The effects of comments and identifier names on program comprehensibility: an experiential study," Journal of Program Languages, 4(3): 143–167, 1996.

[56] S. W. Thomas, B. Adams, A. E. Hassan, and D. Blostein, "Modeling the evolution of topics in source code histories," in Proceedings of the 8th Working Conference on Mining Software Repositories, 2011, pp. 173–182.

[57] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," IEEE Transactions on Software Engineering, vol. 35, no. 3, pp. 347–367, 2009.

[58] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "JDeodorant: Identification and removal of type-checking bad smell," in Proceedings of the 23rd IEEE International Conference on Software Maintenance, Paris, France, 2007.

[59] V. Wahler, D. Seipel, J. W. von Gudenberg, and G. Fischer, "Clone detection in source code by frequent itemset techniques," in Proceedings of the IEEE Working Conference on Source Code Analysis and Manipulation, 2004, pp. 128–135.

[60] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller, "How long will it take to fix this bug?" in Proceedings of the 4th International Workshop on Mining Software Repositories. Minneapolis, Minnesota, USA: IEEE CS Press, 2007, pp. 1–8.

[61] A. Yamashita and L. Moonen, "Exploring the impact of inter-smell relations on software maintainability: an empirical study," in Proceedings of the 35th International Conference on Software Engineering, San Francisco, CA, USA, 2013, pp. 682–691.

[62] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll, "Predicting source code changes by mining change history," IEEE Transactions on Software Engineering, 30(9): 574–586, 2004.

[63] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," in Proceedings of the 26th International Conference on Software Engineering, 2004, pp. 563–572.