Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets, © JOT 2011
Online at http://www.jot.fm.

Automatic detection of bad smells in code: An experimental assessment

Francesca Arcelli Fontana[a] Pietro Braione[a] Marco Zanoni[a]

a. DISCo, University of Milano-Bicocca, Italy

Abstract Code smells are structural characteristics of software that may indicate a code or design problem that makes software hard to evolve and maintain, and may trigger refactoring of code. Recent research is active in defining automatic detection tools to help humans in finding smells when code size becomes unmanageable for manual review. Since the definitions of code smells are informal and subjective, assessing how effective code smell detection tools are is both important and hard to achieve. This paper reviews the current panorama of the tools for automatic code smell detection. It defines research questions about the consistency of their responses, their ability to expose the regions of code most affected by structural decay, and the relevance of their responses with respect to future software evolution. It gives answers to them by analyzing the output of four representative code smell detectors applied to six different versions of GanttProject, an open source system written in Java. The results of these experiments cast light on what current code smell detection tools are able to do and what the relevant areas for further improvement are.

Keywords code smells; code smell detection tools; refactoring; software quality evaluation.

1 Introduction

Nowadays there is an increasing number of software analysis tools available for detecting bad programming practices, highlighting anomalies, and in general increasing the awareness of the software engineer about the structural features of the program under development. As these tools are gaining acceptance in practice, a question arises on how to assess their effectiveness and select the "best" one. For this aspect we have to face the common and very difficult problem of tool comparison and validation of the results.

In this paper we focus our attention on *code smells* and on automatic tools developed for their detection. Code smells are structural characteristics of software that may indicate a code or design problem and can make software hard to evolve and maintain. The concept was introduced by Fowler [Fow99], who defined 22 different

Francesca Arcelli Fontana, Pietro Braione, Marco Zanoni. *Automatic detection of bad smells in code: An experimental assessment*. In Journal of Object Technology, vol. 11, no. 2, 2012, pages 5:1–38.
doi:10.5381/jot.2012.11.2.a5

kinds of smells. Later, other authors (e.g. Mäntylä [MVL04]) identified more smells, and new ones can be discovered. Code smells are strictly related to the practice of *refactoring* software to enhance its internal quality. As developers detect bad smells in code, they should evaluate whether their presence hints at some relevant degradation in the structure of the code, and if positive, decide which refactoring should be applied. Using a metaphor, smells are like the symptoms of possible diseases, and refactoring operations may heal the associated diseases and remove their symptoms. Since the same symptom may be caused by different diseases, or even by no disease at all, human judgement is indispensable for assessing smells in the context of the project where they are found. Automatic tools, on the other hand, can play a relevant role in alleviating the task of *finding* code smells in large code bases.

Not necessarily all the code smells have to be removed: it depends on the system. When they have to be removed, it is better to remove them as early as possible. If we want to remove smells in the code, we have to locate and detect them; tool support for their detection is particularly useful, since many code smells can go unnoticed while programmers are working. However, since the concept of code smell is vague and prone to subjective interpretation, assessing the effectiveness of code smell detection tools is especially challenging. Different tools may provide different results when they analyze the same system for many reasons:

- The ambiguity of the smell definitions and hence the possibly different interpretations given by the tools implementors. There are some words and constraints which are routinely used to define the essence of many smells, such as the words "few", "many", "enough", "large", "intimate", which are obviously ambiguous. In other cases the definitions are too vague and have to be refined and improved to enhance the detection tools.

- The different detection techniques used by the tools. They are usually based on the computation of a particular set of combined metrics, or standard object-oriented metrics, or metrics defined *ad hoc* for the smell detection purpose [LM06].

- The different threshold values used for the metrics, even when the techniques are analogous or identical. To establish these values different factors have to be taken into account, such as the system domain and size, organization best practices, the expertise and the understanding of the software engineers and the programmers who define these values. Changing thresholds obviously has a great impact on the number of detected smells, i.e. too many or too few.

In general, the validation of the results of the existing approaches is scarce, done only on small systems and for few smells [MGDM10, MHB10, MVL04, Mä05]. The validation of the detection results is complex, not only for the reasons cited above, but for the problems related to the manual validation. The manual validation of the results of a tool might not be completely correct, even if the manual validators are given the same criteria used by the tool (e.g. smell definition, detection rule). In fact, the values of some metrics, like those used for example to measure method complexity, can change from a manual inspection done by a programmer to the automatic computation of this metric done by a tool. Moreover, the manual computation can differ from one programmer to another: it is subjective. However, the results of manual and automatic detection of smells detected through simple metrics, like counting entities, tend to have greater accordance.

This paper is focused on tools rather than on humans and extends previous work on the experimentation of code smell detection tools [AFMM+11]. It reviews the current panorama of the tools for automatic code smell detection, and comparatively analyzes some of them against their usefulness in assisting the main stakeholders of the software development process when assessing the internal quality of a software system. We do that by defining a number of research questions about the consistency of tool responses, the ability of tools to expose the regions of code most affected by structural decay, and the relevance of tool responses with respect to future software evolution. We answer these research questions by an experimental analysis of six different versions of GanttProject,[1] an open source system written in Java, by means of four representative code smell detectors. We are not aware of the existence of previous research in literature with a similar aim and scope. Hence, the results of these experiments cast light on what current code smell detection tools are able to do and what the relevant areas for further improvement are. We hope that this work will be valuable not only to researchers, but also to developers who exploit tools to detect critical region of code, assess the internal quality of software and focus on refactoring activities. Reliable tools must yield precise, synthetic and reproducible answers, and prioritize them on their relevance. The experiments discussed in this paper provide practitioners with a view on how well current tools for code smell detection satisfy these requirements.

The paper is organized as follows. Section 2 introduces the smells we focus our attention on in this paper, and discusses how software metrics and code smells can be correlated with the purpose of their automatic detection. Section 3 lists all the tools able to detect some smell we are aware of. Section 4 contains four research questions on code smell detection tools, and the experimental setup we designed to answer them. Section 5 exposes the results of our experiments, answers the research question of Section 4, and discusses the main threats to the validity of these results. Section 6 reviews the current literature. Finally, Section 7 summarizes the research presented in this paper and outlines our current efforts in extending them.

2 Bad smells: Definitions and their detection

Fowler identifies 22 code smells [Fow99] and associates each of them with the refactoring transformations that may be applied to improve the structure of code. He defines code smells informally, maintaining that only human intuition may guide a decision on whether some refactoring is overdue. Some smells indicate real problems in code (e.g., long parameter lists make methods harder to invoke), while other are just possible symptoms of a problem: e.g. a method has Feature Envy when it uses the features of a class different from that where it is declared, which may indicate that the method is misplaced, or that some pattern as Visitor is being applied.

As Zhang et al. [ZHBW08] suggest, Fowler's definitions are too informal to implement in a smell detection tool. The ambiguous definitions of smells have a great impact on their detection. We provide here the definitions [Fow99] of some code smells on which we focus our attention in this paper.

[1] `http://www.ganttproject.biz`

2.1 Smells Definition

Duplicated Code means that the same code structure appears in more than one place.

Feature Envy means that a method is more interested in other class(es) than the one where it is currently located. This method is in the wrong place since it is more tightly coupled to the other class than to the one where it is currently located.

God Class usually called also design flaw, refers to class that tends to centralize the intelligence of the system. A God Class performs too much work on its own, delegating only minor details to a set of trivial classes, and using the data from other classes [LM06]. This smell is comparable to Fowler's Large Class smell [Fow99] and is similar also to Brain Class smell [LM06] (see Appendix B for the definitions of these smells).

Large Class means that a class is trying to do too much. These classes have too many instance variables or methods.

Long Method is a method that is too long, so it is difficult to understand, change, or extend. The Long Method smell is similar to the Brain Method smell defined by Lanza et al. [LM06], which tend to centralize the functionality of a class, in the same way as a God Class centralizes the functionality of an entire subsystem, or sometimes even a whole system.

Long Parameter List is a parameter list that is too long and thus difficult to understand.

2.2 Smells Detection

As we will see in Section 5 the results for the detection of these smells provided by the tools are very different. This subsection outlines how the smells can be detected and discusses some of the relevant problems which we have to face for their automatic detection. Some of the considerations we report here have been described by Mäntylä [Mä05]. We outline below how some tools detect the above smells and we remind to the next section for the descriptions and the references to the tools.

2.2.1 Duplicated Code

Discovering duplicates can be done by measuring the percentage of duplicated code lines in the system. The problem in the measurement of this smell lies in the different possible kinds of duplication. Exact duplication is simple to detect, using diff-like techniques. Other kinds of duplication imply entity renaming or aliasing, so the detection technique needs to be able to manage the syntax of the programming language, and needs more computational time to manage all the possible combinations of candidate renaming. Another issue is that duplicated code is often slightly modified and mixed with different code.

The detection strategy of iPlasma and inFusion is to detect duplicated code through a set of metrics about exact code duplication, but considering also the length of the duplication and the distance between two duplications. Checkstyle detects this smell simply counting 12 line of consecutive duplicated code in the text of the program; the

piece of duplicated code can also span multiple methods or classes. PMD considers as an occurrence of the smell a portion of code that is duplicated at least once and that is at least composed by 25 tokens.

2.2.2 Feature Envy

The detection of Feature Envy can be achieved by measuring the strength of coupling that a method has to methods (or data) belonging to foreign classes. There are no other typical measures exploited for the detection of this smell.

Lanza and Marinescu [LM06] propose the following metrics for the detection of Feature Envy smells:

- methods using directly more than a few attributes of other classes, measured by the ATFD (Access To Foreign Data) metric;
- methods using far more attributes from other classes than their own, measured by the LAA (Locality of Attribute Accesses) metric;
- the used "foreign" attributes that belong to very few other classes, measured by the FDP (Foreign Data Providers) metric.

They detect this smell through the following condition:

$$\text{FDP} \leq \text{FEW} \ \wedge \ \text{ATFD} > \text{FEW} \ \wedge \ \text{LAA} < \frac{1}{3},$$

where for FEW the authors consider the value of 5. Both iPlasma and inFusion adopt the above rule.

JDeodorant approaches the problem by looking for Move Method refactoring opportunities [TC09]: It tries to find methods which will use less foreign resources if moved in another class.

2.2.3 God Class

Lanza and Marinescu [LM06] propose to detect the God Class smell through the computation of the following three metrics:

- Weighted Method Count (WMC): the sum of the statistical complexity of all methods in a class.
- Tight Class Cohesion (TCC): the relative number of methods directly connected via accesses of attributes.
- Access to Foreign Data (ATFD): the number of external classes from which a given class accesses attributes, directly or via accessor methods. This last metric is a new one, defined by the authors for the computation of this smell.

In their approach, a class is a God Class when:

$$\text{WMC} \geq \text{VERY\_HIGH} \ \wedge \ \text{ATFD} > \text{FEW} \ \wedge \ \text{TCC} < \frac{1}{3},$$

where for VERY_HIGH the authors consider the value of 47, and for FEW the value of 5. How these values have been fixed is described in their book. Both iPlasma and inFusion adopt the above rule. JDeodorant [FTCS09] computes God Class as a class that can be decomposed in other classes, and it reports one (or more) Extract Class refactoring opportunity. This kind of detection strategy is not directly related to the size of the class.

2.2.4 Large Class

Many class size measures have been introduced in the past. The traditional way of measuring class size is to measure the number of lines of code, i.e. NLOC, or the number of attributes and methods. Simple size measures such as NLOC can reveal the size of the class, but they offer no help on whether the class is doing too much or not. For example GUI classes, as well as other special cases, can make the definition of Large Class more elusive: e.g. a Large Class which represents a parser cannot be considered a smell, because it is typically generated code.

PMD and Checkstyle both use NLOC as detection strategy. The former uses a threshold of 1000 and the second a threshold of 2000. Another measure of class size is class cohesion, and different metrics can be used.

2.2.5 Long Method

Measuring long methods should be quite easy. However, relying on a too simple size measure such as NLOC will definitely bring a wrong result, because, for example, initiation methods can often be quite long. There is no sense in refactoring long initiation methods, because they usually have very low cyclomatic complexity and therefore they are very easy to understand and modify. Mäntylä [Mä05] suggests using cyclomatic complexity and Halstead measures, which measure the number of operators and operands, and can provide necessary information on method complexity. In the opinion of Mäntylä, the best metric for this smell is a polynomial metric that combines NLOC, Cyclomatic Complexity, and Halstead metrics.

Tools like Checkstyle and PMD detect Long Method by simply considering NLOC, but using different thresholds: for PMD the threshold is 100 and for Checkstyle is 150. JDeodorant uses slicing techniques to determine if a class is eligible for an Extract Method refactoring [TC11].

2.2.6 Long Parameter List

The detection of a Long Parameter List can be achieved simply by counting the number of parameters of each method. The consideration of the type of parameters (i.e. primitives, classes) in combination with this smell can additionally help the refactoring process, since redundant primitive lists make good candidates for new classes. The critical point in the detection of a Long Parameter List is the setting of the threshold value. For example, in PMD the default value of the threshold is 10, while in Checkstyle it is 7.

3 Tools for the automatic detection of code smells

This section briefly surveys the code smell detection tools we are aware of. Table 1 synthesizes some basic facts for the tools, while Table 2 reports the smells detected by the tools. Appendix B provides the definitions of the smells in Table 2 not previously defined.

Some tools, as we will see, have been developed to improve code quality during software development, other tools to support reengineering and maintenance activities. Most of the tools are not able to perform refactoring automatically on detected smells (JDeodorant is the only one which provides refactoring choices), but modern IDEs are usually capable of performing refactoring automatically, also if they need user guidance. It would be desirable for smell detector tools to help the user to understand at least

Table 1 – Code smell detection tools

Tool	Version	Type	Analyzed languages	Refactoring	Link to code
Checkstyle	5.4.1 2011	Eclipse Plugin, Standalone	Java	No	Yes
DECOR	1.0 2009	Standalone	Java	No	No
iPlasma	6.1 2009	Standalone	C++, Java	No	No
inFusion	7.2.11 2010	Standalone	C, C++, Java	No	No
JDeodorant	4.0.4 2010	Eclipse Plugin	Java	Yes	Yes
PMD	4.2.5 2009	Eclipse Plugin, Standalone	Java	No	Yes
Stench Blossom	1.0.4 2009	Eclipse Plugin	Java	No	Yes

Legend:
Analyzed languages: languages that the tool is able to analyze;
Refactoring: whether the tool provides automatic refactoring or not;
Link to code: whether the tool provides the location in the code of the detected smells.

the cause of the smells and, as underlined in the guidelines proposed by Murphy-Hill and Black [MHB10], the tools should also not display smell information in a way that overloads the programmer, in case of smell proliferation.

Checkstyle Checkstyle[2] has been developed to help programmers to write Java code that adheres to a coding standard. It is able to detect the Large Class, Long Method, Long Parameter List, and Duplicated Code code smells.

DECOR Moha et al. [MGDM10, MGM+10] defined an approach that allows the specification and automatic detection of code and design smells (also called anti patterns). They specified six code smells by using a custom language, automatically generated their detection algorithms using templates, and validated the algorithms in terms of precision and recall. This approach is implemented in their DECOR platform for software analysis.[3] In the following, with the name DECOR we mean the component developed for code smell detection.

inFusion inFusion[4] is the current, commercial evolution of iPlasma. inFusion is able to detect more than 20 design flaws and code smells, like Duplicated Code, classes that break encapsulation, i.e. Data Class and God Class, methods and classes that are heavily coupled, or ill-designed class hierarchies.

[2] `http://checkstyle.sourceforge.net/`
[3] `http://www.ptidej.net/download`
[4] `http://www.intooitus.com/inFusion.html`

iPlasma This tool [MMM+05] is an integrated platform for quality assessment of object-oriented systems that includes support for all the necessary phases of analysis, from model extraction, up to high-level metrics based analysis.[5] iPlasma is able to detect what the authors define as code disharmonies, classified into identity disharmonies, collaboration disharmonies, and classification disharmonies. The detailed description of these disharmonies can be found in [LM06]. Several code smells are considered as disharmonies, e.g., Duplicated Code (named Significant Duplication), God Class, Feature Envy, and Refused Parent Bequest.

JDeodorant JDeodorant [TC11] is an Eclipse plugin that automatically identifies the Feature Envy, God Class, Long Method and Switch Statement (in its Type Checking variant) code smells in Java programs.[6] The tool assists the user in determining an appropriate sequence of refactoring applications by determining the possible refactoring transformations that solve the identified problems, ranking them according to their impact on the design, presenting them to the developer, and automatically applying the one selected by the developer.

PMD PMD[7] scans Java source code and looks for potential problems or possible bugs like dead code, empty try/catch/finally/switch statements, unused local variables or parameters, and duplicated code. PMD is able to detect Large Class, Long Method, Long Parameter List, and Duplicated Code smells, and allows the user to set the thresholds values for the exploited metrics.

Stench Blossom Stench Blossom [MHB10] is a smell detector that provides an interactive visualization environment designed to give programmers a quick and high-level overview of the smells in their code, and of their origin. The tool is a plugin for the Eclipse environment that provides the programmer with three different views, which progressively offer more information about the smells in the code being visualized. The feedback is synthetic and visual, and has the shape of a set of petals close to a code element in the IDE editor. The size of a petal is directly proportional to the "strength" of the smell of the code element it refers. The only possible procedure to find code smells is to manually browse the source code, looking for a petal whose size is big enough to make the user suppose that there is a code smell. The tool is able to detect 8 smells.

Other tools Other developed tools or methods for code smell detection have been proposed. CodeVizard [ZA10] is a currently unreleased tool able to detect several smells, essentially following the detection rules defined in [LM06]. inCode[8] is an Eclipse plugin based on inFusion, that provides detection of design problems as code is written. Given its similarity with inFusion we have not considered it. Other tools are: FxCop for .NET, Analyst for Java (commercial), CodeNose (no longer available), JCosmo [vEM02] (for Linux), CloneDigger and ConQat (for clone detection).

[5] `http://loose.upt.ro/iplasma/index.html`
[6] `http://http://www.jdeodorant.com/`
[7] `http://pmd.sourceforge.net/`
[8] `http://pmd.sourceforge.net/`

Table 2 – Code smell support

Smell	Checkstyle	DECOR	inFusion	iPlasma	JDeodorant	PMD	Stench Blossom
Brain Class			×	×			
Brain Method			×	×			
Data Class		×	×	×			
Data Clumps			×				×
Dead Code						×	
Duplicated Code	×		×	×		×	
Extensive Coupling			×	×			
Feature Envy			×	×	×		×
God Class / Large Class (DECOR)		×	×	×	×		
Instanceof							×
Intensive Coupling			×	×			
Large Class	×	×				×	×
Long Method	×	×			×	×	×
Long Parameter List	×	×				×	
Message Chains		×					×
Refused Parent Bequest		×	×	×			
Shotgun Surgery			×	×			
Speculative Generality		×					
Switch Statements / Type Checking (JDeodorant)					×		×
Tradition Breaker		×	×	×			
Typecast							×
Total	4	9	12	11	4	5	8

Table 3 – GanttProject size

Version	Classes	Methods	Lines of code
1.4	20	290	3844
1.6	21	366	4542
1.9.1	47	678	8535
1.9.6	62	1302	14126
1.9.10	114	878	20108
1.10.2	295	1869	31794

4 Experimenting with code smell detection tools

The aim of our study is to experiment with different tools for code smells detection by applying them to different versions of different subject programs. In our experimentation, we decided to use among the tools of Section 3 the following: JDeodorant, inFusion, PMD and Checkstyle. This choice is due to the fact that these four tools are available for download, actively developed and maintained, their results are easily accessible, and we are able to know the detection techniques they use. Moreover, inFusion is the evolution of iPlasma, while DECOR and Stench Blossom do not provide accessible and easy way to export their results. All the other tools reported in Section 3 are either no longer available, or no longer developed, or unable to analyze Java programs.

We applied these tools to the analysis of the GanttProject, JEdit, JFreeChart, JRefactory, and Lucene open source Java software systems. In this paper we report the detailed results for GanttProject, a software for planning and project management assistance. Appendix A reports the results of the experiments on the other systems in a more synthetic fashion. To mitigate the possible effects of code maturity on the number and distribution of smells in code, we applied the tools on six different versions of GanttProject, spanning a period of time from February 2003 to November 2004 (Table 3 reports the analyzed GanttProject versions and their respective size). The v1.4 version is the earliest version whose source code is still available for download. We assume that this version and its immediate successors have a low degree of maturity, thus a high smell density. The experiments consider the six smells shared by at least two of the selected tools: Duplicated Code, Feature Envy, God Class, Large Class, Long Method, Long Parameter List.[9]

In the following we introduce three research questions about current code smell detection tools, and explain the structure of the experiments designed to answer each of them.

Question 1. *Do different detection tools for the same code smell agree when applied to the same subject program?*

Our first experiment investigates whether different tools for code smell detection, based on different algorithms, agree on their results or not. We considered all the smells analyzed by at least two different tools: Duplicated Code (analyzed by Checkstyle and JDeodorant), Feature Envy and God Class (analyzed by JDeodorant and inFusion),

[9] A rough estimate of the effort spent in gathering the experimental results is, on average, one hour of work for analyzing a single code smell with all the tools on 10000 lines of code. Effort is strongly dependent on the number of found smell instances and on the type of smell/tool combination.

Large Class and Long Parameter List (analyzed by Checkstyle and PMD), Long Method (analyzed by Checkstyle, PMD and JDeodorant). We calculated the kappa statistics [Coh60, Fle71] of the tools, which is basically an attempt to balance the amount of accordance between the tools and the amount of accordance due to randomness (tools returned the same results, but not for the same reason).

Question 2. *How relevant are the automatically detected smells to future software development?*

This research question arises from a remark by Fowler [Fow99]: Not every smell may lead to a corresponding refactoring. Refactoring becomes necessary whenever the presence of a bad structure in code starts to be an obstacle to further software evolution and maintenance. Human judgement is necessary to evaluate the relevance of a code smell in the context of the actual project needs.

Our experiment aims at ranking the tools against an approximated evaluation of the future relevance of the smell they detect, based on the assumption that a smell is relevant if it is removed from code in the immediate future after it is introduced. To this end, we determined the version where a code smell instance, found by a given tool in a given version of GanttProject, is removed from the code. The exact procedure was the following. We ran a smell detector on all the six versions of GanttProject we considered for Question 1. Then, we manually tracked the smell through the evolution of the code across the version, abstracting away refactoring transformation as, for example, renaming of classes and methods or their relocation in different packages and classes. This allowed us to assess whether two smell instances found by a tool in two different versions of GanttProject should be considered the "same" smell instance. Finally, we calculated the *persistence* of each smell instance, i.e., the total number of versions where the smell instance is present before its removal. As an example, a smell instance introduced in version 1.9.1 and removed in version 1.9.6 would have persistence 1, since we did not analyze any version between versions 1.9.1 and 1.9.6. The smells that are not removed by version 1.10.2 are classified as either not removed or unassessable, if they have been introduced, respectively, before or at version 1.10.2. Persistence is our proxy measure for the relevance of a smell instance. Our analysis is limited to the Duplicated Code and Feature Envy smells because either the higher granularity (class smells) or the low number of positive occurrences found by the tools do not allow us to draw significant conclusions for the other smells.

Question 3. *Is the presence of smells related to some observable feature of the source code or of the process?*

The change in the number of smells found usually reflects some significant change in the source code that hinders its degradation. We therefore ask ourselves whether, assuming that the tools may be imprecise, or may have a poor recall, they still can be used by managers to observe, on a broader scale, the evolution of software and assess the general trend of its internal quality. Our experiment will be based on the information on the density ratio of the smells reported by the tools for each version of the project, and on the overall history of the project as deduced from a manual differential analysis of the source code across versions. We will attempt to informally correlate changes, by manual review, in smell density across versions, and the prevalent position of smells in the code, with some basic facts on project development that can be deduced from source code analysis (introduction of new functionalities, refactoring, etc.).

Table 4 – Tools' overall agreement

Version	Duplicated Code	Feature Envy	God Class	Large Class	Long Method	Long Parameter List
1.4	98.28%	95.52%	90.00%	100.00%	90.00%	100.00%
1.6	98.91%	96.72%	85.71%	100.00%	92.35%	100.00%
1.9.1	99.12%	97.05%	89.13%	95.74%	92.77%	99.85%
1.9.6	93.09%	98.08%	88.71%	95.16%	94.09%	99.92%
1.9.10	94.31%	94.87%	86.79%	97.37%	84.28%	99.66%
1.10.2	96.84%	96.74%	86.78%	98.98%	89.83%	99.84%

5 Results

In this section we report the results of the experiments we did to answer the above questions.

5.1 Tools analysis

We recall the research question which motivated the experimental analysis based on tools:

Question 1. *Do different detection tools for a same code smell agree when applied to a same subject program?*

Table 4 shows the proportion (in percentage) of the overall agreement for all the smells on every analyzed GanttProject version. The values range from 85% to 100%, giving an a idea of strong agreement among the tested tools. These values indeed hide the fact that the data are unbalanced. The large majority of the agreement among the tools is on *false* values: Most of the classes (or methods) are classified as not containing any smell. If we trust the results of the tools, we may consider the occurrence of a smell in code as a relatively rare event, a result confirmed by manual analysis.

For each smell and for each version of GanttProject we also report in Table 5 the kappa statistic. We used Cohen's kappa [Coh60] for the smells that are recognized by two tools, and Fleiss' kappa [Fle71] for those recognized by more than two tools. Zero or negative values indicate no accordance, values close to 1 indicate good accordance. The value is accompanied by the 95% confidence interval, which can give an idea of how much the value of kappa is reliable. Another way of evaluating the confidence of the statistic is to know the number of comparisons made. We tested every smell by using the largest sample available: the number of classes for the God Class and Large Class smells, and the number of methods for all the other smells. To create the statistics dataset we assigned every smell instance, for each tool, to its class or method.

The table reports no value in those cases where kappa statistics could not be calculated from the resulting data. This happens, as an example, when no tool detects any instance of a smell, as happened for the Large Class code smell in version 1.4.

Overall, the general accordance among the tools is very low, with six cases where kappa is 0 and four cases where kappa cannot be calculated, where the tools did find

Table 5 – Tools' kappa statistic

Version	Duplicated Code	95% Conf. Interval	Feature Envy	95% Conf. Interval
1.4	0.4384	[0.3432, 0.5337]	0.0000	[0.0000, 0.0000]
1.6	0.4959	[0.4074, 0.5843]	0.0000	[0.0000, 0.0000]
1.9.1	0.3972	[0.3371, 0.4572]	0.0883	[0.0574, 0.1193]
1.9.6	0.3088	[0.2696, 0.3481]	-0.0015	[-0.0226, 0.0196]
1.9.10	0.0303	[-0.0042, 0.0648]	-0.0022	[-0.0213, 0.0168]
1.10.2	0.0000	[0.0000, 0.0000]	0.1350	[0.1083, 0.1617]
Version	God Class	95% Conf. Interval	Large Class	95% Conf. Interval
1.4	0.6154	[0.2108, 1.0000]	-	-
1.6	0.3505	[0.0253, 0.6757]	-	-
1.9.1	0.4051	[0.1753, 0.6349]	0.0000	[0.0000, 0.0000]
1.9.6	0.4182	[0.2158, 0.6207]	0.0000	[0.0000, 0.0000]
1.9.10	0.4493	[0.2961, 0.6025]	0.5604	[0.3955, 0.7253]
1.10.2	0.2325	[0.1594, 0.3057]	0.5672	[0.4644, 0.6701]
Version	Long Method	95% Conf. Interval	Long Parameter List	95% Conf. Interval
1.4	-0.0012	[-0.0676, 0.0653]	-	-
1.6	0.0705	[0.0114, 0.1297]	-	-
1.9.1	0.1002	[0.0568, 0.1437]	0.0000	[0.0000, 0.0000]
1.9.6	0.1337	[0.1023, 0.1650]	0.6663	[0.6151, 0.7175]
1.9.10	0.1322	[0.0940, 0.1704]	0.5700	[0.5103, 0.6297]
1.10.2	0.1277	[0.1015, 0.1539]	0.5708	[0.5298, 0.6117]

no instances at all. Whenever the agreement between some tools is higher, it is because they use similar detection algorithms, as it will be discussed later in this subsection. In these cases, with the exception of the Long Parameter List, the confidence limit tends to be wide, lowering the possibility of concluding that there is good accordance among the tools.

Tables 6–11 contain a more detailed view of the results. We will comment on them by considering each smell separately.

Duplicated code Table 6 reports the total number of smell instances found by the Checkstyle and inFusion tools against the size of the analyzed system expressed as the total number of methods. The number of smell instances is the total number of methods in the system which have at least one row of duplicated code. If a same block of code is duplicated in n different methods, all n methods are counted. The resulting ratio (last two columns) range from 0%, when no duplicates exist in the system, to 100%, when all the methods in the systems contain duplicated code. We notice that Checkstyle consistently reports a higher number of smells than inFusion. Table 7 compares the detection results by each tool. The first four columns detail how many methods are classified as affected by the smell respectively by no tool (first column), by inFusion but not by Checkstyle (second column), by Checkstyle but not by inFusion (third column) and by both tools (fourth column). As we can see, Checkstyle classifies

Table 6 – Duplicated code per-tool results

Version	Checkstyle	inFusion	Size	Checkstyle ratio	inFusion ratio
1.4	7	2	290	2.41%	0.69%
1.6	6	2	366	1.64%	0.55%
1.9.1	8	2	678	1.18%	0.29%
1.9.6	112	22	1302	8.60%	1.69%
1.9.10	48	4	878	5.47%	0.46%
1.10.2	59	0	1869	3.16%	0.00%

Table 7 – Duplicated code tool comparison

Version	None	Checkstyle	inFusion	Both	Same		Different		Size
1.4	283	5	0	2	285	98.28%	5	1.72%	290
1.6	360	4	0	2	362	98.91%	4	1.09%	366
1.9.1	670	6	0	2	672	99.12%	6	0.88%	678
1.9.6	1190	90	0	22	1212	93.09%	90	6.91%	1302
1.9.10	827	47	3	1	828	94.31%	50	5.69%	878
1.10.2	1810	59	0	0	1810	96.84%	59	3.16%	1869

as affected by the smell almost all the methods that are also classified positively by inFusion. This suggests that Checkstyle uses a weaker detection strategy than that of inFusion. Indeed, the detection technique used in Checkstyle is exact text matching of at least 12 consecutive text rows. This technique misses variable renaming or code re-indentation, and reports also sequences of very short methods like getters and setters when they are written in the same order in two different files. inFusion calculates clone detection metrics that try to balance the size of consecutive cloned code, and the number of cloned and different lines in a fragment. Columns from five to eight in Table 7 give more details on the synthetic data of Table 4, reporting both the number of methods and the percentage on which the tools agree or disagree, respectively.

Feature envy Tables 8 and 9 are structured in the same way as Tables 6 and 7. The size of the system is, again, the total number of methods in the system, and the numbers reported in the columns titled with the name of the tools is the number of methods which the corresponding tool classifies as affected by Feature Envy. inFusion is again the most conservative tool, which classifies as affected by the smell only 10 methods of the more than 5000 analyzed. Table 9 shows that the number of positively classified methods on which the tools agree is almost equal to the number of positively classified methods on which the tools disagree (6 against 4).

God Class Tables 10 and 11 report the results of the God Class tool detection experiment. We notice that JDeodorant's positive answers include all of inFusion's positive answers. This yields the third best level of agreement among all the experiments, after Long Parameter List and Large Class. We do not have an explanation of why this happens, as the two tools are based on entirely different detection strategies:

Table 8 – Feature envy per-tool results

Version	inFusion	JDeodorant	Size	inFusion ratio	JDeodorant ratio
1.4	0	13	290	0.00%	4.48%
1.6	0	12	366	0.00%	3.28%
1.9.1	1	21	678	0.15%	3.10%
1.9.6	1	24	1302	0.08%	1.84%
1.9.10	1	44	878	0.11%	5.01%
1.10.2	7	64	1869	0.37%	3.42%

Table 9 – Feature envy tool comparison

Version	None	inFusion	JDeodorant	Both	Same		Different		Size
1.4	277	0	13	0	277	95.52%	13	4.48%	290
1.6	354	0	12	0	354	96.72%	12	3.28%	366
1.9.1	657	0	20	1	658	97.05%	20	2.95%	678
1.9.6	1277	1	24	0	1277	98.08%	25	1.92%	1302
1.9.10	833	1	44	0	833	94.87%	45	5.13%	878
1.10.2	1803	2	59	5	1808	96.74%	61	3.26%	1869

JDeodorant performs clustering on class methods and attributes, inFusion calculates metric for class cohesion, class functional complexity and access to foreign data.

Large class Tables 12 and 13 contain data on the Large Class smell. The size of a system is measured as the total number of classes in it, with the exclusion of anonymous inner classes. We did not consider anonymous inner classes to be stand-alone entities because they typically play the role of reusable code blocks. We considered them as part of the methods where they are instantiated. The column for the tools report the number of classes that each tool classifies as affected by the Large Class smell. It is apparent that all the classes that are positively classified by Checkstyle are also positively classified by PMD. Indeed, both tools use the number of line of codes as the metric to decide whether a class is large or not, and PMD uses by default a lower threshold than Checkstyle to label a class as affected by the smell. This is reflected

Table 10 – God class per-tool results

Version	inFusion	JDeodorant	Size	inFusion ratio	JDeodorant ratio
1.4	2	4	20	10.00%	20.00%
1.6	1	4	21	4.76%	19.05%
1.9.1	2	7	47	4.26%	14.89%
1.9.6	3	10	62	4.84%	16.13%
1.9.10	7	21	114	6.14%	18.42%
1.10.2	7	46	295	2.37%	15.59%

Table 11 – God class comparison

Version	None	inFusion	JDeodorant	Both	Same		Different		Size
1.4	16	0	2	2	18	90.00%	2	10.00%	20
1.6	17	0	3	1	18	85.71%	3	14.29%	21
1.9.1	40	0	5	2	42	89.36%	5	10.64%	47
1.9.6	52	0	7	3	55	88.71%	7	11.29%	62
1.9.10	93	0	14	7	100	87.72%	14	12.28%	114
1.10.2	249	0	39	7	256	86.78%	39	13.22%	295

Table 12 – Large class per-tool results

Version	Checkstyle	PMD	Size	Checkstyle ratio	PMD ratio
1.4	0	0	20	0.00%	0.00%
1.6	0	0	21	0.00%	0.00%
1.9.1	0	2	47	0.00%	4.26%
1.9.6	0	3	62	0.00%	4.84%
1.9.10	2	5	114	1.75%	4.39%
1.10.2	2	5	295	0.68%	1.69%

by the fact that kappa statistics scores the second best degree of agreement between tools after Long Parameter List.

Long Method Tables 14 and 15 report the results of the experiment for the Long Method detectors. This is the only smell which is detected by more than two tools. Correspondingly, Table 15 reports the number of each of the possible 2^3 combinations of tools decisions. JDeodorant is the tool which detects the highest number of smells, about one order of magnitude more than the other tools. The positive outcomes of PMD are mostly contained in those of the other tools. JDeodorant and Checkstyle are the tools that disagree the most on their positive answers. To understand the high number of positive results produced by JDeodorant, we manually inspected some of them, and found that JDeodorant includes methods with significantly shorter size than the other tools. JDeodorant includes in its search for Long Methods all the methods for which it finds an Extract Method refactoring opportunity, *independently* on their actual length [TC11].

Long parameter list Tables 16 and 17 report in details the results of the experiment on the Long Parameter List code smell. The tools which detect this smell simply compare the total number of parameters in all methods signatures against a fix threshold. PMD's default threshold is higher than Checkstyle's (10 against 7), thus Checkstyle classifies positively all the methods that are classified positively by PMD. This yields the highest agreement among all the detection strategies we analyzed.

Table 13 – Large class comparison

Version	None	Checkstyle	PMD	Both	Same		Different		Size
1.4	20	0	0	0	20	100.00%	0	0.00%	20
1.6	21	0	0	0	21	100.00%	0	0.00%	21
1.9.1	45	0	2	0	45	95.74%	2	4.26%	47
1.9.6	59	0	3	0	59	95.16%	3	4.84%	62
1.9.10	109	0	3	2	111	97.37%	3	2.63%	114
1.10.2	290	0	3	2	292	98.98%	3	1.02%	295

Table 14 – Long Method per-tool results

Version	Checkstyle	JDeodorant	PMD	Size	Checkstyle ratio	JDeodorant ratio	PMD ratio
1.4	2	26	2	290	0.69%	8.97%	0.69%
1.6	4	24	3	366	1.09%	6.56%	0.82%
1.9.1	8	43	5	678	1.18%	6.34%	0.74%
1.9.6	11	69	11	1302	0.84%	5.30%	0.84%
1.9.10	12	140	18	878	1.37%	15.95%	2.05%
1.10.2	14	195	18	1869	0.75%	10.43%	0.96%

Table 15 – Long method comparison

Checkstyle		×			×	×		×					
JDeodorant			×		×		×	×					
PMD				×		×	×	×					
Version									Same		Different		Size
1.4	261	2	25	1	0	0	1	0	261	90.00%	29	10.00%	290
1.6	338	2	23	0	0	2	1	0	338	92.35%	28	7.65%	366
1.9.1	628	4	41	0	0	3	1	1	629	92.77%	49	7.23%	678
1.9.6	1224	3	63	0	1	6	4	1	1225	94.09%	77	5.91%	1302
1.9.10	733	2	125	0	0	3	8	7	740	84.28%	138	15.72%	878
1.10.2	1669	2	180	1	0	2	5	10	1679	89.83%	190	10.17%	1869

Table 16 – Long parameter list per-tool results

Version	PMD	Checkstyle	Size	PMD ratio	Checkstyle ratio
1.4	0	0	290	0.00%	0.00%
1.6	0	0	366	0.00%	0.00%
1.9.1	0	1	678	0.00%	0.15%
1.9.6	1	2	1302	0.08%	0.15%
1.9.10	2	5	878	0.23%	0.57%
1.10.2	2	5	1869	0.11%	0.27%

Table 17 – Long parameter list comparison

Version	None	Checkstyle	PMD	Both	Same		Different		Size
1.4	290	0	0	0	290	100.00%	0	0.00%	290
1.6	366	0	0	0	366	100.00%	0	0.00%	366
1.9.1	677	1	0	0	677	99.85%	0	0.00%	678
1.9.6	1300	1	0	1	1301	99.92%	1	0.08%	1302
1.9.10	873	3	0	2	875	99.66%	2	0.23%	878
1.10.2	1864	3	0	2	1866	99.84%	2	0.11%	1869

Conclusions The experiments pointed out that different detectors for a same code smell produce different answers even if they are based on similar detection algorithms. The only notable exception is given by the tools that detect the God Class smell.

5.2 Code smell relevance

Question 2. *How relevant are the automatically detected smells to future software development?*

Figures 1 and 2 track the six analyzed versions of GanttProject the introduction and removal of all the Duplicated Code and Feature Envy smells. Each horizontal line indicates a set of smells[10] that are introduced (dotted end) and removed (crossed end) at the same version. The size of each set of smells is reported on the left of the dotted end of the corresponding line. Table 18 reports the number of detected smells in the six analyzed versions, for each possible value of persistence (we recall that persistence is the number of analyzed versions a smell is present in code). We did not analyze the evolution of the other smells, because either their higher granularity (Large Class, God Class), or the low number of positive occurrences found by the tools (Long Parameter List) do not allow us to draw significant conclusions.

For the Duplicated Code smell, Checkstyle and inFusion produced a total of 193 positive answers over the six versions of GanttProject we analyzed. Of them, 134 (69.4%) are removed from code before the last analyzed version 1.10.2, 22 (11.4%) remain in the code, and the remaining 37 (9.2%) cannot be assessed. It clearly emerges that most of the smells that are removed from the code, are removed within the next version after their introduction—precisely, 124 of 134 (92.5%). These results strongly suggest that the tools are able to detect sensible regions of code, that are subject to refactoring within a short time horizon. The Feature Envy data substantially agree, with 22 smells removed at the next version after their introduction, of 32 that are removed from the code (68.8%).

Conclusions Most of the automatically detected smells are relevant to the future evolution of the software system.

[10] Here we consider all the instances reported by *at least* one tool.

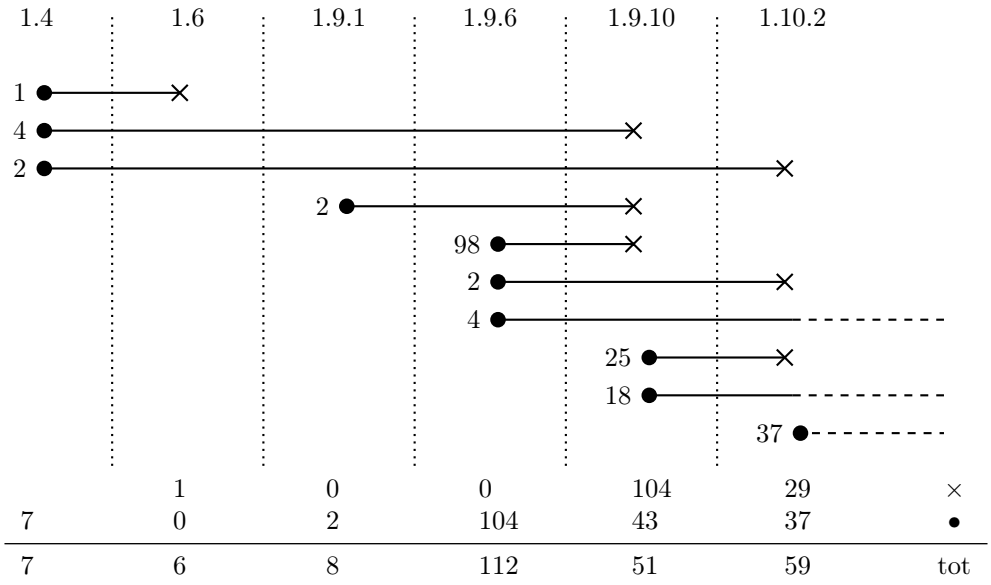

Figure 1 – Introduction and removal of Duplicated Code smells in GanttProject

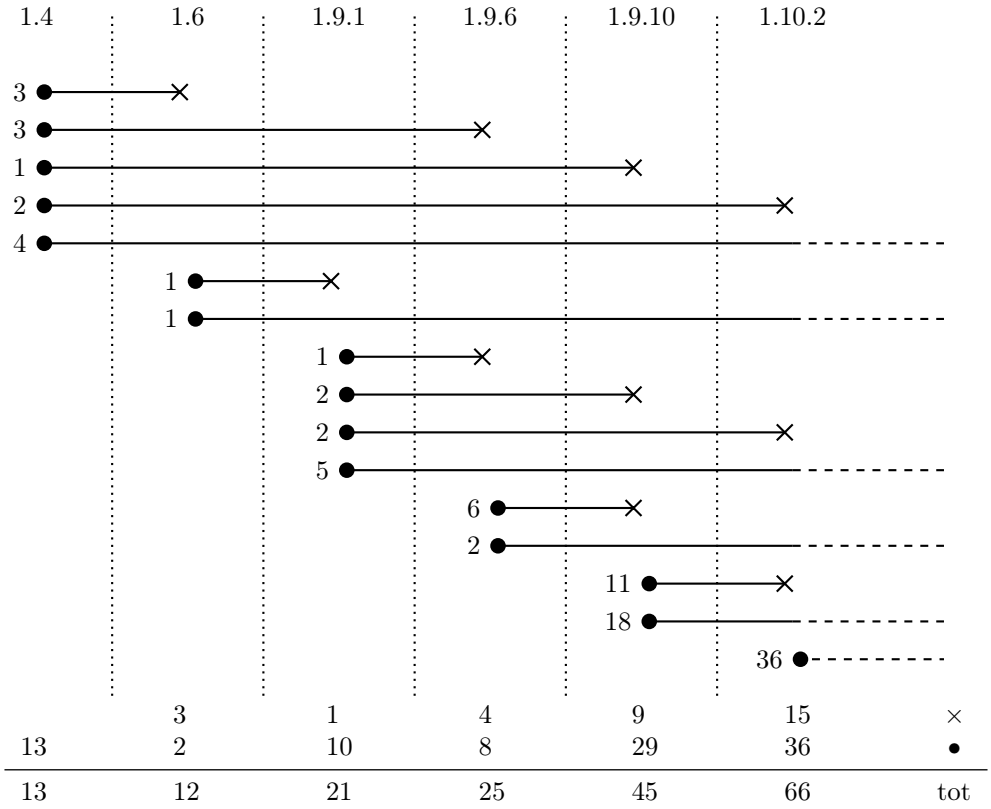

Figure 2 – Introduction and removal of Feature Envy smells in GanttProject

Table 18 – Persistence of smells in code

	Duplicated Code			Feature Envy		
Persistence	Total	Checkstyle	inFusion	Total	JDeodorant	inFusion
1 version	124	122	18	22	21	1
2 versions	4	4	2	2	2	0
3 versions	0	0	0	5	5	0
4 versions	4	4	2	1	1	0
5 versions	2	2	0	2	2	0
not removed	22	22	2	30	29	1
unassessable	37	37	0	36	35	6

5.3 Code smells evolution

Question 3. *Is the presence of smells related to some observable feature of the source code or of the process?*

We briefly summarize the evolution of GanttProject across the analyzed versions.

- *Version 1.4 to 1.6*: Added support to the Spanish language locale (added class `GanttLanguageSpanish`). Class `GanttAppli` has been renamed `GanttProject`.
- *Version 1.6 to 1.9.1* More locales (added classes `GanttLanguageGerman` and `GanttLanguagePortugues`), added management of human resources (added classes `GanttPeople` and `GanttPeoplePanel`), plus facilities for exporting projects to XML, HTML and PNG file formats (added classes `GanttDialogExport`, `GanttHTMLExport`, `GanttIO`, `GanttXMLFileFilter`, `GanttXMLOpen`, `GanttXMLSaver`).
- *Version 1.9.1 to 1.9.6*: More locales (added classes `GanttLanguageChineseBig5`, `GanttLanguageChineseGB`, `GanttLanguageItalian`, `GanttLanguageNorwegian`, `GanttLanguagePolish`, and `GanttLanguageTurkish`), added possibility of assigning tasks to human resources (added class `GanttDialogAssign`), added an unused `GanttGraphicPrint` class very similar to the `GanttGraphicArea` class.
- *Version 1.9.6 to 1.9.10*: Version 1.9.10 is characterized both by an overall refactoring of the code, and by the introduction of many new features. The existing and new classes were organized in packages, locales management was modified to load locales from resource bundles (eliminated `GanttLanguageXxxx` classes), GUI actions were moved to dedicated classes, a more comprehensive support to project resources was introduced (added classes `ProjectResource` and `ResourceLoadGraphicArea`, classes `GanttPeople` and `GanttPeoplePanel` refactored to `HumanResource` and `GanttResourcePanel`), added support to graphic themes and PDF export.
- *Version 1.9.10 to 1.10.2*: Added many packages and classes to implement additional functionalitie, for example time units and automatic task scheduling. The architecture is identical to that of version 1.9.10.

Figures 3 to 8 depict the evolution of the detected code smells in the six analyzed versions of GanttProject. Data are presented in percentage on the total number of classes or methods in the system.

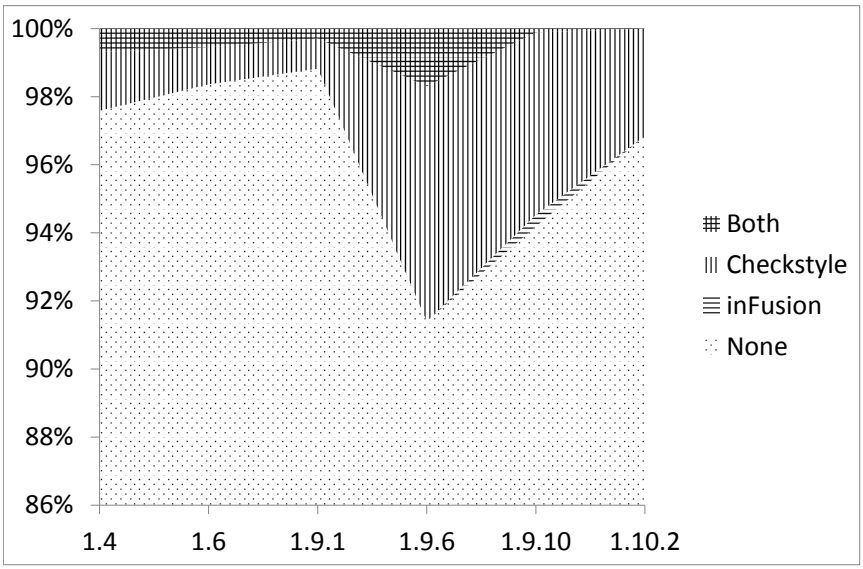

Figure 3 – Duplicated Code results on GanttProject

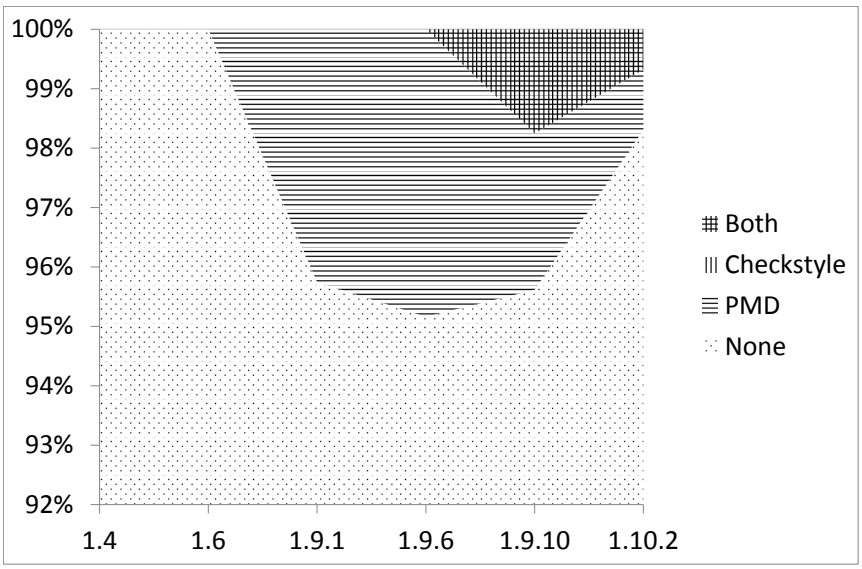

Figure 4 – Large Class results on GanttProject

The figures highlight how the tools report relevant increases (3 to 6 percentage points) in all the smells densities except God Class between versions 1.6 and 1.9.10. In detail, version 1.9.6 has maximum density of Duplicated Code and Large Class smells, and version 1.9.10 has maximum density of Feature Envy, Long Method, and Long Parameter List smells. By manually analyzing the source code, we noticed that starting from version 1.9.6 the size of the system grows rapidly, mainly because of the addition of new functionalities and the extension of existing ones. This evolution is also reflected in the progressive introduction of the use of packages to organize the classes in subsystems.

The rest of this section will report our qualitative analysis of tool data, and our remarks on how this data may be related to the actual evolution of the system.

Duplicated Code and Large Class The increase in the amount of Duplicated Code and Large Classes in version 1.9.6 (see Figure 3 and Figure 4) is mainly due to the introduction of the class `GanttGraphicPrint` which is mostly a copy of the existing `GanttGraphicArea` Large Class. Version 1.9.10 is characterized by the introduction of many new functionalities, implemented in a high number of new classes. Of these, the introduction of several implementations of the abstract `TableModel` class, all sharing Duplicated Code, increase the density of this smell. Finally, the introduction of the `ResourceLoadGraphicArea` class, almost a clone of the `GanttGraphicArea` class, contributes to the increase in density of the Duplicated Code smell. Overall, the classes `GanttGraphicArea`, `GanttGraphicPrint`, the subclasses of the `TableModel` class, and `ResourceLoadGraphicArea` account for more than the 78% of the duplicates in version 1.9.6, more than the 56% of the duplicates in version 1.9.10, and more than the 42% of the duplicates in version 1.10.2.

Feature Envy The increase in density of the Feature Envy (see Figure 5) smell in version 1.9.10 is justified by a decrease in size of the system, while the total number of Feature Envy instances grow slightly because of the introduction of clones (e.g., the `ResourceLoadGraphicArea` class) or new functionalities (e.g., in the `GanttGraphicArea` class). Overall, the classes `GanttProject`, `GanttCalendar`, `GanttResourcePanel`, `GanttGraphicArea` and `ResourceLoadGraphicArea` are always present among the first five most "smelly" classes in all the analyzed versions.

God Class Figure 6 suggests that the presence of the God Class smell is the most stable one: The tools consistently report as God Classes some fixed classes of the system (`GanttProject`, `GanttGraphicArea`, `GanttTree`, `GanttTask`) across all the analyzed versions. A manual analysis highlights how these classes indeed centralize a large part of the core functionalities of the project. The slight increment trend in God Classes is also justified by the introduction of clones (again, the `ResourceLoadGraphicArea` class) and new functionalities, which lower the cohesion of the classes where they are introduced (e.g., `GanttTree`). The overall stability of God Classes is a hint of the fact the developers never considered it important to refactor them. An aggregated package analysis on versions 1.9.10 and 1.10.2 highlights that the most problematic packages are, again, those devoted to the overall management of the application (`net.sourceforge.ganttproject`) and to the GUI (`net.sourceforge.ganttproject.gui.*`). These packages form 76% of the God Classes in version 1.9.10, and 56% of the God Classes in version 1.10.2, followed by the `net.sourceforge.ganttproject.io` package that contributes 13% more smells.

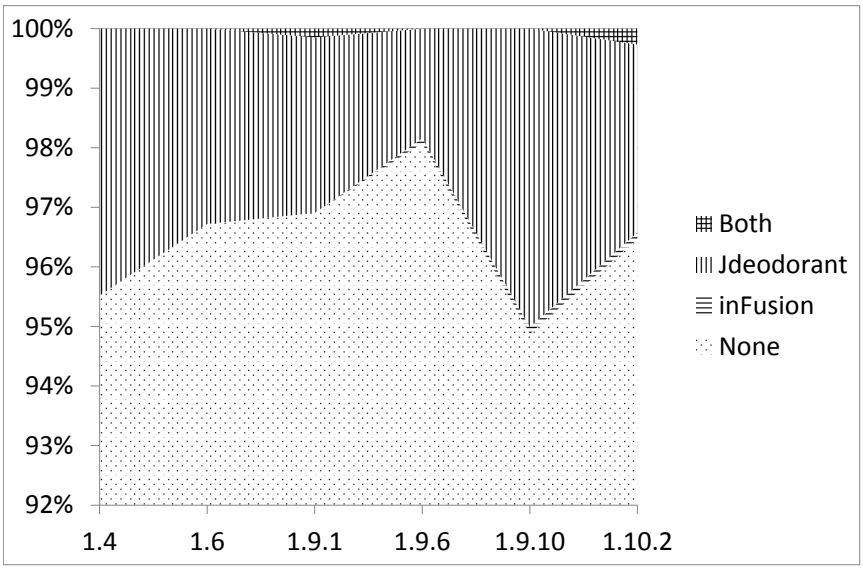

Figure 5 – Feature Envy results on GanttProject

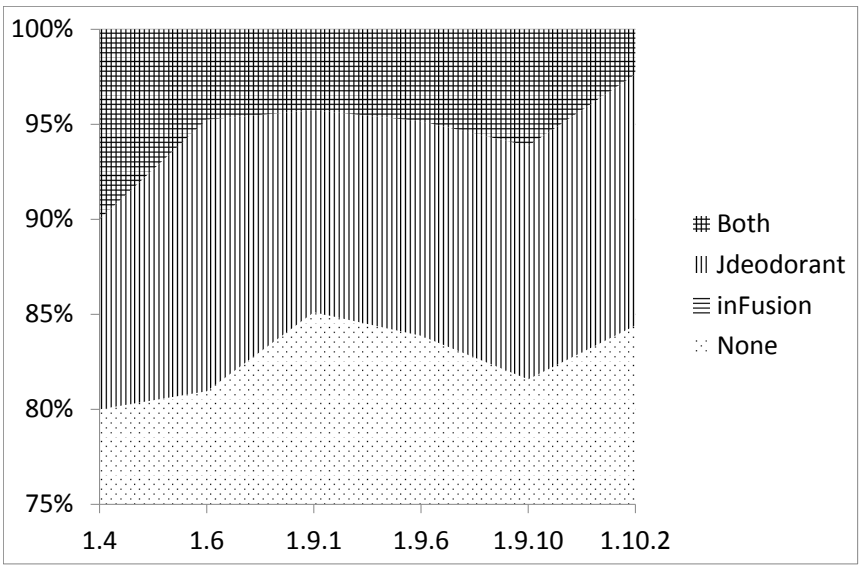

Figure 6 – God Class results on GanttProject

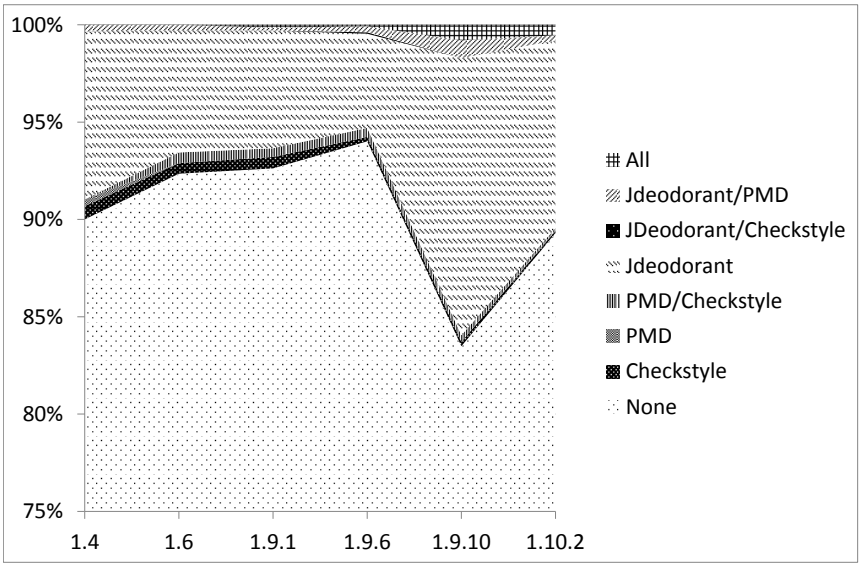

Figure 7 – Long Method results on GanttProject

The latter essentially is a refactoring of the `net.sourceforge.ganttproject` package of version 1.9.10, with some additional classes.

Long Method The increase of the Long Method density (see Figure 7) in version 1.9.10 is due both to the decrease in size of the system, as for the Feature Envy smell, and to the increase in number of found Long Method. All the tools agree in the positive trend, and most of the Long Method instances are in the problematic classes detected by the other smell analyses, i.e., `GanttProject`, `GanttGraphicArea`, `GanttTree`, `GanttGraphicPrint` and `ResourceLoadGraphicArea`. The three classes `GanttProject`, `GanttGraphicArea` and `GanttTree` are the top three most "smelly" classes in all the analyzed versions.

Long Parameter List The data about the Long Parameter List smell (see Figure 8) are not sufficient to derive sound conclusions about the evolution of software. We however remark that they are consistent with the trends highlighted in the rest of this section.

Conclusions The information about density and position of bad smells, as reported by the tools, appear to be related to observable features of the analyzed systems, both in its history and in its structure. It was not possible to perform a significant statistical correlation analysis because of the small size of the available data, and because the observed features are not easily measurable in an objective way.

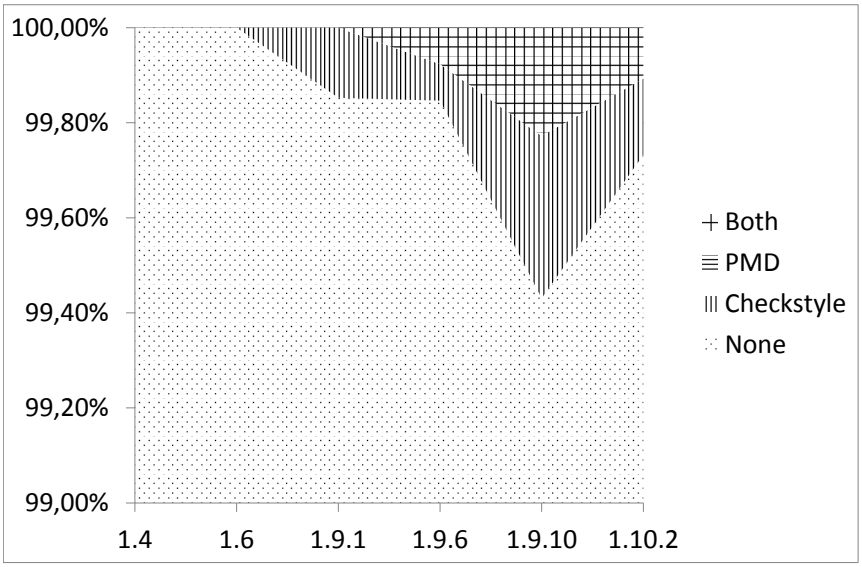

Figure 8 – Long Parameter List results on GanttProject

5.4 Threats to validity

The remainder of the section discusses the main threats to the validity of our experiments. We will first discuss the factors affecting the internal validity, and then those affecting the external validity.

5.4.1 Threats to internal validity

The main factors that negatively affect the internal validity of the above reported experiments are the size of the subject programs, the statistical dependence between different versions of the same program, and the possible errors in the transcription of the results of tools analysis. While we are not able to quantify their effect, we will give our informed opinion on them.

Size of subject programs The subjects of our analysis are small to medium size programs. Our experiments revealed that, at least for the chosen programs, the presence of a smell in a given code element is a low-probability event. This yielded a highly skewed distribution of "smelly" and "non-smelly" code elements, exacerbating the small scale effect and hindering the statistical relevance of the experiments.

Statistical dependence between different versions One may reasonably expect a strong correlation between the smells present in different versions of a software system if they are sufficiently close in time. The experiments for Question 2 highlight that a small but meaningful fraction of the smell instances detected by the tools persist across many versions of the software. A lower temporal granularity would enhance

this phenomenon. This might explain in part why the data reported for Question 1 in many cases does not vary meaningfully across versions.

Transcription errors Different tools produce their output in different formats. We obviated the need for translating tools results to a common format by a mix of automatic and manual translation. Unfortunately, distraction may introduce errors during manual translation; automatic translation is not affected by distraction, but, when it fails, it fails catastrophically. This happened as no tool maker usually documents the format of tool reports. We had therefore to infer these formats from examples of the reports themselves. Some tools did not even produce a report. In such a case, we had no alternative to manually transcribing tools results. Based on our experience we reviewed all produced data for consistency multiple times.

5.4.2 Threats to external validity

The main threats to external validity of our experiment are the nature of the analyzed system, and the temporal extent and granularity of our analysis.

Nature of the analyzed system GanttProject is a small size, open source software system developed in Java by a small team. We do not consider them to be readily extendable to large scale, non-Java, or proprietary projects. Appendix A reports the synthetic results of our replication of the Question 1 experiments on other small-to-medium size open source software systems written in Java, which essentially confirm our conclusions.

Temporal extent and granularity of the analysis We chose to analyze a manageable subset of all the released versions of GanttProject across a period of time (21 months) long enough to allow meaningful changes in software, but short enough to justify the rationale of Question 2 experiments. The selected versions from 1.4 to 1.10.2 can be considered a snapshot of the system evolution in its earliest phase. One can conjecture that a different temporal window, or a different temporal granularity between the chosen versions, would yield different results.

Definition of metrics When designing our experiments we took some decisions on how to measure some observed quantities. Some of these choices arose naturally, given the nature of the project under analysis, but they cannot be generalized. As an example, our choice of not considering anonymous inner classes as independent entities was motivated by the fact that GanttProject has few of them, they are usually found in GUI classes, and are mostly used as callbacks to methods of their container classes. This may not be always the case. Other choices are even less clear, e.g., whether Java interfaces should be counted when measuring the density of class smells (we chose to count them), and how many distinct instances of the Duplicated Code code smell are given by n different replicas of a same code block (we chose to consider them as n different instances of the Duplicated Code smell). These slight variations in the exact definition of the metrics have a very limited impact on the systems we analyzed, but we cannot exclude that they might become meaningful when analyzing different systems.

6 Related work

We have not found in the literature papers describing experience reports on the usage of several tools for code smell detection. The only work, to our knowledge, attempting a quantitative assessment of the detection ability of more than one code smell detection tool is from Moha et al. [MGM+10]. The authors compare their automatic code smell detection tool and iPlasma against precision and recall. Benchmark data are obtained by manual inspection of GanttProject v1.10.2 and Xerces v2.7.0. This is the reason why we decided to start our experiments by analyzing the GanttProject system.

Many publications are available on code smells and refactoring. We cite below some relevant papers which we include in the following subsections.

6.1 Code smell detection tools

We found many papers on single tools for code smell detection, as those we have described in Section 3.

6.2 Code smells and refactoring

Kerievsky defines some new smells in his book [Ker04], but the central aim of the book is on refactoring. In particular, he analyzes how through refactoring one can introduce patterns into a system, exploiting the fact that design patterns can provide targets for the refactoring.

Mens et al. provide a survey of existing research in the field of software refactoring [MT04], where they identify six distinct activities the refactoring process consists of. They do not pay particular attention to the refactoring of code smells.

Wake, in the Refactoring Workbook [Wak03], aims to provide practice in the identification of the most important smells and practice with the most important refactoring techniques, with particular emphasis on discovering new refactorings.

Counsell et al. [CHH+10b, CHH10a] analyze a set of five open-source Java systems; they show very little tendency for smells to be eradicated by developers, and they provide a theoretical enumeration of smell-related refactorings to suggest why smells may be left alone from an effort perspective.

6.3 Code smells evolutions

It is claimed that classes that are involved in certain code smells are liable to be changed more frequently and have more defects than other classes in the code.

A study of God Classes and Brain Classes in the evolution of three open source systems by Olbrich et al. [OCS10] investigated the extent to which this claim is true for God Classes and Brain Classes. The results show that God and Brain Classes were changed more frequently and contained more defects than other kinds of classes. However, when they normalized the measured effects with respect to size, then God and Brain Classes were less subject to change and had fewer defects than other classes. Hence, the presence of God and Brain Classes is not necessarily harmful because such classes may be an efficient way of organizing code.

Zhang et al. [ZHBW08] assert that the empirical basis of using bad smells to direct refactoring and to address "trouble" in code is not clear, and they propose a study which aims to empirically investigate the impact of bad smells on software in terms of their relationship to faults.

Zhang et al. [ZBWH11] investigated the relationship between six of Fowler et al.'s code bad smells (Duplicated Code, Data Clumps, Switch Statements, Speculative Generality, Message Chains, and Middle Man) and software faults. Their study can be used by software developers to prioritise refactoring. In particular, they suggest that source code containing Duplicated Code is likely to be associated with more faults than source code containing the other five code bad smells. As a consequence, Duplicated Code should be prioritised for refactoring. Source code containing Message Chains seems to be associated with a high number of faults in some situations. Consequently, it is another code bad smell which should be prioritised for refactoring. Source code containing only one of the other smells is not likely to be fault-prone.

Vaucher et al. [VKMG09] study in two open-source systems the "life cycle" of God Classes: how they arise, how prevalent they are, and whether they remain or they are removed as the systems evolve over time through a number of versions. They show how to automatically detect the degree of "godliness" of classes. Then they show that by identifying the evolution of "godliness", it is possible to distinguish between those classes by design (good code), from those that occurred by accident (bad code). This methodology can guide software quality teams in their efforts to implement prevention and correction mechanisms.

Khomh et al. [KDPG09] investigate if classes with code smells are more change-prone than classes without smells. They detect 29 code smells in 9 releases of Azureus and in 13 releases of Eclipse and study the relationship between classes with these code smells and class change-proneness. They show that in almost all releases of Azureus and Eclipse, classes with code smells are more change-prone than others, and that specific smells are more correlated than others to change-proneness.

Li et al. [LS07] present the results from an empirical study that investigated the relationship between bad smells and class error probability in three error-severity levels in an industrial-strength open source system. Their research, which was conducted in a context of the post-release system evolution process, showed that some bad smells were positively associated with the class error probability in the three error-severity levels. This finding supports the use of bad smells as a systematic method to identify and refactor problematic classes in this specific context.

6.4 Software quality evaluation for code smell detection and refactoring

Kataoka et al. [KIAF02] propose coupling metrics as an evaluation method to determine the effect of refactoring on the maintainability of programs.

Tahvildari et al. [TK03] analyze the association of refactorings with a possible effect on maintainability enhancements through refactorings. They use a catalogue of object-oriented metrics as an indicator for the transformations to be applied to improve the quality of a legacy system. The indicator is achieved by analysing the impact of each refactoring on these object-oriented metrics.

Similar to this approach, Arcelli et al. [AFS11], according to well known metrics for code and design quality evaluation, analyze the impact of refactoring applied to remove some code smells on the quality evaluation of these metrics, with the aim to prioritize the smells to be removed.

Work on refactoring prioritization using bad code smells is described by Zhang et al. [ZBWH11], where they provide a prioritization among six analyzed smells according to their association with software faults.

6.5 Taxonomies or Classifications of Smells

Fowler in his book on code smells and refactoring [Fow99] does not propose any type of smell classification or taxonomy. The first taxonomy we are aware of has been proposed by Mäntylä et al. [MVL03, MVL04], with the aim to identify possible correlations among smells. They identify six disjoint classes, which include all the smells identified by Fowler.

Wake proposes a different classification in his book [Wak03]. He first distinguishes between smells within classes and smells between classes, he defines new smells, and classifies all the smells of Fowler and the new ones in nine classes. In his book, Wake describes also the refactoring necessary to remove all the described smells and how the classification suggests possible smell correlations.

Lanza et al. [LM06] classify 11 smells in three different categories, 7 from Fowler and 6 new smells. Their classification is particularly oriented to the identification of relationships existing among smells.

Moha et al. [MGDM10] propose a generic classification of the smells as intra-class and inter-class smells, as in the classification of Wake. Then they identify three categories (Structural, Lexical, Measurable), where they classify the smells of Fowler, with some overlapping because they classify some smells in more than one category. They propose this kind of categorization, with the aim of simplifying the detection technique of the different smells. In fact, they assert that "for example, the detection of a structural smell may essentially be based on static analyses; the detection of a lexical smell may rely on natural language processing; the detection of a measurable smell may use metrics".

Marticorena [MLC06] establishes additional criteria for classifying bad smells, criteria related to metric features, and proposes a method to evaluate the suitability of the tools assisting bad code smell detection, as well as selection and implementation of metrics linked with bad code smells.

Counsell et al. [CHH+10b] provide an in-depth deconstruction of both Fowler's and Kerievsky's code smells in an attempt to determine their overlap.

6.6 Experimental evaluations

Few studies have tackled the evaluation of smell occurrences in code. Mäntylä et al. [MVL04, Mä05] study how humans, rather than tools, evaluate code smells by analyzing a number of manual surveys by 12 different software developers of a Finnish software house. From their analysis they address questions on how different evaluators disagree, how their decisions are correlated to software metrics, to the evaluator demographics and experience. Murphy-Hill and Black [MHB10] designed a similar experiment with 12 software developers with heterogeneous background, which inspected code from two open source projects. From the experimental data the authors assessed the usefulness, rather than precision and recall, of their code smell detection tool Stench Blossom, and also analyzed, similarly to Mäntylä, whether answers given by different evaluators disagreed.

7 Conclusions and future work

This paper presented a comparison of four code smell detection tools on six versions of a medium-size software project, and an assessment of the agreement, consistency and relevance of the answers produced. To the best of our knowledge, this is the first

comparative study with similar aims and scope. Our experiments suggest that different detectors for the same smell do not meaningfully agree in their answers. Nevertheless, they can detect problematic regions of code which are relevant for the future evolution of software. It is our opinion that a better understanding of what the tools available today can do is a step towards devising what future tools shall do to support the real needs of developers, software architects and managers. We believe that this paper provides the community with a view on these needs, and an approach to giving a quantitative assessment on how tools come close to fulfilling them.

We can assert that code smell detection tools are certainly useful to assess which parts of the code need to be improved, but we are not able to determine which is the best one. This was one of the aims of our analysis, but as largely outlined in the paper, the agreement in the results is scarce and a benchmark for the comparison of these results is not yet available. We observed that we have the best agreement in the results for the God Class detection and then for the Large Class and Long Parameter List smells. Developers aiming to recover all the parts of the code that need to be improved perhaps need to exploit the output of more than one tool. We remarked that the automatically detected smells usually are in sensible zones of the project, and are removed quickly: This suggests that tools are able to highlight relevant issues in code. Finally, project evolution data strongly suggests that automatic smell detection helps provide an understanding of software evolution. We notice that only one of the considered tools, JDeodorant, can perform refactoring, despite the idea of code smell is strictly related to refactoring opportunities. A user would expect smell detection tools to be able to support at least the removal of the simplest kind of smells, but this functionality is unsupported in most cases.

Our current research efforts are directed both on refining our experiments by gathering more data on different combination of tools, smells, and analyzed systems; we aim at deriving more general results, and at extending our analysis with experiments to investigate how useful code smell detection tools are for assessing internal software quality and directing refactoring activities. There are several issues which make these tasks harder than one would expect. We discussed obstacles to gathering data by applying the available tools to different software systems.

Another objective is to determine how well code smell detection tools approximate human ability for detecting problematic regions of code. This requires sufficient manual code analysis data to perform a statistically significant test. These manual validations will allow a better comparison of results from tools and to create a benchmark dataset. To do that we aim to better refine the definitions of the smells we have analyzed in this paper and the definitions of other common smells as Data Class, Speculative Generality and Intensive Coupling (see Appendix B). The refined definitions can be used to improve the detection techniques of the smells. This should take into account information related to the domain of the analyzed systems, to better perform a sensitivity analysis on metrics thresholds and discard from the detection the domain dependent smells, which do not represent symptoms of code decay.

Finally, we would like to analyze the mutual relationships between code smells, and between smells and other recurrent structures in code, such as design patterns and antipatterns, and how these can be sources of misclassification by tools and humans. As an example, there is a well known relationship [Fow99] between the Feature Envy smell and the Visitor pattern, since a Visitor relocates on purpose some code outside the class where it would naturally belong. A tool that is able to recognize an instance of a Visitor pattern would be, in principle, more precise in detecting real smells than

one that is unable to. We are also doing some preliminary research to correlate code smells and micro patterns [AFZ11a] and both antipatterns and design pattern detection within our MARPLE project [AFZ11b].

References

[AFMM+11] Francesca Arcelli Fontana, Elia Mariani, Alessandro Morniroli, Raul Sormani, and Andrea Tonello. An experience report on using code smells detection tools. In *IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW 2011)*, RefTest 2011, pages 450–457, Berlin, Germany, March 2011. IEEE Computer Society. doi:10.1109/ICSTW.2011.12.

[AFS11] Francesca Arcelli Fontana and Stefano Spinelli. Impact of refactoring on quality code evaluation. In *Proceeding of the 4th workshop on Refactoring tools*, WRT '11, pages 37–40, Waikiki, Honolulu, HI, USA, 2011. ACM. Workshop held in conjunction with ICSE 2011. doi:10.1145/1984732.1984741.

[AFZ11a] Francesca Arcelli Fontana and Marco Zanoni. On investigating code smells correlations. In *IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW 2011)*, RefTest 2011, pages 474–475, Berlin, Germany, March 2011. IEEE Computer Society. doi:10.1109/ICSTW.2011.14.

[AFZ11b] Francesca Arcelli Fontana and Marco Zanoni. A tool for design pattern detection and software architecture reconstruction. *Information Sciences*, 181(7):1306–1324, April 2011. doi:10.1016/j.ins.2010.12.002.

[CHH10a] Steve J. Counsell, Hamza Hamza, and Rob M. Hierons. An empirical investigation of code smell 'deception' and research contextualisation through paul's criteria. *Journal of Computing and Information Technology*, 18(4):333—-340, 2010. doi:10.2498/cit.1001919.

[CHH+10b] Steve J. Counsell, Rob M. Hierons, Hamza Hamza, Sue Black, and M. Durrand. Exploring the eradication of code smells: An empirical and theoretical perspective. *Advances in Software Engineering*, 2010:12, 2010. doi:10.1155/2010/820103.

[Coh60] Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1):37–46, 1960. doi:10.1177/001316446002000104.

[Fle71] Joseph L. Fleiss. Measuring nominal scale agreement among many raters. *Psychological Bulletin*, 76(5):378–382, November 1971.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co. Inc., Boston, MA, USA, 1999. http://www.refactoring.com/.

[FTCS09] Marios Fokaefs, Nikolaos Tsantalis, Alexander Chatzigeorgiou, and Jörg Sander. Decomposing object-oriented class modules using an agglomerative clustering technique. In *Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM'2009)*, pages 93–101, September 2009. doi:10.1109/ICSM.2009.5306332.

[KDPG09] Foutse Khomh, Massimiliano Di Penta, and Yann-Gaël Guéhéneuc. An exploratory study of the impact of code smells on software change-proneness. In *16th Working Conference on Reverse Engineering (WCRE '09)*, pages 75–84, Lille, France, October 2009. IEEE Computer Society. doi:10.1109/WCRE.2009.28.

[Ker04] Joshua Kerievsky. *Refactoring to Patterns*. Pearson Higher Education, 2004.

[KIAF02] Yoshio Kataoka, Takeo Imai, Hiroki Andou, and Tetsuji Fukaya. A quantitative evaluation of maintainability enhancement by refactoring. In *Proceedings of International Conference on Software Maintenance (ICSM 2002)*, pages 576–585, Montréal, Canada, October 2002. IEEE Computer Society. doi:10.1109/ICSM.2002.1167822.

[LM06] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006. doi:10.1007/3-540-39538-5.

[LS07] Wei Li and Raed Shatnawi. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. *Journal of Systems and Software*, 80(7):1120–1128, 2007. Dynamic Resource Management in Distributed Real-Time Systems. doi:10.1016/j.jss.2006.10.018.

[MGDM10] Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Françoise Le Meur. DECOR: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1):20–36, January–February 2010. doi:10.1109/TSE.2009.50.

[MGM+10] Naouel Moha, Yann-Gaël Guéhéneuc, Anne-Françoise Le Meur, Laurence Duchien, and Alban Tiberghie. From a domain analysis to the specification and detection of code and design smells. *Formal Aspects of Computing*, 22:345–361, 2010. doi:10.1007/s00165-009-0115-x.

[MHB10] Emerson Murphy-Hill and Andrew P. Black. An interactive ambient visualization for code smells. In *Proceedings of the 5th international symposium on Software visualization*, SOFTVIS '10, pages 5–14, Salt Lake City, Utah, USA, 2010. ACM. doi:10.1145/1879211.1879216.

[MLC06] Raúl Marticorena, Carlos López, and Yania Crespo. Extending a taxonomy of bad code smells with metrics. In *International Workshop on Object-Oriented Reengineering, co-located with the 20th European Conference on Object-Oriented Programming (ECOOP 2006)*, Nantes, France, July 2006. Software Composition Group (online). URL: http://scg.unibe.ch/wiki/events/woor2006/woor2006paper4?view=PRDownloadView.

[MMM+05] Cristina Marinescu, Radu Marinescu, Petru Mihancea, Daniel Ratiu, and Richard Wettel. iPlasma: An integrated platform for quality assessment of object-oriented design. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM 2005)*, pages 77–80, Budapest, Hungary, September 2005. IEEE Computer Society. (Industrial & Tool Proceedings), Tool Demonstration Track.

[MT04] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, February 2004. doi:10.1109/TSE.2004.1265817.

[MVL03] Mika V. Mäntylä, Jari Vanhanen, and Casper Lassenius. A taxonomy and an initial empirical study of bad smells in code. In *Proceedings of International Conference on Software Maintenance (ICSM 2003)*, pages 381–384, Amsterdam, The Netherlands, September 2003. IEEE Computer Society. doi:10.1109/ICSM.2003.1235447.

[MVL04] Mika V. Mäntylä, Jari Vanhanen, and Casper Lassenius. Bad smells — humans as code critics. In *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM 2004)*, pages 399–408, Chicago, Illinois, September 2004. IEEE Computer Society. doi:10.1109/ICSM.2004.1357825.

[Mä03] Mika Mäntylä. Bad smells in software — a taxonomy and an empirical study. Master's thesis, Helsinki University of Technology, 2003.

[Mä05] Mika V. Mäntylä. An experiment on subjective evolvability evaluation of object-oriented software: Explaining factors and interrater agreement. In *Proceedings of 2005 International Symposium on Empirical Software Engineering*, pages 287–296, Noosa Heads, Queensland, Australia, November 2005. IEEE Computer Society. doi:10.1109/ISESE.2005.1541837.

[OCS10] Steffen M. Olbrich, Daniela S. Cruzes, and Dag I.K. Sjøberg. Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems. In *IEEE International Conference on Software Maintenance (ICSM 2010)*, pages 1–10, Timisoara, Romania, September 2010. IEEE Computer Society. doi:10.1109/ICSM.2010.5609564.

[TC09] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering*, 35(3):347–367, 2009. doi:10.1109/TSE.2009.1.

[TC11] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of extract method refactoring opportunities for the decomposition of methods. *Journal of Systems and Software*, 84(10):1757–1782, 2011. doi:10.1016/j.jss.2011.05.016.

[TK03] Ladan Tahvildari and Kostas Kontogiannis. A metric-based approach to enhance design quality through meta-pattern transformations. In *Proceedings of the Seventh European Conference on Software Maintenance and Reengineering*, pages 183–192, Benevento, Italy, March 2003. doi:10.1109/CSMR.2003.1192426.

[vEM02] Eva van Emden and Leon Moonen. Java quality assurance by detecting code smells. In *Proceedings 9th Working Conference on Reverse Engineering (WCRE 2002)*, Richmond, Virginia, USA, 2002. IEEE Computer Society. doi:10.1109/WCRE.2002.1173068.

[VKMG09] Stéphane Vaucher, Foutse Khomh, Naouel Moha, and Yann-Gaël Guéhéneuc. Tracking design smells: Lessons from a study of god classes. In *16th Working Conference on Reverse Engineering (WCRE '09)*,

pages 145–154, Lille, France, October 2009. IEEE Computer Society. doi:10.1109/WCRE.2009.23.

[Wak03] William C. Wake. *Refactoring Workbook*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1 edition, 2003.

[ZA10] Nico Zazworka and Christopher Ackermann. CodeVizard: a tool to aid the analysis of software evolution. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '10, pages 63:1–63:1, Bolzano, Italy, September 2010. ACM. doi:10.1145/1852786.1852865.

[ZBWH11] Min Zhang, Nathan Baddoo, Paul Wernick, and Tracy Hall. Prioritising refactoring using code bad smells. In *IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW 2011), RefTest Workshop*, pages 458–464, Berlin, Germany, March 2011. IEEE Computer Society. doi:10.1109/ICSTW.2011.69.

[ZHBW08] Min Zhang, Tracy Hall, Nathan Baddoo, and Paul Wernick. Do bad smells indicate "trouble" in code? In *Proceedings of the 2008 workshop on Defects in large software systems*, DEFECTS '08, pages 43–44, Seattle, Washington, 2008. ACM. doi:10.1145/1390817.1390831.

A Experiments on other systems

In this section we report additional data gathered on other systems using the same code smell detectors. The data reported in this section does not have the same coverage (in terms of number of analyzed versions and smell types) of the data reported in the paper, because of the manual effort needed for the merge, check and transcription of the results from different tools. We reported all the experimental data whose quality we were able to align with that of GanttProject data, and disregarded all the data whose inconsistencies we were not able to adjust or justify.

Table 19 reports results on the detection of the Long Method smell using PMD and Checkstyle. In the JEdit 3.0, 3.2 and 4.2 experiments we also used JDeodorant. In Table 20 the results of the detection of the Long Parameter List smell on four systems/versions are reported. The experiments have been made using PMD and Checkstyle. Table 21 and Table 22 show respectively the agreement and kappa statistics values about the detection of the Feature Envy and the God Class smells using inFusion and JDeodorant.

Looking at the data it emerges clearly that the agreement is very low in most cases, and it is not homogeneous from system to system and from version to version. We do not have an explanation for this disparity. We conjecture that it is due to some instability of the tools, or to errors during the manual data merge and transcription phase.

B Other code smell definitions

We give below the definitions of the smells we have mentioned in the paper and in Table 2, which have been not the subject of our detailed analysis. The definitions are reported verbatim from their sources.

B.1 Fowler definitions

Data Class These are classes that have fields, getting and setting methods for the fields, and nothing else. Such classes are dumb data holders and are almost certainly being manipulated in far too much detail by other classes [Fow99, p. 86].

Data Clumps Often you'll see the same three or four data items together in lots of places: fields in a couple of classes, parameters in many method signatures. Bunches of data that hang around together really ought to be made into their own object [Fow99, p. 81].

Message Chains You see message chains when a client asks one object for another object, which the client then asks for yet another object, which the client then asks for yet another another object, and so on. (...) Navigating this way means the client is coupled to the structure of the navigation. [Fow99, p. 84].

Refused Bequest Subclasses get to inherit the methods and data of their parents. But what if they don't want or need what they are given? (...) this means the hierarchy is wrong [or] the subclass is reusing behavior but does not want to support the interface of the superclass [Fow99, p. 87].

Table 19 – Results on Long Method comparing PMD and Checkstyle (also JDeodorant for JEdit 3.0, 3.2, 4.2)

System	# methods	Agreement	Kappa	95% conf interval
JEdit 3.0	2973	88.66%	0.0498	[0.0290, 0.0705]
JEdit 3.2	3568	88.99%	0.0520	[0.0331, 0.0710]
JEdit 4.0	4127	99.10%	0.4093	[0.3828, 0.4358]
JEdit 4.2	5288	98.85%	0.2240	[0.2084, 0.2395]
JEdit 4.4	6941	99.27%	0.4242	[0.4040, 0.4445]
JFreeChart 0.9.15	5812	99.55%	-0.0022	[-0.0280, 0.0235]
JRefactory 2.9.9	8475	99.86%	0.6660	[0.6447, 0.6872]
JRefactory 2.9.18	10660	99.94%	0.8418	[0.8228, 0.8608]
JRefactory 2.9.19	10660	99.69%	0.1523	[0.1333, 0.1713]
Lucene 1.9.1	340	96.47%	0.0000	[-0.0000, 0.0000]

Table 20 – Results on Long Parameter List (PMD and Checkstyle)

System	# methods	Agreement	Kappa	95% conf interval
JFreeChart 0.9.15	5812	98.73%	-0.0064	[-0.0321, 0.0193]
JRefactory 2.9.9	8475	99.98%	0.8570	[0.8357, 0.8783]
JRefactory 2.9.18	10660	99.98%	0.8999	[0.8809, 0.9189]
JRefactory 2.9.19	10660	99.80%	0.1590	[0.1400, 0.1780]

Table 21 – Results on Feature Envy (JDeodorant and inFusion)

System	# methods	Agreement	Kappa	95% conf interval
Lucene 1.9.1	340	97.65%	0.0000	[0.0000, 0.0000]
Lucene 2.2.0	403	83.87%	-0.0013	[-0.0577, 0.0551]
Lucene 2.4.0	538	84.57%	-0.0004	[-0.0441, 0.0434]
Lucene 2.9.2	698	81.95%	-0.0245	[-0.0600, 0.0110]
Lucene 3.0.3	624	84.29%	-0.0064	[-0.0487, 0.0359]

Table 22 – Results on God Class (JDeodorant and inFusion)

System	# classes	Agreement	Kappa	95% conf interval
Lucene 2.2.0	160	71.25%	0.1873	[0.0970, 0.2776]
Lucene 2.4.0	220	70.00%	0.1352	[0.0454, 0.2250]
Lucene 3.0.3	268	76.12%	0.1845	[0.1038, 0.2651]

Shotgun Surgery [This smell occurs] when every time you make a kind of change, you have to make a lot of little changes to a lot of different classes. When the changes are all over the place, they are hard to find, and it's easy to miss an important change. [Fow99, p. 80].

Speculative Generality You get it when people say, "Oh, I think we need the ability to this kind of thing someday" and thus want all sorts of hooks and special cases to handle things that aren't required [Fow99, p. 83].

Switch Statements Often you find the same switch statement scattered about a program in different places. If you add a new clause to the switch, you have to find all these switch statements and change them. The object-oriented notion of polymorphism gives you an elegant way to deal with this problem. (...) Often the switch statement switches on a type code [Fow99, p. 82].

B.2 van Emden and Moonen definitions

Instanceof A concentration of instanceof operators in the same block of code may indicate a place where the introduction of an inheritance hierarchy or the use of method overloading might be a better solution [vEM02, p. 101].

Typecast Typecasts are used to explicitly convert an object from one class type into another. Many people consider typecasts to be problematic since it is possible to write illegal casting instructions in the source code which cannot be detected during compilation but result in runtime errors [vEM02, p. 101].

B.3 Mäntylä definitions

Dead Code Fowler and Beck did not present a smell for dead code, which is quite surprising (...). With Dead Code I mean code that has been used in the past, but is not currently used. [Mä03, p. 97].

B.4 Lanza and Marinescu definitions

Brain Class [Brain Classes are] complex classes that tend to accumulate an excessive amount of intelligence, usually in the form of several methods affected by Brain Method. (...) [Brain Classes] are not detected as God Classes either because they do not abusively access data of "satellite" classes, or because they are a little more cohesive [LM06, p. 33].

Brain Method Brain Methods tend to centralize the functionality of a class, in the same way as a God Class centralizes the functionality of an entire subsystem, or sometimes even a whole system [LM06, p. 92].

Extensive (Dispersed) Coupling This is the case where a single operation communicates with an excessive number of provider classes, whereby the communication with each of the classes is not very intense i.e., the operation calls one or a few methods from each class [LM06, p. 127].

Intensive Coupling One of the frequent cases of excessive coupling that can be improved is when a method is tied to many other operations in the system, whereby these provider operations are dispersed only into one or a few classes (...). In other words, this is the case where the communication between the client method and (at least one of) its provider classes is excessively verbose [LM06, p. 120].

Tradition Breaker [A] derived class should not break the inherited "tradition" and provide a large set of services which are unrelated to those provided by its base class. (...) if the child class hardly specializes any inherited services and only adds brand new services which do not depend much on the inherited functionality, then this is a sign that something is wrong either with the definition of the child's class interface or with its classification relation [LM06, p. 152].

About the authors

Francesca Arcelli Fontana is an associate professor of software engineering at the Department of Computer Science of the University of Milano Bicocca, where she works on software evolution and reverse engineering. Contact her at `arcelli@disco.unimib.it`, or visit `http://essere.disco.unimib.it`.

Pietro Braione is a full-time researcher at the Department of Computer Science of the University of Milano Bicocca. His research interests include testing and analysis of software and formal methods. Contact him at `pietro.braione@disco.unimib.it`, or visit `http://www.lta.disco.unimib.it`.

Marco Zanoni is a PhD Student at the Department of Computer Science of the University of Milano Bicocca; he took his M.S. degree in 2008. He is currently working on techniques for design pattern detection on object oriented systems. Contact him at `marco.zanoni@disco.unimib.it`, or visit `http://essere.disco.unimib.it`.