Mining Version Histories for Detecting Code Smells

Fabio Palomba, *Student Member, IEEE*, Gabriele Bavota, *Member, IEEE Computer Society*,
Massimiliano Di Penta, Rocco Oliveto, *Member, IEEE Computer Society*,
Denys Poshyvanyk, *Member, IEEE Computer Society*, and Andrea De Lucia, *Senior Member, IEEE*

Abstract—Code smells are symptoms of poor design and implementation choices that may hinder code comprehension, and possibly increase change- and fault-proneness. While most of the detection techniques just rely on structural information, many code smells are intrinsically characterized by how code elements change over time. In this paper, we propose *H*istorical *I*nformation for *S*mell de*T*ection (HIST), an approach exploiting change history information to detect instances of five different code smells, namely Divergent Change, Shotgun Surgery, Parallel Inheritance, Blob, and Feature Envy. We evaluate HIST in two empirical studies. The first, conducted on 20 open source projects, aimed at assessing the accuracy of HIST in detecting instances of the code smells mentioned above. The results indicate that the precision of HIST ranges between 72 and 86 percent, and its recall ranges between 58 and 100 percent. Also, results of the first study indicate that HIST is able to identify code smells that cannot be identified by competitive approaches solely based on code analysis of a single system's snapshot. Then, we conducted a second study aimed at investigating to what extent the code smells detected by HIST (and by competitive code analysis techniques) reflect developers' perception of poor design and implementation choices. We involved 12 developers of four open source projects that recognized more than 75 percent of the code smell instances identified by HIST as actual design/implementation problems.

Index Terms—Code smells, mining software repositories, empirical studies

<hr>

1 INTRODUCTION

CODE smells have been defined by Fowler [14] as symptoms of poor design and implementation choices. In some cases, such symptoms may originate from 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 [10]. For example a *Blob* is a large and complex class that centralizes the behavior of a portion of a system and only uses other classes as data holders. *Blob* classes can rapidly grow out of control, making it harder and harder for developers to understand them, to fix bugs, and to add new features.

Previous studies have found that smells hinder comprehension [1], and possibly increase change- and fault-proneness [23], [24]. In summary, 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 for detecting smells in source code to alert developers of their presence [30], [33], [48]. 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* [33], *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 [32]. Other smells, such as *Blob*, might use more complex rules.

Although existing approaches exhibit good detection accuracy, they still might not be adequate for detecting many of the smells described by Fowler [14]. In particular, there are some smells that, rather than being characterized by source code metrics or other information extracted from source code snapshots, 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, additionally 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 rather than with those of the same class.

Based on such considerations, we propose an approach, named *H*istorical *I*nformation for *S*mell de*T*ection (HIST), to detect smells based on change history information mined from versioning systems, and, specifically, by analyzing co-changes occurring between source code artifacts. HIST is

- F. Palomba and A. De Lucia are with the University of Salerno, Fisciano (SA), Italy. E-mail: {fpalomba, adelucia}@unisa.it.
- G. Bavota is with the Free University of Bozen-Bolzano, Bolzano, Italy. E-mail: gabriele.bavota@unibz.it.
- M. Di Penta is with the University of Sannio, Benevento, Italy. E-mail: dipenta@unisannio.it.
- R. Oliveto is with the University of Molise, Pesche (IS), Italy. E-mail: rocco.oliveto@unimol.it.
- D. Poshyvanyk is with the College of William and Mary, Williamsburg, VA. E-mail: denys@cs.wm.edu.

aimed at detecting five smells from Fowler [14] and Brown [9] catalogues.[1] Three of them—*Divergent Change, Shotgun Surgery*, and *Parallel Inheritance*—are symptoms that can be intrinsically observed from the project's history even if a single project snapshot detection approach has been proposed for the detection of *Divergent Change* and *Shotgun Surgery* [42]. For the remaining two—*Blob* and *Feature Envy*—there exist several single project snapshot detection approaches [33], [48]. 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 smells remained in the system for a substantial amount of time [29], [44]. Also, Gîrba et al. [17] exploited formal concept analysis (FCA) for detecting co-change patterns, that can be used to detect some smells. However, to the best of our knowledge, the use of historical information for smell detection remains a premiere of this paper.

We have evaluated HIST in two empirical studies. The first, conducted on 20 Java projects, aimed at evaluating HIST detection accuracy in terms of precision and recall against a manually-produced oracle. Furthermore, wherever possible, we compared HIST with results produced by approaches that detect smells by analyzing a single project snapshot, such as JDeodorant [13], [48] (for the *Feature Envy* smell) and our re-implementations of the *DECOR's* [33] detection rules (for the *Blob* smell) and of the approach by Rao and Raddy [42] (for *Divergent Change* and *Shotgun Surgery*). The results of our study indicate that HIST's precision is between 72 and 86 percent, and its recall is between 58 and 100 percent. When comparing HIST to alternative approaches, we observe that HIST tends to provide better detection accuracy, especially in terms of recall, since it is able to identify smells that other approaches omit. Also, for some smells, we observe a strong complementarity of the approaches based on a single snapshot (SS) analysis with respect to HIST, suggesting that even better performances can be achieved by combining these two complementary sources of information.

Despite the good results achieved in the previous study, it is important to point out that a smell detection technique is actually useful only if it identifies code design problems that are recognized as such by software developers. For this reason we conducted a second study—involving 12 developers of four open source systems—and aimed at investigating to what extent the smells detected by HIST (and by the competitive single snapshot techniques) reflect developers' perception of poor design and implementation choices. Results of this second study highlight that over 75 percent of the smell instances identified by HIST are considered as design/implementation problems by developers, that generally suggest refactoring actions to remove them.

Summarizing, the contributions of this paper are:

1) HIST, a novel approach to identify smells in source code by relying on change history information.
2) A study on 20 systems aimed at assessing the detection accuracy of HIST and of state-of-the-art smell

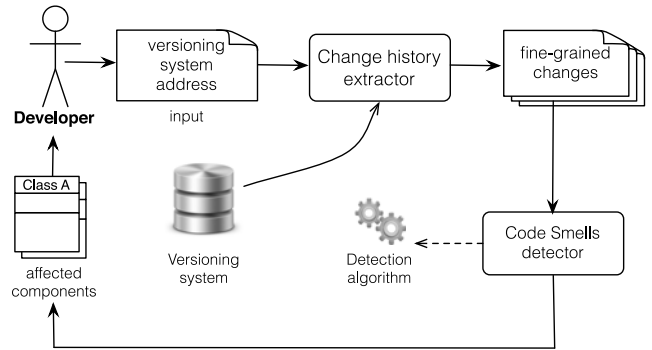1. Definitions of these five smells are provided in Section 2.

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

detection techniques (based on the analysis of a single snapshot) against a manually-produced oracle.
3) A study with 12 developers of four open source systems aimed at understanding to what extent the smells identified by HIST and by state-of-the-art techniques actually represent design/implementation problems from a developer's point of view.
4) A comprehensive replication package [40], including (i) the manually built oracles for the 20 systems, and (ii) the raw data of all our experimentations.

Paper structure. Section 2 presents the proposed approach HIST. Section 3 describes the design and the results of the first case study aimed at evaluating the HIST detection accuracy. The design and the results of the second study are presented in Section 4, while Section 5 discusses the threats that could affect the validity of our empirical evaluation. Section 7 summarizes our observations and outlines directions for future work, after a discussion on the related literature (Section 6).

2 HIST OVERVIEW

The key idea behind HIST is to identify classes affected by smells via change history information derived from version control systems. Fig. 1 overviews the main steps behind the proposed approach. Firstly, HIST extracts information needed to detect smells from the versioning system through a component called *Change history extractor*. This information—together with a specific detection algorithm for a particular 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 smells characterized in the specific detection algorithm.

The *Code smell detector* uses different detection heuristics for identifying target smells. In this paper, we have instantiated HIST for detecting the five smells summarized in Table 1:

- *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 [14] helps understanding 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* [14]. Thus, this type of smell

TABLE 1
Code Smells Detected by HIST

Code Smell	Brief Description
Divergent Change	A class is changed in different ways for different reasons
Shotgun Surgery	A change to the affected class (i.e., to one of its fields/methods) triggers many little changes to several other classes
Parallel Inheritance	Every time you make a subclass of one class, you also have to make a subclass of another
Blob	A class implementing several responsibilities, having a large number of attributes, operations, and dependencies with data classes
Feature Envy	A method is more interested in another class than the one it is actually in

clearly triggers *Extract Class* refactoring opportunities.[2] Indeed, the goal of *Extract Class* refactoring is to split a class implementing different responsibilities into separated classes, each one grouping together methods and attributes related to a specific responsibility. The aim is to (i) obtain smaller classes that are easier to comprehend and thus to maintain and (ii) better isolate the change.

- *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 [14]. The presence of a Shotgun Surgery smell can be removed through a *Move Method/Field* refactoring. In other words, the method/field causing the smell is moved towards the class in which its changes trigger more modifications.
- *Parallel inheritance*. This smell occurs when *"every time you make a subclass of one class, you also have to make a subclass of another"* [14]. This could be symptom of design problems in the class hierarchy that can be solved 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 [9]. The obvious way to remove this smell is to use *Extract Class* refactoring.
- *Feature envy*. As defined by Fowler [14], this smell occurs when *"a method is more interested in another class 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 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 smells (*Blob* and *Feature Envy*) can be detected relying solely on structural information, and several approaches based on static source code analysis of a single system's snapshot have been proposed for detecting those smells [33], [48].

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

2.1 Change History Extraction

The first operation performed by the *Change history extractor* is to mine the versioning system log, reporting the entire 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 at file level of granularity. Such a granularity level is not sufficient to detect most of the smells defined in the literature. In fact, many of them describe method-level behavior (see, for instance, *Feature Envy* or *Divergent Change*).[3] In order 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.[4] 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) added/removed/moved/renamed classes, (ii) added/removed class attributes, (iii) added/removed/ moved/renamed methods, (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.

The code analyzer parses source code by relying on the *srcML* toolkit [12]. To distinguish cases where a method/ class was removed and a new one added from cases when a method/class was moved (and possibly its source code changed), the MARKOS code analyzer uses heuristics that map methods/classes with different names if their source code is similar based on a metric fingerprint similar to the one used in metric-based clone detection [31]. For example, each method is associated with a 12 digits fingerprint

---

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

3. Note that some versioning systems allow to obtain line diffs of the changes performed in a commit. However, the mapping between the changed lines and the impacted code components (e.g., which methods are impacted by the change) is not provided.

4. www.markosproject.euverified on September 2014.

containing the following information: LOCs, number of statements, number of *if* statements, number of *while* statements, number of *case* statements, number of *return* statements, number of specifiers, number of parameters, number of thrown exceptions, number of declared local variables, number of method (NOM) invocations, and number of used class attributes (i.e., instance variables). The accuracy of such heuristics has been evaluated at two different levels of granularity:

- *Method level*, by manually checking 100 methods reported as moved by the MARKOS code analyzer. Results showed that 89 of them were actually moved methods.
- *Class level*, by manually checking 100 classes reported as moved by the MARKOS code analyzer. Results showed that 98 of them were actually moved classes.

Typical cases of false positives were those in which a method/class was removed from a class/package and a very similar one—in terms of fingerprint—was added to another class/package.

2.2 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 smells. While the exploited underlying information is the same for all target smells (i.e., the change history information), HIST uses custom detection heuristics for each 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 smell in the past does not exist in the current version of the system, e.g., because it has been refactored by the developers. Thus, once HIST identifies a component that is affected by a 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 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 3.1.4.

2.2.1 Divergent Change Detection

Given the definition of this smell provided by Fowler [14], 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 [3] 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 [3]. In HIST, the dataset is composed of a sequence of change sets—e.g., methods—that have been committed (changed) together in a version control repository [55]. 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 [3]:

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

$$Confidence = \frac{|M_{left} \cup M_{right}|}{|M_{left}|},$$

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* [3]. Note that, minimum *Support* and *Confidence* to consider an association rule as valid can be set in the *Apriori* algorithm. Once HIST detects these change rules between methods of the same class, it identifies classes affected by *Divergent Change* as those containing at least two 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; and
3) Each method in the set does not change with methods in other sets as detected by the association rules.

2.2.2 Shotgun Surgery Detection

In order to define a 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.

2.2.3 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"* [14]. Thus, the *Code Smell detector* identifies 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.

2.2.4 Blob Detection

A *Blob* is a class that centralizes most of the system's behavior and has dependencies towards data classes [9]. Thus, our conjecture is that *despite the kind of change developers have 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 α percent of commits involving at least another class. This last condition is used to better reflect the nature of *Blob* classes that are expected to change despite the type of change being applied, i.e., the set of modified classes.

TABLE 2
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
Android API (framework-opt-telephony)	Aug. 2011-Jan. 2013	218-225	73-78
Android API (frameworks-base)	Oct. 2008-Jan. 2013	1,698-3,710	534-1,043
Android API (frameworks-support)	Feb. 2011-Nov. 2012	199-256	58-61
Android API (sdk)	Oct. 2008-Jan. 2013	132-315	14-82
Android API (tool-base)	Nov. 2012-Jan. 2013	471-714	80-134
Apache Commons Lang	Jul. 2002-Oct. 2013	30-242	14-165
Apache Cassandra	Mar. 2009-Oct. 2013	313-1,008	115-935
Apache Commons Codec	Apr. 2004-Jul. 2013	23-107	4-25
Apache Derby	Aug. 2008-Oct. 2013	1,298-2,847	159-179
Eclipse Core	Jun. 2001-Sep. 2013	824-1,232	120-174
Apache James Mime4j	Jun. 2005-Sep. 2013	106-269	91-532
Google Guava	Sep. 2009-Oct. 2013	65-457	4-35
Aardvark	Nov. 2010-Jan. 2013	16-157	13-25
And Engine	Mar. 2010-Jun. 2013	215-613	14-24
Apache Commons IO	Jan. 2002-Oct. 2013	13-200	3-56
Apache Commons Logging	Aug. 2001-Oct. 2013	5-65	1-54
Mongo DB	Jan. 2009-Oct. 2013	13-27	10-25

2.2.5 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 into. 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, HIST identifies methods affected by this smell as those involved in commits with methods of another class of the system β percent more than in commits with methods of their class.

3 EVALUATING THE ACCURACY OF HIST

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

3.1 Study Design

This section provides details about the design and planning of the study aimed at assessing HIST's effectiveness and comparing it with alternative approaches.

3.1.1 Context Selection

The *context* of the study consists of 20 software projects. Table 2 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). Among the analyzed projects we have:

- Nine projects belonging to the Apache ecosystem[5]: ANT, TOMCAT, COMMONS LANG, CASSANDRA, COMMONS CODEC, DERBY, JAMES MIME4J, COMMONS IO, and COMMONS LOGGING.
- Five projects belonging to the Android APIs[6]: FRAMEWORK-OPT-TELEPHONY, FRAMEWORKS-BASE, FRAMEWORKS-SUPPORT, SDK, and TOOL-BASE. Each of these projects is responsible for 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.
- Six open source projects from elsewhere: JEDIT,[7] ECLIPSE CORE,[8] GOOGLE GUAVA,[9] AARDVARK,[10] AND ENGINE,[11] and MONGO DB.[12]

Note that our choice of the subject systems is not random, but guided by specific requirements of our underlying infrastructure. Specifically, the selected systems:

1) are written in Java, since the MARKOS code analyzer is currently able to parse just systems written in this programming language;
2) have their entire development histories tracked in a versioning system;
3) have different development history lengths (we start with a minimum of three months for TOOL-BASE up to 13 years for APACHE ANT); and
4) have different sizes (we go from a minimum of 25 KLOCs for COMMONS CODEC up to 1,043 KLOCs for FRAMEWORK-BASE).

---

6. https://android.googlesource.com/verified on September 2014.
7. http://www.jedit.org/verified on September 2014.
8. http://www.eclipse.org/eclipse/platform-core/verified on September 2014.
9. https://code.google.com/p/guava-libraries/verified on September 2014.
10. http://karmatics.com/aardvark/verified on September 2014.
11. http://www.andengine.org/verified on September 2014.
12. http://www.mongodb.org/verified on September 2014.

5. http://www.apache.org/verified on September 2014.

TABLE 3
Snapshots Considered for the Smell 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
Android API (framework-opt-telephony)	b3a03455	Feb. 2012	223	75
Android API (frameworks-base)	b4ff35df	Nov. 2011	2,766	770
Android API (frameworks-support)	0f6f72e1	Jun. 2012	246	59
Android API (sdk)	6feca9ac	Nov. 2011	268	54
Android API (tool-base)	cfebaa9b	Dec. 2012	532	119
Apache Commons Lang	4af8bf41	Jul. 2009	233	76
Apache Cassandra	4f9e551	Sep. 2011	826	117
Apache Commons Codec	c6c8ae7a	Jul. 2007	103	23
Apache Derby	562a9252	Jun. 2006	1,746	166
Eclipse Core	0eb04df7	Dec. 2004	1,190	162
Apache James Mime4j	f4ad2176	Mar. 2009	250	280
Google Guava	e8959ed0	Aug. 2012	153	16
Aardvark	ff98d508	Jun. 2012	103	25
And Engine	f25236e4	Oct. 2011	596	20
Apache Commons IO	c8cb451c	Oct. 2010	108	27
Apache Commons Logging	d821ed3e	May 2005	61	23
Mongo DB	b67c0c43	Oct. 2011	22	25

3.1.2 Research Questions

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 accuracy of HIST in detecting instances of the five smells described in Section 2, namely *Divergent Change*, *Shotgun Surgery*, *Parallel Inheritance*, *Blob*, and *Feature Envy*.
- RQ_2. *How does HIST compare to the smell detection techniques based on the analysis of a single project snapshot?* This research question aims at comparing the accuracy of HIST in detecting the five smells above with the accuracy achieved by applying a more conventional approach based on the analysis of a single project snapshot. The results of this comparison will provide insights into the usefulness of historical information while detecting smells.

3.1.3 Study Procedure, Data Analysis and Metrics

In order to answer RQ_1 we simulated the use of HIST in a realistic usage scenario. In particular, we split the history of the 20 subject systems into two equal parts, and ran our tool on all snapshots of the first part. For instance, given the history of APACHE ANT going from January 2000 to January 2013, we selected a system snapshot s from June 2006. Then, HIST analyzed all snapshots from January 2000 to June 2006 in order to detect smell instances on the selected snapshot s. This was done aiming at simulating a developer performing smell detection on an evolving software system. On the one hand, considering some early snapshot in the project history, there could have been the risk of performing smell detection on a software system still exhibiting some ongoing, unstable design decisions. On the other hand, by considering snapshots occurring later in the project history (e.g., the last available release) there could have been the risk of simulating some unrealistic scenario, i.e., in which developers put effort

in improving the design of a software system when its development is almost absent. Table 3 reports the list of selected snapshots, together with their characteristics.

To evaluate the detection accuracy of HIST, we need an oracle reporting the instances of smells in the considered systems' snapshots. Unfortunately, there are no annotated sets of such smells available in literature. Thus, 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 (see Table 1), the student manually analyzed the source code of each snapshot, 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. This process took four weeks of work. Then, a second Master's student (still from the University of Salerno) validated the produced oracle, to verify that all affected code components identified by the first student were correct. Only six of the smells identified by the first student were classified as false positives by the second student. After a discussion performed between the two students, two of these six smells were classified as false positives (and thus removed from the oracle). Note that, while this does not ensure that the defined oracle is complete (i.e., it includes all affected components in the systems), it increases our degree of confidence on the correctness of the identified smell instances. To avoid any bias in the experiment, students were not aware of the experimental goals and of specific algorithms used by HIST for identifying smells. The number of code smell instances in our oracle is shown in Table 4 for each of the 20 subject systems. As we can see *Parallel Inheritance*, *Blob*, and *Feature Envy* code smells are quite diffused, presenting more than 50 instances each. A high number (24) of *Divergent Change* instances is also present in our oracle, while the *Shotgun Surgery* smell seems to be poorly diffused across open source projects, with just six instances identified.

TABLE 4
Code Smell Instances in the Manually Defined Oracle

Project	Divergent Change	Shotgun Surgery	Parallel Inheritance	Blob	Feature Envy
Apache Ant	0	0	7	8	8
Apache Tomcat	5	1	9	5	3
jEdit	4	1	3	5	10
Android API (framework-opt-telephony)	0	0	0	13	0
Android API (frameworks-base)	3	1	3	18	17
Android API (frameworks-support)	1	1	0	5	0
Android API (sdk)	1	0	9	10	3
Android API (tool-base)	0	0	0	0	0
Apache Commons Lang	1	0	6	3	1
Apache Cassandra	3	0	3	2	28
Apache Commons Codec	0	0	0	1	0
Apache Derby	0	0	0	9	0
Eclipse Core	1	1	8	4	3
Apache James Mime4j	1	0	0	0	9
Google Guava	0	0	0	1	2
Aardvark	0	1	0	1	0
And Engine	0	0	0	0	1
Apache Commons IO	1	0	1	2	1
Apache Commons Logging	2	0	2	2	0
Mongo DB	1	0	0	3	0
Overall	**24**	**6**	**51**	**92**	**86**

Once we defined the oracle and obtained the set of smells detected by HIST on each of the systems' snapshots, we evaluated its detection accuracy by using two widely-adopted Information Retrieval (IR) metrics, namely recall and precision [5]:

$$recall = \frac{|correct \cap detected|}{|correct|}\%,$$

$$precision = \frac{|correct \cap detected|}{|detected|}\%,$$

where *correct* and *detected* represent the set of true positive smells (those manually identified) and the set of 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:

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

Turning to RQ_2, we executed smell detection techniques based on the analysis of a single snapshot on the same systems' snapshots previously selected when answering RQ_1. To the best of our knowledge, there is not a single approach detecting all the smells that we considered in our study. For this reason, depending on the specific smell being detected, we considered different competitive techniques to compare our approach against. As for the *Blob*, we compared HIST with DECOR, the detection technique proposed by Moha et al. [33]. Specifically, we implemented the detection rules used by DECOR for the detection of *Blob*. Such rules are available online.[13] For the *Feature Envy* we considered JDeodorant as a competitive technique [48], which is a publicly

available Eclipse plug-in.[14] The approach implemented in JDeodorant analyzes all methods for a given system, and forms a set of candidate target classes where a method should be moved into. This set is obtained by examining the entities (i.e., attributes and methods) that a method accesses from the other classes.

As for *Divergent Change* and *Shotgun Surgery*, we compared HIST against our implementation of the approach proposed by Rao and Raddy [42] that is purely based on structural information. This technique starts by building an $n \times n$ matrix (where n is the number of classes in the system under analysis), named Design Change Propagation Probability (DCPP). A generic entry A_{ij} in DCPP represents the probability that a change in the class i triggers a change to the class j. Such a probability is given by the *cdegree* [43], i.e., an indicator of the number of dependencies that class i has with a class j (note that *cdegree* is not symmetric, i.e., $A_{ij} \neq A_{ji}$). Once the DCPP matrix is built, a *Divergent Change* instance is detected if a column in the matrix (i.e., a class) has several (more than λ) non-zero values (i.e., the class has dependencies with several classes). The conjecture is that if a class depends on several other classes, it is likely that it implements different responsibilities divergently changing during time. Regarding the detection of the *Shotgun Surgery*, instances of such a smell are identified when a row in the matrix (i.e., a class) contains several (more than η) non-zero values (i.e., several classes have dependencies with the class). The conjecture is that changes to this class will trigger changes in classes depending on it. From now on we will refer to this technique as DCPP.

Concerning the *Parallel Inheritance* smell, we are not aware of publicly available techniques in the literature to detect it. Thus, in order to have a meaningful baseline for

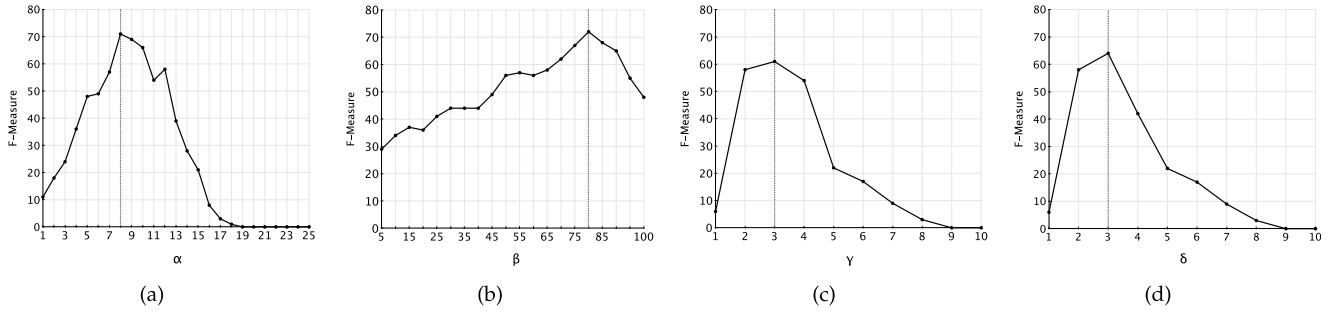13. http://www.ptidej.net/research/designsmells/grammar/Blob.txt

14. http://www.jdeodorant.com/ verified on September 2014.

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).

HIST, we implemented a detection algorithm based on the analysis of a single project snapshot. Note that this analysis was not intended to provide evidence that HIST is the best method for detecting *Parallel Inheritance* instances. Instead, the goal was to conduct an investigation into the actual effectiveness of historical information while detecting smells as compared to information extracted from a single project snapshot.

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 (from now on coined 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*" [14].

To compare the performances of HIST against the competitive techniques described above, we used recall, precision, and F-measure. Moreover, to analyze the complementarity of static code information and historical information when performing smell detection, we computed the following overlap metrics:

$$correct_{HIST \cap SS} = \frac{|correct_{HIST} \cap correct_{SS}|}{|correct_{HIST} \cup correct_{SS}|}\%,$$

$$correct_{HIST \setminus SS} = \frac{|correct_{HIST} \setminus correct_{SS}|}{|correct_{HIST} \cup correct_{SS}|}\%,$$

$$correct_{SS \setminus HIST} = \frac{|correct_{SS} \setminus correct_{HIST}|}{|correct_{HIST} \cup correct_{SS}|}\%,$$

where $correct_{HIST}$ and $correct_{SS}$ represent the sets of correct smells detected by HIST and the competitive technique, respectively. $correct_{HIST \cap SS}$ measures the overlap between the set of true smells detected by both techniques, and $correct_{HIST \setminus SS}$ ($correct_{SS \setminus HIST}$) measures the true smells detected by HIST (SS) only and missed by SS (HIST). The latter metric provides an indication on how a smell detection strategy contributes to enriching the set of correct smells identified by another method.

3.1.4 Calibrating HIST and the Competitive Approaches

While for JDeodorant and DECOR parameter tuning has already been empirically investigated by their respective authors, we needed to calibrate parameters for HIST and

DCPP as well. Indeed, in the work presenting the DCPP approach no best values for its parameters were recommended [42]. We performed this calibration on a software system which was not used in our experimentation, i.e., APACHE XERCES.[15] Also on this system, we asked two Master's students to manually identify instances of the five considered smells in order to build an oracle. The procedure adopted by the students was exactly the same described before and used to build the study oracle. Then, we evaluated the F-measure value obtained by the detection approaches using different settings.

Results of the calibration are reported in Fig. 2 for the HIST parameters α, β, γ, and δ, and in Fig. 3 for the DCPP λ and the DCPP η parameters. As for the confidence and support, the calibration was not different from what was done in other work using association rule discovery [10], [19], [53], [55]. In particular, we tried all combinations of confidence and support obtained by varying the confidence between 0.60 and 0.90 by steps of 0.05, and the support between 0.004 and 0.04 by steps of 0.004, and searching for the one ensuring the best F-measure value on XERCES. Table 5 summarizes the calibration process, reporting 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).

3.1.5 Replication Package

The raw data and working data sets used in our study are publicly available in a replication package [40] where we provide: (i) links to the Git repositories from which we extracted historical information; (ii) complete information on the change history in all the subject systems; (iii) the oracle used for each system; and (iv) the list of smells identified by HIST and by the competitive approaches.

3.2 Analysis of the Results

This section reports the results aimed at answering the two research questions formulated in Section 3.1.2. Note that to avoid redundancies, we report the results for both research questions together, discussing each smell separately.

Tables 6, 7, 8, 9, and 10 report the results—in terms of recall, precision, and F-measure—achieved by HIST and approaches based on the analysis of a single snapshot on the 20 subject systems. In addition, each table also reports (i) the number of smell instances present in each system
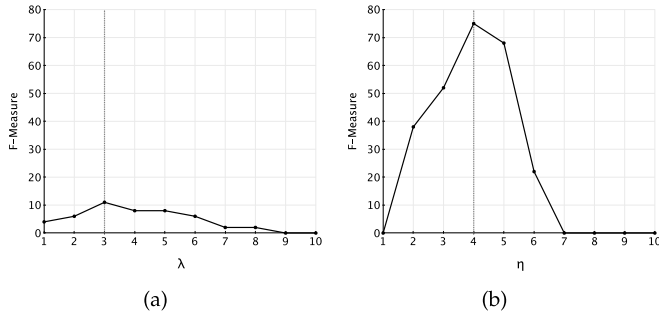
---

15. http://xerces.apache.org/ verified on September 2014.

Fig. 3. Parameters' calibration for DCPP-divergent change λ (a), and DCPP-Shotgun Surgery η (b).

(column "#Smell Instances"), (ii) the number of smell instances identified by each approach (column "Identified"), (iii) the number of true positive instances identified by each approach (column "TP"), and (iv) the number of false positive instances identified by each approach (column "FP"). Note that each table shows the results for one of the five smells considered in our study and in particular: Table 6 for *Divergent Change*, Table 7 for *Shotgun Surgery*, Table 8 for *Parallel Inheritance*, Table 9 for *Blob*, and Table 10 for *Feature Envy*.

As explained in Section 3.1.3 for *Divergent Change* and *Shotgun Surgery* we compared HIST against DCCP approach proposed by Rao and Raddy [42], while for *Parallel Inheritance* we used an alternative approach that we developed (PICA). Finally, for *Blob* and *Feature Envy* we used DECOR rules [33] and the JDeodorant tool [13], respectively.

When no instances of a particular smell were present in the oracle (i.e., zero in the column "#Smell Instances"), it was not possible to compute the recall (that is, division by zero), while the precision would be zero if at least one false positive is detected (independently of the number of false positives). In these cases a "-" is indicated in the corresponding project row. Similarly, when an approach did not retrieve any instances of a particular smell, it was not possible to compute precision, while recall would be zero if at least one false positive is retrieved. In this case a "N/A" is included in the project row. However, to have an accurate estimation of the performances of the experimented techniques, we also report in each table the results achieved by considering all systems as a single dataset (rows "Overall"). In such a dataset, it never happens that recall or precision cannot be computed for the reasons described above. Thus, all true positives and all false positives identified by each technique are taken into account in the computation of the overall recall, precision, and F-measure.

Finally, Table 11 reports the overlap and differences between HIST and the techniques based on code analysis of a single snapshot: column "HIST ∩ SS Tech." reports the number (#) and percentage (%) of smells correctly identified by both HIST and the competitive technique; column "HIST\SS Tech." reports the number and percentage of smells correctly identified by HIST but not by the competitive technique; column "SS Tech.\HIST" reports the number and percentage of smells correctly identified by the competitive technique but not by HIST. In the following, we discuss the results for each kind of smell.

3.2.1   Divergent Change

We identified 24 instances of *Divergent Change* in the 20 systems (see Table 6). The results clearly indicate that the use of historical information allows to outperform DCCP (i.e., the approach based on the analysis of a single snapshot). Specifically, the F-measure achieved by HIST on the overall dataset is 82 percent (83 percent of recall and 80 percent of precision) against 10 percent (13 percent of recall and 11 percent of precision) achieved by DCCP. This is an expected result, since the *Divergent Change* is by definition a "historical smell" (see Section 2), and thus we expected difficulties in capturing this kind of smell by just relying on the analysis of a single system's snapshot.

One of the *Divergent Change* instances captured by HIST is depicted in Fig. 4 and related to the `ViewPager` class from the ANDROID FRAMEWORKS-SUPPORT project. `ViewPager` allows users of Android apps to flip left and right through pages of data. In this class, HIST identified three sets of methods divergently changing during the project's history (see Section 2.2.1 for details on how these sets were identified). The three sets are highlighted in Fig. 4 by using different shades of gray. Starting from the top of the Fig. 4, the first set groups together methods somewhat related to the management of the `items` to be displayed in the `View` (e.g., menu, buttons, etc.). The middle set gathers methods allowing to manage the `View` layout (i.e., setting margins, page offsets, etc.), while the set at the bottom of Fig. 6 is mainly related to the `View` configuration (e.g., init the page viewer, define the change listeners, etc.). Thus, the three identified sets of methods, not only change independently one from the other, but also seem to represent quite independent responsibilities implemented in the `ViewPager` class. Of course, no speculations can be made on the need for refactoring of this class, since developers having high experience on the system are needed to evaluate both pros and cons.

TABLE 5
Calibration of the Parameters Required by the Different Detection Techniques Used in the Study

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 percent by steps of 1 percent	8%
HIST (Feature Envy)	β	From 5 to 100 percent by steps of 5 percent	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
DCPP (Divergent Change)	λ	From 1 to 10 by steps of 1	3
DCPP (Shotgun Surgery)	η	From 1 to 10 by steps of 1	4

TABLE 6
Divergent Change—HIST Accuracy as Compared to the Single Snapshot Technique

Project	#Smell Instances	HIST						Single Snapshot technique					
		Identified	TP	FP	Prec.	Recall	F-measure	Identified	TP	FP	Prec.	Recall	F-measure
Apache Ant	0	0	0	0	-	-	-	1	0	1	-	-	-
Apache Tomcat	5	6	3	3	50%	60%	55%	0	0	0	N/A	N/A	N/A
jEdit	4	3	3	0	100%	75%	86%	1	1	0	100%	25%	40%
Android API (framework-opt-telephony)	0	0	0	0	-	-	-	0	0	0	-	-	-
Android API (frameworks-base)	3	3	3	0	100%	100%	100%	0	0	0	N/A	N/A	N/A
Android API (frameworks-support)	1	1	1	0	100%	100%	100%	2	0	2	0%	0%	0%
Android API (sdk)	1	1	1	0	100%	100%	100%	0	0	0	N/A	N/A	N/A
Android API (tool-base)	0	1	0	1	-	-	-	0	0	0	-	-	-
Apache Commons Lang	1	1	1	0	100%	100%	100%	0	0	0	N/A	N/A	N/A
Apache Cassandra	3	2	2	0	100%	67%	80%	7	1	6	14%	34%	20%
Apache Commons Codec	0	0	0	0	-	-	-	0	0	0	-	-	-
Apache Derby	0	0	0	0	-	-	-	0	0	0	-	-	-
Eclipse Core	1	1	1	0	100%	100%	100%	0	0	0	N/A	N/A	N/A
Apache James Mime4j	1	1	1	0	100%	100%	100%	1	1	0	100%	100%	100%
Google Guava	0	0	0	0	-	-	-	0	0	0	-	-	-
Aardvark	0	1	0	1	-	-	-	0	0	0	-	-	-
And Engine	0	0	0	0	-	-	-	14	0	14	-	-	-
Apache Commons IO	1	1	1	0	100%	100%	100%	3	0	3	0%	0%	0%
Apache Commons Logging	2	2	2	0	100%	100%	100%	0	0	0	N/A	N/A	N/A
Mongo DB	1	1	1	0	100%	100%	100%	0	0	0	N/A	N/A	N/A
Overall	**24**	**25**	**20**	**5**	**80%**	**83%**	**82%**	**29**	**3**	**26**	**10%**	**13%**	**11%**

TABLE 7
Shotgun Surgery—HIST Accuracy Compared to the Single Snapshot Techniques

Project	#Smell Instances	HIST						Single snapshot echnique					
		Identified	TP	FP	Prec.	Recall	F-measure	Identified	TP	FP	Prec.	Recall	F-measure
Apache Ant	0	0	0	0	-	-	-	4	0	4	-	-	-
Apache Tomcat	1	1	1	0	100%	100%	100%	13	0	13	0%	0%	0%
jEdit	1	1	1	0	100%	100%	100%	3	0	3	0%	0%	0%
Android API (framework-opt-telephony)	0	1	0	1	-	-	-	3	0	3	-	-	-
Android API (frameworks-base)	1	1	1	0	100%	100%	100%	1	0	1	0%	0%	0%
Android API (frameworks-support)	1	1	1	0	100%	100%	100%	2	0	2	0%	0%	0%
Android API (sdk)	0	0	0	0	-	-	-	0	0	0	-	-	-
Android API (tool-base)	0	0	0	0	-	-	-	0	0	0	-	-	-
Apache Commons Lang	0	0	0	0	-	-	-	0	0	0	-	-	-
Apache Cassandra	0	0	0	0	-	-	-	0	0	0	-	-	-
Apache Commons Codec	0	0	0	0	-	-	-	0	0	0	-	-	-
Apache Derby	0	0	0	0	-	-	-	0	0	0	-	-	-
Eclipse Core	1	1	1	0	100%	100%	100%	0	0	0	N/A	N/A	N/A
Apache James Mime4j	0	0	0	0	-	-	-	0	0	0	-	-	-
Google Guava	0	0	0	0	-	-	-	0	0	0	-	-	-
Aardvark	1	1	1	0	100%	100%	100%	0	0	0	N/A	N/A	N/A
And Engine	0	0	0	0	-	-	-	0	0	0	-	-	-
Apache Commons IO	0	0	0	0	-	-	-	0	0	0	-	-	-
Apache Commons Logging	0	0	0	0	-	-	-	0	0	0	-	-	-
Mongo DB	0	0	0	0	-	-	-	0	0	0	-	-	-
Overall	**6**	**7**	**6**	**1**	**86%**	**100%**	**92%**	**26**	**0**	**26**	**0%**	**0%**	**0%**

Our second study presented in Section 4 aims at answering exactly this question.

Going back to the quantitative results, DCCP was able to detect only three correct occurrences of *Divergent Change* and one of them was also captured by HIST. The instances missed by HIST (and identified by DCCP) affect the RE class of JEDIT and the CassandraServer class of APACHE CASSANDRA. Both of these classes do not have enough change history

TABLE 8
Parallel Inheritance—HIST Accuracy as Compared to the Single Snapshot Techniques

Project	#Smell Instances	HIST						Single snapshot technique					
		Identified	TP	FP	Prec.	Recall	F-measure	Identified	TP	FP	Prec.	Recall	F-measure
Apache Ant	7	8	5	3	63%	71%	67%	52	4	48	8%	57%	14%
Apache Tomcat	9	10	6	4	60%	67%	63%	61	4	57	7%	44%	12%
jEdit	3	0	0	0	N/A	N/A	N/A	15	3	12	20%	100%	33%
Android API (framework-opt-telephony)	0	0	0	0	-	-	-	9	0	9	-	-	-
Android API (frameworks-base)	3	1	0	1	0%	0%	0%	111	0	111	0%	0%	0%
Android API (frameworks-support)	0	0	0	0	-	-	-	9	0	9	-	-	-
Android API (sdk)	9	12	8	4	67%	89%	76%	59	3	56	5%	33%	12%
Android API (tool-base)	0	0	0	0	-	-	-	0	0	0	-	-	-
Apache Commons Lang	6	6	6	0	100%	100%	100%	6	6	0	100%	100%	100%
Apache Cassandra	3	1	1	0	100%	34%	50%	35	1	34	3%	34%	5%
Apache Commons Codec	0	0	0	0	-	-	-	3	0	3	-	-	-
Apache Derby	0	0	0	0	-	-	-	53	0	53	-	-	-
Eclipse Core	8	8	7	1	88%	88%	88%	31	2	29	6%	25%	10%
Apache James Mime4j	0	0	0	0	-	-	-	10	0	10	-	-	-
Google Guava	0	0	0	0	-	-	-	0	0	0	-	-	-
Aardvark	0	0	0	0	-	-	-	0	0	0	-	-	-
And Engine	0	0	0	0	-	-	-	60	0	60	-	-	-
Apache Commons IO	1	1	1	0	100%	100%	100%	8	1	7	13%	100%	22%
Apache Commons Logging	2	1	1	0	100%	50%	67%	3	1	2	34%	50%	40%
Mongo DB	0	0	0	0	-	-	-	0	0	0	-	-	-
Overall	**51**	**48**	**35**	**13**	**73%**	**69%**	**71%**	**525**	**25**	**500**	**5%**	**49%**	**9%**

TABLE 9
Blob—HIST Accuracy as Compared to DECOR

Project	#Smell Instances	HIST						DECOR					
		Identified	TP	FP	Prec.	Recall	F-measure	Identified	TP	FP	Prec.	Recall	F-measure
Apache Ant	8	10	6	4	60%	75%	67%	10	3	7	30%	38%	33%
Apache Tomcat	5	1	1	0	100%	20%	33%	6	4	2	67%	80%	73%
jEdit	5	3	2	1	67%	40%	50%	5	3	2	60%	60%	60%
Android API (framework-opt-telephony)	13	10	10	0	100%	77%	87%	10	7	3	70%	54%	61%
Android API (frameworks-base)	18	13	9	4	70%	50%	58%	14	9	5	65%	50%	57%
Android API (frameworks-support)	5	7	5	2	71%	100%	83%	8	3	5	38%	60%	49%
Android API (sdk)	10	7	6	1	86%	60%	71%	7	2	5	29%	20%	24%
Android API (tool-base)	0	0	0	0	-	-	-	0	0	0	-	-	-
Apache Commons Lang	3	2	2	0	100%	67%	80%	0	0	0	N/A	N/A	N/A
Apache Cassandra	2	0	0	0	N/A	N/A	N/A	0	0	0	N/A	N/A	N/A
Apache Commons Codec	1	2	1	1	50%	100%	67%	0	0	0	N/A	N/A	N/A
Apache Derby	9	0	0	0	N/A	N/A	N/A	7	4	3	57%	44%	50%
Eclipse Core	4	3	2	1	67%	50%	57%	4	2	2	50%	50%	50%
Apache James Mime4j	0	3	0	3	-	-	-	0	0	0	-	-	-
Google Guava	1	1	1	0	100%	100%	100%	0	0	0	N/A	N/A	N/A
Aardvark	1	1	1	0	100%	100%	100%	1	1	0	100%	100%	100%
And Engine	0	0	0	0	-	-	-	0	0	0	-	-	-
Apache Commons IO	2	3	2	1	67%	100%	80%	0	0	0	N/A	N/A	N/A
Apache Commons Logging	2	3	2	1	67%	100%	80%	2	2	0	100%	100%	100%
Mongo DB	3	5	3	2	60%	100%	75%	0	0	0	N/A	N/A	N/A
Overall	**92**	**74**	**53**	**21**	**72%**	**58%**	**64%**	**74**	**40**	**34**	**54%**	**43%**	**48%**

data 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 DCCP results reported in Table 11 is quite expected: among the sets of smells correctly detected by two techniques, there is just a 4 percent overlap, HIST is the only one retrieving 87 percent of

TABLE 10
Feature Envy—HIST Accuracy as Compared to JDeodorant

Project	#Smell Instances	HIST						JDeodorant					
		Identified	TP	FP	Prec.	Recall	F-measure	Identified	TP	FP	Prec.	Recall	F-measure
Apache Ant	8	9	6	3	67%	75%	71%	13	2	11	15%	25%	19%
Apache Tomcat	3	1	1	0	100%	33%	50%	3	2	1	67%	67%	67%
jEdit	10	10	8	2	100%	100%	100%	3	3	0	100%	27%	43%
Android API (framework-opt-telephony)	0	0	0	0	-	-	-	0	0	0	-	-	-
Android API (frameworks-base)	17	24	15	9	63%	88%	73%	16	16	0	100%	94%	96%
Android API (frameworks-support)	0	0	0	0	-	-	-	0	0	0	-	-	-
Android API (sdk)	3	1	1	0	100%	33%	50%	0	0	0	N/A	N/A	N/A
Android API (tool-base)	0	0	0	0	-	-	-	0	0	0	-	-	-
Apache Commons Lang	1	2	1	1	50%	100%	67%	2	1	1	50%	100%	67%
Apache Cassandra	28	28	28	0	100%	100%	100%	28	28	0	100%	100%	100%
Apache Commons Codec	0	1	0	1	-	-	-	0	0	0	-	-	-
Apache Derby	0	0	0	0	-	-	-	0	0	0	-	-	-
Eclipse Core	3	5	3	2	60%	100%	75%	0	0	0	N/A	N/A	N/A
Apache James Mime4j	9	0	0	0	N/A	N/A	N/A	11	9	2	82%	100%	90%
Google Guava	2	2	2	0	100%	100%	100%	3	0	3	0%	0%	0%
Aardvark	0	0	0	0	-	-	-	0	0	0	-	-	-
And Engine	1	2	1	1	50%	100%	67%	0	0	0	N/A	N/A	N/A
Apache Commons IO	1	0	0	0	N/A	N/A	N/A	6	0	6	0%	0%	0%
Apache Commons Logging	0	0	0	0	-	-	-	8	0	8	-	-	-
Mongo DB	0	0	0	0	-	-	-	1	0	1	-	-	-
Overall	**86**	**85**	**66**	**19**	**78%**	**77%**	**77%**	**94**	**61**	**33**	**65%**	**71%**	**68%**

the smells, while DCCP is the one detecting only two smells described above and missed by HIST (9 percent). Thus, the complementarity between HIST and DCCP is rather low.

3.2.2 Shotgun Surgery

Shotgun Surgery is the smell with the lowest number of instances in the subject systems, i.e., with only six systems affected for a total of six instances (one per system). HIST was able to detect all the instances of this smell (100 percent recall) with 86 percent precision outperforming DCCP (i.e., the competitive approach). Specifically, DCCP was not able to detect any of the six instances of this smell present in the subject systems. Thus, no meaningful observations can be made in terms of overlap metrics. This result highlights the fact that it is quite difficult to identify characteristics of such a smell by solely analysing a single system's snapshot, as the smell is intrinsically defined in terms of a change triggering many other changes [14].

TABLE 11
Overlap between HIST and Single Snapshot Techniques for Blob the SS Tech. Is DECOR, for Feature Envy It Is JDeodorant.

Code Smell	HIST∩SS Tech.		HIST\SS Tech.		SS Tech.\HIST	
	#	%	#	%	#	%
Divergent Change	1	4%	19	87%	2	9%
Shotgun Surgery	0	0%	6	100%	0	0%
Parallel Inheritance	20	50%	15	38%	5	12%
Blob	13	16%	40	51%	27	33%
Feature Envy	44	54%	22	27%	17	19%

It is also 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

```
┌─────────────────────────────────────────────────────────┐
│                        ViewPager                         │
├─────────────────────────────────────────────────────────┤
│ ...                                                      │
├─────────────────────────────────────────────────────────┤
│ addNewItem(int,int)                                      │
│ setCurrentItem(int)                                      │
│ setCurrentItem(int,boolean)                              │
│ getCurrentItem()                                         │
│ setCurrentItemInternal(int,boolean,boolean)              │
│ setCurrentItemInternal(int,boolean,boolean,int)          │
│ populate()                                               │
│ populate(int)                                            │
│ setPageMargin(int)                                       │
│ getPageMargin()                                          │
│ setPageMarginDrawable(Drawable)                          │
│ setPageMarginDrawable(int)                               │
│ calculatePageOffsets(ItemInfo,int,ItemInfo)              │
│ distanceInfluenceForSnapDuration(float)                  │
│ smoothScrollTo(int,int)                                  │
│ smoothScrollTo(int,int,int)                              │
│ initViewPager()                                          │
│ onDetachedFromWindow()                                   │
│ setScrollState()                                         │
│ setAdapter(PagerAdapter)                                 │
│ removeNonDecorViews()                                    │
│ getAdapter()                                             │
│ setOnAdapterChangeListener(OnAdapterChangeListener)      │
│ getOffscreenPageLimit()                                  │
│ setOffscreenPageLimit(int)                               │
│ setOnPageChangeListener(OnPageChangeListener)            │
│ setInternalPageChangeListener(OnPageChangeListener)      │
│ verifyDrawable(Drawable)                                 │
│ drawableStateChanged()                                   │
│ dataSetChanged()                                         │
└─────────────────────────────────────────────────────────┘
```

Fig. 4. One of the identified *Divergent Change* instances: the class `ViewPager` from ANDROID FRAMEWORKS-SUPPORT.
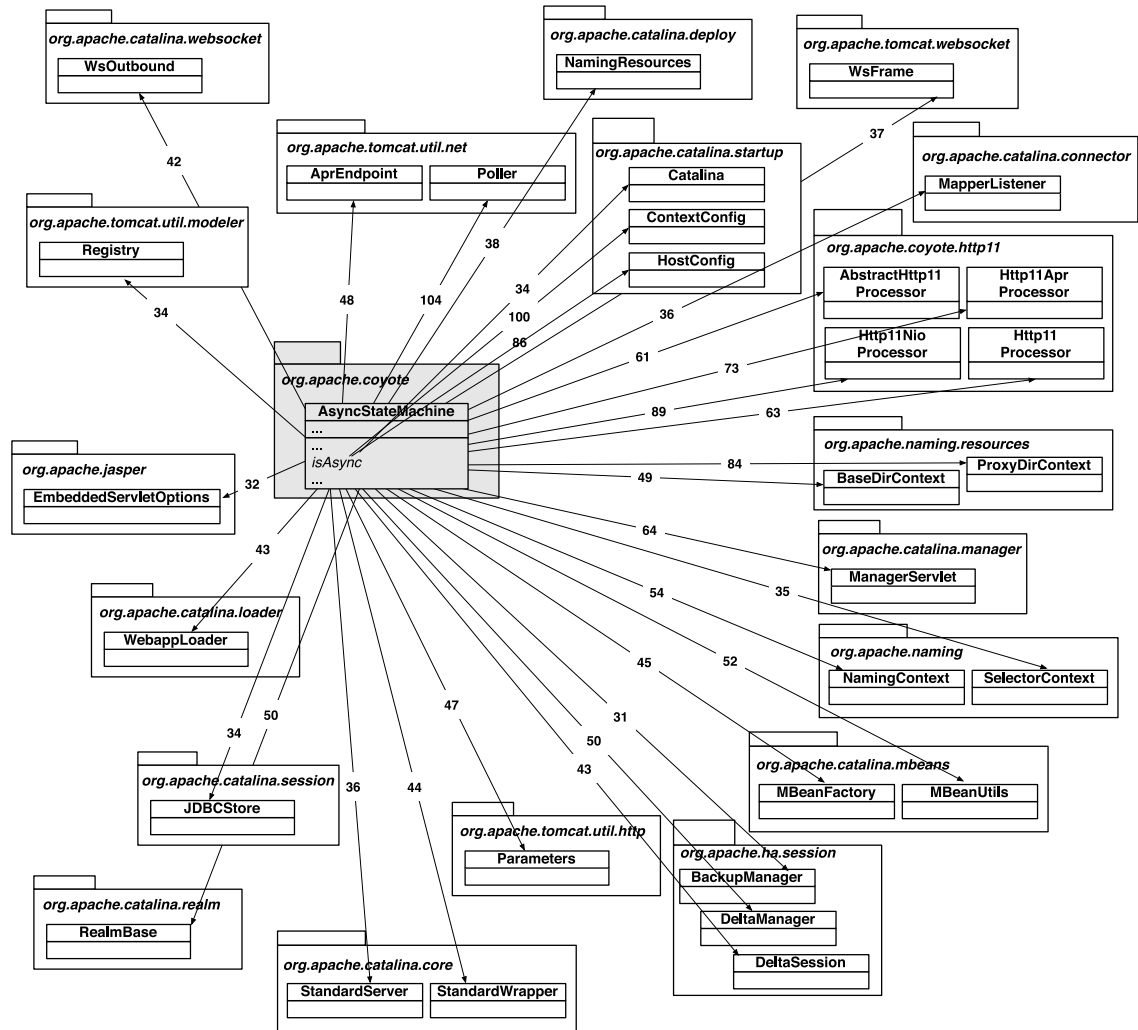
Fig. 5. One of the identified *Shotgun Surgery* instances: the `AsyncStateMachine.isAsync` method from APACHE TOMCAT.

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. Fig. 5 shows all 31 classes involved: each arrow going from the `isAsync` method to one of these 31 classes is labeled with the number of times `isAsync` co-changed with methods of that class in the analyzed time period. Note that the total number of changes performed in the analyzed time period to `isAsync` is 110. For instance, `isAsync` co-changed 104 (95 percent) times with two methods contained in the `Poller` class. What is also very surprising about this instance of *Shotgun Surgery* is that it triggers changes in over 19 different packages of the software system. This clearly highlights the fact that such smell could be very detrimental in software evolution and maintenance context.

As for the only false positive instance identified by HIST, it concerns the method `dispose` from the class `GsmData-ConnectionTracker` of the FRAMEWORK-OPT-TELEPHONY Android APIs (see Table 7). HIST identified association rules between this method and three other methods in the system, and in particular: `CdmaDataConnection-Tracker.dispose()`, `SMSDispatcher.handleSend-Complete()`, and `GsmCallTracker.dump()`. However, this behavior was not considered as "smelly" by the

students building the oracle because: (i) differently from what discussed for the `isAsync` method, the triggered changes in this case are spread just across three classes, and (ii) even if the four involved methods tend to change together, they are correctly placed into different classes splitting well the system's responsibilities. For instance, while the two `dispose` methods contained in classes `GsmDataConnectionTracker` and `CdmaDataConnec-tionTracker` are both in charge of cleaning up a data connection, the two protocols they manage are different (i.e., GSM versus CDMA). Thus, even if they co-change during time, there is no apparent reason for placing them in the same class with the only goal of isolating the change (poorly spread in this case). Indeed, as a side effect, this refactoring operation could create a class managing heterogeneous responsibilities (i.e., a *Blob* class).

3.2.3 Parallel Inheritance

Among the 51 instances of the *Parallel Inheritance* smell, HIST was able to correctly identify 35 of them (recall 69 percent) with a price to pay of 13 false positives, resulting in a precision of 73 percent. By using the competitive technique (i.e., PICA) we were able to retrieve 25 correct instances of the smell (recall of 49 percent) while also

Fig. 6. An identified *Parallel Inheritance* instance: the pair of classes `CompletitionOnQualifiedNameReference` and `SelectionOnQuali-fiedNameReference` from ECLIPSE JDT.

retrieving 473 false positives (precision of 5 percent). One of the *Parallel Inheritance* instances detected by HIST and missed by PICA is depicted in Fig. 6. The pair of classes affected by the smell is `CompletitionOnQualifiedNa-meReference` and `SelectionOnQualifiedNameRefer-ence` from ECLIPSE JDT. As shown in Fig. 6, these two classes have been committed together on June 27, 2008 in the same package and since then, the hierarchies having them as top superclasses evolved in parallel. Indeed, the first subclass (`QualifiedNameReference`) has been added to both superclasses on September 3, 2008 followed by the second subclass (`NameReference`) on September 25, 2008. Note that while the name of the subclasses added to the two superclasses is the same, we are talking about two different subclasses. Indeed, as show in Fig. 6, these subclasses are from different packages. For instance, the `NameReference` subclass of `CompletitionOnQuali-fiedNameReference` is from the `org.eclipse.jdt.internal.codeassist` package, while the correspond-ing subclass of `SelectionOnQualifiedNameReference` is from the `org.eclipse.jdt.internal.compiler.ast` package.

Looking at the overlap metrics reported in Table 11, we can see an overlap of 50 percent among the set of smells cor-rectly identified by the two techniques, while 38 percent of the correct instances are retrieved only by HIST and the remaining 12 percent are identified only by PICA. For example, an instance of *Parallel Inheritance* detected by PICA and missed by HIST is the one affecting the pair of classes `Broken2OperationEnum` and `Broken5OperationEnum` belonging to APACHE COMMONS LANG. In this case, while the two hierarchies co-evolved synchronously, the (too high) thresholds used for the support and confidence of the asso-ciation rule mining algorithm used in HIST did not allow capturing this specific instance (and thus, to identify the smell). Obviously, this instance could have been detected when using lower values for support and confidence, how-ever, this would naturally result in drastically decreasing precision while somewhat increasing recall values.

3.2.4 Blob

As for detecting the *Blobs*, HIST was able to achieve a precision of 72 percent and a recall of 58 percent (F-measure = 64%), while DECOR was able to achieve a precision of 54 percent and a recall of 43 percent (F-measure = 48%). In more details, HIST achieved better precision values on 13 systems (on average, + 45 percent), DECOR on two systems (on average, + 45 percent), while on one system there was a tie. Thus, for most of the systems containing *Blob* instances (13 out of 16) HIST requires less effort to developers looking for instances of Blobs due to the lower number of false positives that will be inspected and discarded. Also, HIST ensured better recall on nine out of the 16 systems containing at least one *Blob* class, and a tie has

been reached on five other systems. On the contrary, HIST was outperformed by DECOR on Apache Tomcat and jEdit (see Table 9). However, on the overall dataset, HIST was able to correctly identify 53 of the 92 existing Blobs, against the 40 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 (64 percent against 48 percent). Noticeably, the two approaches seem to be highly complementary. This is highlighted by the overlap results in Table 11. Among the sets of smells correctly identified by the two techniques, there is an overlap of just 16 percent. Specifically, HIST is able to detect 51 percent of smells that are ignored by DECOR, and the latter retrieves 33 percent of correct smells that are 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 single-snapshot 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 APACHE 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.

3.2.5 Feature Envy

For the *Feature Envy* smell, we found instances of this smell in 12 out of the 20 systems, for a total of 86 affected methods. HIST was able to identify 66 of them (recall of 77 percent) against the 61 identified by JDeodorant (recall of 71 percent). Also, the precision obtained by HIST is higher than the one achieved by JDeodorant (78 percent against 65 percent). However, it is important to point out 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 a program's behavior does not change after applying the suggested refactoring operation [48]. An example of considered preconditions is that *the envied class does not contain a method having the same signature as the moved method* [48]. To perform a fair comparison (especially in terms of recall), we filtered the *Feature Envy* instances retrieved by HIST by using the same set of preconditions defined by JDeodorant [48]. This resulted in the removal of three correct instances, as well as three false positives previously retrieved by HIST, thus decreasing the recall from 78 to 74 percent and increasing the precision from 78 to 80 percent. Still, HIST achieves better recall and precision values as compared to JDeodorant.

It is interesting to observe that the overlap data reported in Table 11 highlights, also in this case, some complementarity

between historical and single snapshot techniques, with 54 percent of correct smell instances identified by both techniques (overlap), 27 percent identified only by HIST, and 19 percent 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 RQ₁. HIST provided acceptable performances in detecting all smells considered in our study (F-measure between 64 to 92 percent). 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 RQ₂. HIST was able to outperform single snapshot techniques and tools in terms of recall, precision, and F-measure. While such a result is somewhat expected for "intrinsically historical" smells, i.e., for (*Divergent Change*, *Shotgun Surgery*, and *Parallel Inheritance*), noticeably HIST is also able to perform well for detecting other smells (i.e., *Blob* and *Feature Envy*), provided that historical information is available. Last, but not least, for *Blob* and *Feature Envy*, our findings suggest that techniques based on code analysis of a single snapshot are complementary to HIST and these two families of approaches can be integrated to potentially improve performance even further.

4 DEVELOPERS' PERCEPTION OF SMELLS

Despite having achieved good results in terms of detection capability, it is also important to point out that a smell detection technique is actually useful only if it identifies code design problems that are recognized as *relevant problems* by developers. For this reason, we performed a second study aimed at investigating to what extent the smells detected by HIST (and by the competitive techniques) reflect developers' perception of poor design and implementation choices and, in this case (i) what is their perceived severity of the problem, and (ii) if they consider as necessary a refactoring operation aimed at removing the smell. The design of this second study was based on the results obtained in the first study. Specifically:

- For purely historical smells (i.e., *Divergent Change*, *Parallel Inheritance*, and *Shotgun Surgery*) we consider only instances that are identified by HIST. Indeed, the results discussed in Section 3 demonstrate low complementarity between HIST and the competitive techniques for detecting these smells, with HIST playing the major role.

TABLE 12
Smell Instances Selected for Each System

System	Divergent Change	Parallel Inheritance	Shotgun Surgery	Blob			Feature Envy		
				onlyHIST	onlyDECOR	both	onlyHIST	onlyJD	both
Apache Ant	-	1	-	1	1	1	1	1	-
Eclipse	1	1	1	1	1	-	1	-	-
Android sdk	1	1	-	1	1	1	1	-	-
Apache frameworks-base	-	1	1	1	1	-	1	1	1
Overall	**2**	**4**	**2**	**4**	**4**	**2**	**4**	**2**	**1**

- For the structural smells (i.e., *Blob* and *Feature Envy*) we consider instances identified (i) only by HIST (*onlyHIST* group), (ii) only by the competitive technique (*onlyDECOR* for *Blobs* and *onlyJD* for *Feature Envy*), and (iii) by both techniques (*both* group). Indeed, the results achieved for these two smells show that historical and structural information can both be good alternatives for identifying smells. Thus, it is interesting to understand which of the above mentioned groups contains smells that are recognized as actual problems by developers.

4.1 Study Design

In the following, we report the design and planning of the study, by detailing the context selection, the research questions, the data collection process, as well as the analysis method.

4.1.1 Context Selection

A needed requirement for this study is, of course, software developers. In order to recruit participants, we sent invitations to active developers of ten of the 20 systems considered in our first study. In particular, we just considered systems exhibiting instances of at least three of the code smells investigated in this paper. The active developers have been identified by analyzing the systems' commit history.[16] In total, we invited 109 developers receiving responses from 12 of them: two developers from APACHE ANT, two from ECLIPSE, two from ANDROID SDK, and six from ANDROID FRAMEWORKS-BASE. Note that, even if the number of respondents appears to be low (11 percent of response rate), we are inline with the suggested minimum response rate for the survey studies defined below 20 percent [6].

4.1.2 Research Questions

This study (Study II) aims at addressing the following two research questions:

- *RQ3. Are the historical code smells identified by HIST recognized as design problems by developers?* This research question focuses its attention on the *Divergent Change*, *Parallel Inheritance*, and *Shotgun Surgery* smells. HIST is the first technique able to effectively identify instances of these smells. Thus, it is worthwhile to know if

16. We considered developers that performed at least one commit in the last two years.

the instances of the smells it identifies really represent design problems for developers.
- *RQ4. Which detection technique aimed at Identifying structural code smells better reflects developers' perception of design problems?* This research question aims at investigating how developers working on the four open-source systems perceive the presence of structural code smells identified by different detection techniques. In particular, we focus on smells identified by HIST only, by the techniques based on code analysis of a single snapshot only, and by both.

We answer both research questions through a survey questionnaire that participants filled-in online.

4.1.3 Survey Questionnaire Design

We designed a survey aimed at collecting developers' opinions needed to answer two of our research questions. Specifically, given the subject system S_i, the following process was performed:

1) *Smell instances selection.* The smell instances to consider in our study were selected as follows:
   - For each *purely historical* smell c_j (i.e., *Divergent Change*, *Parallel Inheritance*, and *Shotgun Surgery*) having at least one instance in S_i detected by HIST, we randomly selected one instance or took the only one available. Note that we refer to the "instance" as code component(s) affected by the smell. For example, it could be a single class affected by the *Divergent Change* smell, as well as a pair of classes affected by the *Parallel Inheritance* smell.
   - For each *structural* smell c_j (i.e., *Blob* and *Feature Envy*) having at least one instance in S_i we randomly selected (i) one instance detected only by HIST (if any), (ii) one instance detected only by the competitive technique—i.e., DECOR or JDeodorant—(if any), and (iii) one instance detected by both techniques (if any).

Note that this study excluded entities affected by more than one smell instance (e.g., a method affected by both *Shotgun Surgery* and *Feature Envy*). The smells selected on each system are summarized in Table 12. As it can be seen, we were not able to get the same number of instances for all the smells and for all the groups of structural smells. However, we were able to cover all smells and groups of smells (i.e., *onlyHIST*, *onlyDECOR/JD*, *both*) with at least one smell instance.

2) *Defining survey questions.* For each selected smell instance, study participants had to look at the source code and answer the following questions:

- In your opinion, does this code component[17] exhibit any design and/or implementation problem?
- If YES, please explain what are, in your opinion, the problems affecting the code component.
- If YES, please rate the severity of the design and/or implementation problem by assigning a score on the following five-points Likert scale [37]: 1 (very low), 2 (low), 3 (medium), 4 (high), 5 (very high).
- In your opinion, does this class need to be refactored?
- if YES, how would you refactor this class?

On the one side, for questions related to *purely historical* smell instances detected by HIST, we also added hints on the change history of the code component (i.e., the same information exploited by HIST to detect that smell instance). This was needed to provide participants with information related to the historical behavior of the involved code components. Indeed, it is impossible to spot a problem as a *Parallel Inheritance* without knowing the number of times the addition of a subclass to a class C_i also resulted in the addition of a subclass to a class C_j. On the other side, for *structural* smells, no metrics were shown for instances identified by HIST as well as by the competitive techniques.

The questionnaires included six tasks[18] for APACHE ANT, ECLIPSE JDT, and ANDROID SDK, and seven tasks for APACHE FRAMEWORKS-BASE.

Besides the above described survey, we also asked participants to fill-in a brief pre-questionnaire in order to assess their background. In particular, we asked:

- How many years of experience do you have in programming?
- How many years of experience do you have in industry?
- Rate your programming skills from 1 = very low to 5 = very high.

Note that all the questions in the survey, as well as the background questions prefacing the survey, were designed to make sure that the survey could be completed within approximately 60 minutes. This is why we limited (i) the number of tasks and (ii) the number of questions in the background section, since a higher number could have resulted in a higher dropout rate before even starting the main survey.

4.1.4   Data Collection

To automatically collect the answers, the survey and background questions were hosted on a web application, *eSurveyPro*.[19] Developers were given 40 days to respond to the survey. Note that the web application allowed developers to complete a questionnaire in multiple rounds, e.g., to answer the first two questions in one session and finish the rest sometime later. At the end of the response period (i.e., of the 40 days), we collected developers' answers in a spreadsheet in order to perform data analysis. As explained before, in the end we collected 12 complete questionnaires (two developers from APACHE ANT, two from ECLIPSE, two from ANDROID SKD, and six from ANDROID FRAMEWORKS-BASE). Note that the developers of the four systems were invited to evaluate only the smells identified from the system that they were working on. Indeed, we are interested in gathering only data coming from original developers having sufficient knowledge of the analyzed source code components. Also, developers were not aware of the types of code smell investigated in our study.

4.1.5   Analysis Method

To answer RQ_3 we computed, for each type of historical smell:

1) The percentage of cases where the smell has been *perceived* by the participants. By *perceived*, we mean cases where participants answered *yes* to the question: "In your opinion, does this code component exhibit any design or coding problem?"

2) The percentage of times the smell has been *identified* by the participants. The term *identified* indicates cases where besides perceiving the smell, participants were also able to identify the exact smell affecting the analyzed code components, by describing it when answering to the question "If yes, please explain what are, in your opinion, the problems affecting the code component". We consider a smell as *identified* only if the design problems described by the participant are clearly traceable onto the definition of the smell affecting the code component. For example, given the following smell description for the *Feature Envy* smell: "*a method making too many calls to methods of another class to obtain data and/or functionality*", examples of "correct" descriptions of the problem are "*the method is too coupled with the C_i class*", or "*the method invokes too many methods of the C_i class*" where C_i is the class *envied* by the method. On the other hand, an answer like "*the method performs too many calls*" is not considered as sufficient to mark the smell as *identified*.

3) Descriptive statistics of answers provided by the participants to the question "please rate the severity of the coding problem". Note that for this point we just considered answers provided by developers that correctly *identified* the code smell.

4) The percentage of participants that answered yes to the question "does this class need to be refactored?". For participants answering "yes", we also report their responses to the question "how would you refactor this class?".

By performing this analysis for each historical code smell we should be able to verify if the instances of historical smells detected by HIST represent actual design problems for original developers.

---

17. Depending on the smell object of the question, a code component could be a method, a class, or a pair of classes.

18. By "task" we refer to the set of questions provided to a participant for each of the evaluated smell instances.

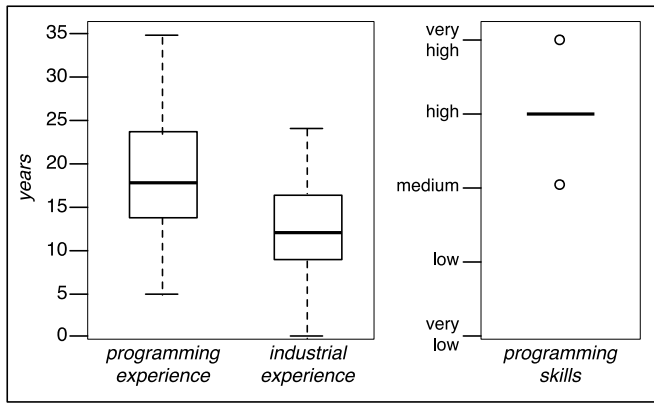19. http://www.esurveyspro.comverified on September 2014.

Fig. 7. Experience of the involved developers.

As for RQ_4, we perform the same exact analysis for each structural smell as described above for the historical smells. In addition, we compared the answers provided by participants for smell instances falling into three different categories (i.e., *onlyHIST*, *onlyDECOR/onlyJD*, and *both*). Given the limited number of data points, this comparison is limited to descriptive statistics only, since we could not perform any statistical tests.

4.1.6 Replication Package

All the data used in our second study are publicly available [40]. Specifically, we provide: (i) the text of the email sent to the developers; (ii) the raw data for the answers (anonymized) provided by the developers.

4.2 Analysis of the Results

Before discussing the results of our two research questions, it is worthwhile to comment on the experience of the developers involved in our study. Fig. 7 reports the boxplots of the distribution of answers provided by developers to questions related to their experience in the background section. Twelve developers claimed a programming experience ranging between 5 to 35 years (mean = 18.5, median = 17.5), industrial experience ranging between 1 to 24 years (mean = 12.7, median = 12). Most of them rated their programming skills as *high*. Thus, all 12 participants had some sort of industrial experience and, most importantly, several years of programming experience.

4.2.1 Are the Historical Code Smells Identified by HIST Recognized as Design Problems by Developers?

Fig. 8 reports the percentage of developers that *correctly identified* the smell present in the code component. As explained in the design, we computed both the percentage of developers that *perceived* and *identified* the smell.[20] However, in the context of this research question all developers who *perceived* the smell were also able to *identify* it. In addition, Fig. 9 reports the percentage of developers assigning each of the five levels of severity (going from *very low* to *very high*) to the identified design/implementation problems. Finally, Table 13 reports the percentage of developers

20. Note that the percentage of identified smells is a subset of the perceived one (see Section 4.1.3).
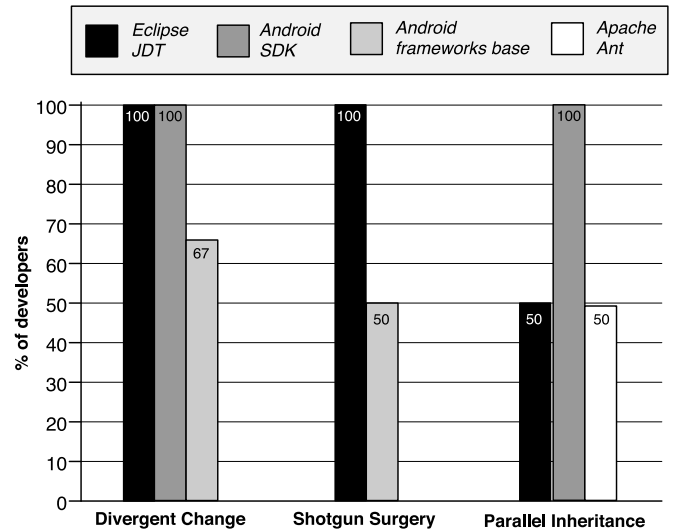
Fig. 8. RQ_3: percentage of *identified* smell instances.

that suggested refactoring operations for the identified smells. Their answers on how to refactor the smells are discussed in the text.

Starting from the *Divergent Change* instances identified by HIST, Fig. 8 shows that developers generally recognized them as a design/implementation problems. Indeed, the two ECLIPSE JDT developers, the two ANDROID SDK developers, and four out of the six involved ANDROID FRAMEWORKS-BASE developers were able to perceive and identify the presence of a *Divergent Change* instance in the analyzed code components. Most of these developers pointed out low cohesion of the class as the root cause for the identified design problem. Low cohesion of classes is clearly a symptom of a *Divergent Change* smell. Indeed, classes having low cohesion tend to implement different responsibilities, that are likely to be changing divergently during time. Interesting is the refactoring suggested by one of the developers of ANDROID FRAMEWORKS-BASE recognizing this smell in the `PackageManagerService` class:

> Make a new separate helper class for talking to the phone's file system.

In other words, the developer is suggesting performing an Extract Class refactoring aimed at removing one responsibility from the `PackageManagerService`, and in particular the management of the phone file system. Concerning the severity of the problem as assessed by the developers identifying the smell, Fig. 9 shows that 25 percent of them
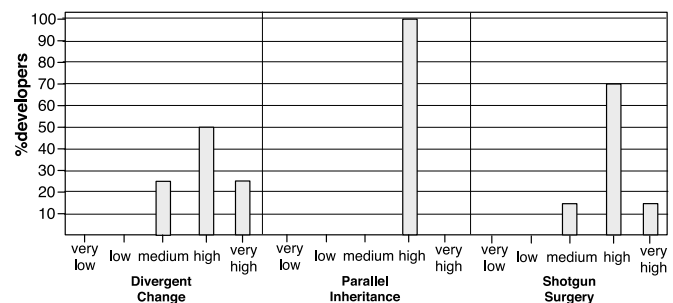

Fig. 9. RQ_3: severity assigned by developers to the *identified* instances of poorly historical smells detected by HIST.

TABLE 13
RQ_3: Percentage of Developers in Favor of Refactoring
the Class among Those Correctly Identifying the Smells

Code Smell	% in favor
Divergent Change	100%
Parallel Inheritance	100%
Shotgun Surgery	75%

rate the severity as *medium*, 50 percent as *high*, and 25 percent as *very high*. Also, all of them agreed on the need to refactor the classes affected by *Divergent Change*.

As for the *Shotgun Surgery* smell, we have instances of this smell just in two out of the four subject systems (i.e., ECLIPSE JDT and ANDROID FRAMEWORKS-BASE). The two involved ECLIPSE JDT developers recognized presence of this smell, explaining how *the high (and spread) coupling of some of the methods contained in the* `MethodLocator` *class could represent a design/implementation problem*. Indeed, the basic ingredient for the appearance of a *Shotgun Surgery* smell is to have methods depending on several other classes, like the `isAsync` method showed in Fig. 5 in the context of our first study.

Three of the ANDROID FRAMEWORK-BASE developers (50 percent) identified the presence of a *Shotgun Surgery* instance in the `Handler` class as an implementation/design problem. One of them pointed out that:

> *Handler is tightly coupled to a few other classes: Message, MessageQueue and Looper. Each class has knowledge about members of the other classes. From a strict Object Oriented Programming perspective this is not optimal.*

However, the developer explained that from his point of view in this case the class affected by the smell should not be refactored, because:

> *At first glance the coupling looks like a problem, but these classes are best viewed as one unit. If you accept that perspective, the design problem just isn't there. There may also be performance benefits of accessing members in the other classes directly. For example, mLooper.mQueue instead of mLooper.getQueue(). It makes sense to trade design for performance for a class at the very core of the message loop.*

This example shows exactly what a smell is all about: it is a symptom in the code that may (or may not) indicate a design problem. Also, the example highlights the importance of this evaluation. Indeed, a smell detection tool should be able to point out smell instances representing an implementation/design problem that software developers are interested in refactoring. Note that the developer above is the only one who did not recognize the need to refactor `Handler` class. Concerning the severity of the identified *Shotgun Surgery* instances, 70 percent of developers assessed the severity as *high*, 15 percent to *very high*, and the remaining 15 percent to *medium*. Thus, the instances of *Shotgun Surgery* identified by HIST are mostly recognized as serious problems by the developers of these subject systems.

The *Parallel Inheritance* smell affects three of the subject systems (see Fig. 8). This smell was the one among the least perceived (and identified) by developers. Still, one of the
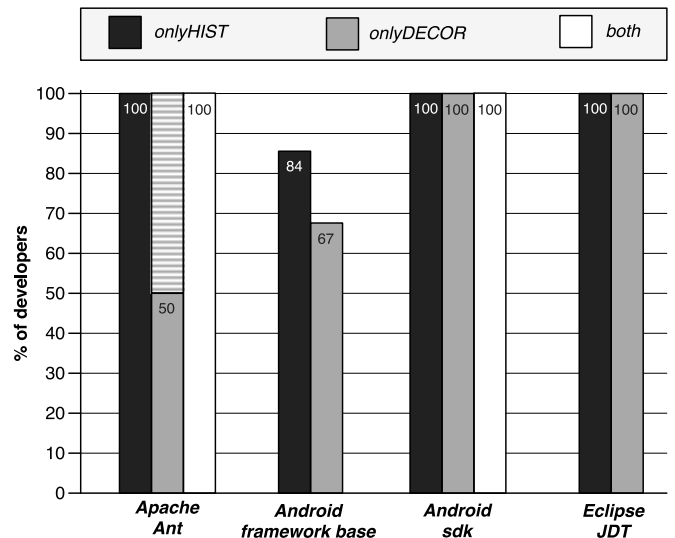
Fig. 10. RQ_4: percentage of *perceived* and *identified Blob* instances.

two involved developers of ECLIPSE JDT and APACHE ANT systems as well as both the developers of ANDROID SDK recognized its presence, talking about *problems in the design hierarchy*. All four developers recognizing the smell, assessed its severity as *high* and suggested to refactor it by moving responsibilities across the hierarchies. This could be done by applying move method refactoring as well as pull up/push down method/field refactorings.

Summary for RQ_3. Developers recognized most of the instances of historical smells identified by HIST as design/implementation problems. Indeed, they recognized 71 percent of the evaluated smell instances (17 out of 24) as such. Also, developers mostly assessed the severity of the problems caused by the presence of the historical smells as *high*, manifesting the willingness to refactor affected classes in 100 percent of cases in the presence of *Divergent Change* and *Parallel Inheritance* instances and in 75 percent of cases in the presence of *Shotgun Surgery* instances. Thus, we conclude that *the historical code smells identified by HIST are recognized as actual design problems by developers in most of the cases.*

4.2.2 Which Detection Technique Aimed at Identifying Structural Code Smells Better Reflects Developers' Perception of Design Problems?

Starting from the *Blob* smell, Fig. 10 reports the percentage of developers who *perceived* (the striped columns) and *identified* (the filled columns) the *Blob* instances belonging to the *onlyHIST*, *onlyDECOR*, and *both* groups. Also, the left part of Fig. 11 reports the percentage of developers assigning each of the five severity levels (going from *very low* to *very high*) to the identified *Blobs*. Finally, the top part of Table 14 reports the percentage of developers that suggested a refactoring for the identified *Blobs*.

We have instances of *Blobs* identified only by HIST on all four subject systems. Among the 12 involved developers, only one developer of ANDROID FRAMEWORKS-BASE did not recognize the evaluated *Blob* instance belonging to the *onlyHIST* group. The remaining eleven developers (92 percent) clearly described the problem affecting the analyzed class. For
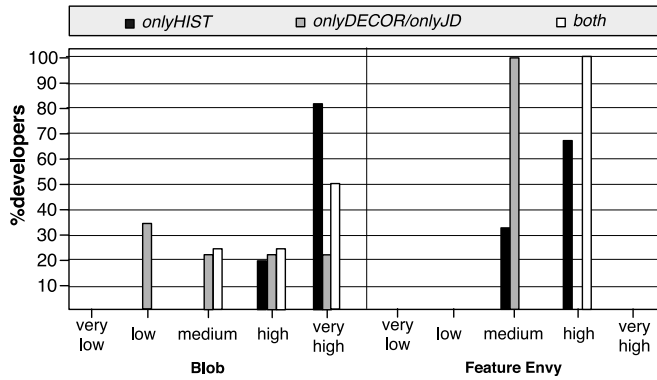
Fig. 11. RQ_4: severity assigned by developers to the *identified* instances of poorly historical smells detected by HIST (*onlyHIST*), by the competitive technique (*onlyDECOR* or *onlyJD*), and by *both*.

example, an ECLIPSE JDT developer, referring to the analyzed class SourceMapper, wrote: *"this is a well known Blob in Eclipse"*; an *Android frameworks-base* developer explained, evaluating the class WindowManagerService: *"it is a very large and complex class"*. The eleven developers recognizing the *Blob* instances also evaluated the severity of the problem as *high* (18 percent) or *very high* (82 percent)—see left part of Fig. 11—manifesting the willingness to refactor such classes in 100 percent of cases (see Table 14). Most of the developers suggested to perform an Extract Class refactoring to remove the smell (e.g., *"make the class easier to comprehend by splitting its responsibilities into different classes"*, from an *Android frameworks-base* developer). Thus, the *Blob* instances detected by HIST and missed by the competitive technique (i.e., DECOR) have been mostly recognized by developers as design/implementation problems. Also, the developers recognized the high severity of the issue caused by the presence of the smell, manifesting the willingness to refactor such classes.

As for the *Blob* instances detected by DECOR and missed by HIST, nine out of the 12 developers (75 percent) recognized them as design/implementation problems. In addition, one of the Apache Ant developers *perceived* the smell but failed to identify it[21] (see Fig. 10). Concerning the severity assessed for the *Blob* instances identified in the *onlyDECOR* group, Fig. 11 shows that 34 percent of developers selected a *low* severity, 22 percent *medium*, 22 percent *high*, and 22 percent *very high*. Also, 78 percent of developers recognized the need to refactor those *Blob* instances.

The third group of *Blob* instances to analyze is the one grouping together *Blobs* detected by both HIST and DECOR (*both* groups). We have instances of these *Blobs* only in APACHE ANT and ANDROID SDK. Interestingly, all developers recognized the *Blob* instances belonging to the *both* group, even if the severity assigned to them is lower than the severity assigned to the instances belonging to the *onlyHIST* group (see Fig. 11). This result is quite surprising. Indeed, one would expect a very high severity for smells identified by both detection techniques. Still, the assessed severity is *medium* (25 percent), *high* (25 percent), or *very high* (50 percent). Moreover, in 100 percent of the cases developers agreed on the importance of refactoring the *Blob* instances belonging to the *both* group.

21. The developer described problems in a method manipulating jar files.

TABLE 14
RQ_4: Percentage of Developers in Favor of Refactoring the Class among Those Correctly Identifying the Smells

Code Smell	Detected by	% in favor
Blob	*onlyHIST*	100%
	onlyDECOR	78%
	both	100%
	onlyHIST	100%
Feature Envy	*onlyJD*	100%
	both	100%

Summarizing, the *Blob* instances detected by both techniques are the ones that are mostly recognized by developers (100 percent of the developers), followed by the ones detected by HIST only (95 percent) and those detected by DECOR only (75 percent). The instances recognized as more severe problems are those identified by HIST only (82 percent *very high*), followed by those detected by both techniques (50 percent *very high*), and those detected by DECOR only (22 percent *very high*). Finally, all the developers agreed on refactoring the *Blob* instances detected by both techniques as well as those detected by HIST only, while 78 percent of developers agreed on refactoring the *onlyDECOR* instances.

Thus, when comparing HIST to DECOR, the *Blob* instances detected by DECOR only are (i) identified by fewer developers, (ii) evaluated with a much lower severity level, and (iii) recognized as less likely refactoring opportunities by developers. Still, the fact that 75 percent of developers recognized the smells points out to the conclusion that complementing HIST with structural information (e.g., DECOR) could be a worthwhile direction in order to identify currently missed *Blob* instances. This result confirms the results of our first study, further highlighting complementarity of the two techniques.

Turning the attention on the *Feature Envy* smell, Fig. 12 shows the percentage of developers who *perceived* (the
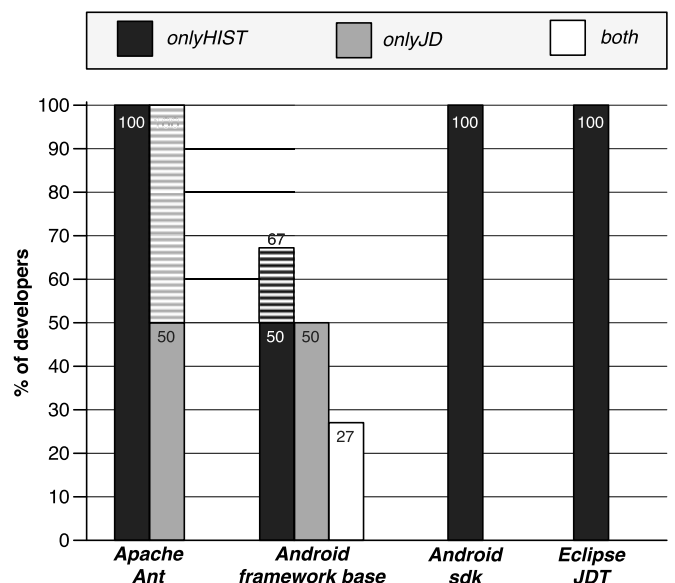

Fig. 12. RQ_4: percentage of *perceived* and *identified Feature Envy* instances.

striped columns) and *identified* (the filled columns) the *Feature Envy* instances belonging to the *onlyHIST*, *onlyJD*, and *both* groups. As before, Fig. 11 (right part) reports the severity assigned by developers to the identified smell instances, while Table 14 reports the percentage of developers that would like to refactoring the smell.

The *Feature Envy* instances falling in the *onlyHIST* group (black columns in Fig. 12) have been recognized as design/implementation problems by nine out of 12 (75 percent) involved developers. In particular, all the developers of APACHE ANT, ANDROID SDK, and ECLIPSE JDT identified the smell instances, while only three of the six ANDROID FRAMEWORK BASE developers recognized the problem. Developers recognizing the problem generally described the issue explaining that the analyzed method has high coupling with another class (i.e., the envied class). For example, while analyzing the ECLIPSE JDT method `generateCode` from class `AND_AND_Expression`, one of the developers explained that *"generateCode is a very complex method and it is highly coupled with the CodeStream class"*. `CodeStream` is exactly the class identified by HIST as the envied class for the `generateCode` method.

Concerning the severity assigned to the smell by the nine developers identifying it, 67 percent rated it as *high*, while 33 percent as *medium* (see Fig. 11). Moreover, all nine developers suggested to refactor this smell (see Table 14) proposing a *Move Method* toward the envied class, or an *Extract Method* followed by a *Move Method*.

As for the *Feature Envy* instances identified by JDeodorant only, we have instances of them just on APACHE ANT and ANDROID FRAMEWORK BASE. On APACHE ANT both developers perceived a problem in the analyzed *Feature Envy* instance (i.e., the RUN method from the `ClearCase` class), but only one correctly identified the smell. On the ANDROID FRAMEWORK BASE, among the six involved developers three identified a *Feature Envy* in the method under analysis (i.e., `executeLoad` from the `FrameLoader` class). Thus, four out of the eight evaluators (50 percent) identified the *Feature Envy* instances in the *onlyJD* group. All of them assessed the severity of the spotted instances of the smell as *medium*, manifesting some readiness to refactor them.

Finally, the only instance falling in the *both* group belongs to the ANDROID SDK system. This instance has been identified by both involved developers, that assessed its severity as *high*, and suggested a *Move Method* refactoring to solve the problem. This confirms, in part, what we observed for the *Blob* smell: when both HIST and the techniques based on a single snapshot analysis detect a code smell, all involved developers identify the smell and suggest appropriate refactoring operation.

Summarizing, the *Feature Envy* instances detected by both techniques are the most recognized by developers (100 percent of developers), followed by the ones detected by HIST only (75 percent) and those detected by JDeodorant only (38 percent). Also, the instances recognized as more severe problems are those detected by both techniques (100 percent *high*), followed by those detected by HIST only (67 percent *high*), and those detected by JDeodorant only (100 percent *medium*). Despite these differences, all the developers identifying the *Feature Envy* instances falling in the three different groups (i.e., *onlyHIST*, *onlyJD*, and *both*) suggested to refactor them.

Summary for RQ₄. The smells that perfectly reflect the developers' perception of design problems are those identified by both HIST and techniques based on code analysis of a single snapshot—100 percent of the involved developers for both *Blob* and *Feature Envy* instances. However, the smells identified by HIST only are highly recognized as design problems by developers—95 percent of participants for *Blob* instances and 75 percent for *Feature Envy* instances. Also, smells belonging to the *onlyHIST* group are more frequently recognized as problems by developers than those solely identified by the competitive technique. In summary, this study confirms that there is a potential to combine historical and structural information to achieve a better smell detection.

5 THREATS TO VALIDITY

This section discusses the threats that could affect the validity of the HIST evaluation.

5.1 Construct 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 Study I, this is mainly due to how the oracle was built (see Section 3.1.3). It is important to remark that to mitigate the bias for such a task, the students who defined the oracle were not aware of how HIST actually works. However, we cannot exclude that such manual analysis could have potentially missed some smells, or else identified some false positives. Another threat is due to the baselines—i.e., competitive approaches—against which we compared HIST. While for *Blob*, *Feature Envy*, *Divergent Change*, and *Shotgun Surgery* we compared HIST against existing techniques/tools, this was not possible for the *Parallel Inheritance* smell, for which we had to define an alternative static detection technique, 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 (for the *Blob* detection) and the approach by Rao and Raddy [42] (for *Divergent Change* and *Shotgun Surgery*) ourselves, these are precisely defined by the authors.

As for Study II, threats to *construct validity* are mainly related to how we measured the developers' perception of smells. As explained in Section 4.1.3, we asked developers to tell us whether they perceived a problem in the code shown to them. In addition, we asked them to explain what kind of problem they perceived to understand whether or not they actually identify the smell affecting the code component as the design and/or implementation problem. Finally, for the severity we used a Likert scale [37] that permits the comparison of responses from multiple respondents. We are aware that questionnaires could only reflect a subjective perception of the problem, and might not fully capture the extent to which the smell instances identified by HIST and by the competitive techniques are actually perceived by developers.

5.2 Internal Validity

Threats to *internal validity* concern factors that could have influenced our results. In both studies, a possible threat is represented by the calibration of the HIST parameters, as well as of those of the alternative static approaches. We performed the calibration of these parameters on one project (Xerces) not used in our study, by computing F-measures for different possible values of such parameters (see Section 3.1.4).

A factor that could have affected the results of Study II is also the response rate: while appearing not very high (11 percent), it is inline what it is normally expected in survey studies (i.e., below 20 percent [6]). Note also that we just targeted for this study original developers of the four open source systems, without taking into account the possibility of involving students or people with no experience on the object systems. Still, we cannot ensure that the involved developers had a good knowledge of the specific code components used in our surveys. An alternative design would have been to invite only developers actually involved in the development of the specific code components evaluated in our survey. However, (i) the different code components present in our survey are evolved and maintained by different developers, and (ii) this would have resulted in a much lower number of developers invited, having as a consequence a very likely drop in the number of participants in our study.

Also, we tried to keep the questionnaire as short as possible to have more developers answering our survey. For instance, we did not include any questions on non-smelly code entities as sanity check in our survey. Thus, we cannot exclude that participants always indicated that the analyzed code components contained a design/implementation problem and the problem was a serious one. However, this holds for the smell instances identified by HIST as well as for those identified by the competitive techniques.

Still in the context of Study II, it must be clear that even if developers recognized most of the code smell instances identified by HIST and declared that they wanted to refactor them, this does not always mean that it is possible to take proper refactoring actions aimed at removing those smells. Indeed, some systems—e.g., Eclipse JDT—contain classes that naturally tend to become smelly. For example, parsers (largely present in the Eclipse JDT) are often affected by the *Blob* code smell [7], and are difficult to remove without taking important (and expensive) refactoring actions.

5.3 External Validity

Threats to *external validity* concern the generalization of the results. HIST only deals with five smells, while there might be many more left uncovered [9], [14]. However, as explained in Section 2 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 in part, by observing source code changes over time. However, we cannot exclude that there could be other smells that can be modeled similarly.

As for the first study, we conducted it on 20 Java projects ensuring a good generalization of our findings. We
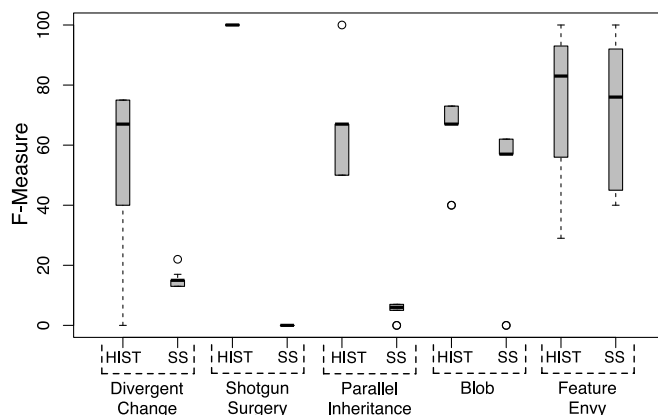

Fig. 13. HIST versus single-snapshot competitive techniques (SS): F-Measure achieved for each smell type on the 10 Cassandra releases.

evaluated HIST and the competitive techniques on a specific system's snapshot selected by splitting the history of each object system in two equal parts. Thus, the achieved results, and in particular our main finding in the context of RQ_2 (i.e., *HIST was able to outperform single snapshot techniques and tools in terms of recall, precision, and F-measure*), might be influenced by the specific selected snapshot. To mitigate such a threat, we replicated Study I on 10 snapshots representing 10 different releases of a single system, namely APACHE CASSANDRA. In particular, we considered CASSANDRA's releases from 0.5 to 1.1.[22] Note that we just performed this analysis on a single system since it required the manual definition of 10 new oracles (i.e., one for each release) reporting the smell instances present in each release. The oracle definition was performed by two Master's students (one of which was also involved in the definition of the 20 oracles exploited in Study I) by adopting the same procedure described in Section 3.1.3. We run HIST and the competitive techniques on the 10 snapshots representing the 10 releases. Such snapshots have been identified by exploiting the git tagging mechanism. The results achieved are high consistent when comparing HIST and the competitive techniques. Fig. 13 reports the boxplots of the F-Measure achieved by HIST and by the competitive techniques on the 10 CASSANDRA releases for each of the five considered code smells. The achieved results can be summarized as follows:

- *Divergent change*. HIST achieves a higher F-Measure with respect to the competitive technique (i.e., DCPP) in nine out of the 10 considered releases (all but CASSANDRA 0.5).
- *Shotgun surgery*. HIST achieves a higher F-Measure with respect to the competitive technique (i.e., DCPP) in nine out of the 10 considered releases (all but CASSANDRA 0.5). In CASSANDRA 0.5, a tie is reached, since no instances of the Shotgun Surgery smell are present, and both detection techniques do not retrieve any false positive.

22. We discarded the first four releases (i.e., from release 0.1 to release 0.4) since change-history information for these four releases was not present in the versioning system.

- *Parallel inheritance*. HIST achieves a higher F-Measure with respect to the competitive technique (i.e., PICA) in all 10 considered releases.
- *Feature envy*. HIST achieves a higher F-Measure with respect to the competitive technique (i.e., JDeodorant) in six out of the 10 considered releases, JDeodorant works better on two releases (the first two, CASSANDRA 0.5 and 0.6), while a tie is reached on the remaining two releases.
- *Blob*. HIST outperforms the competitive technique (i.e., DECOR) in all 10 considered releases.

Interestingly, when the competitive techniques outperform HIST, the releases involved are the 0.5 (in case of Divergent Change, Shotgun Surgery, and Feature Envy) and the 0.6 (in case of Feature Envy), representing the first two considered releases. Thus, we can conclude that a shorter change history penalizes HIST as compared to the competitive techniques. Such a limitation is typical of all approaches exploiting historical information to derive recurring patterns. Despite that, the overall results achieved on the release-snapshots confirm our main finding reported while answering RQ_2: HIST outperforms the competitive detection techniques based on code analysis of a single system snapshot. The interested reader can find detailed results about this analysis in our replication package [40].

Despite the effort we put in extending our evaluation to a high number of systems, it could be worthwhile to replicate the evaluation on other projects having different evolution histories or different architectures (e.g., plugin-based architecture). Also, the number of code smell instances present in our oracle was quite low for the *Shotgun Surgery* smell (six instances). However, while this means evaluating the HIST performances on a small number of "true positive" instances, it is worth noting that achieving high precision levels is even harder when the number of correct instances in the oracle is low. Indeed, it is easier to identify a high number of false positives when the true positives in the oracle are very few. Despite this, HIST achieved an average precision of 86 percent for such a smell.

Concerning Study II, *external validity* threats can be related to the set of chosen objects and to the pool of the participants to the study. Concerning the chosen objects, we are aware that our study is based on smell instances detected in four Java systems only, and that further studies are needed to confirm our results. In this study we had to constrain our analysis to a limited set of smell instances, because the task to be performed by each respondent had to be reasonably small (to ensure a decent response rate).

6   RELATED WORK

This section analyzes the literature related to (i) the identification of code smells in source code; and (ii) the analysis of the evolution of code smells in existing software systems.

6.1   Methods and Tools to Detect Bad Smells

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: [10], [14], [45], [50]. The first by Webster [50] 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 [45] defines more than 60 guidelines to rate the integrity of a software design. Fowler [14] defines 22 code smells together with refactoring operations to remove them from the system. Finally, Brown et al. [10] 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. [47] 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. [46] 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. If it is possible to identify different sets of cohesive attributes and methods in a class, then an Extract Class refactoring opportunity is identified.

van Emden and Moonen [49] 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*. The first occurs when there is a concentration of *instanceof* operators in the same block of code making the code difficult to understand. As for the *typecast* code smell, it appears when an object is explicitly converted from one class type into another, possibly performing illegal casting which results in a runtime error.

Kamiya et al. [21] introduced the tool CCFinder in order to identify clones in source code. In particular, they used a syntactic-based approach where the program is divided in lexemes and the token sequences are compared in order to find matches between two subsequences. However, such approach appear to be ineffective in cases where duplicated code suffers from several modifications during its evolution. To mitigate such a problem, Jiang et al. [18] introduced DECKARD, a technique able to identify clones using a mix of tree-based and syntactic-based approaches. They first translate the source code into syntax tree, and then complement it with the syntactic information in form of vectors that are subsequently clustered. To detect clones, heuristic rules are applied on the clusters.

Marinescu [30] proposed 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* for measuring 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 a set of thresholds based on the measurement of the exploited metrics in real software systems. The detection strategies are formulated in different steps. First, the *symptoms* that characterize a particular smell are defined. Second, a proper *set of metrics* measuring these symptoms is identified. Having this information, the next step is to define thresholds to classify the class as affected (or not) by the

defined symptoms. Finally, AND/OR operators are used to correlate the symptoms, leading to the final rule for detecting the smells.

Munro [35] presented a metric-based detection technique able to identify 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, LOC, weight methods per class (WMC), and Coupling Between Objects (CBO).

Khomh et al. [25] proposed 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. [36] for identifying instances of *Blobs* in source code.

Tsantalis and Chatzigeorgiou [48] presented 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. In its current version JDeodorant[23] is also able to detect other three code smells (i.e., *State Checking*, *Long Method*, and *God Classes*).

Moha et al. [33] introduced DECOR, a technique 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*.

Kessentini et al. [22] presented a technique to detect design defects by following the assumption that what significantly diverges from good design practices is likely to represent a design problem. The advantage of their approach is that it does not look for specific code smells (as most of approaches, including HIST, do) but for design problems in fwnweL. Also, in the reported evaluation [22] the approach was able to achieve a 95 percent precision in identifying design defects.

Boussaa et al. [8] proposed the use of competitive coevolutionary search to the code-smells detection problem. In particular, two populations evolve simultaneously: the first generates detection rules with the aim of detecting the highest possible proportion of code smells, whereas the second population generates smells that are currently not detected by the rules of the other population.

Ligu et al. [28] introduced the identification of *Refused Bequest* code smell using a combination of static source code analysis and dynamic unit test execution. Their approach aims at discovering classes that really *wants to support the interface of the superclass* [14]. In order to understand what are the methods really called on subclass instances, they 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 occur. Otherwise, the method is never called and an instance of *Refused Bequest* is found.

All the approaches mentioned so far exploit information extracted from source code—e.g., quality metrics—to detect code smells. Differently, HIST, the approach described in this paper, exploits change-history information extracted from versioning systems for the identification of code smells in source code. From this point of view, the two most related approaches are those by Ratiu et al. [44] and by Gîrba et al. [17].

Ratiu et al. [44] describe an approach for detecting smells based on evolutionary information of problematic code components (as detected by code analysis) over their lifetime. The aim is to measure persistence of the problem and related maintenance effort spent on the suspected components. This work is the closest to HIST since it discusses the role of historical information for smell detection. However, Ratiu et al. did not explicitly use historical information for detecting smells (as done by HIST), but they only performed multiple code analysis measurements of design problems during the history of code components. Historical information have also been used by Lozano et al. [29] to assess the impact of code smells on software maintenance.

Gîrba et al. [17] exploited formal concept analysis for detecting co-change patterns. In other words, they identified code components that change in the same way and in the same time. The authors explain how the proposed technique could be applied for detecting instances of *Shotgun Surgery* and *Parallel Inheritance*. However, the performances of their approach for the detection of these two smells are not reported. It is worth noting that while the basic idea behind the work by Gîrba et al. is similar to the one behind HIST, our approach exploits a totally different underlying mechanism (i.e., association rules versus FCA) providing additional information on the degree to which code components co-change during time (i.e., support and confidence). Also, while the approach by Gîrba et al. performs a change analysis at release level (i.e., the changes to each code component are detected each time a release is issued), HIST relies on finer-grained information extracting the changes at commit level.

Historical information has been also exploited by Ouni et al. [38] in the context of a multi-objective optimization-based approach aimed at identifying the best sequence of refactoring operations that minimizes the number of bad-smells in a system under analysis while maximizing the consistence with the development history. While we share with Ouni et al. the use of historical information to improve the internal quality of software, HIST is a smell detector also focused on historical smells, while the approach by Ouni et al. is a refactoring recommendation system.

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. [53], Zimmermann et al. [54], [55], Gall et al. [15], and Kagdi [16], [19], [20] for identifying logical change couplings, and by Adams et al. [2], Mulder and Zaidman [34], and Canfora et al. [10] for the identification of crosscutting concerns. Although the underlying technique

---

23. http://www.jdeodorant.com/verified on February 2014.

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 2, a fine-grained analysis, identifying co-changes at method-level, is often required.

6.2 Empirical Studies on Bad Smells

Code code smells have also been widely studied in order to investigate their evolution and their effect on maintenance activities.

As far as the evolution of code smells is concerned, Chatzigeorgiou and Manakos [11] demonstrated that the number of code smells increases during the evolution of the system and that the developers are reluctant to perform activities aimed at their removal. Also Peters and Zaidman [41] obtained similar results showing how, even if the developers are aware about the presence of code smells, they do not care to perform refactoring activities to remove them from the system. The reasons of this behavior are explained by Arcoverde et al. [4], who reported a survey in order to understand the longevity of code smells and showed that code smells often remain in source code for a long time and the main reason to postpone their removal through refactoring activities is to avoid API modifications [4].

As for the studies aimed at analyzing evolution of code smells, there are several empirical studies targeted at investigating their impact on maintenance properties, such as comprehensibility and change- or fault- proneness. Abbes et al. [1] investigated how two types of code smells—Blob and Spaghetti Code—impact program comprehension. The results demonstrate that, while the presence of a code smell does not decrease significantly the developers' performance, a combination of more code smells in the same class is able to destabilize the developers and, consequently, reduce their performance. Also Yamashita and Moonen [52] studied the interaction of different code smells, obtaining similar results. Indeed, they showed that the maintenance problems are strictly related to the presence of more code smells in the same file. At the same time, they studied the impact of code smells on maintainability characteristics [51]. In particular, they were the first to investigate the key maintainability factors that are relevant for the developers and then identify which code smells related to these maintainability problems.

Regarding to change- and fault-proneness, Khomh et al. [23], [24] demonstrated how code smells make the source code much more change prone [23] and fault-prone [24] with respect to the other files. Moreover, they provided evidence that code involved in code smells is more fault-prone than the other files in the system. Khomh et al. also demonstrated that there are some code smells—such as Message Chains—where the phenomenon is amplified [24].

The correlation between the presence of code smells and the probability that the class has errors has also been empirically evaluated by Li and Shatnawi [27]. They studied the post-release system evolution process demonstrating that there are many code smells positively correlated with class errors.

All these studies provide evidence that code smells have negative effects on maintenance properties such as understandability. However, it is still unclear whether developers would actually consider code smells as actual symptoms of suboptimal design/implementation choices. In other words, there seems to be a gap between the theory and the practice. Our second study partially bridged this gap, providing empirical evidence on the extent to which code smells identified by HIST and by techniques based on the analysis of a single snapshot are perceived by original developers as actual design or implementation problems.

7 CONCLUSION AND FUTURE WORK

We presented HIST, an approach aimed at detecting five different code bad smells by exploiting 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 [3] or analyzing the set of classes/methods co-changed with the suspected smell.

We evaluated HIST through two empirical studies. In the first study, we assessed its recall and precision over a manually-built oracle of smells identified in 20 Java open source projects, and compared it with alternative smell detection approaches based on the analysis of a single project snapshot. The results of our study indicate that HIST exhibits a precision between 72 and 86 percent, and a recall between 58 and 100 percent. For "intrinsically historical" smells—such as *Divergent Change*, *Shotgun Surgery*, *Parallel Inheritance*—HIST clearly outperforms approaches based on the analysis of a single snapshot, and generally performs as well these latter (if not better) for *Blob* and *Feature Envy* smells. Besides the better detection accuracy (in terms of precision and recall), HIST has a further advantage: it highlights smells that are subject to frequent changes, and therefore be possibly more problematic for the maintainer. In other words, a *Blob* detected based on structural information might not be necessarily a problem if it rarely (or never) changes, whereas it is worthwhile to bring to the attention of the developer those changing very frequently, hence identified by HIST. Finally, it is important to remark that in several cases the sets of smells detected by HIST and by techniques analyzing a single system's snapshot are quite complementary, suggesting that better techniques can be built by combining them.

In a second study, we involved 12 original developers of four open source systems to understand to what extent smell instances identified by HIST and by the competitive techniques are felt as design/implementation problems by developers. The results achieved indicated that over 75 percent of smell instances identified by HIST are also recognized by developers as actual design/implementation problems. In addition, this study showed that smell instances identified by both HIST and the single-snapshot techniques are the ones that perfectly match developers' perception of design/implementation problems. This result, together with the high complementarity between HIST and some of the competitive techniques observed in our first study, triggers our future research agenda, aimed at developing a hybrid smell detection approach, obtained by combining static code analysis with analysis of co-changes.

Also, we are planning on investigating the applicability of HIST to other types of smells. Finally, we would like to perform a deeper investigation into the characteristics causing a smell instance to represent/not represent a problem for developers.

ACKNOWLEDGMENTS

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 *Proc. 15th Eur. Conf. Softw. Maintenance Reeng.*, Oldenburg, Germany, Mar. 1–4, 2011, pp. 181–190.

[2] B. Adams, Z. M. Jiang, and A. E. Hassan, "Identifying crosscutting concerns using historical code changes," in *Proc. 32nd ACM/IEEE Int. Conf. Softw. Eng.*, Cape Town, South Africa, May 1–8, 2010, pp. 305–314.

[3] R. Agrawal, T. Imielinski, and A. N. Swami, "Mining association rules between sets of items in large databases," in *Proc. 1993 ACM SIGMOD Int. Conf. Manage. Data*, 1993, pp. 207–216.

[4] R. Arcoverde, A. Garcia, and E. Figueiredo, "Understanding the longevity of code smells: Preliminary results of an explanatory survey," in *Proc. Int. Workshop Refactoring Tools*, 2011, pp. 33–36.

[5] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*. Reading, MA, USA: Addison-Wesley, 1999.

[6] Y. Baruch, "Response rate in academic studies a comparative analysis," *Hum. Relat.*, vol. 52, no. 4, pp. 421–438, 1999.

[7] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto, "Automating extract class refactoring: An improved method and its evaluation," *Empir. Softw. Eng.*, vol. 19, no. 6, pp. 1617–1664, 2013.

[8] M. Boussaa, W. Kessentini, M. Kessentini, S. Bechikh, and S. Ben Chikha, "Competitive coevolutionary code-smells detection," in *Proc. 5th IEEE Symp. Search Based Softw. Eng.*, 2013, pp. 50–65.

[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*, 1st ed. New York, NY, USA: Wiley, Mar. 1998.

[10] G. Canfora, L. Cerulo, and M. Di Penta, "On the use of line co-change for identifying crosscutting concern code," in *Proc. 22nd IEEE Int. Conf. Softw. Maintenance*, Philadelphia, PA, USA, Sep. 24–27, 2006, pp. 213–222.

[11] A. Chatzigeorgiou and A. Manakos, "Investigating the evolution of bad smells in object-oriented code," in *Proc. Int. Conf. Qual. Inf. Commun. Technol.*, 2010, pp. 106–115.

[12] M. L. Collard, H. H. Kagdi, and J. I. Maletic, "An xml-based lightweight c++ fact extractor," in *Proc. 11th Int. Workshop Program Comprehension*, Portland, OR, USA, May 10–11, 2003, pp. 134–143.

[13] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "JDeodorant: Identification and application of extract class refactorings," in *Proc. 33rd Int. Conf. Softw. Eng.*, Waikiki, Honolulu, HI, USA, May 21–28, 2011, pp. 1037–1039.

[14] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Reading, MA, USA: Addison-Wesley, 1999.

[15] H. Gall, K. Hajek, and M. Jazayeri, "Detection of logical coupling based on product release history," in *Proc. 14th IEEE Int. Conf. Softw. Maintenance*, 1998, pp. 190–198.

[16] M. Gethers, B. Dit, H. Kagdi, and D. Poshyvanyk, "Integrated impact analysis for managing software changes," in *Proc. 34th IEEE/ACM Int. Conf. Softw. Eng.*, 2012, pp. 430–440.

[17] T. Gîrba, S. Ducasse, A. Kuhn, R. Marinescu, and R. Daniel, "Using concept analysis to detect co-change patterns," in *Proc. 9th Int. Workshop Principles Softw. Evol.: Conjunction 6th ESEC/FSE Joint Meeting*, 2007, pp. 83–89.

[18] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and accurate tree-based detection of code clones," in *Proc. Int. Conf. Softw. Eng.*, 2007, pp. 96–105.

[19] H. Kagdi, M. Gethers, D. Poshyvanyk, and M. Collard, "Blending conceptual and evolutionary couplings to support change impact analysis in source code," in *Proc. 17th Working Conf. Reverse Eng.*, 2010, pp. 119–128.

[20] H. Kagdi, M. Gethers, and D. Poshyvanyk, "Integrating conceptual and logical couplings for change impact analysis in software," *Empirical Softw. Eng.*, vol. 18, no. 5, pp. 933–969, 2013.

[21] T. Kamiya, S. Kusumoto, and K. Inoue, "CCfinder: A multilinguistic token-based code clone detection system for large scale source code," *J. Trans. Softw. Eng.*, vol. 28, no. 7, pp. 654–670, Jul. 2002.

[22] M. Kessentini, S. Vaucher, and H. Sahraoui, "Deviance from perfection is a better criterion than closeness to evil when identifying risky code," in *Proc. IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2010, pp. 113–122.

[23] F. Khomh, M. Di Penta, and Y.-G. Guéhéneuc, "An exploratory study of the impact of code smells on software change-proneness," in *Proc. 16th Working Conf. Reverse Eng.*, Lille, France, Oct. 13–16, 2009, pp. 75–84.

[24] 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 Softw. Eng.*, vol. 17, no. 3, pp. 243–275, 2012.

[25] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "A Bayesian approach for the detection of code and design smells," in *Proc. 9th Int. Conf. Qual. Softw.*, 2009, pp. 305–314.

[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*. New York, NY, USA: Springer, 2006.

[27] W. Li and R. Shatnawi, "An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution," *J. Syst. Softw.*, vol. 80, pp. 1120–1128, 2007.

[28] E. Ligu, A. Chatzigeorgiou, T. Chaikalis, and N. Ygeionomakis, "Identification of refused bequest code smells," in *Proc. 29th IEEE Int. Conf. Softw. Maintenance*, 2013, pp. 392–395.

[29] A. Lozano, M. Wermelinger, and B. Nuseibeh, "Assessing the impact of bad smells using historical information," in *Proc. 9th Int. Workshop Principles Softw. Evol.: Conjunction 6th ESEC/FSE Joint Meeting*, 2007, pp. 31–34.

[30] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *Proc. 20th Int. Conf. Softw. Maintenance*, Chicago, IL, USA, Sep. 11–17, 2004, pp. 350–359.

[31] J. Mayrand, C. Leblanc, and E. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics," in *Proc. Int. Conf. Softw. Maintenance*, Monterey, CA, USA, Nov. 4–8, 1996, pp. 244–253.

[32] T. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976.

[33] 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 Trans. Softw. Eng.*, vol. 36, no. 1, pp. 20–36, Jan./Feb. 2010.

[34] F. Mulder and A. Zaidman, "Identifying cross-cutting concerns using Software repository mining," in *Proc. 10th Int. Workshop Principles Softw. Evol.*, 2010, pp. 23–32.

[35] M. J. Munro, "Product metrics for automatic identification of 'bad smell' design problems in Java source-code," in *Proc. 11th Int. Softw. Metrics Symp.*, Sep. 2005, p. 15.

[36] R. Oliveto, F. Khomh, G. Antoniol, and Y.-G. Guéhéneuc, "Numerical signatures of antipatterns: An approach based on b-splines," in *Proc. 14th Conf. Softw. Maintenance Reeng.*, Mar. 2010, pp. 248–251.

[37] A. N. Oppenheim, *Questionnaire Design, Interviewing and Attitude Measurement*. London, U.K.: Pinter, 1992.

[38] A. Ouni, M. Kessentini, H. Sahraoui, and M. S. Hamdi, "The use of development history in software refactoring using a multi-objective evolutionary algorithm," in *Proc. 15th Annu. Conf. Genet. Evol. Comput.*, 2013, pp. 1461–1468.

[39] 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 *Proc. IEEE/ACM 28th Int. Conf. Autom. Softw. Eng.*, Nov. 2013, pp. 268–278.

[40] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "HIST: Replication," package http://dx.doi.org/10.6084/m9.figshare.1157374, 2014.

[41] R. Peters and A. Zaidman, "Evaluating the lifespan of code smells using Software repository mining," in *Proc. Eur. Conf. Softw. Maintenance ReEng.*, 2012, pp. 411–416.

[42] A. Rao and K. Raddy, "Detecting bad smells in object oriented design using design change propagation probability matrix," in *Proc. Int. MultiConf. Eng. Comput. Sci.*, 2008, pp. 1001–1007.

[43] A. Rao and D. Ram, "Software design versioning using propagation probability matrix," presented at the 3rd Int. Conf. Comput. Appl., Yangon, Myanmar, 2005.

[44] D. Ratiu, S. Ducasse, T. Gîrba, and R. Marinescu, "Using history information to improve design flaws detection," in *Proc. 8th Eur. Conf. Softw. Maintenance Reeng.*, Tampere, Finland, Mar. 24–26, 2004, pp. 223–232.

[45] A. J. Riel, *Object-Oriented Design Heuristics*. Reading, MA, USA: Addison-Wesley, 1996.

[46] F. Simon, F. Steinbr, and C. Lewerentz, "Metrics based refactoring," in *Proc. 5th Eur. Conf. Softw. Maintenance Reeng.*, 2001, pp. 30–38.

[47] G. Travassos, F. Shull, M. Fredericks, and V. R. Basili, "Detecting defects in object-oriented designs: Using reading techniques to increase software quality," in *Proc. 14th Conf. Object-Oriented Programming, Syst., Lang., Appl.*, 1999, pp. 47–56.

[48] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Trans. Softw. Eng.*, vol. 35, no. 3, pp. 347–367, May/Jun. 2009.

[49] E. van Emden and L. Moonen, "Java quality assurance by detecting code smells," in *Proc. 9th Working Conf. Reverse Eng.*, Oct. 2002.

[50] B. F. Webster, *Pitfalls of Object Oriented Development*, 1st ed. Baltimore, MD, USA: M & T Books, Feb. 1995.

[51] A. Yamashita and L. Moonen, "Do code smells reflect important maintainability aspects?" in *Proc. Int. Conf. Softw. Maintenance*, 2012, pp. 306–315.

[52] A. Yamashita and L. Moonen, "Exploring the impact of inter-smell relations on Software maintainability: An empirical study," in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 682–691.

[53] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll, "Predicting source code changes by mining change history," *IEEE Trans. Softw. Eng.*, vol. 30, no. 9, pp. 574–586, Sep. 2004.

[54] T. Zimmermann, A. Zeller, P. Weißgerber, and S. Diehl, "Mining version histories to guide software changes," *IEEE Trans. Softw. Eng.*, vol. 31, no. 6, pp. 429–445, Jun. 2005.

[55] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," in *Proc. 26th Int. Conf. Softw. Eng.*, 2004, pp. 563–572.

Fabio Palomba received the bachelor's degree in computer science from the University of Molise, Italy, master's degree in computer science from the University of Salerno, Italy. He is currently working toward the PhD degree at the Department of Management & Information Technology, University of Salerno, under the supervision of Prof. Andrea De Lucia and Prof. Rocco Oliveto. His research interests include mining software repositories, refactoring and remodularization, software maintenance and evolution, and empirical software engineering. He is a student member of the IEEE.

Gabriele Bavota received the PhD degree in computer science from the University of Salerno, Italy, in 2013. He is an assistant professor at the Free University of Bozen-Bolzano, Italy. From January 2013 to October 2014, he has been a research fellow at the University of Sannio, Italy. His research interests include software maintenance, empirical software engineering, mining software repository, refactoring of software systems, and information retrieval. He is an author of more than 50 papers appeared in international journals, conferences, and workshops. He serves and has served as an organizing and program committee member of international conferences in the field of software engineering, such as ICSME, MSR, ICPC, SANER, SCAM, and others. He is a member of the IEEE Computer Society.

Massimiliano Di Penta is an associate professor at the University of Sannio, Italy. His research interests include software maintenance and evolution, mining software repositories, empirical software engineering, search-based software engineering, and service-centric software engineering. He is an author of more than 190 papers appeared in international journals, conferences, and workshops. He serves and has served in the organizing and program committees of more than 100 conferences such as ICSE, FSE, ASE, ICSM, ICPC, GECCO, MSR WCRE, and others. He has been a general co-chair of various events, including the 10th IEEE Working Conference on Source Code Analysis and Manipulation (SCAM 2010), the second International Symposium on Search-Based Software Engineering (SSBSE 2010), and the 15th Working Conference on Reverse Engineering (WCRE 2008). Also, he has been a program chair of events such as the 28th IEEE International Conference on Software Maintenance (ICSM 2012), the 21st IEEE International Conference on Program Comprehension (ICPC 2013), the ninth and 10th Working Conference on Mining Software Repository (MSR 2013 and 2012), the 13th and 14th Working Conference on Reverse Engineering (WCRE 2006 and 2007), the first International Symposium on Search-Based Software Engineering (SSBSE 2009), and other workshops. He is currently a member of the steering committee of ICSME, MSR, SSBSE, and PROMISE. Previously, he has been a steering committee member of other conferences, including ICPC, SCAM, and WCRE. He is in the editorial board of the *IEEE Transactions on Software Engineering*, the *Empirical Software Engineering Journal* edited by Springer, and of the *Journal of Software: Evolution and Processes* edited by Wiley.

Rocco Oliveto received the PhD degree in computer science from the University of Salerno, Italy, in 2008. He is an assistant professor in the Department of Bioscience and Territory at University of Molise, Italy. He is the chair at the Computer Science program and the director at the Laboratory of Computer Science and Scientific Computation, University of Molise. His research interests include traceability management, information retrieval, software maintenance and evolution, search-based software engineering, and empirical software engineering. He is an author of about 100 papers appeared in international journals, conferences, and workshops. He serves and has served as an organizing and program committee member of international conferences in the field of software engineering. He is a member of the IEEE Computer Society and ACM.

Denys Poshyvanyk received the MS and MA degrees in computer science from the National University of Kyiv-Mohyla Academy, Ukraine, and Wayne State University in 2003 and 2006, respectively. He received the PhD degree in computer science from Wayne State University in 2008. He is an associate professor at the College of William and Mary in Virginia. He serves as a program co-chair for ICSME'16. He also served as a program co-chair for ICPC'13, WCRE'12 and WCRE'11. His research interests include software engineering, software maintenance and evolution, program comprehension, reverse engineering, software repository mining, source code analysis and metrics. He is a member of the IEEE and ACM.

Andrea De Lucia received the Laurea degree in computer science from the University of Salerno, Italy, in 1991, the MSc degree in computer science from the University of Durham, United Kingdom, in 1996, and the PhD degree in electronic engineering and computer science from the University of Naples "Federico II," Italy, in 1996. He is a full professor of software engineering at the Department of Management & Information Technology, University of Salerno, Italy, the head at the International Summer School on Software Engineering, and the coordinator of the PhD program in Management & Information Technology. His research interests include software maintenance and testing, reverse engineering and reengineering, empirical software engineering, search-based software engineering, collaborative development, workflow and document management, and elearning. He has published more than 200 papers on these topics in international journals, books, and conference proceedings and has edited books and journal special issues. He also serves on the editorial boards of international journals and on the organizing and program committees of several international conferences. He is a senior member of the IEEE and the IEEE Computer Society. He was also at-large member of the executive committee of the IEEE Technical Council on Software Engineering (TCSE).

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.