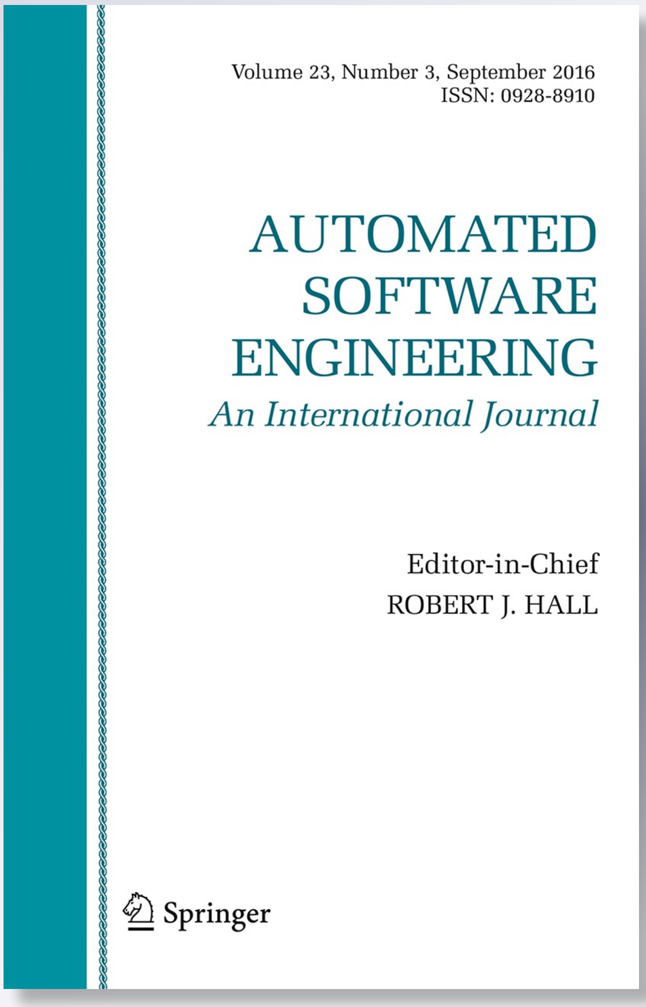
An approach to prioritize code smells for refactoring

Santiago A. Vidal, Claudia Marcos & J. Andrés Díaz-Pace

Automated Software Engineering
An International Journal

ISSN 0928-8910
Volume 23
Number 3

Autom Softw Eng (2016) 23:501-532
DOI 10.1007/s10515-014-0175-x



Autom Softw Eng (2016) 23:501–532
DOI 10.1007/s10515-014-0175-x

An approach to prioritize code smells for refactoring

Santiago A. Vidal · Claudia Marcos · J. Andrés Díaz-Pace

Received: 21 March 2014 / Accepted: 28 November 2014 / Published online: 14 December 2014

Abstract Code smells are a popular mechanism to find structural design problems in software systems. Consequently, several tools have emerged to support the detection of code smells. However, the number of smells returned by current tools usually exceeds the amount of problems that the developer can deal with, particularly when the effort available for performing refactorings is limited. Moreover, not all the code smells are equally relevant to the goals of the system or its health. This article presents a semi-automated approach that helps developers focus on the most critical problems of the system. We have developed a tool that suggests a ranking of code smells, based on a combination of three criteria, namely: past component modifications, important modifiability scenarios for the system, and relevance of the kind of smell. These criteria are complementary and enable our approach to assess the smells from different perspectives. Our approach has been evaluated in two case-studies, and the results show that the suggested code smells are useful to developers.

Keywords Code smells · Refactoring · Software evolution · Design problems

S. A. Vidal (✉) · J. A. Díaz-Pace · C. Marcos
ISISTAN, UNICEN, Tandil, Argentina
e-mail: svidal@exa.unicen.edu.ar

J. A. Díaz-Pace
e-mail: adiaz@exa.unicen.edu.ar

S. A. Vidal · J. A. Díaz-Pace
CONICET, Tandil, Argentina

C. Marcos
CIC, Buenos Aires, Argentina
e-mail: cmarcos@exa.unicen.edu.ar

1 Introduction

Software evolution and maintenance involve high costs in the development process (April and Abran 2008; Seacord et al. 2003; Erlikh 2000), particularly as systems become larger and complex. A usual concern that makes system maintenance and evolution difficult is the existence of structural design problems, which were not sufficiently taken care of in early development stages. These design problems are often described as code smells (Fowler 1999). A *code smell* is a symptom in the source code that helps to identify a design problem. In this way, code smells allow developers to detect fragments of code that should be re-structured, in order to improve the quality of the system. A technique commonly used to fix code smells is *refactoring* (Fowler 1999; Kim et al. 2012).

Different semi-automated tools have been proposed (Moha et al. 2010; Lanza and Marinescu 2006) for identifying code smells in a system. However, a major limitation of existing tools is that they usually find numerous code smells. This is a challenging problem for the developer, for a number of reasons. First, she can get overwhelmed by the amount of information to be analyzed. Second, the efforts needed to fix all the smells usually exceeds the budget that the developer has available for refactoring. Third, in practice, not all smells are equally important for the goals of the system or its health (Demeyer et al. 2003). For example, some long classes, such as those that implement a parser, are not necessarily a design problem. Therefore, the developer has to manually peruse the list of smells and select a set of smells that will be fixed. In this context, the provision of tool support for assisting the developer to quickly identify high-priority code smells becomes essential.

In this work, we propose a semi-automated approach called smart identification of refactoring opportunities (*SpIRIT*) that prioritizes the code smells of a system according to their criticality. We define the critical problems as those smells that compromise the architecture of the system. In particular, we use modifiability scenarios (Clements and Kazman 2003) to capture goals (or desired properties) of the system with respect to evolution and architecture design. A scenario specifies a type of change that the system must accommodate. For example, a scenario can specify changes to a GUI feature wanted by a customer, which might affect several components (if the feature is not properly encapsulated). Normally, developers seek to confine the effects of changes specified by scenarios to narrow system areas in order to avoid the change propagation across the system. From this perspective, code smells are obstacles to satisfy the modifiability scenarios of the system. Modifiability problems can also be spotted by analyzing change patterns across system versions (Gîrba et al. 2004).

Given an object-oriented system with a number of code smells, *SpIRIT* assists the developer in the prioritization of the smells. The identification of the smells relies on existing catalogs (Fowler 1999; Lanza and Marinescu 2006). The novel aspect of our approach is that the prioritization of smells is based on assessing their relationships with modifiability issues. Our assessment of a code smell instance is determined by three factors: (i) the stability of the components that participate in a smell, (ii) the impact of a smell on modifiability scenarios, and (iii) the relevance of the kind of code smell. The relevance is a subjective value that a developer can assign to each kind of smell to indicate how harmful she considers it. This value might vary from developer to developer, or might also be system-specific (Mkaouer et al. 2014).

The contribution of this article is twofold. First, we develop a prioritization strategy for code smells that combines different criteria, which account for code-level, evolution and design-level information. Second, we propose a novel scenario-based criterion (Kazman et al. 1996) to drive the prioritization, incorporating design knowledge about modifiability.

We have evaluated our approach by means of two case-studies. In these studies, *SpIRIT* was applied to Java applications of different sizes. We compared the rankings of smells generated by *SpIRIT* with the smells ranked by expert developers. The results show that *SpIRIT* ranks first the most critical smells.

The rest of this article is structured as follows. Section 2 discusses the main problems of fixing code smells. Section 3 describes the *SpIRIT* approach. Section 4 presents the case-studies and their main results. Section 5 discusses related work. Finally, Sect. 6 presents the conclusions and outlines future work.

2 Improving design with code smells detection

Code smells are useful to identify structural problems of a system that relate to modifiability problems. In this way, a smell acts as an anti-pattern indicating code that should be improved (Fowler 1999). Each smell can affect several components (e.g. packages, classes, methods) of a system. Some of the symptoms used by code smells include: duplicated code, very large methods or classes, long lists of parameters or violations in the encapsulation of a class, among others. A popular catalog of code smells is the one proposed by Fowler (Fowler 1999). Usually, in the catalogs, for each smell a refactoring (or a group of them) is proposed to solve the problem. For example, the smell God Class identifies the situation in which a class centralizes the intelligence of the system (or a subsystem). In this case, the suggested refactoring is to extract groups of related methods into new classes by using the Move Method and Move Field refactorings (Fowler 1999). To help developers to find smells in systems, several tools such as PMD,[1] FindBugs[2] and iPlasma[3] are available.

A problem of existing tools is that they usually produce a large number of smells. For example, after analyzing 9 kinds of code smells in SweetHome3D,[4] a 84K LOC open source application, 787 smells were found (Table 1). Refactoring of these smells would be ideal but also time-consuming. In these situations, the developer might end up overwhelmed by the analysis of all the smells. Furthermore, the refactorization of some smells might not be urgent. For example, the refactoring of code smells in a class with no change since its initial implementation (and not expected to be modified in the future) may have low priority when compared to a code smell in a class that received modifications in the last 10 most significant revisions. That is, fixing some code smells

[1] http://pmd.sourceforge.net/.

[2] http://findbugs.sourceforge.net/.

[3] http://loose.upt.ro/reengineering/research/iplasma.

[4] http://www.sweethome3d.com.

Table 1 Code smells found in SweetHome3D

Code smell	# instances
Brain class	6
Brain method	127
Data class	1
Dispersed coupling	210
Feature envy	114
God class	38
Intensive coupling	89
Refuse parent bequest	12
Shotgun surgery	190
Total	787

can be more urgent than fixing others. Regarding the priority of the components to be refactored, some researchers have suggested that those components whose code suffered many changes in the past, are more likely to be modified in the future than those that did not changed (Gîrba et al. 2004; Mens and Demeyer 2001). For example, in SweetHome3D 85 % of the detected smells were modified only in one or two versions of the last 25 versions. For this reason, the refactoring effort should probably be focused on the remaining 15 % of code smells.

Moreover, we argue that the impact of the code smells on key modifiability scenarios of the system should be taken into account, in order to determine the priority of fixing a given smell. That is, if a given smell is touching a code area that is sensitive to one or more key modifiability scenarios, the developer should pay close attention to fixing that smell, in order to improve the satisfaction of the scenarios. Conversely, less attention should be paid to smells that do not directly affect key scenarios. For example, the God Class *HomeComponent3D* of SweetHome3D is directly involved in the realization of a scenario that allows developers to change the 3D visualization engine of the application. The visualization engine is critical in the architecture of SweetHome3D, because it involves one of its main features. By capturing the scenario above, it is meant that the engine should be easy to change, or that the effects of the change should be as localized as possible in the design. Therefore, it is important to fix the involved God Class, as it can negatively affect the satisfaction of the scenario.

We think that the developer should not only be assisted in the detection of code smells, as several tools currently do, but she should be assisted in the prioritization of the detected smells as well.

3 SpIRIT approach

We propose a semi-automated approach that helps developers to achieve the refactoring of an object-oriented system by focusing on the most critical code smells of the system. We call this approach *SpIRIT*: Smart Identification of Refactoring opportunITies.

We envision the use of the *SpIRIT* approach (Fig. 1) in the following situation. Let us assume a developer that is working in a project within an iterative and incremental development process. The developer only has a couple of hours per week to refactor the

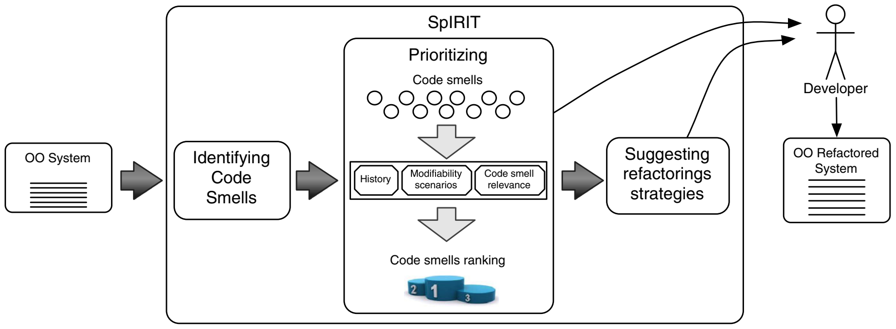


Fig. 1 The *SpIRIT* approach

system because she has to spend most of the time developing user-oriented features. Given a large list of smells to be refactored, the developer will have to pick the smells that are top-priority considering different criteria: the relation of the smells with the architecture, the importance of the kind of smell; and the analysis of how likely is that the source code related to the smell will be modified in future versions. We argue that the use of multiple criteria helps to examine the smells from different perspectives and, in this way, to discover if the smell is a critical problem. In this context, the developer will use *SpIRIT* to analyze components of the system (such as packages, classes or methods) that have potential modifiability problems. Then, the smells will be ranked by *SpIRIT* based on their importance, which is determined by the aforementioned criteria. Once the developer chooses a smell to be fixed, *SpIRIT* is expected to suggest different refactoring alternatives for it. However, the assistance with refactorings is out of the scope of this article.

The proposed approach is implemented as a tool.[5] *SpIRIT* is built using Moose,[6] a platform for software analysis. To analyze a system in *SpIRIT*, the system must be loaded using a MSE file. MSE is a generic file format, similar to XML, used by Moose to describe models of systems. A MSE file saves all the information related to a system such as packages, classes, methods, attributes as well as the relationships between them (e.g. invocations and inheritance). There are several applications available to generate MSE files from source code, such as VerveineJ[7] and inFamix.[8]

In the next sections, we provide details of the techniques used by the approach.

3.1 Identifying code smells

The *SpIRIT* approach begins by identifying the code smells of an application. Currently, *SpIRIT* supports the identification of 10 smells (Table 2) following the detection strategies presented in the catalog of Lanza and Marinescu (Lanza and Marinescu

[5] The latest version of *SpIRIT* is available from http://sites.google.com/site/santiagoavidal/projects/spirit.

[6] http://moosetechnology.org/.

[7] https://gforge.inria.fr/projects/verveinej/.

[8] http://www.intooitus.com/products/infamix.

Table 2 Code smells supported by *SpIRIT*

Code smell	Short description
Brain class	Complex class that accumulates intelligence by brain methods
Brain method	Long and complex method that centralizes the intelligence of a class
Data class	Class that contains data but not behavior related to the data
Disperse coupling	Method that calls one or few methods of several classes
Feature envy	Method that calls more methods of single external class that their own
God class	Long and complex class that centralizes the intelligence of the system
Intensive coupling	Method that calls several methods that are implemented in one or few classes
Refused parent bequest	Subclass that does not use the protected methods of its superclass
Shotgun surgery	Method called by many methods that are implemented in different classes
Tradition breaker	Subclass that does not specialize the superclass

2006). In this detection strategy, each smell is expressed as a rule combining different metrics, which have to reach predetermined thresholds. For example, the rule to identify a God Class combines three metrics: weighted method count (WMC) to measure the complexity of the class, access to foreign data (ATFD) to measure the coupling with external, and tight class cohesion (TCC) to measure the internal cohesion of the class. In this way, a God Class is determined by the rule:

$$GodClass = (WMC > VERY\ HIGH)\ and\ (ATFD > FEW)\ and\ (TCC < LOW)$$

The threshold values, such as, *FEW*, *LOW* or *VERY HIGH* are also the ones proposed by Lanza and Marinescu. Although these detection strategies are predefined in the *SpIRIT* tool, they are not per se a part of our approach, they are just a pluggable module in the tool.

Once the model of a system is generated and loaded into *SpIRIT* through a MSE file, the tool automatically detects the possible code smells. For each smell, *SpIRIT* shows the elements that compose the smell. We distinguish two kinds of constitutive elements: the class or method in which the smell is mainly implemented (we call this class/method the *main* class/method of the code smell) and the affected components. For example, in the case of the God Class *HomeComponent3D* presented in Sect. 2, *SpIRIT* shows the main class (*HomeComponent3D*) and all the external methods invoked by the class and also the external methods that are invoking the class.

3.2 Prioritizing code smells

Once the smells are discovered they should be ranked according to their importance. We argue that a code smell is important if it compromises the architecture of the system. This aspect was analyzed by some works showing that by refactoring the smells related to architectural problems the degradation of the architecture could be stopped (Macia et al. 2012b, a; Arcoverde 2012). Additionally, since not all kinds of smells are equally relevant to the architecture, the kind of code smell should also be taken into account to determine the importance of a smell. Finally, since refactoring is generally more beneficial in changing environments (Tsantalis and Chatzigeorgiou 2011), the smells found in classes or packages that are more likely to change should be more important. We argue that the combination of these aspects should be used to determine a ranking of smells. Along this line, we apply the following three criteria:

(1) Stability of related components (SRC): this criterion checks if the component in which a smell is found has undergone many changes over the history of the application.
(2) Relevance of a code smell (RCS): the developer can choose the kinds of smells that are more important using an ordinal scale (e.g. 1–5 where 5 means that the smell is very important).
(3) Related modifiability scenarios (RMS): this criterion helps to focus on those smells that affect modifiability scenarios of the system.

We chose these criteria because they take into account: the stability of the component in which the smell was found, the assessment that the developer makes of each kind of smell, and furthermore, they allow the developer to focus in those parts of the system that affects the quality of the architecture through the analysis of modifiability-specific requirements that should be satisfied. The importance of using different criteria is because they are complementary. For example, if only the relevance of the smell were used, smells that are not architecturally relevant or that have not changed since their implementation could be ranked first. Moreover, if only the criterion of history were used, unimportant smells for the developer and the architecture could be prioritized first. Similarly, if just the criterion of scenarios were used, while the prioritized smells would still be architecturally relevant, there could be smells that have not changed for a long time or kinds of smells that are not relevant for the developer. That is to say, by using the three criteria the smells are analyzed from multiple perspectives with the goal of determining the most critical smells.

In the following sections, each criteria is explained in detail as well as the calculation of the overall ranking.

3.2.1 Stability of related components (SRC)

The stability determines how often the class in which the code smell is mainly implemented (main class) was modified during the lifetime of the system. By determining the stability of the main class of the smell, we want to find if the smell is implemented in a part of the system that is usually modified. Our assumption is that the smells appearing in classes that changed often should be fixed first. For instance, a God Class that has not been modified since it was implemented might not represent a real problem (Demeyer et al. 2003).

Previous works have analyzed the history of systems to determine the classes that will change in the future based on those classes that change often (Gîrba et al. 2004; Wong et al. 2011; Tsantalis and Chatzigeorgiou 2011). We follow this same hypothesis to detect the unstable classes of the system.

To measure the stability of a class we use a Beta analysis (β). Beta is a financial indicator that measures the volatility of a given asset relative to the volatility of the market (Levy 2002). We have adapted Beta to use it in our context. An asset represents a class and the market represents the system in which the class is defined. That is, Beta allows us to know how important the changes are in the class with respect to the changes in the system. By important we mean a large addition or modification of methods. For example, a high value of Beta will indicate that when many methods are changed in the system (comparing to the total number methods of the system), many methods are also changed in the class (comparing to the total number of methods of the class). We define the Beta of a class as:

$$\beta_c = \frac{Cov(r_c, r_s)}{Var(r_s)}$$

where r_c measures the rate of return of the class, r_s measures the rate of return of the system, $Cov(r_c, r_s)$ is the covariance between the rates of return, and $Var(r_s)$ is the variance of the rate of return of the system. The rate of return of a class for version i is calculated based on the metric LENON which is one of the underlying metrics used by Gîrba et al. (2004) to predict class changes. This metric identifies the classes that experienced most changes in the last versions of the system. In LENON, the classes that most frequently changed are identified by weighting the change in the number of methods (NOM) of a class between two adjacent versions. More formally:

$$LENOM_{j..k}(C) = \sum_{i=j+1}^{k} \mid NOM_i(C) - NOM_{i-1}(C) \mid \times 2^{i-k}$$

where $1 \leq j < k \leq n$ being j the first version of the system analyzed, k the last version analyzed and n the total number of versions of the system.

We have found that our combination of Beta with LENON has in average a better predictive precision than using only LENON (Hurtado et al. 2013). The rate of return using LENON for version i of a class c is calculated as follows:

$$r_{c_i} = \mid NOM_i(C) - NOM_{i-1}(C) \mid \times 2^{i-n}$$

where n is the total number of versions of the system and $NOM_i(C)$ is the number of method of class C in version i. In this way, in our context, the rate of return is the gain or loss of number of method in a class or system over a specified period which are weighted benefiting the latest changes in history. For example, Table 3 shows the variation of the NOM in a class *Foo* for 5 different versions of a system. The rate of return of a class is calculated for each version taking into account the previous version. For instance, the rate of return for version 2 is calculated as $r_{Foo_2} = \mid 4 - 3 \mid \times 2^{2-5} = 0.125$. Note that while the change of NOM is the same in version 2 as in version 5, the r_c of version 5 is higher than the one of version 2 since 5 is the latest version.

The r_s is calculated in the same way that r_c but NOM is the sum of all the NOM of the system classes for a specific version. The $Var(r_s)$ and $Cov(r_c, r_s)$ are calculated using the rate of return values of all the versions.

Table 3 Rate of return example

	NOM v1	NOM v2	NOM v3	NOM v4	NOM v5
Foo	3	4	6	6	5
Rate of return	–	$1 \times 2^{-3} = 0.125$	$2 \times 2^{-2} = 0.5$	$0 \times 2^{-1} = 0$	1

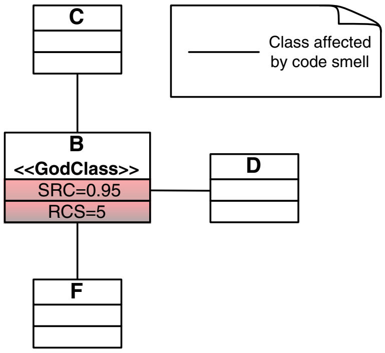


Fig. 2 Example of God Class impact

While *SpIRIT* calculates the rate of return using NOM, this metric can be easily configured by the user to other metric such as cyclomatic complexity or LOC, among others. A discussion of the use of LENOM with other metrics can be found in Vidal (2013).

Regarding the meanings of different values of β, if a class has a positive Beta it means that the NOM of the class tends to increase when the NOM of the system increases, and conversely, the class NOM tends to decrease when the system NOM decreases. In contrast, if a class has a negative Beta, it means that the class NOM generally moves opposite to the system NOM. A Beta value of zero indicates no correlation between the class and the system. Beta = 1 means that the class is exactly correlated with the system and if Beta>1 means that the class is correlated with the system but the class is more volatile than the system (i.e. the class changes often and at a greater rate than the system).

Finally, to compare the Beta values of classes among them, we normalize all the beta values to the highest value of Beta. In this way, the class with the highest Beta will have a normalized value of 1. For example, Fig. 2 shows that class *B* is a God Class and that it affects classes *C*, *D*, and *F*. The main class of the smell is class *B* and its β (or SRC) is 0.95. We here interpret that the class is unstable because it changed almost in each system version. For this reason, the criterion is highlighted in the figure.

The *SpIRIT* tool allows to load the history of a system as a set of versions, and each version is loaded by using its own MSE file. Once the history is loaded, *SpIRIT* calculates automatically the SRC value for each class affected by a code smell.

3.2.2 Relevance of a code smell (RCS)

This second criterion specifies how relevant a kind of smell is for the developer. *SpIRIT* allows the developer to choose a [1..5] ordinal scale for each kind of smell. In this context, 1 means that the code smell is not relevant for the system and 5 means that it is very relevant. In the example of Fig. 2, the RCS value chosen for God classes is the maximum: 5.

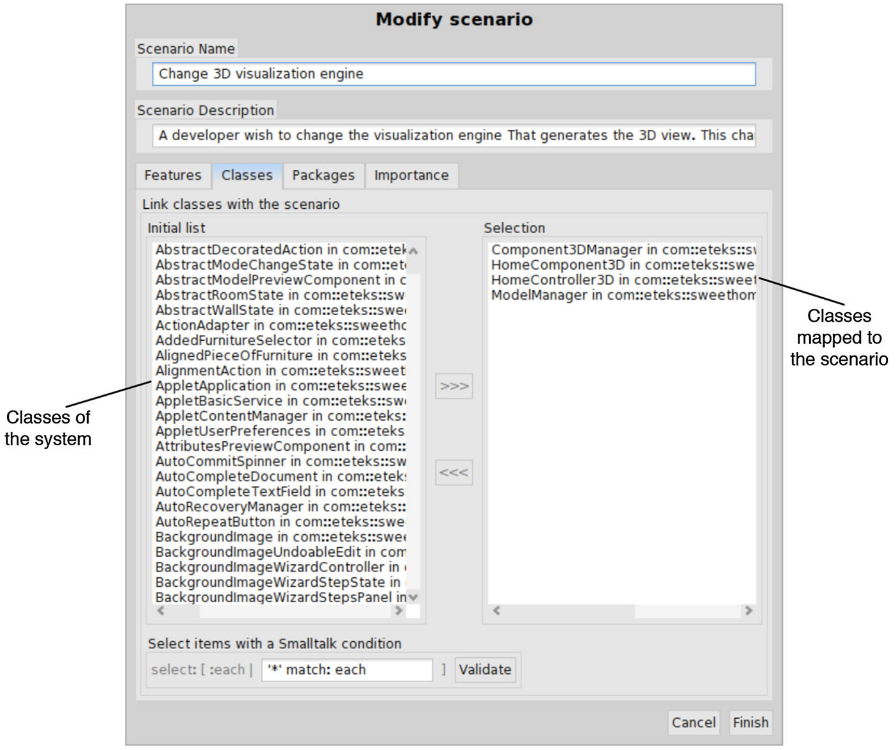
This criterion allows *SpIRIT* to adapt the recommendation of code smells to the preferences of the developer. By selecting the relevance of a kind of code smell, the developer can select the smells that she believes are the most important to the system or the kind of smells she is most familiar with. Additionally, the developer can indirectly select which problems to deal with such as coupling, cohesion, or complexity. For example, if the developer wants to improve the coupling of the system, she would select Intensive Coupling, Dispersed Coupling or Shotgun Surgery (Fowler 1999) as most relevant by giving them values close to 5. Conversely, if the developer wants to improve the cohesion, she would select God Class, Brain Method, or Intensive Coupling as most relevant. Notice that some smells can influence positively or negatively more than one aspect of design. For example, the refactoring of an Intensive Coupling smell can reduce the coupling between classes but it can raise the complexity of the provider class.

3.2.3 Related modifiability scenarios (RMS)

The third criterion analyzes the relation of smells with modifiability scenarios. By using these scenarios, *SpIRIT* includes architectural information in the prioritization of smells. We use modifiability scenarios because they can express the main goals and constraints of the evolution of a system using natural language (Ozkaya et al. 2010). A scenario briefly describes some anticipated or desired use of a system (Kazman et al. 1996). For example, a modifiability scenario that describes the change of a 3D visualization engine could be as follow: "A developer wish to change the visualization engine that generates the 3D view. This change will be made to the code at design time " (Fig. 3). In order to define the scenarios, it is desirable to have a software architecture model of the system, so that the architects and developers can refer their scenarios to that design model. In practice, architects generally know (e.g., via interactions with the stakeholders, or just by their own experience) the parts of the system that need to be modifiable. This perspective assumes that the architecture should be designed in such way it somehow addresses the constraints imposed by each scenario. Although scenarios clearly involve architectural aspects, they are also dependent on code-level aspects of the architecture (Woods 2012). This is the main reason for us to link scenarios to code smells. Along this line, developers can normally identify the system features implied by each scenario, and map them to the detailed design of the system (e.g., packages, classes, or methods).

An interesting benefit of using modifiability scenarios is that this information is available since early stages of development. That is, developer can describe the changes that the system must support since the architecture starts to be materialized. For this reason, scenarios complement the analysis of historical changes, which is more useful in late development stages.

Each system scenario can be mapped to one or more components or features of the system that make it possible to realize the scenario. A feature is useful to define a specific requirement that is implemented by a group of classes or packages. For



Modify scenario

Scenario Name

Change 3D visualization engine

Scenario Description

A developer wish to change the visualization engine That generates the 3D view. This cha

Features | Classes | Packages | Importance

Link classes with the scenario

Initial list

AbstractDecoratedAction in com::etek
AbstractModeChangeState in com::ete
AbstractModelPreviewComponent in c
AbstractRoomState in com::eteks::sw
AbstractWallState in com::eteks::swe
ActionAdapter in com::eteks::sweethc
AddedFurnitureSelector in com::eteks
AlignedPieceOfFurniture in com::eteks
AlignmentAction in com::eteks::sweet
AppletApplication in com::eteks::swee
AppletBasicService in com::eteks::sw
AppletContentManager in com::eteks:
AppletUserPreferences in com::eteks
AttributesPreviewComponent in com::
AutoCommitSpinner in com::eteks::sw
AutoCompleteDocument in com::etek
AutoCompleteTextField in com::eteks
AutoRecoveryManager in com::eteks::
AutoRepeatButton in com::eteks::swe
BackgroundImage in com::eteks::swe
BackgroundImageUndoableEdit in com
BackgroundImageWizardController in
BackgroundImageWizardStepState in
BackgroundImageWizardStepsPanel in

>>>
&lt;&lt;&lt;

Selection

Component3DManager in com::eteks::sv
HomeComponent3D in com::eteks::swe
HomeController3D in com::eteks::sweet
ModelManager in com::eteks::sweethom

Select items with a Smalltalk condition

select: [:each | '*' match: each] Validate

Cancel Finish

Fig. 3 Scenario wizard in *SpIRIT* tool

each scenario, the developer should specify the features, packages and/or classes that compose it. For example, scenario I in Fig. 4 is mapped to classes *A*, *B*, and *C*. Also, to distinguish the importance of each scenario, the developer could select from a ordinal scale from 0 to 1 the importance of each scenario, being 1 the most important one. In this example, scenario I has an importance of 0.6.

The *SpIRIT* tool presents a simple interface to load scenarios (Fig. 3). Specifically, the developer must define the name of the scenario and provide a brief description of it. Additionally, the developer must select the classes, packages, and/or features of the system that compose the scenario by choosing from different lists given by *SpIRIT* (the features can be defined by using other wizard provided by the tool). For example, Fig. 3 shows the definition of a scenario called "Change 3D visualization engine". The left part of the wizard lists all the classes of the system under analysis. The right part shows the classes that belong to the scenario.

A scenario is mapped to certain components that are key for fulfilling the scenario. For this reason, our intuition behind the use of modifiability scenarios is that fixing first the smells whose components compose the scenarios will make easier the satisfaction of the scenarios. Thus, if the modifiability scenarios are satisfied, the impact of future changes should be narrow. Along this line, using related modifiability scenarios (RMS) we perform a change impact analysis of the smells vis-a-vis with the scenarios. That is, we determine which classes affected by a smell are also mapped by a scenario.

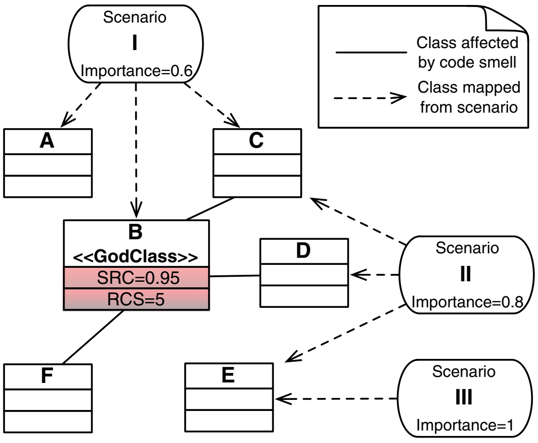
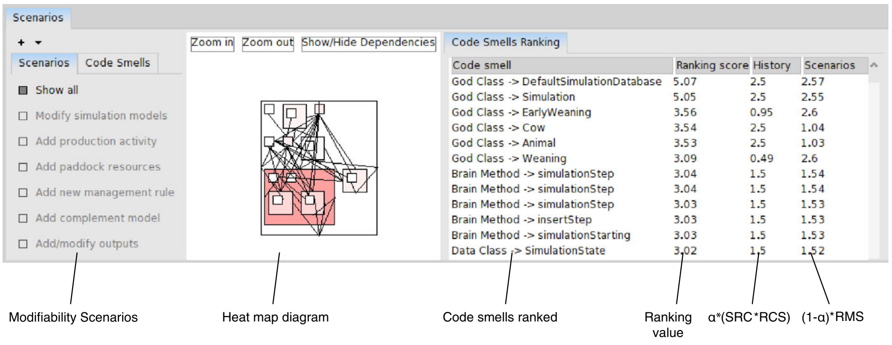


Fig. 4 Example of scenario



Code smell	Ranking score	History	Scenarios
God Class -> DefaultSimulationDatabase	5.07	2.5	2.57
God Class -> Simulation	5.05	2.5	2.55
God Class -> EarlyWeaning	3.56	0.95	2.6
God Class -> Cow	3.54	2.5	1.04
God Class -> Animal	3.53	2.5	1.03
God Class -> Weaning	3.09	0.49	2.6
Brain Method -> simulationStep	3.04	1.5	1.54
Brain Method -> simulationStep	3.04	1.5	1.54
Brain Method -> simulationStep	3.03	1.5	1.53
Brain Method -> insertStep	3.03	1.5	1.53
Brain Method -> simulationStarting	3.03	1.5	1.53
Data Class -> SimulationState	3.02	1.5	1.52

Fig. 5 Ranking of code smells with $\alpha = 0.5$ in the *SpIRIT* tool

The classes affected by a smell are determined for each kind of code smell taking into account the classes that should be refactored to fix the smell (the worst case is supposed). For example, the affected classes for a God Class are the main class in which the scenario is implemented and also the classes that invoke and are invoked by the God Class.

SpIRIT shows this kind of change impact analysis by means of the so-called heat maps (D'Ambros and Lanza 2009). In Fig. 5, *SpIRIT* shows a heat map of the distribution of the affected packages by the smells of the system. The packages in color are the ones in which the largest amount of classes affected by the smells of the system under analysis were found. By means of this kind of visual inspection of the scenarios, *SpIRIT* helps developers to focus on the change impact that the refactoring of a smell (or a group of them) will have in the scenarios.

The RMS value for a code smell is computed on the basis of the number of components that belong to the scenarios that are affected by the smell. First, we sum up the RCS value multiplied by the importance of the scenario that includes the affected class if the class in which the smell is implemented is mapped to at least one scenario. Then, for each class affected by the smell that is mapped to at least one scenario, we sum the RCS value multiplied by the importance of the scenario and this value is divided by the number of classes that are mapped by at least one scenario. In those cases in which more than one scenario maps to a class, the scenario with the highest importance is used. More formally:

$$RMS = RCS \times importanceScenarios + \frac{\sum RCS \times importanceScenarios}{allClassesAffectedByScenarios}$$

The intuition behind this calculation is to give more importance to those code smells whose main classes and affected classes are both directly involved in scenarios.

For example, consider the situation presented in Fig. 4. Three scenarios are defined in the system. Scenario I is mapped to classes *A*, *B*, and *C* and has an importance of 0.6. Scenario II is mapped to classes *C*, *D*, and *E* and has an importance of 0.8. Finally, scenario III is mapped to class *E* and has an importance of 1. To calculate the RMS value, the main class of the smell is analyzed. Since class B is mapped by scenario I, the importance of scenario I (0.6) is multiplied by the RCS of the God Class (5). Then, the classes affected by the smell must be analyzed. The class C is mapped by scenarios I and II. Since scenario II has a higher importance than I, the importance of the scenario II is used and 5×0.8 is summed. The class D is only mapped by the scenario II thus 5×0.8 is summed again. Since no scenario defines class *F*, it is not used in the RMS calculation. The resulting RMS for this example is:

$$RMS = 5 \times 0.6 + \frac{5 \times 0.8 + 5 \times 0.8}{5} = 4.6$$

Note that in order to draw conclusions from this RMS value, such as whether it is a high or low value, it would be necessary to compare it with the RMS values of other smells. For instance, if there is a code smell in class C (Fig. 4) whose RCS is 4 but it affects classes A, D, and E, its RMS value will be higher than the one presented before:

$$RMS = 4 \times 0.8 + \frac{4 \times 0.6 + 4 \times 0.8 + 4 \times 0.8}{5} = 4.96$$

3.2.4 Ranking calculation

The ranking is calculated by aggregating SRC, RCS, and RMS. Specifically, a ranking value is determined for each code smell as follows:

$$Ranking = \alpha \times (SRC \times RCS) + (1 - \alpha) \times RMS$$

where $0 \leq \alpha \leq 1$. As it is shown, the values of SRC and RMS are increased according to the value of RCS (note that RCS is also used to calculated RMS). Then the increased values are combined to create the ranking. The first part of the equation, the one that involves SRC, determines the severity of the smell taking into account the stability of the component in which the smell is defined. Moreover, the second part, the one involving the use of RMS, determines the impact of the smell from the perspective of scenarios. The α value allows the developer to weight the contribution of a particular part of the equation to the final ranking. For instance, using the example of Fig. 4 and $\alpha = 0.5$ the ranking is calculated as:

$$Ranking = 0.5 \times (0.95 \times 5) + 0.5 \times 4.6 = 4.675$$

To create the ranking, *SpIRIT* allows the developer to use all the defined scenarios or only a subset of them. Figure 5 shows a snapshot of a ranking generated by the *SpIRIT* tool. On the left of the figure, the modifiability scenarios are listed. On the right, the resulting ranking is shown. The ranking has four columns: (1) the kind of code smell and the name of the system element in which the smell is mainly implemented, (2) the ranking value for the smell, (3) the weight of the history in the ranking value [i.e. $\alpha \times (SRC \times RCS)$], and (4) the weight of the scenarios in the ranking value [i.e. $(1 - \alpha) \times RMS$].

4 Case-studies

In order to evaluate our approach, we conducted two empirical studies, guided by the following questions:

(1) Does *SpIRIT* ranks first the most critical code smells of a system?
(2) Are the smells ranked first by *SpIRIT* relevant to the developer?
(3) What value (or values) of α ranks first the most critical smells?

Questions 1 and 3 are answered in the study presented in Sect. 4.1 in which a Java application of 8.5K lines of code with 47 smells is analyzed. In this case-study, the full ranks of smells generated by *SpIRIT* with different values of α are compared against a rank created by an application expert. The goal is to empirically test how the smells of *SpIRIT* are ranked when compared to a baseline made by the application expert.

Question 3 is also analyzed in Sect. 4.2 along with question 2. In this study, a middle-size Java application of 38K lines of code was analyzed. The last version of this application reported a total of 523 smells. Since the analysis of each smell by an expert was impractical, we analyzed if the top-10 smells ranked by *SpIRIT*, with different values of α, were relevant to the expert.

The scope of both case-studies involves the identification and prioritization of smells whose refactoring could contribute positively to the evolution of the system. However, we do not consider the refactoring step of *SpIRIT* in which the smells are actually fixed. Also, in both case-studies *SpIRIT* is tested in smalls team settings where developers work collocated.

4.1 Case-study #1: subscribers DB application

The target application is a Java sub-system of a publishing house[9]. This application manages data related to the subscribers of its publications and it supports different

[9] For confidentiality reasons, we can not publish details about the company or the source code of the application.

queries on the data. Also, the application manages a printing service for the postal labels that go with the publications when they are periodically distributed to subscribers. The application has more than 15 releases. Its latest version has around 8.5K lines of code and 193 classes. The main subject in our study was the lead application developer, who has experience in software design and code smells. The analysis of the latest version of the application reported 47 smells.

Since the lead developer is requested to give explicit feedback on each smell, we chose an application with a low number of smells. An application with numerous smells would do this experiment impractical.

4.1.1 Hypotheses and operation

In this study we analyze how similar the ranking given by *SpIRIT* is regarding the ranking given by the lead developer of the application under analysis (the comparison is made position by position). Our hypothesis is that a strong correlation should exist between both rankings (i.e. they are very similar). We test this hypothesis by generating *SpIRIT* rankings with different values of α.

This study involved an interview with the application developer. During the interview the developer was asked to define the main modifiability scenarios of the application. The developer defined three scenarios that map to 16 classes of the application (8.3 % of the total number of classes). The developer spent 60 min to define the scenarios of the application. Table 4 describes the scenarios and the importance given by the developer.

Also, during the interview, the developer was asked to select the relevance of different kinds of code smells by using the ordinal scale of *SpIRIT*. The developer selected as the most important smells those kinds of smells whose main entity is a class (i.e. God Class, Brain Class and Data Class).

Finally, after defining the scenarios and smell relevances, the list of smells of the latest application version was presented to the developer and she was asked to rank all the smells according to their importance. During this process, the developer analyzed the source code of each smell to have a deeper understanding of the problem.

Table 4 Scenarios mapping

Scenario name	Short description	# of mapped classes	Importance
Add personal data	A developer wish to add new personal information to be stored in the db	12	1
Modify labels	A developer wish to change the information that is printed in the postal labels	2	0.4
Add search criterion	A developer wish to add a new kind of search to be accessible by the users	2	0.6

Table 5 Smells marked to be refactored by the developer

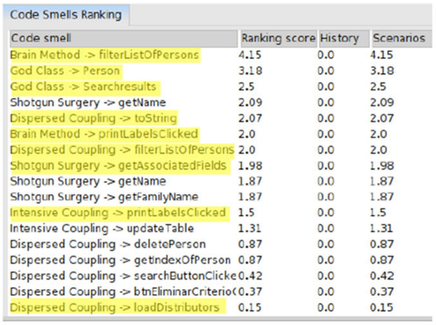
# Ranking	Kind	Main entity
1	*God class*	*SearchResults*
2	*Brain method*	*SearchResults.printLabelsClicked()*
3	*Brain method*	*SearchCriteriaForLabel.filterListOfPersons(List)*
4	*God class*	*Person*
5	*Intensive coupling*	*SearchResults.printLabelsClicked()*
6	*Shotgun surgery*	*Label.getAssociatedFields()*
7	Intensive coupling	DatabaseManager.updatePerson(Person)
7	Intensive coupling	AddPersonFrame.acceptButtonClicked()
8	*Dispersed coupling*	*SearchCriteriaForDonation.filterListOfPersons(List)*
8	*Dispersed coupling*	*SearchCriteriaForLabel.toString()*
9	Dispersed coupling	Person.compare(Person, Person)
9	Dispersed coupling	AddLabel.okButtonClicked()
9	Dispersed coupling	AddEvent.okButtonClicked()
9	Dispersed coupling	AddPersonToEvent.btnAddCurrentDateClicked()
9	Dispersed coupling	AddPersonFrame.loadDistributors()
9	Dispersed coupling	AddDonation.saveButtonClicked()
9	Dispersed coupling	EditLabelsForPerson.okButtonClicked()

4.1.2 Analysis and interpretation

While no indications were given to the developer on how to prioritize the smells, she decided to mainly rank them using the kind of code smell as the criterion. However, the developer also used other criteria such as the importance of the class in which the smell was implemented, the frequency of class modifications, and the relationship with scenarios. Also, she ranked the smells taking into account if she considered that the smell should be refactored in the short term (this information is shown in Table 5). The developer gave us several reasons for not refactoring a smell, namely: it was implemented in a class that is not usually modified, the problem is not really important (e.g. an intensive coupling that is coupled with an UI class), the smell is a false positive [e.g. a shotgun surgery method that gets the instance of a singleton (Gamma et al. 1995) class], among others. From a total of 47 smells, the developer indicated 17 smells to be refactored. These 17 smells were, logically, the first ones in the ranking (Table 5). In some ranking positions the developer indicated ties.

After analyzing the ranking we found that the developer ranked first many smells that are related to the modifiability scenarios she previously had defined. A total of 14 smells out of 47 are related to these scenarios. The developer ranked 8 of these 14 smells to be refactored. The smells related to modifiability scenarios are those highlighted in italic in Table 5.

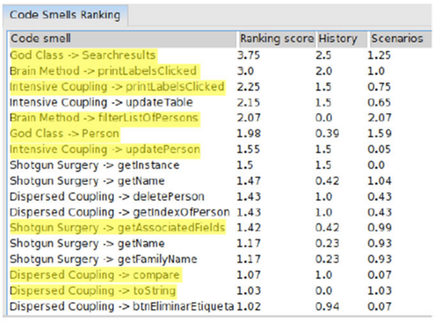
Once the interview was finished, the modifiability scenarios and the smell relevances defined by the developer where loaded into *SpIRIT* along with the history of the



Code Smells Ranking

Code smell	Ranking score	History	Scenarios
Brain Method -> filterListOfPersons	4.15	0.0	4.15
God Class -> Person	3.18	0.0	3.18
God Class -> Searchresults	2.5	0.0	2.5
Shotgun Surgery -> getName	2.09	0.0	2.09
Dispersed Coupling -> toString	2.07	0.0	2.07
Brain Method -> printLabelsClicked	2.0	0.0	2.0
Dispersed Coupling -> filterListOfPersons	2.0	0.0	2.0
Shotgun Surgery -> getAssociatedFields	1.98	0.0	1.98
Shotgun Surgery -> getName	1.87	0.0	1.87
Shotgun Surgery -> getFamilyName	1.87	0.0	1.87
Intensive Coupling -> printLabelsClicked	1.5	0.0	1.5
Intensive Coupling -> updateTable	1.31	0.0	1.31
Dispersed Coupling -> deletePerson	0.87	0.0	0.87
Dispersed Coupling -> getIndexOfPerson	0.87	0.0	0.87
Dispersed Coupling -> searchButtonClicke	0.42	0.0	0.42
Dispersed Coupling -> btnEliminarCriterio(	0.37	0.0	0.37
Dispersed Coupling -> loadDistributors	0.15	0.0	0.15

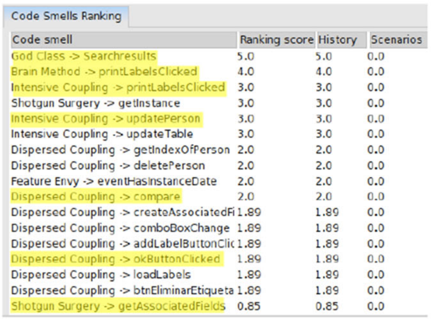
(a) $\alpha = 0$



Code Smells Ranking

Code smell	Ranking score	History	Scenarios
God Class -> Searchresults	3.75	2.5	1.25
Brain Method -> printLabelsClicked	3.0	2.0	1.0
Intensive Coupling -> printLabelsClicked	2.25	1.5	0.75
Intensive Coupling -> updateTable	2.15	1.5	0.65
Brain Method -> filterListOfPersons	2.07	0.0	2.07
God Class -> Person	1.98	0.39	1.59
Intensive Coupling -> updatePerson	1.55	1.5	0.05
Shotgun Surgery -> getInstance	1.5	1.5	0.0
Shotgun Surgery -> getName	1.47	0.42	1.04
Dispersed Coupling -> deletePerson	1.43	1.0	0.43
Dispersed Coupling -> getIndexOfPerson	1.43	1.0	0.43
Shotgun Surgery -> getAssociatedFields	1.42	0.42	0.99
Shotgun Surgery -> getName	1.17	0.23	0.93
Shotgun Surgery -> getFamilyName	1.17	0.23	0.93
Dispersed Coupling -> compare	1.07	1.0	0.07
Dispersed Coupling -> toString	1.03	0.0	1.03
Dispersed Coupling -> btnEliminarEtiqueta	1.02	0.94	0.07

(b) $\alpha = 0.5$



Code Smells Ranking

Code smell	Ranking score	History	Scenarios
God Class -> Searchresults	5.0	5.0	0.0
Brain Method -> printLabelsClicked	4.0	4.0	0.0
Intensive Coupling -> printLabelsClicked	3.0	3.0	0.0
Shotgun Surgery -> getInstance	3.0	3.0	0.0
Intensive Coupling -> updatePerson	3.0	3.0	0.0
Intensive Coupling -> updateTable	3.0	3.0	0.0
Dispersed Coupling -> getIndexOfPerson	2.0	2.0	0.0
Dispersed Coupling -> deletePerson	2.0	2.0	0.0
Feature Envy -> eventHasInstanceDate	2.0	2.0	0.0
Dispersed Coupling -> compare	2.0	2.0	0.0
Dispersed Coupling -> createAssociatedFi	1.89	1.89	0.0
Dispersed Coupling -> comboBoxChange	1.89	1.89	0.0
Dispersed Coupling -> addLabelButtonClic	1.89	1.89	0.0
Dispersed Coupling -> okButtonClicked	1.89	1.89	0.0
Dispersed Coupling -> loadLabels	1.89	1.89	0.0
Dispersed Coupling -> btnEliminarEtiqueta	1.89	1.89	0.0
Shotgun Surgery -> getAssociatedFields	0.85	0.85	0.0

(c) $\alpha = 1$

Fig. 6 Comparison of top ranked smells **a** $\alpha = 0$ **b** $\alpha = 0.5$ **c** $\alpha = 1$

application (15 releases). We were interested in the influences of the code history and the modifiability scenarios on the ranking. As discussed in Sect. 3, this influence is weighted by parameter α. For this reason, after loading the scenarios and relevances, the *SpIRIT* ranking was obtained using different configurations of α in the range $0 \leq \alpha \leq 1$.

For example, Fig. 6 shows the first 17 positions of the rankings of *SpIRIT* for three reference values of α: no influence of the system history ($\alpha = 0$), equal influence of both history and scenarios ($\alpha = 0.5$), no influence of the scenarios ($\alpha = 1$). The smells highlighted are those smells that are also present in the top-17 smells suggested by the developer. Note that the rankings generated with $\alpha = 0$ and $\alpha = 0.5$ contain 9 out of the 17 most critical smells in their first 17 positions. Regarding the ranking of $\alpha = 1$, it contains 7 smells. As expected, the ranking generated with $\alpha = 0$ contains in the first positions the 8 smells related to scenarios marked to be refactored by the developer. However, these 8 smells are also top-ranked with $\alpha = 0.5$, meaning that most of these smells are not only influenced by scenarios but also by history (although in a smaller percentage).

Regarding the positions of the smells, Table 6 compares the positions suggested by the developer for the smells marked to be refactored with their counterparts after running *SpIRIT*. The cells highlighted in italic indicate the positions of the *SpIRIT*

Table 6 Comparison of positions of smells marked to be refactored

Code smell	Ranking			
	Developer	$\alpha = 0$	$\alpha = 0.5$	$\alpha = 1$
God class—SearchResults	1	*3*	*1*	*1*
Brain method—SearchResults. printLabelsClicked()	2	6.5	*2*	*2*
Brain method—SearchCriteriaForLabel. filterListOfPersons(List)	3	*1*	*5*	40
God class—Person	4	*2*	*6*	21
Intensive coupling—SearchResults. printLabelsClicked()	5	11	*3*	*4.5*
Shotgun surgery—Label.getAssociatedFields()	6	*8*	12	17.5
Intensive coupling—DatabaseManager. updatePerson(Person)	7	25	*7*	4.5
Intensive coupling—AddPersonFrame. acceptButtonClicked()	7	25	33.5	29
Dispersed coupling—SearchCriteriaForDonation. filterListOfPersons(List)	8	*6.5*	20.5	40
Dispersed coupling—SearchCriteriaForLabel. toString()	8	5	16	40
Dispersed coupling—Person.compare(Person, Person)	9	19.5	15	*8.5*
Dispersed coupling—AddLabel.okButtonClicked()	9	30	37	40
Dispersed coupling—AddEvent.okButtonClicked()	9	40.5	43	40
Dispersed coupling—AddPersonToEvent. btnAddCurrentDateClicked()	9	40.5	28	19.5
Dispersed coupling—AddPersonFrame. loadDistributors()	9	19.5	35	32
Dispersed coupling—AddDonation. saveButtonClicked()	9	40.5	43	25
Dispersed coupling—EditLabelsForPerson. okButtonClicked()	9	19.5	18	13.5

ranking that are less than two positions away from the positions proposed by the developer. For example, the God Class *SearchResults* ranked first by the developer was also in the first position in the *SpIRIT* ranking for $\alpha = 1$. Note that the ranking generated with $\alpha = 0.5$ is the most accurate when the ten first positions are compared. In this case, 6 of the first 7 smells are ranked less than two positions away from the positions proposed by the developer. This fact is important because it shows that the first smells ranked by *SpIRIT* are not only those indicated as the most critical ones by the developer but also that they are the ones marked to be refactored.

While the analysis above provides evidence that *SpIRIT* ranks in the first positions the majority of the critical smells, it is important to analyze how accurate these positions are. Given the ranking proposed by the developer, we need to determine how close *SpIRIT* comes to the proposed ranking. Since the developer's ranking preferences are known, we apply the Spearman's correlation coefficient (p) (Ricci et al. 2011). This coefficient measures the strength of association between the developer's and *SpIRIT* rankings. Since the developer's and *SpIRIT* rankings contain ties (i.e. more than one smell have the same ranking position) we use the following method to calculate the Spearman's correlation:

$$p = \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_i (x_i - \bar{x})^2 \sum_i (y_i - \bar{y})^2}}$$

where x_i and y_i are the ranking positions for the same code smell instance in the developer's and *SpIRIT* rankings respectively. Regarding $\bar{x}$ and $\bar{y}$, they are the means of the ranking values of each ranking. In the case of tied smells, their ranking value should be the average of the ranking position. For example, if the smells in positions 1, 2 and 3 of the *SpIRIT* ranking have the same ranking score, the ranking value assigned to each of them must be $\frac{1+2+3}{3} = 2$. The coefficient can take values between 1 and -1. If p=1, it indicates a perfect association between both rankings. If p=0, it indicates no correlation between the rankings. If p=-1 indicates a negative association between the rankings.

Figure 7 shows the variations of the p coefficient for different values of α. When all the smells ranked were analyzed, we found that the coefficient varied between $0.58 \leq p \leq 0.62$ for α values between 0 and 0.5. The highest value of p was found after generating the ranking with $\alpha = 0.2$. The Spearman's correlation value sharply decreased for a α value higher than 0.6. This means that the rankings proposed by *SpIRIT*, when the modifiability scenarios are taken (to some extent) into account, are very close to the ranking proposed by the developer. A similar behavior is observed when only the smells marked to be refactored are taken into account to calculate the correlation. In this case, the values of p are higher than those for the cases in which all the smells were considered. The highest values of p ranges $0.8 \leq p \leq 0.88$ for α values between 0.1 and 0.6. In this case, the highest value of p was found for $\alpha = 0.4$. Such high values of p, when only the smells marked to be refactored are taken into account, means that this subset of smells is ranked by *SpIRIT* in almost the same positions as the ones proposed by the developer.

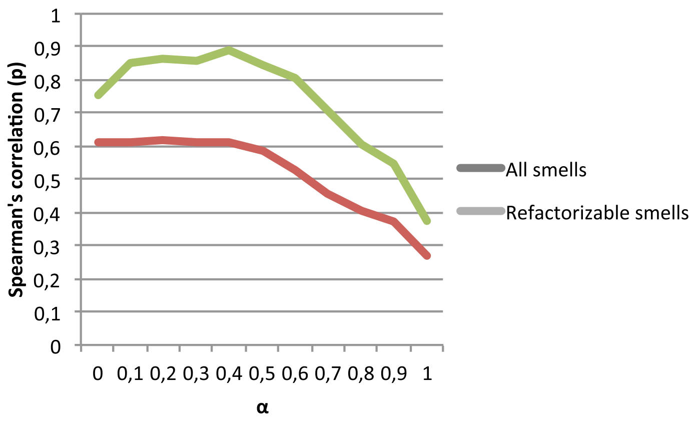


Fig. 7 Spearman's correlation results

Results Overall, we found that *SpIRIT* ranks first the most critical smells of an application. The analysis of all the smells versus the ones marked to be refactored by the developer shows that a combination of scenarios and history is appropriate to generate the *SpIRIT* ranking. This is because the highest values of p where found for values of α close to 0.5.

4.1.3 Threats to validity

Next, we analyze threats to four types of validity for this study.

Conclusion validity This threat concerns the statistical analysis of the results. In this case, the main concern is that the study was made over one single application with one developer. This could reduce the ability to draw correct results. For this reason, we think that further experiments with other applications are necessary in order to generalize our results.

Internal validity This threat concerns causes that can affect the independent variable of the experiment without the researcher's knowledge. The main threat in this case is that we did not have a "second opinion" to contrast the scenarios and ranking given by the developer. For example, other developers could have defined more scenarios or made different mappings leading to different results. Also, it is unknown if other developers could have prioritized first those smells related to modifiability scenarios. However, we argue that the developer prioritized first the smells related to scenarios because they are related with classes that are the main sources of problems. The person that defined the scenarios and the ranking was the application developer, who had a deep understanding of the application and of the kind of changes that could typically occur. For this reason, the internal validity is not considered to be critical. Anyway, the intervention of a second developer to contrast the definition of the ranking and the scenarios should be considered in future works to mitigate this threat.

Construct validity It is concerned with the design of the experiment and the behavior of the subjects. Our main concern if that the developer's ranking could have been influenced by the relevances of the smells and the description of the scenarios made at the beginning of the experiment. That is, the developer could have followed a different approach to rank the smells if the definition of the smell relevances and the scenarios have been done after creating the ranking. However, it was emphasized during the interview that the ranking did not necessarily depend on the relevances and scenarios.

External validity It is concerned with having a subject that is not representative of the population. The main treat is that the application analyzed was small with a reduced amount of code smells. However, the same analysis in larger applications is not always viable because a developer must manually analyze each smell and suggest its ranking. Also, another threat is that the experiment was conducted in a small team setting with all developers collocated. Although scenario-based techniques have been regarded as useful in the literature, the definition of a comprehensive set of scenarios can be an expensive process in industrial contexts, particularly because it requires both architects' expertise and stakeholders' commitment. This adoption barrier has been discussed in (Bashroush et al. 2004; Woods 2012). In our approach, the value of the modifiability scenarios stems from their ability to focus architects, stakeholders and developers on important system areas. In addition to the scenario definition (at the architectural level), a related threat has to do with relationships between the scenarios and the code. In this experiment, having a small team of developers working collocated helped them to reason and discuss about the system. Nonetheless, in larger or geographically-distributed teams, reasoning about the internal system details and their effects on code smells can be harder.

4.2 Case-study #2: beef-cattle farm simulator

We conducted an empirical study on a non-trivial Java application. The target application is a beef-cattle farm simulator (BCFS) (Mangudo et al. 2012; Marcos et al. 2011) developed by a local software factory, and currently being used by several agro-livestock companies. This application has been developed for 5 years so far following an iterative, agile process. BCFS has around 38K lines of Java code and 425 classes. The case-study involved the analysis of the latest version of this application. The main subject in our study was the lead architect of BCFS, who has vast experience in the design and implementation of this kind of systems. A first run of *SpIRIT* on BCFS reported 523 smells. Figure 8 shows the distribution of the different kinds of smells. While three kinds of smells represents the 70 % of the smells found, this does not mean that these types of smells are the most relevant ones. It is in this situation that the prioritization of *SpIRIT* becomes helpful.

4.2.1 Hypotheses and operation

SpIRIT is assumed to rank first the most important smells of the system. Our goal was to empirically evaluate this hypothesis, from the perspective of a domain and

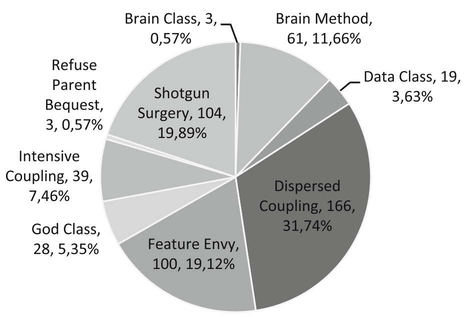


Fig. 8 Code smells of BCFS

development expert. Unlike case-study #1, it is not possible to ask the developer to rank all the possible smells of the system, mainly because the number of smells is very large. For this reason, in this case study we analyze the relevance for the developer of the *SpIRIT* top-ranked smells. As in case-study #1, three reference values of α were used (0, 0.5 and 1 respectively).

Hypotheses In this study, we consider that *SpIRIT* will rank first the most important smells if at least 7 smells of the first 10 ranked are judged as important by the developer (i.e. more than 60 % of the top-10 smells must be important). Along this line, we formulated the following null hypotheses:

- $H1_0$ number of important code smells $\leq 60\ \%$ of the first 10 smells ranked with a configuration of $\alpha = 0$.
- $H2_0$ number of important code smells $\leq 60\ \%$ of the first 10 smells ranked with a configuration of $\alpha = 0.5$.
- $H3_0$ number of important code smells $\leq 60\ \%$ of the first 10 smells ranked with a configuration of $\alpha = 1$.

Also, we formulated the following alternative hypotheses:

- $H1_1$ number of important code smells $> 60\ \%$ of the first 10 smells ranked with a configuration of $\alpha = 0$.
- $H2_1$ number of important code smells $> 60\ \%$ of the first 10 smells ranked with a configuration of $\alpha = 0.5$.
- $H3_1$ number of important code smells $> 60\ \%$ of the first 10 smells ranked with a configuration of $\alpha = 1$.

As for the variables selection in the study, the independent variables are those involved in the configuration of *SpIRIT*, namely: the parameter α, the relevance of each kind of smell, and the modifiability scenarios inputted into the tool. The main dependent variable in our analysis is the ranking proposed by the tool.

Operation The evaluation process involved two rounds of interviews with the lead developer of BCFS. While the developer had previous knowledge of code smells and modifiability scenarios, she was given an introduction to the topic. However, the developer was not aware of our hypotheses for the tool. In the first interview, we tried to capture the main structural problems of BCFS. To this end, the developer provided us with a list of the 10 classes that she believed were the most problematic ones for the current system version. Then, we asked the developer to define a set of modifiability scenarios that she considered as key for the current system goals. She assigned a priority to every scenario, mapping each one to a group of related classes. As a separate activity, she also defined the relevance of each kind of code smell, in the context of BCFS.

After this first interview, we fed the modifiability scenarios and the relevances of the smells into *SpIRIT*. Also, we loaded into *SpIRIT* the previous versions of the application to take its history into account. Based on the three α configurations above (0, 0.5, and 1), we executed the tool and obtained three corresponding rankings of smells. Then, we came back to the lead developer for a second interview. We presented the first 10 smells of each ranking to the developer in a random order, and asked her to grade the importance of the smells for BCFS using a five-level ordinal scale (not at all important, somewhat unimportant, neutral, somewhat important, and very important). In particular, for every smell of the rankings, he was required to answer the question: how important do you think code smell X is?

4.2.2 Analysis and interpretation

In this section the interviews with the developer are described and the results of them are analyzed.

Analysis of the First Interview During the first interview, the developer ranked the 10 classes that he believed were the most problematic ones of the application (Table 7). In addition, the developer defined 6 modifiability scenarios for BCFS. These scenarios mapped to 64 different classes, which represents the 15 % of the application classes. The developer spent 120 min to define the scenarios of the application.

Also, the developer defined the God Class and the Brain Class as the most relevant smells (Table 8). Note that the developer defined God Class and Brain Class as the most relevant smells even when both had very few occurrences in the total list of smells for BCFS (5.35 and 0.57 % respectively, as shown in Fig. 8). This supports the

Table 7 Developer's ranking of most problematic classes

Class
Farm
Animal
FeedIntake
GrowAnimals
FeedIntakeCalf
GrowCalf
MobExcelExportDecorator
FarmExcelExportDecorator
Cow
OrganizeMobs

Table 8 Relevance of code smells given by the developer

Code smell	Relevance
God class	5
Brain class	4
Brain method	3
Data class	3
Dispersed coupling	2
Feature envy	2
Intensive coupling	2
Shotgun surgery	2
Refused parent bequest	1
Tradition breaker	1

Table 9 Top-10 smells of the rankings generated by *SpIRIT*

$\alpha = 0$	$\alpha = 0.5$	$\alpha = 1$
God class—Reproduction	God class—DefaultSimulationDatabase	God class—DefaultSimulationDatabase
God class—Weaning	**God class—Simulation**	***God class—Cow***
God class—EarlyWeaning	**God class—EarlyWeaning**	God class—ReproDataSummary
God class—OrganizeMobs	***God class—Cow***	***God class—Animal***
God class—MoveMobs	***God class—Animal***	God class—SimulationFrontEnd
God class—DefaultSimulationDatabase	**God class—Weaning**	**God class—Simulation**
God class—Simulation	***Brain method—GrowAnimals. SimulationStep(Simulation)***	**God class—FeedlotType**
God class—GrowPasture	***Brain method—FeedIntake. simulationStep(Simulation)***	**God class—Mob**
God class—Economico	**Brain method—ProductiveDataRecopilation. simulationStep(Simulation)**	God class MobCowCalfFeatures
Brain class—Hibrido	**Brain method—DefaultSimulationDatabase. insertStep(Simulation)**	God class—ProductiveSummary

intuition that some smells are more important than others for the developer, regardless of their density in the code.

With these values as input, *SpIRIT* generated three rankings. Table 9 shows the top-10 smells of each ranking. All the rankings were primarily composed by God classes, as follows::

- $\alpha = 0$ 9 instances of God Class and 1 instance of Brain Class.
- $\alpha = 0.5$ 6 instances of God Class and 4 instances of Brain Class.
- $\alpha = 1$ 10 instances of God Class.

These results can be explained by the fact that the God Class was defined as the most relevant smell, and the main classes of the ranked God classes have high SRC (relevance of a code Smell) values or they are mapped by scenarios.

Table 10 Overlapping between *SpIRIT* rankings

α	0 (%)	0.5 (%)	1 (%)
0	–	40	20
0.5	40	–	40
1	20	40	–

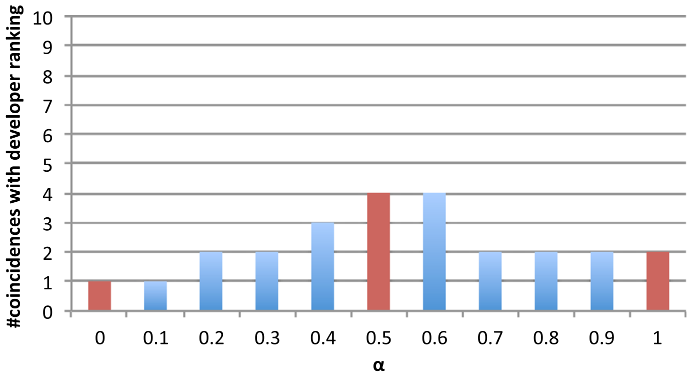


Fig. 9 Coincidences between classes of the *SpIRIT* ranking and the developer's ranking

The overlapping between the rankings generated with the three values of α was relatively low (considering just the first 10 smells of each ranking). Only two smell instances were present in the three rankings (although in different positions). This means that the values chosen for α are sufficiently different to conduct the experiment. Table 10 shows the percentage of overlapping between the top-10 smells of the three rankings. For example, only four smells ranked in the top-10 of the ranking generated with $\alpha = 0$ are present in the top-10 generated with $\alpha = 0.5$.

We also compared the main classes of the first 10 smells of the rankings generated by *SpIRIT* against the 10 classes provided by the developer. Remember that the developer defined her ranking before knowing the lists of smells outputted by *SpIRIT*. In the *SpIRIT* ranking with $\alpha = 0.5$, we found that the main classes of the first 4 smells (of the list of 10 smells) matched some of the classes in the developer' ranking (Fig. 9). This matching shows that 40 % of the smells in the *SpIRIT* ranking are implemented in problematic classes from the list given by the developer. If we analyze the values for $\alpha = 0$ and $\alpha = 1$, the matching classes are lower than those for $\alpha = 0.5$. Note that, although a 40 % of matching is relatively low in terms of predictive power of *SpIRIT*, it is important to know if the remaining smells (those whose main class is not in the developer's ranking) are actually problems that the developer might have overlooked. This aspect is analyzed during the second interview. Table 9 highlights in italic those smells of the *SpIRIT* ranking whose main class was ranked by the developer as a source of problems. For example, the God Class *Cow* is highlighted in italic in Table 9 because this class was identified as a source of problems, as shown in Table 7.

We analyzed the number of main classes (of the smells) that received mappings from at least one scenario, as a sanity check about the influence of scenarios according to the value of α. This analysis was made for the first 10 smells ranked by *SpIRIT*. Interestingly, we found that all main classes of the *SpIRIT* ranking using $0 \leq \alpha \leq 0.5$

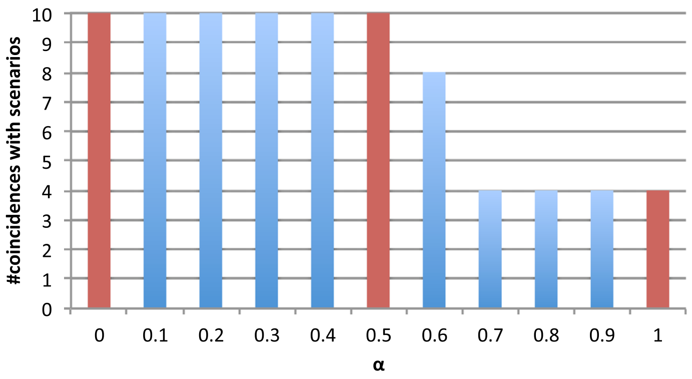


Fig. 10 Coincidences between main classes of the code smells ranked and classes mapped by scenarios

Table 11 Developer's agreement with the ranked smells

α	Not at all important	Somewhat unimportant	Neutral	Somewhat important	Very important
0	–	–	1	2	7
0.5	–	–	1	2	7
1	–	4	1	–	5

had mappings from one or more scenarios (Fig. 10). This level of matching indicates that when the scenarios are used for generating the ranking, the top positions of the ranking are smells strongly related to those scenarios. This finding is worth noticing because the scenarios only covered 15 % of the system classes. Regarding the main classes of all the code smells, only 49 % of them are mapped by some scenario.

Analysis of the second interview To test the hypotheses, we used a binomial test because we consider each smell as an independent trial and evaluated a finite number of them. The test inputs were the developer's answers to the statement "How important do you think each code smell is?". These answers are summarized in Table 11. The table shows the agreement of the developer with the importance of the top-10 smells of the rankings generated by *SpIRIT* with different values of α.

We consider that a smell is important if the developer's answer was *Somewhat important* or *Very important*. That is, for $\alpha = 0$ and $\alpha = 0.5$ we have 9 important smells. Analogously, for $\alpha = 1$ we have 5 important smells. Table 9 shows in bold those smells of the *SpIRIT* ranking that were judged as important by the developer. Note the differences (and overlaps) with the smells considered as related to problematic classes by the developer during the first interview (italic cells in the table). The differences essentially say that *SpIRIT* uncovered problems that went undetected to the developer (in her first analysis). Thus, with this additional information, the 40 % of problem matching observed during the first interview for the ranking generated with $\alpha = 0.5$ now goes up to a 90 % of matching.

The facts above, however, need to be checked from a statistical perspective. Let X be the number of important smells and $k = 0.6$ the probability that we want to test in the null hypotheses (H_0). Then, the binomial test statistic is $X \sim B(10, 0.6)$. After testing the data for $\alpha = 0$ and $\alpha = 0.5$ with a significance level of 5 %, we obtained a p-value = 0.04636. Since p-value <0.05, it is possible to reject the null hypotheses with a one-tailed test. This means that we accept the alternatives hypotheses $H1_1$ and $H2_1$, which state that the number of important code smells (ranked in the first 10 positions) is greater or equal than 7 when $\alpha = 0$ or $\alpha = 0.5$ is used.

Similarly, we tested the data for $\alpha = 1$ ($H3_0$) with a significance level of 5 % . In this case, we obtained a p-value = 0.8338. Since p-value $\nless$0.05, it is not possible to reject the null hypothesis with a one-tailed test. Thus, the number of important smells (ranked in the first 10 positions) is equal or less than 60 % of the smells.

Results Overall, we investigated how important the first 10 smells ranked by *SpIRIT* were, from the point of view of the developer. We verified that *SpIRIT* ranks at least 7 important smells in the first 10 positions of the ranking when $\alpha = 0$ or $\alpha = 0.5$. That is, the best results are found when the ranking is influenced (to some extent) by the modifiability scenarios. Since only 10 and 40 % (for $\alpha = 0$ and $\alpha = 0.5$ respectively) of the smells ranked by *SpIRIT* were shared by the developer's ranking, these results indicate that *SpIRIT* is indeed useful to the developer as it reveals "new" problems in the form of code smells. This finding is graphically shown in Table 9 by the smells in bold that are not highlighted in italic.

Another interesting finding derived from the good results with $\alpha = 0$ is that the approach could also generate acceptable rankings when the history it is not available. This case is specially relevant in early development stages of an application or when there is little or no history of previous versions available. However, the mixture of criteria had the best performance. This is because the ranking generated with $\alpha = 0.5$ contains more smells that affects problematic classes than the rankings generated with $\alpha = 0$ and $\alpha = 1$ see the smells highlighted in italic in Table 9).

4.2.3 Threats to validity

Our case-study and its results are subject to validity threats.

Conclusion validity While well-known statistical techniques were applied in the experiment, the main concern is that the experiment was conducted over one single application. This fact could reduce the power of the statistical test. For this reason, we think that further experiments with other applications are necessary in order to generalize our results.

Internal validity In our case, the main threat is the selection of the developer. Since she defined the scenarios and also indicated the relevance of each code smell, having other expert developers with a different background or perception of the system could generate different results. To mitigate this threat, a second developer of the application could be interviewed to corroborate the answers of the first developer.

Construct validity The main concern is that the previous knowledge of the developer about certain kinds of code smells could have made her prefer specific relevance values for those smells, regardless of their importance in the application context. We think that this is a minor threat because each developer would have different preferences about the smells relevances.

External validity In this case, this threat concerns the generalization of the experiment results to other environments. A minor threat is that the developer only analyzed the first 10 positions of each ranking. However, the analysis of all possible values was not viable. Also, this experiment is threatened by the concerns that we detail on case-study #1 about the difficulty to define modifiability scenarios in some contexts and also on how to map scenarios to source code in large or geographically-distributed teams.

5 Related work

The literature proposes various approaches to prioritize problems in object-oriented systems.

Tsantalis and Chatzigeorgiou (2011) rank refactoring suggestions to deal with code smells based on the analysis of past modifications. In this ranking, those refactorings whose target code was modified in the past will have the highest priority. Similar to us, the approach uses historical volatility models taken from the field of financial markets to calculate the probability of change of a given component. However, the way in which the volatility is calculated by the authors varies for each kind of code smell. Thus, the volatility is based in each instance of the smell. That means that the volatility can only be calculated since the creation of the smell ignoring important information of changes that could have been done before. Additionally, since only the volatility is taken into account to rank the smells, this approach can only be used in late development stages. Also, instead of generating a unique ranking, this approach generates a different ranking for each kind of code smell. This could confuse developers when a catalog of smells is used because of the high number of rankings. Moreover, the developer would not know which kind of smell to refactor first. Another difference between this approach and the Beta analysis used by us is that Beta calculates the volatility by comparing the change between the system and a class. That is, Beta allows us to know how important the changes are in the class with respect to the changes in the system.

Tsantalis and Chatzigeorgiou also present approaches to identify state-checking problems (the use of *if* clauses to decide the behavior of an object instead of polymorphism) and the decomposition of large methods (Tsantalis and Chatzigeorgiou 2010, 2011). Differently from us, the ranking in these approaches is done according to specific characteristics of the problems detected (e.g. number of instance variables used). Also, this work only ranks problems of the same kind.

Gîrba et al. (2004) follow the hypothesis that those fragments of code that were modified in the past are more likely to be modified in the future. This work analyzes each class of a version of a system using an algorithm that determines the probability of change of the class in future versions. The algorithm simply makes a ranking of the most changed classes in the last n versions of the system, then selects the top X classes. If the class is likely to change, then it is marked as candidate class to be refactored. That is, differently to us the output of the approach is not a list of smells but simply a list of classes that were modified frequently in the last versions of the system under analysis. The work does not cover the analysis of possible problems in the classes to be refactored or the suggestion of refactorings. To measure the change of a class between versions, the authors introduce a metric called latest evolution of number of methods (LENOM) that identifies the classes that experienced more changes in the last versions of the system.

Lanza and Marinescu (2006) present a strategy to identify code problems called disharmonies. The disharmonies are defined based on a combination of different metrics that have to exceed a predetermined threshold. They also propose the use of a visualization technique called Class Blueprint in order to complement the detection strategy. To prioritize the disharmonies to be refactored, the authors propose a naive approach in which the classes (or methods) that present a high number of disharmonies should be refactored first. We think that analyzing the concentration of smells can be an interesting complement for *SpIRIT* in the task of understanding the system under analysis.

Marinescu (2012) proposes the measurement of the impact of code smells based on three factors: (i) the negative influence of a kind of code smell in the architecture of a system; (ii) the kind of entity that the smell affects (such as a method or a class); and (iii) the values of the metrics used to identify each kind of code smell. This impact score can be used to rank smells. Factors (i) and (ii) are similar to our RCS criteria. In the case of factor (i) it is pre-calculated for each kind of code smells using a three value nominal scale (low, medium, high). That is, not real assessment is performed of the impact of a smell's instance in the architecture of the system but an estimation of the influence of a kind of smell. Regarding factor (iii), we think that a similar calculation can be used to measure the benefits of fixing a smell.

Arcoverde (2012) presents an approach to prioritize code smells using heuristics. The heuristics are based on different characteristics, namely: the number of changes of a component (e.g. packages, classes) during the history of a system, the number of bugs found in the component during its history, the concentration of smells in a class or package, and architecture roles played by classes. The heuristics are used to rank first those smells that affect components that meet the heuristics. While this approach takes into account architectural information through the use of architectural roles, the definition of architectural roles requires a thorough knowledge of architecture (for example, some form of documentation of the architecture, which typically is not available). Differently, to define the modifiability scenarios of our approach this kind of information is not always necessary since developers generally know the parts of the system that needs to be modifiable. Another difference with *SpIRIT* is that the work does not take into account the relevance of each kind of code smell.

Moreover, in contrast to the aforementioned approaches, our approach relies on modifiability scenarios as the lenses to look at important code smells. In this way, *SpIRIT* prioritizes those parts of the application that need to be as modifiable as possible.

6 Conclusion

This article presents a semi-automated approach for prioritizing code smells before deciding on suitable refactorings for them. The approach is based on three criteria: the history of changes of the components in which a smell is implemented, the relevance of the kind of smell for the developer, and how smells are affected by key modifiability scenarios. The approach is intended to help the developer choose which code smells should be fixed based on how critical the smells are.

In order to validate the benefits of the approach we conducted two case-studies. These case-studies corroborated our assumptions about the advantages of our approach, allowing the developer to refactor first the most critical smells. In the first study, we compared the smells ranked by *SpIRIT* for a Java application with the smells ranked by an expert in the application. We found that the ranking proposed by *SpIRIT* generated with $0 \leq \alpha \leq 0.5$ were highly correlated with the one proposed by the expert. In the second study, we analyzed how important the top-ten smells ranked by Spirit are for the developer in the context of a mid-size Java application. We found that *SpIRIT* ranks at least 7 important smells for the developer in the first 10 positions of the ranking when $\alpha = 0$ or $\alpha = 0.5$. That is, we found in both studies that *SpIRIT* ranks first the most critical smells of an application when the three criteria are combined. However, we also found that *SpIRIT* returns acceptable results when the criterion based on modifiability scenarios is preferred over the history. This is important because the history criterion has some drawbacks when used in isolation. For example, since a number of past versions needs to be available for the analysis, the results often come “late” in the lifecycle, when problems are harder to fix or important efforts might have been committed to implementation. Moreover, results from the second study showed that *SpIRIT* helps to reveal “new” problems in the form of code smells that the developer was not aware of.

Although we found that the approach helps the developer during the prioritization and refactoring of smells, the approach has still some limitations. First, since the Beta analysis is currently based on changes in the number of methods, it does not identify as a change the situation in which a number of methods of a class are removed and a same number of new methods are added. It neither takes into account the case in which the whole body of a method (or an important part of it) is replaced. Second, an adoption challenge is that the definition of scenarios by the developer can take some time and might require experience and knowledge of the system. Anyway, we believe it pays off in terms of aligning the code smell analysis with the system goals. Third, another limitation is that the criteria that takes into account the history of changes of the application can only be used in late stages of development (when enough history is available). However, we think that this criterion is complemented by the other two which can be used from the beginning of the development.

As future work, we will propose strategies to suggest refactorings to fix smells. Along with the suggestion of refactoring alternatives, we also plan to measure not only their benefits but also the cost of applying each alternative. These measurements should be based in different aspects: the number of refactorings of an alternative, the kind of refactorings to apply, the number of classes affected, among others. Moreover, this cost measurement could be integrated into SpIRIT as a new prioritization criterion. In this way, the developer could prioritize the most critical smells with the lowest refactoring costs. We also plan to test *SpIRIT* in other applications. For instance, we are interested in using *SpIRIT* to assist novice developers who are tasked to do refactoring of a system they might not be familiar with. In this situation, we conjecture that *SpIRIT* can help these developers to be more productive in their analysis of smells. Also, we will replicate some of the experiments and also analyze applications written in other programming languages than Java.

Acknowledgments This work was partially supported by CONICET (Argentina) through PIP Project No. 112-201101-00078. We are grateful to Mauricio Arroqui and Carlos Machado for their valuable collaboration in the BCFS case-study.

References

April, A., Abran, A.: Software maintenance management: evaluation and continuous improvement. IEEE Computer Society (2008)

Arcoverde, R.L.: Prioritization of code anomalies basead on architecture sensitiveness. Master's thesis, Pontifícia Universidade Católica do Rio de Janeiro (2012)

Bashroush, R., Spence, I.T.A., Kilpatrick, P., Brown, T.J.: Towards an automated evaluation process for software architectures. In: IASTED Conference on Software Engineering, pp. 54–58 (2004)

Clements, P., Kazman, R.: Software Architecture in Practice. Addison-Wesley Longman Publishing Co., Inc., Boston (2003)

D'Ambros, M., Lanza, M.: Visual software evolution reconstruction. J. Softw. Maint. **21**(3), 217–232 (2009)

Demeyer, S., Ducasse, S., Nierstrasz, O.: Object-Oriented Reengineering Patterns. Morgan Kaufmann, San Francisco (2003)

Erlikh, L.: Leveraging legacy system dollars for e-business. IT Prof. **2**(3), 17–23 (2000). doi:10.1109/6294.846201

Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Longman Publishing Co., Inc., Boston (1999)

Gamma, E., Helm, R., Johnson, R.E.: Design Patterns. Elements of Reusable Object-Oriented Software, 1st edn. Addison-Wesley Longman, Amsterdam (1995)

Gîrba, T., Ducasse, S., Lanza, M.: Yesterday's weather: guiding early reverse engineering efforts by summarizing the evolution of changes. In: ICSM, IEEE Computer Society, pp 40–49 (2004)

Hurtado, J.F., Sabadini, F., Vidal, S., Marcos, C.: Predicción del cambio a través de la historia del sistema. In: 14th Argentine Symposium on Software Engineering (ASSE 2013), 42 JAIIO (Jornadas Argentinas de Informática) (2013). In Spanish

Kazman, R., Abowd, G.D., Bass, L.J., Clements, P.C.: Scenario-based analysis of software architecture. IEEE Softw. **13**(6), 47–55 (1996)

Kim, M., Zimmermann, T., Nagappan, N.: A field study of refactoring challenges and benefits. In: Proceedings of 20th International Symposium on the Foundations of Software Engineering (FSE) (2012)

Lanza, M., Marinescu, R.: Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems. Springer, New York (2006)

Levy, H.: Fundamentals of investments. Financial Times, Prentice Hall (2002)

Macia, I., Arcoverde, R., Cirilo, E., Garcia, A., von Staa, A.: Supporting the identification of architecturally-relevant code anomalies. In: 28th IEEE International Conference on Software Maintenance (ICSM), pp. 662–665 (2012a). doi:10.1109/ICSM.2012.6405348

Macia, I., Arcoverde, R., Garcia, A., Chavez, C., von Staa, A.: On the relevance of code anomalies for identifying architecture degradation symptoms. In: CSMR (2012b)

Mangudo, P., Arroqui, M., Marcos, C., Machado, C.: Rescue of a whole-farm system: crystal clear in action. Int. J. Agile Extrem. Softw. Dev. **1**, 6–22 (2012)

Marcos, C., Vidal, S., Abait, E., Arroqui, M., Sampaoli, S.: Refactoring of a beef-cattle farm simulator. IEEE Lat. Am. Trans. **9**, 1099–1104 (2011)

Marinescu, R.: Assessing technical debt by identifying design flaws in software systems. IBM J. Res. Dev. **56**(5), 9 (2012)

Mens, T., Demeyer, S.: Future trends in software evolution metrics. In: Proceedings of the 4th International Workshop on Principles of Software Evolution, ACM, New York, IWPSE '01, pp. 83–86 (2001). doi:10.1145/602461.602476

Mkaouer, M.W., Kessentini, M., Bechikh, S., Ó Cinnéide, M.: A robust multi-objective approach for software refactoring under uncertainty. In: Le Goues, C., Yoo, S. (eds.) Search-Based Software Engineering, Lecture Notes in Computer Science, vol. 8636, pp. 168–183. Springer International Publishing, New York (2014)

Moha, N., Guéhéneuc, Y.G., Duchien, L., Meur, A.F.L.: Decor: a method for the specification and detection of code and design smells. IEEE Trans. Softw. Eng. **36**(1), 20–36 (2010)

Ozkaya, I., Díaz Pace, J.A., Gurfinkel, A., Chaki, S.: Using architecturally significant requirements for guiding system evolution. In: CSMR, IEEE, pp. 127–136 (2010)

Ricci, F., Rokach, L., Shapira, B., Kantor, P.B. (eds.): Recommender Systems Handbook. Springer, New York (2011). http://dblp.uni-trier.de/db/reference/rsh/rsh2011.html

Seacord, R., Plakosh, D., Lewis, G.: Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices. Addison-Wesley Professional, Boston (2003)

Tsantalis, N., Chatzigeorgiou, : Identification of refactoring opportunities introducing polymorphism. J. Syst. Softw. **83**(3), 391–404 (2010)

Tsantalis, N., Chatzigeorgiou, A.: Identification of extract method refactoring opportunities for the decomposition of methods. J. Syst. Softw. **84**(10), 1757–1782 (2011a)

Tsantalis, N., Chatzigeorgiou, A.: Ranking refactoring suggestions based on historical volatility. In: Kanellopoulos, Y., Winter, A., Mens, T. (eds.) CSMR, pp. 25–34. IEEE Computer Society, Los Alamitos (2011b)

Vidal, S.A.: (2013) Spirit: Smart identification of refactoring opportunities. Ph.D. thesis, UNICEN University

Wong, S., Cai, Y., Kim, M., Dalton, M.: Detecting software modularity violations. In: Taylor, R.N., Gall, H., Medvidovic, N. (eds.) ICSE, pp. 411–420. ACM (2011)

Woods, E.: Industrial architectural assessment using tara. J. Syst. Softw. **85**(9), 2034–2047 (2012)