Detecting Bad Smells in Source Code using Change History Information

Fabio Palomba[1], Gabriele Bavota[2], Massimiliano Di Penta[2],
Rocco Oliveto[3], Andrea De Lucia[1], Denys Poshyvanyk[4]
[1]University of Salerno, Fisciano (SA), Italy
[2]University of Sannio, Benevento, Italy
[3]University of Molise, Pesche (IS), Italy
[4]The College of William and Mary, Williamsburg, VA, USA

***Abstract*—Code smells represent symptoms of poor implementation choices. Previous studies found that these smells make source code more difficult to maintain, possibly also increasing its fault-proneness. There are several approaches that identify smells based on code analysis techniques. However, we observe that many code smells are intrinsically characterized by how code elements change over time. Thus, relying solely on structural information may not be sufficient to detect all the smells accurately.**

We propose an approach to detect five different code smells, namely Divergent Change, Shotgun Surgery, Parallel Inheritance, Blob, and Feature Envy, by exploiting change history information mined from versioning systems. We applied approach, coined as HIST (Historical Information for Smell deTection), to eight software projects written in Java, and wherever possible compared with existing state-of-the-art smell detectors based on source code analysis. The results indicate that HIST's precision ranges between 61% and 80%, and its recall ranges between 61% and 100%. More importantly, the results confirm that HIST is able to identify code smells that cannot be identified through approaches solely based on code analysis.

***Index Terms*—Code Smells; Change History Information.**

I. Introduction

Code smells have been defined by Fowler [1] as symptoms of poor design and implementation choices. In some cases, such symptoms may originate by activities performed by developers while in a hurry, e.g., implementing urgent patches or simply making suboptimal choices. In other cases, smells come from some recurring, poor design solutions, also known as anti-patterns. Previous studies have found that code smells hinder comprehensibility [2], and possibly increase change- and fault-proneness [3], [4]. In summary, these smells need to be carefully detected and monitored and, whenever necessary, refactoring actions should be planned and performed to deal with them.

There exist a number of approaches that detect smells in source code to alert developers of their presence [5], [6], [7]. These approaches rely on structural information extracted from source code, for example, by means of constraints defined on some source code metrics. For instance, according to some existing approaches, such as *DECOR* [5], *LongMethod* or *LargeClass* smells are based on the size of the source code component in terms of LOC, whereas other smells like *ComplexClass* are based on the McCabe cyclomatic complexity [8]. Other smells, such as *Blob*, might use more complex rules.

Although existing approaches demonstrate good performances in terms of precision and recall, they still might not be adequate to detect many of the smells described by Fowler [1]. In particular, there are some smells that, rather than being characterized by source code metrics or other information extracted from a source code snapshot, are *intrinsically characterized by how source code changes over time*. For example, a *Parallel Inheritance* means that two or more class hierarchies evolve by adding code to both classes at the same time. Also, there are smells that are traditionally detected using structural information, where historical information can aid in capturing complementary, additional useful properties. For example, a *Feature Envy* may manifest itself when a method of a class tends to change more frequently with methods of other classes than with those of the same class.

Based on such considerations, we propose an approach, named as HIST (**H**istorical **I**nformation for **S**mell de**T**ection), to detect source code smells based on change history information extracted from versioning systems, and, specifically, by analyzing co-changes occurring between source code artifacts. Specifically, HIST is able to detect five smells from Fowler [1] and Brown [9] catalogues. Three of them—*Divergent Change*, *Shotgun Surgery*, and *Parallel Inheritance*—are symptoms that can be intrinsically observed from the project's history. For the remaining two—*Blob* and *Feature Envy*—there exist static detection approaches [5], [6]. However, as explained for the *Feature Envy*, those smells can also be characterized and possibly detected using source code change history.

In the past, historical information has been used in the context of smell analysis for the purpose of assessing to what extent code smells remained in the system for a substantial amount of time [10], [11]. However, to the best of our knowledge, the use of historical information for the detection of smells remains a premiere of this paper.

We have evaluated HIST on the change history of eight Java projects, namely Apache Ant and Tomcat, jEdit, and five different Android API projects. The study aims at evaluating HIST performances in terms of precision and recall against a manually-produced oracle. Furthermore, wherever possible, we compared HIST with results produced by structural smell detectors, such as JDeodorant [6], [12] and our re-implementation of the *DECOR's* [5] detection rules. The results of our study indicate that HIST's precision is between

978-1-4799-0215-6/13 © 2013 IEEE

Accepted for publication by IEEE. © 2013 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

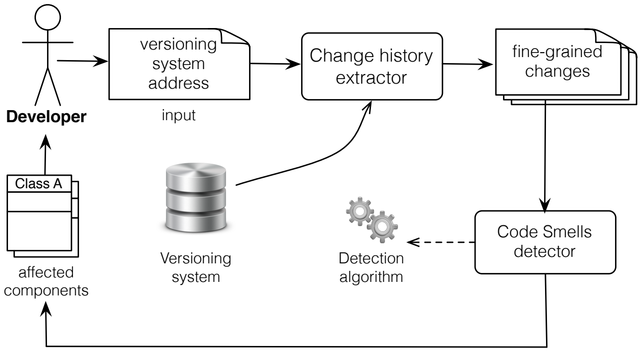


Fig. 1. HIST: The proposed code smell detection process.

61% and 80%, and its recall is between 61% and 100%. When comparing HIST to alternative approaches, we observe that HIST tends to provide better performances, especially in terms of recall, since it is able to identify smells that other approaches omit, because they do not consider historical information. Also, for some smells, we observe a strong complementarity of the approaches based on code analysis with respect to HIST, suggesting that even better performances can be achieved by combining these two orthogonal sources of information.

Paper organization. Section II presents the proposed approach HIST. Section III describes the design of the case study aimed at evaluating HIST. The results are reported and discussed in Section IV, while Section V discusses the threats that could affect the validity of our study. Section VI discusses the related literature, while Section VII summarizes our observations and outlines directions for future work.

II. HIST: Historical Information for Smell Detection

The key idea behind HIST is to identify classes affected by code smells via change history information derived from version control systems. Fig. 1 overviews the main steps of the proposed approach. First, HIST extracts information needed to detect the smells from the versioning system through a component called *Change history extractor*. This information—together with a specific detection algorithm for a particular code smell—is then provided as an input to the *Code smell detector* for computing the list of code components (i.e., methods/classes) affected by the characterized code smells.

The *Code smell detector* uses different detection heuristics for identifying target code smells. In this paper, we have instantiated HIST for detecting five different smells:

- *Divergent Change*: this smell occurs when a class is changed in different ways for different reasons. The example reported by Fowler in his book on refactoring [1] helps to understand this smell: *If you look at a class and say, "Well, I will have to change these three methods every time I get a new database; I have to change these four methods every time there is a new financial instrument", you likely have a situation in which two classes are better than one* [1]. Thus, this type of smell clearly triggers *Extract Class* refactoring opportunities[1].
- *Shotgun Surgery*: a class is affected by this smell when a change to this class (i.e., to one of its fields/methods) triggers many little changes to several other classes [1]. The presence of a Shotgun Surgery smell can be removed through a *Move Method/Field* refactoring.
- *Parallel Inheritance*: this smell occurs when *"every time you make a subclass of one class, you also have to make a subclass of another"* [1]. This could be symptom of something wrong in the class hierarchy that can be corrected by redistributing responsibilities among the classes through different refactoring operations, e.g., *Extract Subclass*.
- *Blob*: a class implementing several responsibilities, having a large number of attributes, operations, and dependencies with data classes [13]. The obvious way to remove this smell is to use *Extract Class* refactoring.
- *Feature Envy*: as defined by Fowler [1], this smell occurs when *"a method is more interested in a class other than the one it is actually in"*. For instance, there can be a method that frequently invokes accessor methods of another class to use its data. This smell can be removed via *Move Method* refactoring operations.

Our choice of instantiating the proposed approach on these smells is not random but driven by the need to have a benchmark including smells that can be naturally identified using change history information and smells that do not necessarily require this type of information. The first three code smells, namely *Divergent Change*, *Shotgun Surgery*, and *Parallel Inheritance*, are by definition historical smells, that is, their definition inherently suggests that they can be detected using revision history. Instead, the last two code smells (*Blob* and *Feature Envy*) can be detected solely relying on structural information, and several approaches based on static source code analysis have been proposed to detect them [5], [6]. Thus, we can compare HIST directly to these code analysis based approaches for detecting *Blob* and *Feature Envy* smells to assess to what extent change history data might be of some value in the detection also of these types of smells. In addition, these two code smells are among the most studied smells in the literature [2], [3], [4]. Thus, considering smells that can be naturally identified through change history information and smells that can also be identified without using this type of information represent a good benchmark for providing a practical indication on the performances of HIST.

The following subsections detail how HIST extracts change history information from versioning systems and then uses it for detecting the above smells.

A. Change History Extraction

The first operation performed by the *Change history extractor* is to mine the versioning system log, reporting the entire

[1] Further details about refactoring operations existing in literature can be found in the refactoring catalog available at http://refactoring.com/catalog/

change history of the system under analysis. This can be done for a range of versioning systems, such as SVN, CVS, or git. However, the logs extracted through this operation report code changes only at file level of granularity. Such a granularity level is not sufficient to detect most of the code smells defined in literature. In fact, many of them describe method-level behavior (see, for instance, *Feature Envy* or *Divergent Change*). To extract fine-grained changes, the *Change history extractor* includes a code analyzer component that is developed in the context of the Markos European project[2]. We use this component to capture changes at method level granularity. In particular, for each pair of subsequent source code snapshots extracted from the versioning system, the code analyzer (i) checks out the two snapshots in two separate folders and (ii) compares the source code of these two snapshots, producing the set of changes performed between them. The set of changes includes: (i) classes added/removed/moved/renamed, (ii) class attributes added/removed/moved/renamed, (iii) methods added/removed/moved/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.

B. Code Smells Detection

The set of fine-grained changes computed by the *Change history extractor* is provided as an input to the *Code Smell detector*, that identifies the list of code components (if any) affected by specific code smells. While the exploited underlying information is the same for all target code smells (i.e., the change history information), HIST uses custom detection heuristics for each code smell. Note that, since HIST relies on the analysis of change history information, it is possible that a class/method that behaved as affected by a code smell in the past does not exist in the current version of the system (e.g., it has been refactored by the developers). Thus, once HIST identifies a component that is affected by a code smell, HIST checks the presence of this component in the current version of the system under analysis before presenting the results to the user. If the component does not exist anymore, HIST removes it from the list of components affected by code smells.

In the following we describe the heuristics we devised for detecting the different kinds of smells described above, while the process for calibrating the heuristic parameters is described in Section III-B.

Divergent Change Detection. Given the definition of this smell provided by Fowler [1], our conjecture is that *classes affected by Divergent Change present different sets of methods each one containing methods changing together but independently from methods in the other sets*. The *Code Smell detector* mines association rules [14] for detecting subsets of methods in the same class that often change together. Association rule discovery is an unsupervised learning technique used for local pattern detection highlighting attribute value conditions that occur together in a given dataset [14]. In our approach, the dataset is composed of a sequence of change sets—e.g., methods—that have been committed (changed) together in a version control repository [15]. 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 [14], defined as:

$$\mathit{Support} = \frac{|M_{left} \cup M_{right}|}{T} \quad \mathit{Confidence} = \frac{|M_{left} \cup M_{right}|}{|M_{left}|}$$

where T is the total number of change sets extracted from the repository. In this paper, we perform association rule mining using a well-known algorithm, namely *Apriori* [14]. Note that, minimum *Support* and *Confidence* to consider an association rule as valid can be set in the *Apriori* algorithm. We empirically calibrated these two parameters in Section III-B. Once HIST detects these change rules between methods of the same class, our approach identifies classes affected by *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.

Shotgun Surgery Detection. To define the detection strategy for this smell, we exploited the following conjecture: *a class affected by Shotgun Surgery contains at least one method changing together with several other methods contained in other classes*. Also in this case, the *Code Smell detector* uses association rules for detecting methods (in this case methods from different classes) often changing together. Hence, a class is identified as affected by a *Shotgun Surgery* smell if it contains at least one method that changes with methods present in more than δ different classes.

Parallel Inheritance Detection. Two classes are affected by *Parallel Inheritance* smell if *"every time you make a subclass of one class, you also have to make a subclass of the other"* [1]. Thus, the *Code Smell detector* identifies the pairs of classes for which the addition of a subclass for one class implies the addition of a subclass for the other class using generated association rules. These pairs of classes are candidates to be affected by the *Parallel Inheritance* smell.

Blob Detection. A *Blob* is a class that centralizes most of the system's behavior and has dependencies towards data classes [13]. Thus, our conjecture is 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*. Given this conjecture, Blobs are identified as classes modified (in any way) in more than $\alpha\%$ of commits

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

TABLE I
CHARACTERISTICS OF THE SOFTWARE SYSTEMS USED IN THE STUDY.

Project	Period	#Classes	KLOC
Apache Ant	Jan 2000-Jan 2013	44-1,224	8-220
Apache Tomcat	Mar 2006-Jan 2013	828-1,548	254-350
jEdit	Sep 2001-July 2010	279-544	85-175
Andr. API (framework-opt-telephony)	Aug 2011-Jan 2013	218-225	73-78
Andr. API (frameworks-base)	Oct 2008-Jan 2013	1,698-3,710	534-1,043
Andr. API (frameworks-support)	Feb 2011-Nov 2012	199-256	58-61
Andr. API (sdk)	Oct 2008-Jan 2013	132-315	14-82
Andr. API (tool-base)	Nov 2012-Jan 2013	471-714	80-134

TABLE II
SNAPSHOTS CONSIDERED FOR THE SMELLS DETECTION.

Project	git snapshot	Date	Classes	KLOC
Apache Ant	da641025	Jun 2006	846	173
Apache Tomcat	398ca7ee	Jun 2010	1,284	336
jEdit	feb608el	Aug 2005	316	101
Andr. API (framework-opt-telephony)	b3a03455	Feb 2012	223	75
Andr. API (frameworks-base)	b4ff35df	Nov 2011	2,766	770
Andr. API (frameworks-support)	0f6f72e1	Jun 2012	246	59
Andr. API (sdk)	6feca9ac	Nov 2011	268	54
Andr. API (tool-base)	cfebaa9b	Dec 2012	532	119

involving at least another class. This last condition is used to better reflect the nature of the *Blob* classes that are expected to change despite the type of change being applied (i.e., the set of modified classes).

Feature Envy Detection. Our goal here is to identify methods placed in the wrong class or, in other words, methods having an envied class which they should be moved in. Thus, our 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, our approach identifies methods affected by this smell as those involved in commits with methods of another class of the system $\beta\%$ more than in commits with methods of their class.

III. EMPIRICAL STUDY DEFINITION AND DESIGN

The *goal* of the study is to evaluate HIST, with the *purpose* of analyzing its effectiveness in detecting code smells in software systems. The *quality focus* is on the detection accuracy and completeness as compared to the approaches based on static code analysis, while the *perspective* is of researchers, who want to evaluate the effectiveness of historical information in identifying code smells to build better recommenders for developers.

The *context* of the study consists of eight software projects, namely Apache Ant[3], Apache Tomcat[4], jEdit[5], and five projects belonging to the Android APIs[6]. Apache Ant is a build tool and library specifically conceived for Java applications (though it can be used for other purposes). Apache Tomcat is a Web container allowing the execution of Java Servlets and Java Server Pages (JSP) web applications. jEdit is a text editor for programmers that provides syntax highlighting and native support for over 130 file formats. As for the remaining five software projects, they are responsible of implementing parts of the Android APIs. For example, framework-opt-telephony provides APIs for developers of Android apps allowing them to access services such as texting. Table I reports the characteristics of the analyzed systems, namely the software history that we investigated, and the size range (in terms of KLOC and # of classes).

A. Research Questions, Data Analysis and Metrics

Our study aims at addressing the following two research questions:

- **RQ$_1$:** *How does HIST perform in detecting code smells?* This research question aims at quantifying the performances of HIST in detecting instances of the five smells described in Section II, namely *Divergent Change*, *Shotgun Surgery*, *Parallel Inheritance*, *Blob*, and *Feature Envy*.
- **RQ$_2$:** *How does HIST compare to the techniques based on static code analysis?* This research question assesses the performances of HIST in detecting the five above mentioned smells by comparing it with the performances achieved by applying a more conventional approach based on static source code analysis. The results of this comparison will provide insights on the usefulness of historical information while detecting code smells.

To answer **RQ$_1$** we simulated the use of HIST in a real-usage scenario. In particular, we split the history of eight subject systems in two equal parts, and run our tool on the corresponding snapshot. For instance, given the history of Apache Ant going from January 2000 to January 2013, we selected a system snapshot from June 2006. This was done aiming at simulating a developer performing code smell detection on an evolving software system. In fact, considering some early snapshot in the project history, there was the risk to perform code smell detection on a very unstable snapshot, still presenting ongoing design decisions. On the other side, by considering snapshots occurring later in the project history (e.g., the last available release) there was the risk of replicating some unrealistic scenario, i.e., developers putting effort in improving the design of a software system when its development is almost motionless. In fact, as shown in Table I, for some of the considered software systems change activities have been stopped some time ago (see for instance jEdit). The list of selected snapshots is reported in Table II together with their characteristics.

To evaluate the detection performances of HIST, we need an oracle reporting the instances of code smells in the considered systems' snapshots. Unfortunately, since there are no annotated sets of such smells available in literature, we had to manually build our own oracle. A Master's student from the University of Salerno manually identified instances of the five considered smells in each of the systems' snapshots. Starting from the definition of the five smells reported in literature, the student manually analyzed each snapshot's source code looking for instances of those smells. Clearly, for smells having an intrinsic historical nature, he analyzed the changes performed by developers on different code components. A second Master's student validated the produced oracle, to

[3] http://ant.apache.org/

[4] http://tomcat.apache.org/

[5] http://www.jedit.org/

[6] https://android.googlesource.com/

verify that all affected code components identified by the first student were correct[7]. Note that, while this does not ensure that the defined oracle is complete (i.e., it includes all affected components in the systems), it provides a certain degree of confidence about the correctness of the identified smell instances. To avoid any bias in the experiment, students were not aware of the experimental goals and of the way that HIST identifies code smells.

Once we defined the oracle and obtained the set of smells detected by HIST on each of the systems' snapshots, we evaluated its performances by using two widely-adopted Information Retrieval (IR) metrics, namely recall and precision [16]:

$$recall = \frac{|cor \cap det|}{|cor|}\% \qquad precision = \frac{|cor \cap det|}{|det|}\%$$

where cor and det represent the set of true positive smells (those manually identified) and the set of code smells detected by HIST, respectively. As an aggregate indicator of precision and recall, we report the F-measure, defined as the harmonic mean of precision and recall:

$$F\text{-}measure = 2 * \frac{precision * recall}{precision + recall}\%$$

To answer **RQ**$_2$, we executed the smell detection techniques based on static analysis of the source code on the same systems' snapshots previously used for HIST when addressing **RQ**$_1$. To the best of our knowledge, we are not aware of any approaches detecting all the smells that we considered in our study. For this reason, for different code smells we considered different competitive techniques to compare our approach with. As for the *Blob*, we compared HIST with DECOR, the detection technique proposed by Moha *et al.* [5]. Specifically, we implemented the detection rules used by DECOR for the detection of *Blob*. Such rules are available online[8]. For the *Feature Envy* we considered JDeodorant as a competitive technique [6], which is an Eclipse plug-in publicly available[9]. The approach implemented in JDeodorant analyzes all methods of a given system, and forms a set of candidate target classes where a method should be moved in. This set is obtained by examining the entities (i.e., attributes and methods) that a method accesses from the other classes.

Concerning the remaining three code smells, we are not aware of publicly available tools in literature to detect them. Thus, to have a meaningful baseline for HIST, we implemented three detection algorithms based on static code analysis and/or quality metrics. Note that such an analysis was not intended to provide evidence that HIST is the best method for detecting considered smells. Instead, the goal was to provide some insight into the actual effectiveness of historical information while detecting code smells as compared to information extracted by static analysis of source code.

To detect classes affected by *Divergent Change*, we implemented an algorithm (from now on coined as DCCA) based on the Connectivity metric [17]. Connectivity is a class cohesion metric defined in the interval $[0 \ldots 1]$, and computed as the number of method pairs in a class sharing an instance variable or having a method call among them, divided by the total number of method pairs in the class. Our conjecture is that if a class has low values for the Connectivity measure, it is likely to contain unrelated subsets of methods that most likely change divergently during the software history. DCCA retrieves those classes affected by *Divergent Change*, which have Connectivity lower than a threshold λ. As well as for HIST parameters, we also empirically calibrated λ, as reported in Section III-B.

As for the *Shotgun Surgery*, we analyzed method calls among classes. The algorithm (named as SSCA) detects all such classes containing at least one method invoking methods of at least η external classes. The conjecture is that if you are changing this method, it is likely that you also have to change methods in other classes.

Finally, we detect classes affected by *Parallel Inheritance* as pairs of classes having (i) both a superclass and/or a subclass (i.e., both belonging to a class hierarchy), and (ii) the same prefix in the class name. This detection algorithm (named as PICA) directly comes from the Fowler's definition of *Parallel Inheritance*: *"You can recognize this smell because the prefixes of the class names in one hierarchy are the same as the prefixes in another hierarchy."* [1].

To compare the performances of HIST with those of the above-mentioned code analysis detection techniques we used recall, precision, and F-measures. Moreover, to provide a further comparison of HIST with the experimented code analysis techniques we computed the following overlap metrics:

$$correct_{m_i \cap m_j} = \frac{|correct_{m_i} \cap correct_{m_j}|}{|correct_{m_i} \cup correct_{m_j}|}\%$$

$$correct_{m_i \setminus m_j} = \frac{|correct_{m_i} \setminus correct_{m_j}|}{|correct_{m_i} \cup correct_{m_j}|}\%$$

where $correct_{m_i}$ represents the set of correct code smells detected by method m_i, $correct_{m_i \cap m_j}$ measures the overlap between the set of true code smells detected by both methods m_i and m_j, and $correct_{m_i \setminus m_j}$ measures the true smells detected by m_i only and missed by m_j. The latter metric provides an indication on how a code smell detection strategy contributes to enriching the set of correct code smells identified by another method. This information can be used to analyze the complementarity of static code information and historical information when performing code smell detection.

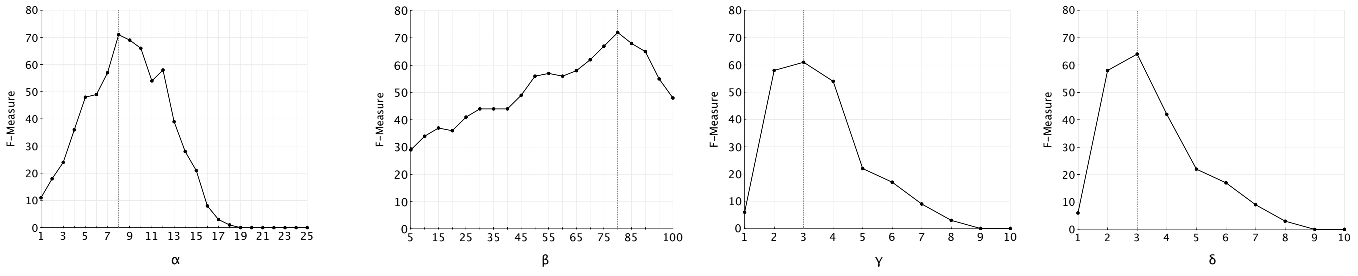
B. Parameters Calibration

While for JDeodorant and DECOR the parameter tuning has already been empirically assessed by their respective authors, to run HIST and the three code analysis detection algorithms described above (DCCA, SSCA, and PICA) we needed to calibrate their parameters. We performed this calibration on a software system which was not used in our experimentation, i.e.,

[7] Only one of the smells identified by the first student was classified as false positive by the second student, and classified as such after reaching a consensus.

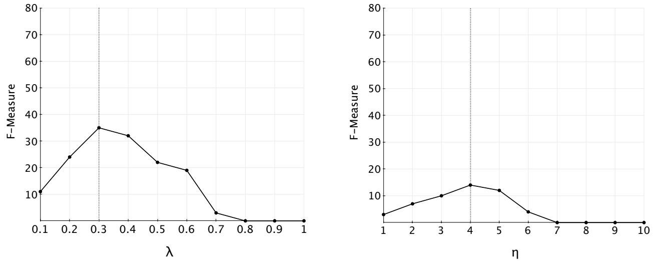
[8] http://www.ptidej.net/research/designsmells/grammar/Blob.htm/file_view

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



(a) (b) (c) (d)

Fig. 2. Parameters calibration for HIST (Blob) α (a), HIST (Feature Envy) β (b), HIST (Divergent Change) γ (c), and HIST (Shotgun Surgery) δ (d).



(a) (b)

Fig. 3. Parameters calibration for DCCA λ (a), and SSCA η (b).

TABLE III
PARAMETERS' CALIBRATION

Technique	Parameter	Experimented Values	Best Value
HIST (Assoc. Rules)	*Support*	From 0.004 to 0.04 by steps of 0.004	0.008
HIST (Assoc. Rules)	*Confidence*	From 0.60 to 0.90 by steps of 0.05	0.70
HIST (Blob)	α	From 1% to 25% by steps of 1%	8%
HIST (Feature Envy)	β	From 5% to 100% by steps of 5%	80%
HIST (Divergent Change)	γ	From 1 to 10 by steps of 1	3
HIST (Shotgun Surgery)	δ	From 1 to 10 by steps of 1	3
DCCA	λ	From 0 to 1 by steps of 0.1	0.3
SSCA	η	From 1 to 10 by steps of 1	4

Apache Xerces[10]. Also on this system, we manually identified instances of five considered code smells and then evaluated F-measure value achieved by the detection approaches using different settings.

Table III reports the values for each parameter that we experimented with and the values that achieved the best results (that is the one that we used in answering the research questions). Results of the calibration are reported in Fig. 2 for the HIST parameters α, β, γ, and δ, and in Fig. 3 for the DCCA λ and the SSCA η parameters. As for the confidence and support, the calibration was not different from what was done in other work using association rule discovery [15], [18], [19], [20].

C. Replication Package

All the data used in our study are publicly available[11]. In the replication package we provide: (i) links to the git software repositories from which we extracted historical information; (ii) complete information on the change history in all the subject systems; (iii) the oracle used on each system; and (iv) the code smells identified by HIST as well as by the competitive approaches.

IV. Analysis of the Results

This section reports the results of our study, with the aim of addressing the research questions formulated in Section III-A. Note that to avoid redundancies, we report the results for both research questions together, discussing each smell separately.

Table IV reports the results—in terms of recall, precision, and F-measure—achieved by HIST and by the approaches based on static code analysis on the eight subject systems. As explained in Section III-A for *Divergent Change*, *Shotgun Surgery*, and *Parallel Inheritance* we used alternative code analysis approaches that we developed (DCCA, SSCA, PICA), whilst for *Blob* and *Feature Envy* we used DECOR rules [5] and the JDeodorant tool [12], respectively. When no instances of a particular code smell were present in the oracle (i.e., zero in the column "Affected Components"), it was not possible to compute the recall (that is, division by zero). In these cases a "-" is indicated in the corresponding code smell row. Similarly, when an approach did not retrieve any instances of code components affected by a particular smell, it was not possible to compute the precision (a "N/A"is included in the code smell row). For each code smell, we also report the results achieved by considering all systems as a single dataset (rows "Overall"). In addition, Table V reports values concerning overlap and differences between HIST and the code analysis techniques: column "HIST $\cap$ CA Tech." reports the percentage of correct code smells identified by both HIST and code analysis technique; column "HIST $\setminus$ CA Tech." reports the percentage of correct code smells identified by HIST but not by the code analysis technique; column "CA Tech. $\setminus$ HIST" reports the percentage of correct code smells identified by the code analysis technique but not by HIST. In the following, we discuss the results for each kind of smell.

Divergent Change. We identified 14 instances of *Divergent Change* in the five systems. The results clearly indicate that the use of historical information allows HIST to outperform DCCA (i.e., the approach based on static code analysis). Specifically, the F-measure on the overall dataset of HIST is 76% (79% of recall and 73% of precision) against 10% (7% of recall and 20% of precision) achieved by DCCA. This is an

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

[11] http://www.rcost.unisannio.it/mdipenta/papers/ase2013/

TABLE IV
HIST PERFORMANCES AS COMPARED TO THE STATIC CODE ANALYSIS TECHNIQUES/TOOLS.

Code Smell	Project	Affected	HIST			Code analysis techniques		
		Components	Precision	Recall	F-measure	Precision	Recall	F-measure
Divergent Change	Apache Ant	0	-	-	-	-	-	-
	Apache Tomcat	5	50%	60%	55%	0%	0%	0%
	jEdit	4	100%	75%	86%	100%	25%	40%
	Android API (framework-opt-telephony)	0	-	-	-	-	-	-
	Android API (frameworks-base)	3	100%	100%	100%	0%	0%	0%
	Android API (frameworks-support)	1	100%	100%	100%	0%	0%	0%
	Android API (sdk)	1	100%	100%	100%	0%	0%	0%
	Android API (tool-base)	0	-	-	-	-	-	-
	Overall	**14**	**73%**	**79%**	**76%**	**20%**	**7%**	**10%**
Shotgun Surgery	Apache Ant	0	-	-	-	-	-	-
	Apache Tomcat	1	100%	100%	100%	0%	0%	0%
	jEdit	1	100%	100%	100%	0%	0%	0%
	Android API (framework-opt-telephony)	0	-	-	-	-	-	-
	Android API (frameworks-base)	1	100%	100%	100%	0%	0%	0%
	Android API (frameworks-support)	1	100%	100%	100%	0%	0%	0%
	Android API (sdk)	0	-	-	-	-	-	-
	Android API (tool-base)	0	-	-	-	-	-	-
	Overall	**4**	**100%**	**100%**	**100%**	**0%**	**0%**	**0%**
Parallel Inheritance	Apache Ant	7	63%	71%	67%	8%	57%	14%
	Apache Tomcat	9	60%	67%	63%	7%	44%	12%
	jEdit	3	N/A	N/A	N/A	2%	100%	4%
	Android API (framework-opt-telephony)	0	-	-	-	-	-	-
	Android API (frameworks-base)	3	N/A	N/A	N/A	N/A	N/A	N/A
	Android API (frameworks-support)	0	-	-	-	-	-	-
	Android API (sdk)	9	67%	89%	76%	5%	33%	12%
	Android API (tool-base)	0	-	-	-	-	-	-
	Overall	**31**	**61%**	**61%**	**61%**	**4%**	**45%**	**7%**
Blob	Apache Ant	8	60%	75%	67%	30%	38%	34%
	Apache Tomcat	5	100%	20%	33%	67%	80%	73%
	jEdit	5	67%	40%	50%	60%	60%	60%
	Android API (framework-opt-telephony)	13	100%	77%	87%	70%	54%	61%
	Android API (frameworks-base)	18	70%	50%	58%	65%	50%	57%
	Android API (frameworks-support)	5	71%	100%	83%	38%	60%	49%
	Android API (sdk)	10	86%	60%	71%	29%	20%	24%
	Android API (tool-base)	0	-	-	-	-	-	-
	Overall	**64**	**76%**	**61%**	**68%**	**52%**	**49%**	**50%**
Feature Envy	Apache Ant	9	78%	78%	78%	15%	25%	19%
	Apache Tomcat	3	100%	33%	50%	67%	67%	67%
	jEdit	10	100%	100%	100%	100%	27%	43%
	Android API (framework-opt-telephony)	0	-	-	-	-	-	-
	Android API (frameworks-base)	17	63%	88%	73%	100%	94%	96%
	Android API (frameworks-support)	0	-	-	-	-	-	-
	Android API (sdk)	3	100%	33%	50%	100%	67%	80%
	Android API (tool-base)	0	-	-	-	-	-	-
	Overall	**42**	**71%**	**81%**	**76%**	**68%**	**60%**	**63%**

expected result, since the *Divergent Change* is by definition (see Section II) a "historical smell", and thus we expected difficulties in capturing this kind of smell by just using source code analysis. As shown, DCCA was able to detect only one occurrence of *Divergent Change* on jEdit, missing all other instances. It is interesting to note how the only smell detected by DCCA was not detected by HIST. This smell occurred in the class `RE` of jEdit, having a low value of cohesion as measured by the Connectivity metric used by the static code analysis technique), but not enough historical information about divergent changes to be captured by HIST. This clearly highlights the main limitation of HIST, that requires sufficient amount of historical information to infer useful association rules. Given these observations, the overlap between the smells detected by HIST and DCCA results reported in Table V is quite expected: among the set of correct smells detected by the two techniques, there is no overlap, as HIST retrieves 93% of the smells, while DCCA detects the one described above.

Shotgun Surgery. *Shotgun Surgery* is the code smell with the least number of instances in the subject systems, i.e., with only four systems affected for a total of four instances (one per system). HIST was able to detect all the instances of this smell (100% of recall) with 100% of precision. HIST outperformed SCCA (i.e., the detection approach based on code analysis). Specifically, SCCA was not able to detect any of the three instances of this smell present in the subject systems. Thus, no meaningful observations can be made in terms of overlap metrics. Objectively, this can be due to the limited capabilities of the SCCA detection technique that we formulated. However,

TABLE V
OVERLAP BETWEEN HIST AND CODE ANALYSIS (CA) TECHNIQUES. FOR BLOB THE CA TECH. IS DECOR, FOR FEATURE ENVY IS JDEODORANT.

Code Smell	HIST ∩ CA Tech.	HIST \ CA Tech.	CA Tech. \ HIST
Divergent Change	0%	93%	7%
Shotgun Surgery	0%	100%	0%
Parallel Inheritance	43%	40%	17%
Blob	17%	48%	35%
Feature Envy	39%	41%	20%

we argue that it is quite difficult to identify characteristics of such a smell solely based on code analysis, as the smell is intrinsically defined in terms of a change triggering many other changes [1]. It is worthwhile to discuss an example of *Shotgun Surgery* we identified in Apache Tomcat and represented by the method `isAsync` implemented in the class `AsyncStateMachine`. HIST identified association rules between this method and 48 methods in the system, belonging to 31 different classes. This means that, whenever the `isAsync` method is modified, also these 48 methods, generally, undergo a change.

Parallel Inheritance. Among the 31 instances of the *Parallel Inheritance* smell, HIST was able to correctly identify 19 of them (recall 61%) with a price to pay of 12 false positives, resulting in a precision of 61%. By using the detection rule based on code analysis (i.e., PICA) we were able to retrieve 14 correct instances of the smell (recall of 45%) while also retrieving 336 false positives (precision of 4%). It is interesting to analyze the overlap metrics reported in Table V. Among the set of correct smells identified by the two techniques, there is an overlap of 43%, while 40% of the correct instances are retrieved only by HIST and a remaining 17% is identified only by PICA. This highlights a tangible potential of combining structural and historical information for detecting this type of smell. We plan to further investigate such a combination as part of our future work.

Blob. As for the detection of *Blobs*, HIST is able to achieve a precision of 76% and a recall of 61% (F-measure=68%), while DECOR is able to achieve a precision of 52% and a recall of 49% (F-measure=50%). In more details, HIST achieved better precision values on all the systems (on average, +24%). This clearly results in less effort for a developer looking for instances of code smells in a software system due to the lower number of false positives to discard. Also, HIST ensured a better recall on four out of the seven systems containing at least a *Blob* class, and a tie has been reached on Android framework-base. On the contrary, HIST was outperformed by DECOR on Apache Tomcat and jEdit (see Table IV). However, on the overall dataset, HIST was able to correctly identify 39 of the 64 existing Blobs, against the 31 identified by DECOR. Thus, as also indicated by the F-measure value computed over the whole dataset, the overall performance of HIST is better than that one of DECOR (68% against 50%). Noticeably, the two approaches seem to be highly complementary. This is highlighted by the overlap results in Table V. Among the set of code smells correctly identified by the two techniques, there is an overlap of just 17%. Specifically, HIST is able to detect 48% of smells ignored by DECOR, and the latter retrieves 35% of correct smells not identified by HIST. Similarly to the results for the *Parallel Inheritance* smell, this finding highlights the possibility of building better detection techniques by combining static code analysis and change history information.

An example of *Blob* correctly identified by HIST and missed by DECOR is the class `ELParser` from Apache Tomcat, that underwent changes in 178 out of the 1,976 commits occurred in the analyzed time period. `ELParser` is not retrieved by DECOR because this class has a one-to-one relationship with data classes, while a one-to-many relationship is required by the DECOR detection rule. Instead, a *Blob* retrieved by DECOR and missed by HIST is the class `StandardContext` of Tomcat. While this class exhibits all the structural characteristics of a *Blob* (thus allowing DECOR to detect it), it was not involved in any of the commits (i.e., it was just added and never modified), hence making the detection impossible for HIST.

Feature Envy. For the *Feature Envy* smell, we found instances of this smell in five out of the eight systems, for a total of 42 affected methods. HIST was able to identify 34 of them (recall of 81%) against the 25 identified by JDeodorant (recall of 60%). Also, the precision ensured by HIST is slightly higher than the one achieved by JDeodorant (71% against 68%). However, it is important to remark that JDeodorant is a refactoring tool and, as such, it identifies *Feature Envy* smells in software systems with the sole purpose of suggesting move method refactoring opportunities. Thus, the tool reports the presence of *Feature Envy* smells only if the move method refactoring is possible, by checking some preconditions ensuring that the program behavior does not change after the application of the suggested refactoring [6]. An example of the considered preconditions is that *the envied class does not contain a method having the same signature as the moved method* [6]. To perform a fair comparison (especially in terms of recall), we filtered the *Feature Envy* instances retrieved by our approach by using the same set of preconditions defined by JDeodorant [6]. This resulted in the removal of three correct instances, as well as of three false positives previously retrieved by HIST, thus decreasing the recall from 81% to 74% and increasing the precision from 71% to 74%. Still, HIST achieves better recall and precision values with respect to JDeodorant.

It is interesting to observe that the overlap data reported in Table V highlights, also in this case, some complementarity between the historical and static analysis techniques, with 39% of correct smell instances identified by both techniques (overlap), 41% identified only by HIST, and 20% only by JDeodorant.

An example of correct smell instance identified by HIST only is the method `buildInputMethodListLocked` implemented in the class `InputMethodManagerService` of the Android framework-base API. For this method, HIST identified `WindowManagerService` as the envied class, since there are just three commits in which the method `buildInputMethodListLocked` is co-changed with methods of its class, against the 16 commits in which it is co-changed together with methods belonging to the envied class. Instead, JDeodorant was the only technique able to correctly identify the *Feature Envy* smell present in Apache Ant and affecting the method `isRebuildRequired` of class `WebsphereDeploymentTool`. In this case, the envied class is `Project` and HIST was not able to identify it due to the limited number of observed co-changes.

Summary for RQ$_1$. HIST provided good performances in detecting all code smells considered in our study (F-measure between 61% to 89%). While this result was quite expected on smells which intrinsically require the use of historical information for their detection, it is promising to observe that HIST provided good performances also when detecting *Blob* and *Feature Envy* smells.

Summary for RQ$_2$. HIST was able to outperform static analysis techniques and tools in terms of recall, precision, and F-measure. While such a result is somewhat expected for "intrinsically historical" smells (*Divergent Change*, *Shotgun Surgery*, and *Parallel Inheritance*), noticeably HIST is also able to perform well on other smells (*Blob* and *Feature Envy*) provided that historical information is available. Last, but not least, for *Parallel Inheritance*, *Blob*, and *Feature Envy*, our findings suggest that static code analysis techniques and HIST could be nicely complemented to obtain better performances.

V. Threats to Validity

Threats to *construct validity* concern relationships between theory and observation. This threat is generally due to imprecision in the measurements performed in the study. In the context of our study, this is mainly due to how the oracle was built (see Section III-A). It is important to remark that in order to mitigate the bias in such a task, the people who defined the oracle were not aware of how the HIST approach actually worked. However, we cannot exclude that such manual analysis could have missed some smells, or else identified some false positives. Another threat is due to the terms of comparison. While for *Blob* and *Feature Envy* we compared HIST with existing techniques/tools (DECOR and JDeodorant), this was not possible for the other smells, for which we had to define alternative static detection techniques, that may or may not be the most suitable ones among those based solely on structural information. Last, but not least, note that although we implemented the DECOR rules ourselves, these are precisely defined by the author of such approach[12].

Threats to *internal validity* concern factors that could have influenced our results. In our study, this is mainly due to the calibration of the HIST parameters, as well as of those of the alternative static approaches. We performed the calibration of such parameters on one project (Xerces) not used in our study, by computing F-measure for different possible values of such parameters (see Section III-B).

Threats to *external validity* concern the generalization of the results. HIST only deals with five code smells, while there might be many more left uncovered [1], [13]. However, as explained in Section II we focused on (i) three smells—*Divergent Change*, *Shotgun Surgery*, and *Parallel Inheritance*—that are clearly related to how source code elements evolve over time, rather than to their structural characteristics, and (ii) two smells—*Blob* and *Feature Envy*—whose characteristics can be captured, at least partially, by observing source code changes. However, we cannot exclude that there could be other smells that can be modeled similarly.

We conducted the evaluation on eight Java projects, five of which belong to different Android APIs, while others belonging to different domains. It could be worthwhile to replicate the evaluation on other projects having different evolution histories or different architectures (e.g., plugin-based architecture).

VI. Related Work

All the techniques for detecting code smells in source code have their roots in the definition of code design defects and heuristics for identifying those that are outlined in well-known books: [1], [9], [21], [22]. The first by Webster [21] describes pitfalls in Object-Oriented (OO) development going from the management of a project through the implementation choices, up to the quality insurance policies. The second by Riel [22] defines more than 60 guidelines to rate the integrity of a software design. Fowler [1] defines 22 code smells together with refactoring operations to remove them from the system. Finally, Brown *et al.* [9] describe 40 anti-patterns together with heuristics for detecting them in code.

Starting from the information reported in these books, several techniques have been proposed to detect design defects in source code. Travassos *et al.* [23] define manual inspection rules (called "reading techniques") aimed at identifying design defects that may negatively impact the design of object-oriented systems.

Simon *et al.* [24] provide a metric-based visualization tool able to discover design defects representing refactoring opportunities. For example, to identify *Blobs*, each class is analyzed to verify the structural relationships (i.e., method calls and attribute accesses) among its methods.

van Emden and Moonen [25] present jCOSMO, a code smell browser that detects and visualizes smells in Java source code. They focus their attention on two code smells related to Java programming language, i.e., *instanceof* and *typecast*.

Marinescu [7] proposes a mechanism called "detection strategies" for formulating metric-based rules that capture deviations from good design principles and heuristics. Such strategies are based on identifying *symptoms* characterizing smells and *metrics* to measure such symptoms, and then by defining rules based on thresholds on such metrics. Then, Lanza and Marinescu [26] describe how to exploit quality metrics to identify "disharmony patterns" in code by defining

[12] http://www.ptidej.net/research/designsmells/

a set of thresholds based on the measurement of the exploited metrics in real software systems. Also, Munro [27] presents a metric-based detection technique able of identifying instances of two smells, namely *Lazy Class* and *Temporary Field*, in source code. In particular, a set of thresholds is applied to the measurement of some structural metrics to identify those smells. For example, to retrieve *Lazy Class*, three metrics are used: Number of Methods (NOM), LOC, Weight Methods per Class (WMC), and Coupling Between Objects (CBO).

Khomh *et al.* [28] propose an approach based on Bayesian belief networks to specify and detect smells in programs. The main novelty of that approach is represented by the fact that it provides a likelihood that a code component is affected by a smell, instead of a boolean value like previous techniques. This is also one of the main characteristics of the approach based on quality metrics and B-splines proposed by Oliveto *et al.* [29] for identifying instances of *Blobs* in source code.

Tsantalis *et al.* [6] presents JDeodorant, a tool for detecting *Feature Envy* smells with the aim of suggesting move method refactoring opportunities. In particular, for each method of the system, their approach forms a set of candidate target classes where a method should be moved. This set is obtained by examining the entities (i.e., attributes and methods) that a method accesses from the other classes.

Moha *et al.* [5] introduce DECOR, a method for specifying and detecting code and design smells. DECOR uses a Domain-Specific Language (DSL) for specifying smells using high-level abstractions. Four design smells are identified by DECOR, namely *Blob*, *Swiss Army Knife*, *Functional Decomposition*, and *Spaghetti Code*.

Ratiu *et al.* [10] describe an approach for detecting smells based on evolutionary information of problematic code components (as detected by code analysis) over their life-time. The aim is to measure persistence of the problem and related maintenance effort spent on the suspected components. This is the closest approach to the one defined in this paper, since it discusses the role of historical information for smell detection. However, Ratiu *et al.* do not explicitly use historical information for detecting smells (as done by HIST), but they only perform multiple code analysis measurements of design problems during the history of code components. Historical information have also been used by Lozano *et al.* [11] to assess the impact of code smells on software maintenance.

All previously discussed approaches exploit information extracted from source code—e.g., quality metrics—to detect code smells. To the best of our knowledge, HIST, the approach described in this paper, is the first approach explicitly using change-history information extracted from versioning systems for the identification of code smells in source code.

Finally, it is worthwhile to mention that co-change analysis has been used in the past for other purposes, for example by Ying *et al.* [19], Zimmermann *et al.* [15], Gall *et al.* [30], and Kagdi [20] for identifying logical change couplings, and by Adams *et al.* [31] and Canfora *et al.* [18] for the identification of crosscutting concerns. Although the underlying technique is similar—i.e., based on the identification of code elements that co-change—for our purpose (smell detection) appropriate rules are needed, and as explained in Section II, a fine-grained analysis, identifying co-changes at method-level, is often required.

VII. Conclusion and Future Work

This paper described an approach, named HIST (Historical Information for Smell deTection), for detecting five different code smells by analyzing co-changes extracted from versioning systems. We identified five smells for which historical analysis can be helpful in the detection process: *Divergent Change*, *Shotgun Surgery*, *Parallel Inheritance*, *Blob*, and *Feature Envy*. For each smell we defined a historical detector, using association rule discovery [14] or analyzing the set of classes/methods co-changed with the suspected smell.

We evaluated HIST over a manually-built oracle of smells identified in eight Java open source projects, and compared it with alternative smell detection approaches based solely on static source code analysis, where possible (for *Blob* and *Feature Envy*) available in literature, i.e., DECOR rules [5] and JDeodorant [6], [12]. The results of our study indicate that the approach exhibits a precision between 61% and 80%, and a recall between 61% and 100%. For "intrinsically historical" smells—such as *Divergent Change*, *Shotgun Surgery*, *Parallel Inheritance*—HIST clearly outperforms static code analysis, and generally performs as well as code analysis (if not better) for *Blob* and *Feature Envy* smells. Besides the better performance in terms of precision and recall, the HIST approach has a further advantage: it highlight smells that are subject to frequent changes and therefore be possibly more problematic for the maintainer.

The main limitation of HIST is represented by the need for having sufficient history of observable co-changes, without which the approach falls short. Finally, it is important to remark that in most cases the sets of smells detected by HIST and by code analysis techniques are quite complementary, suggesting that better techniques can be built by combining them.

For the aforementioned reason, our future research agenda includes the development of a hybrid smell detection approach, combining static code analysis with analysis of co-changes. Also, we are planning to investigate the applicability of HIST to other types of smells. Last, but not the least, we will further validate HIST technique on more software systems.

Acknowledgment

The authors would like to thank the students defining the oracle used in the HIST evaluation. Gabriele Bavota and Massimiliano Di Penta are partially funded by the EU FP7-ICT-2011-8 project Markos, contract no. 317743. Denys Poshyvanyk was supported in part by the NSF CCF-1016868 grant. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

REFERENCES

[1] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley, 1999.

[2] 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 *15th European Conference on Software Maintenance and Reengineering, CSMR 2011, 1-4 March 2011, Oldenburg, Germany*. IEEE Computer Society, 2011, pp. 181–190.

[3] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change- and fault-proneness," *Empirical Software Engineering*, vol. 17, no. 3, pp. 243–275, 2012.

[4] F. Khomh, M. Di Penta, and Y.-G. Guéhéneuc, "An exploratory study of the impact of code smells on software change-proneness," in *16th Working Conference on Reverse Engineering, WCRE 2009, 13-16 October 2009, Lille, France*. IEEE Computer Society, 2009, pp. 75–84.

[5] 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*, vol. 36, no. 1, pp. 20–36, 2010.

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

[7] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *20th International Conference on Software Maintenance (ICSM 2004), 11-17 September 2004, Chicago, IL, USA*. IEEE Computer Society, 2004, pp. 350–359.

[8] T. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, 1976.

[9] 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*, 1^{st} ed. John Wiley and Sons, March 1998.

[10] D. Ratiu, S. Ducasse, T. Gîrba, and R. Marinescu, "Using history information to improve design flaws detection," in *8th European Conference on Software Maintenance and Reengineering (CSMR 2004), 24-26 March 2004, Tampere, Finland, Proceeding*. IEEE Computer Society, 2004, pp. 223–232.

[11] A. Lozano, M. Wermelinger, and B. Nuseibeh, "Assessing the impact of bad smells using historical information," in *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*, ser. IWPSE '07. New York, NY, USA: ACM, 2007, pp. 31–34.

[12] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "Jdeodorant: identification and application of extract class refactorings," in *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*. ACM, 2011, pp. 1037–1039.

[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] 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.

[15] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," in *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, 2004, pp. 563–572.

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

[17] L. C. Briand, J. W. Daly, and J. Wüst, "A unified framework for cohesion measurement in object-orientedsystems," *Empirical Software Engineering*, vol. 3, pp. 65–117, July 1998.

[18] G. Canfora, L. Cerulo, and M. Di Penta, "On the use of line co-change for identifying crosscutting concern code," in *22nd IEEE International Conference on Software Maintenance (ICSM 2006), 24-27 September 2006, Philadelphia, Pennsylvania, USA*. IEEE Computer Society, 2006, pp. 213–222.

[19] 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*, vol. 30, no. 9, pp. 574–586, 2004.

[20] H. Kagdi, M. Gethers, D. Poshyvanyk, and M. Collard, "Blending conceptual and evolutionary couplings to support change impact analysis in source code," in *Reverse Engineering (WCRE), 2010 17th Working Conference on*, 2010, pp. 119–128.

[21] B. F. Webster, *Pitfalls of Object Oriented Development*, 1^{st} ed. M & T Books, February 1995.

[22] A. J. Riel, *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.

[23] G. Travassos, F. Shull, M. Fredericks, and V. R. Basili, "Detecting defects in object-oriented designs: using reading techniques to increase software quality," in *Proceedings of the 14^{th} Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, 1999, pp. 47–56.

[24] F. Simon, F. Steinbr, and C. Lewerentz, "Metrics based refactoring," in *Proceedings of 5th European Conference on Software Maintenance and Reengineering*. Lisbon, Portugal: IEEE CS Press, 2001, pp. 30–38.

[25] E. van Emden and L. Moonen, "Java quality assurance by detecting code smells," in *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE'02)*. IEEE CS Press, Oct. 2002.

[26] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer, 2006.

[27] M. J. Munro, "Product metrics for automatic identification of "bad smell" design problems in java source-code," in *Proceedings of the 11^{th} International Software Metrics Symposium*. IEEE Computer Society Press, September 2005.

[28] 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*. Hong Kong, China: IEEE CS Press, 2009, pp. 305–314.

[29] R. Oliveto, F. Khomh, G. Antoniol, and Y.-G. Guéhéneuc, "Numerical signatures of antipatterns: An approach based on b-splines," in *Proceedings of the 14^{th} Conference on Software Maintenance and Reengineering*, R. Capilla, R. Ferenc, and J. C. Dueas, Eds. IEEE Computer Society Press, March 2010.

[30] H. Gall, K. Hajek, and M. Jazayeri, "Detection of logical coupling based on product release history," in *Proceedings of 14th IEEE International Conference on Software Maintenance*, 1998, pp. 190–198.

[31] B. Adams, Z. M. Jiang, and A. E. Hassan, "Identifying crosscutting concerns using historical code changes," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*. ACM, 2010, pp. 305–314.