JSpIRIT: A Flexible Tool for the Analysis of Code Smells

Santiago Vidal*, Hernán Vázquez*, J. Andrés Díaz-Pace*, Claudia Marcos† Alessandro Garcia, Willian Oizumi
ISISTAN Research Institute, UNICEN, Argentina PUC-Rio, Brazil
Email: {svidal, hvazquez, adiaz, cmarcos}@exa.unicen.edu.ar Email: {afgarcia, woizumi}@inf.puc-rio.br
*CONICET, Argentina
†CIC-Buenos Aires, Argentina

Abstract—Code smells are a popular mechanism to identify structural design problems in software systems. Since it is generally not feasible to fix all the smells arising in the code, some of them are often postponed by developers to be resolved in the future. One reason for this decision is that the improvement of the code structure, to achieve modifiability goals, requires extra effort from developers. Therefore, they might not always spend this additional effort, particularly when they are focused on delivering customer-visible features. This postponement of code smells are seen as a source of technical debt. Furthermore, not all the code smells may be urgent to fix in the context of the system's modifiability and business goals. While there are a number of tools to detect smells, they do not allow developers to discover the most urgent smells according to their goals. In this article, we present a flexible tool to prioritize technical debt in the form of code smells. The tool is flexible to allow developers to add new smell detection strategies and to prioritize smells, and groups of smells, based on the configuration of their manifold criteria. To illustrate this flexibility, we present an application example of our tool. The results suggest that our tool can be easily extended to be aligned with the developer's goals.

I. Introduction

The pressing needs for development of customer-visible features induces developers to usually make shortsighted design decisions, or write poor-quality source code. All these factors are justified in order to satisfy short-term goals, but can hinder the maintenance and evolution of the applications. This kind of problems is usually explained with the metaphor of technical debt [1], which is similar to a financial debt. In this metaphor, shortsighted design decisions or postponement of poorly-written code refactoring are considered a debt [2]. One way of paying the technical debt is by refactoring later the problematic code that originated it [3], although this payment comes at a cost.

A code smell (or code anomaly) is a symptom in the source code that helps to identify a design problem [3]. For instance, the so-called God Class is a smell that refers to a large and complex class that centralizes the intelligence of the system. In general, code smells allow developers to detect fragments of code that should be re-structured, in order to improve the quality of the system. When smells are not refactored out from an evolving program, other smells can increasingly affect the same program locations. These same groups of smells (e.g. such as various smells affecting the same class) are called agglomerations and are key indicators of design problems [4], [5], [6], [7]. Therefore, the occurrence of smell agglomerations indicates that the technical debt is increasing. A number of tools have been proposed for detecting single instances of code smells (but not agglomerations) [8], [9], [10]. Once detected, the smells can be fixed through a number of refactoring strategies [3].

978–1–4673–9817–6/15/$31.00

In an ideal world, fixing all smells would "pay" much of the technical debt; however, this activity would also be very time-consuming. Furthermore, not all smells will be equally urgent for the goals of the system. Unfortunately, existing tools for analyzing smells often output a large list of smells, and developers need to examine such a list manually in order to select the "right" smells to be tackled. Therefore, developers often need to be empowered with flexible means to define the criteria for detecting critical smells. Unfortunately, existing tools have a lack of support for selecting or defining specific criteria, not allowing developers to have their own ranking of critical smells.

In this context, this article presents a tool called JSpIRIT (Java Smart Identification of Refactoring opportunITies) that supports the identification and prioritization of technical debt in the form of code smells. The main benefit of using JSpIRIT is that developers can configure and extend the tool by providing different strategies to identify and rank the smells. To do so, the tool provides hooks for implementing smell identification strategies and evaluation criteria. For example, a criterion can be based on the relationship between the code smells and the modifiability goals of the system [11], [12]. Moreover, JSpIRIT allows the combination of different strategies for identifying agglomerations of smells, as well as the creation of flexible prioritization strategies for ranking the smells.

The remainder of this article is structured as follows. Section II reviews related work. Section III presents the tool design. Section IV describes an application example of JSpIRIT with several criteria. Finally, Section V gives the conclusions and discusses future work.

II. Related Work

Several tools have been proposed for automatically identifying code smells [8], [9], [13]. The approaches are usually based on a set of software metrics and thresholds. However, these detection strategies by themselves do not prioritize the usually long list of smells identified by the tools. Moreover,

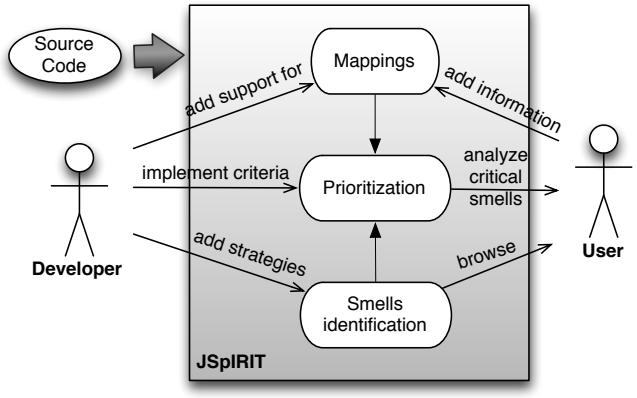


Fig. 1: JSpIRIT tool workflow

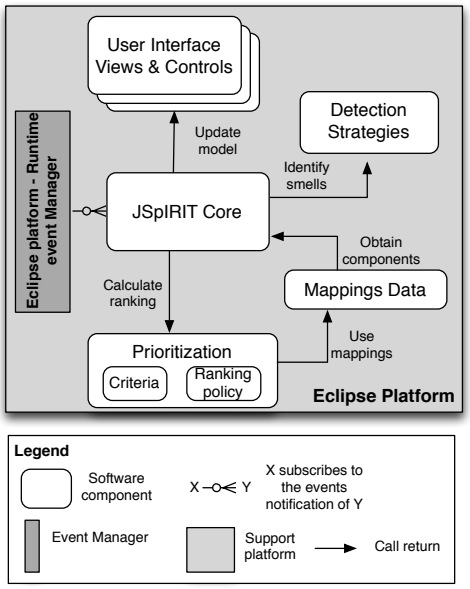


Fig. 2: JSpIRIT Architecture

as far as we know, no tool has yet the ability to identify agglomerations of code smells.

A few approaches and tools have tried to prioritize code smells [14], [13], [15], [16] as a complement to the detection strategies. These works use different criteria to rank code smells, such as: the kind of code smell [15], [16], software metrics [15], changes in the history of the component in which the smell is located [13], [16], or the dependencies between the smells [14]. In general, these works are not designed to be extended with new criteria or support the composition of the criteria. For example, some tools only rely on historical data to rank the smells [13]. However, while a historical analysis is important to determine which parts of the system received more changes, using this criterion alone implies that the approach can be most effective in late development stages when enough history is available to "predict" high-priority changes. Unfortunately, many code smells are already present in systems since the first versions [17]. For this reason, it is important to support multiple criteria, based on different information sources, in order to rank the smells according to their relevance.

III. JSpIRIT

In this section, we present the design of the JSpIRIT tool[1]. From its conception, JSpIRIT was intended to be flexible enough to strike a balance between i) prioritization criteria that require little developers' intervention but work best after several system versions, and ii) criteria that rely on (a reasonable amount of) external information provided in advance by developers but produce good results for early system versions.

Essentially, the JSpIRIT tool takes as input the Java source code of an application and produces as output a ranking of code smells (Fig. 1). To generate the ranking, a developer must instantiate the tool by providing:

- One or more kind of external information to the code, to be used by the prioritization criteria.
- One or more detection strategies for smells or agglomerations of smells.
- One or more criteria to determine how important a given smell is for the system.

Once JSpIRIT is instantiated, the user can interact with it to do one or more of the following activities that are directly related to the instantiation:

- Mapping of information: the user can include into JSpIRIT external information by mapping it to the source code. For example, the architectural design of the application can be mapped to the code in order to indicate in which classes is implemented each component of the architecture.
- Identification of smells: the application code is scanned, and the smells are automatically detected via detection strategies.
- Prioritization of smells: the smells are evaluated according to their importance based on different prioritization criteria.

These activities are supported by the architecture of the tool (Fig. 2), which is implemented as an Eclipse plug-in. The architecture involves six main components:

- JSpIRIT Core: it manages the loading and analysis of applications by interacting with the Eclipse platform. Also, it manages the JSpIRIT workflow by detecting the code smells and by interacting with the Prioritization component in order to generate the ranking of smells.
- Detection Strategies: this component implements the strategies for identifying code smells and agglomerations. A wide range of strategies is supported, both for anomalies and agglomerations. In addition, new detection rules for code smells can be easily added in this component.

[1] https://sites.google.com/site/santiagoavidal/projects/jspirit

- Mappings Data: this component supports the structure for mapping external information (to the source code) to classes. The developer can also configure new kinds of mappings to be used by the prioritization strategies.
- Prioritization Criteria: this component calculates a ranking for a set of code smells obtained from JSpIRIT Core. The ways in which the calculation is made and the criteria are applied are extensible. That is, new prioritization criteria can be added without changing the architecture, thanks to a set of interfaces.
- User Interface Views & Controls: it is a set of user interface components (views, dialog boxes and action handlers), which provide JSpIRIT functionality to its users. The user can interact with the user interface to map the external information to the source code, rank code smells, and analyze the source code of the detected smells.

A. *Mapping of External Information*

One way of taking advantage of external information when the code smells are analyzed and prioritized is to map the elements of the design model (e.g. modules, responsibilities, scenarios, concerns) to elements of the implementation (e.g. packages, classes, methods). In this way, JSpIRIT can be instantiated by a developer to take into account different kinds of requirements and architecture documentation such as component and connector views, blueprints, concerns, quality attribute scenarios, among others. In all the cases, an abstract architectural concept (a component, concern, etc.) will be mapped, by the final user of the tool, to one or more classes (or packages) to indicate its traceability. For example, JSpIRIT could be instantiated to allow the user to link modifiability scenarios [11] with elements of the source code. A modifiability scenario describes a change-related property that is desirable in a system. The definition of modifiability scenarios can help to assess how easy to modify a particular group of system elements should be. For example, a modifiability scenario could be stated as follow: "A developer wishes to change the visualization engine that generates a 3D view. This change will be made to the code at design time. It will take less than one hour to make and test the change and no side effect changes other than the new engine will occur in the behavior".

In order to map the external information to the source code, JSpIRIT provides different wizards to be extended. For example, in the case of modifiability scenarios, JSpIRIT should present a wizard (Fig. 3) to allow the user to initially define the name of the scenario and provide a brief description of it. Additionally, the user should select the classes, and packages of the system that compose the scenario by choosing from different lists given by JSpIRIT. Also, she can select an importance value between [0..1] to indicate how critical is the satisfaction of the scenario. Figure 3 shows the definition of a scenario called "Change 3D visualization engine". The left part of the wizard lists all the classes/packages of the system under analysis. The right part shows the classes/packages that belong to the scenario. In a similar way to scenarios, JSpIRIT allows developers to instantiated the tool with other kind of external information.

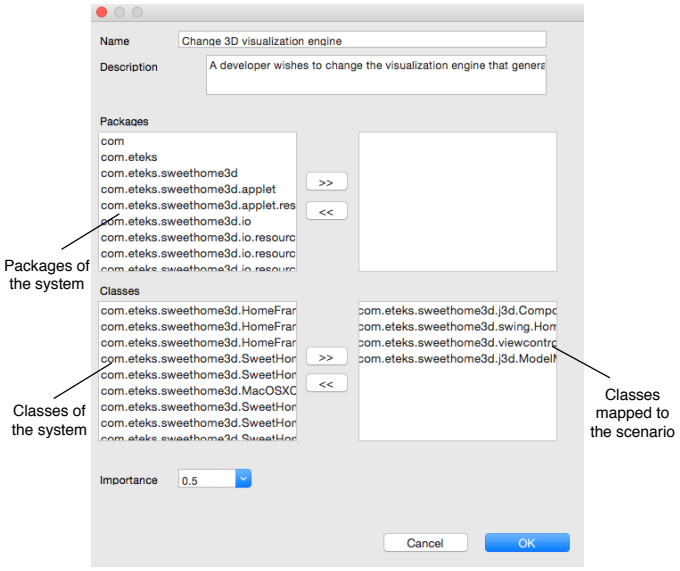


Fig. 3: Scenario wizard in JSpIRIT

TABLE I: Kinds of Smells Supported by JSpIRIT

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
Dispersed Coupling	Method that calls more methods of single external class that their own
Feature Envy	Method that calls one or few methods of several classes
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

B. *Identifying Smells*

The JSpIRIT tool begins by identifying the technical debt of the application in the form of code smells. Currently, JSpIRIT is pre-instantiated with the identification of 10 kinds of code smells (Table I) following the detection strategies presented by Lanza and Marinescu [8] and 3 kinds of agglomerations of smells (Table II) defined in [6]. Although these detection strategies are predefined in the JSpIRIT tool,

TABLE II: Kinds of Agglomerations Supported by JSpIRIT

Agglomeration	Short description
Intra-component	This agglomeration identifies smells that are implemented by the same architectural component
Cross-component	This agglomeration identifies smells that are syntactically related but that they are implemented by different architectural components
Hierarchical	This agglomeration identifies smells that occur across the same inheritance tree involving one or more components

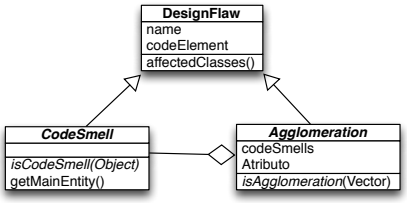
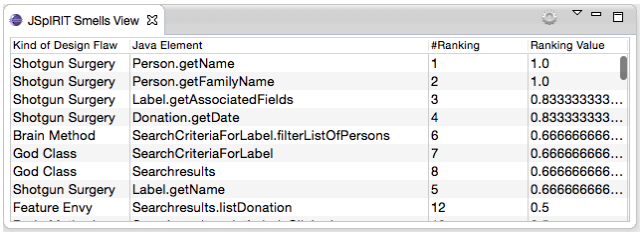
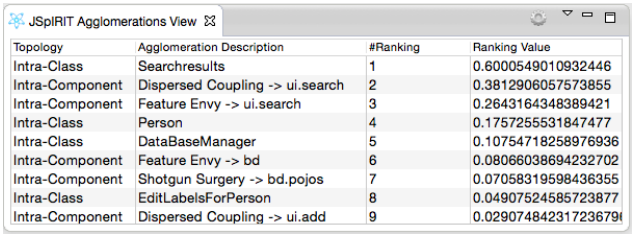


Fig. 4: Smells and agglomerations design



(a) Single instances of smells



(b) Agglomerations

Fig. 5: Code smells identification

note that they are not per se a part of our approach, they are just a pluggable module in the architecture of the tool. Thus, developers can easily add new smells (and their corresponding detection rules) by extending the abstract classes *CodeSmell* or *Agglomeration* (Fig. 4) that are part of Detection Strategies component (Fig. 2).

Once a system is loaded in the Eclipse environment, the user can instruct JSpIRIT to analyze the possible code smells (and agglomerations). This is done by clicking the "JSpiRIT->Find Code Smells" button from a contextual menu of the project. After clicking, the code smells are detected and ranked, and they are listed in the *JSpIRIT Smells View* and the *JSpIRIT Agglomerations View* (Figure 5). For each smell, JSpIRIT shows its kind and the Java element that compose the smell. We distinguish two kinds of constitutive elements: i) the class or method in which the smell is mainly implemented (we call this class/method the main class/method of the code smell), ii) the affected components. For example, for each instance of Shotgun Surgery, JSpIRIT shows the method that suffers from this kind of smell (main method). In the first smell listed in Figure 5, *getName* is the main method that is implemented by the main class *Person*. Additionally, when an element of a smell is clicked, JSpIRIT shows its source code.

In addition, using JSpIRIT enables the selection of different thresholds for the metrics used when detecting the code smells (Figure 6). While the developer that instantiate the tool can

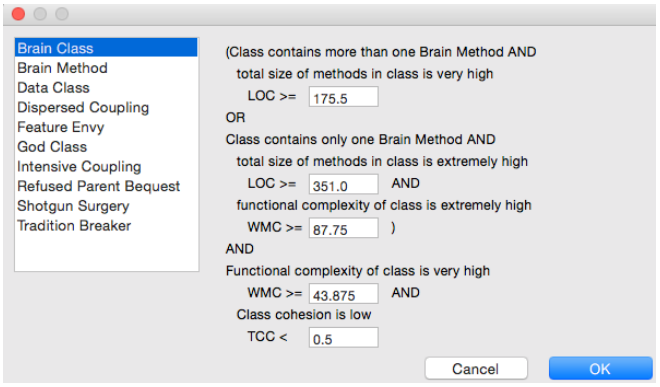


Fig. 6: Detection metrics configuration

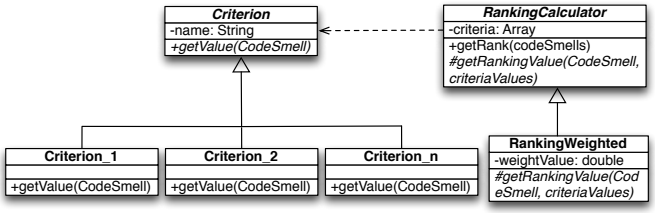


Fig. 7: Criteria and ranking detailed design

pre-configure thresholds by default, the final user of the tool can change the thresholds to adapt the detection strategies to the characteristics of the application being analyzed.

C. Prioritizing Smells

Once the smells are discovered, they should be ranked according to their importance. A developer can instantiated JSpIRIT to prioritize smells by different criteria. For example, the relevance of the kind of code smells, the history of the system, or different software metrics, among others. Additionally, the developer can use external information to improve the prioritization. For example, some works have shown that by refactoring the code smells related to design problems, the degradation of the architecture could be stopped [16], [18], [4]. In this case, the developer could create a criterion to rank first those smells that compromises the architecture of the system in its modifiability. This analysis can be achieved, for instance, by defining a criterion based on the relationship between the smells and the modifiability scenarios of the system. Thus, the external information incorporated into JSpIRIT (Section III-A) becomes important to determine the criticality of each smell instance.

As stated before, JSpIRIT supports the definition of multiple criteria to calculate the ranking of smells. The developer must instantiate the tool by adding their own criteria and also by specifying how the criteria will be combined to create the ranking. Fig. 7 shows a fragment of the detailed design for the module Prioritization (Fig. 2), which deals with the definition of criteria and the computation of an aggregated ranking. These classes realize the *Strategy* design pattern [19]. *Criterion* is an abstract class that defines the behavior (or strategy) that each criterion should follow. In this way, if a new

TABLE III: Change Analysis Example

	v1	v2	v3
LOC	10	15	20
% of change	-	50	33.3

criterion must be added to JSpIRIT, it should be implemented as a subclass of *Criterion*. The subclass must implement the *getValue(CodeSmell)* method that returns the value of the criterion for a smell instance. This is the strategic method of the pattern. Regarding the computation of the ranking, this is achieved by class *RankingCalculator*. This class combines the criteria following a policy implemented in a subclass, such as *RankingWeighted*, to prioritize the smells. For this reason, changes in the way criteria get combined are restricted only to this subclass. While JSpIRIT provides this policy, it allows the developer to instantiate her own policy by extending from *RankingCalculator*.

Figure 8 shows a sequence diagram of the creation of a ranking. In this case, an instance of class *RankingWeighted* receives the message *getRank()* with a collection of code smells as input parameter. Then, this method iterates over all the smells and for each one invokes method *getValue()* of the instances of Criterion contained in the criteria array. The values obtained from each criterion are sent to method *getRankingValue()* that returns the ranking score after combining the criteria values.

IV. Case Study

In order to reflect upon the flexibility of the tool to support different evaluation criteria, we present here a concrete usage of JSpIRIT with three prioritization criteria and a mapping of external information presented in a previous work [20]. In this case, we made the assumption that JSpIRIT was pre-instantiated with the identification strategies indicated in Section III-B. The criteria are based on: the analysis of modifiability scenarios vis-a-vis with code smells, the importance of the kind of smell, and an analysis of how likely the source code related to a smell will be modified in future versions. These criteria are described below:

a) Change Analysis (CA):: it provides clues about the stability of the components that participate in a code smell. The more often the components were modified in past system versions, the more unstable these components will be under new changes. Towards this goal, the number of lines of code (LOC) of the classes is analyzed. Specifically, this criterion returns the average of the percentage of changes in the LOC of a class in which a code smell is mainly implemented.

For example, consider a code smell whose main class is *A*. Table III shows the LOC values of *A* for three versions of the system in which it is implemented. Given this LOC values, the percentages of changes will be $\frac{15*100}{10} - 100 = 50\%$ for version 2 and $\frac{20*100}{15} - 100 = 33.3\%$ for version 3. In this way, the CA value will be $\frac{50+33.3}{2} = 41.6\%$.

b) Smell Relevance (SR):: by using this criterion the user can give to each kind of code smell a value of importance. This is because not all kinds of code smells are equally harmful. Specifically, the relevance is indicated using an ordinal scale from 1 to 10 (where 10 means that the code smell is very important). For example, a user could consider that instances of Feature Envy are more harmful for her system (given it a relevance of 10) because she want to reduce the coupling. This criterion allows JSpIRIT to adapt the recommendation of code smells to the preferences of the user. By selecting the relevance of a kind of code smell, the user can select the smells that he/she believes are the most important to the system or the kind of smells with which he/she is most familiar.

c) Scenario Analysis (SA):: this criterion helps to focus on those code smells that affect modifiability scenarios. Specifically, the SA value for a smell is calculated based on whether the class in which the smell is mainly implemented (as well as the classes affected by the smell) is mapped by one or more scenarios. Specifically, for each scenario that maps the main class of the smell, an X value will be added. Then, for each scenario that maps a class affected by the smell a Y value is added. The values of X and Y can be easily configured by the user. The intuition behind this calculation is to give more importance to those smells whose main classes and affected classes are both directly involved in scenarios. For example, consider the situation presented in Figure 9. Scenario 1 is mapped to classes *Bar* and *Foobar*. Class *Bar* is the main class of a code smell and it affects classes *Foo* and *Foobar*. In this case, if X=2 and Y=1, the Scenario Analysis value for class *Bar* will be $SA_{Bar} = 2 + 1 = 3$ since the main class of the smell (*Bar*) is mapped by one scenario and only one of the affected classes (*Foobar*) is also mapped.

These criteria were implemented in JSpIRIT using the classes described in Section III-C. As it is shown, our tool is flexible to implement criteria that are based on different kind of information: source code history, explicit feedback from an ordinal scale, and a mapping of modifiability scenarios to source code. Also, we instantiated the ranking calculation by implementing class *RankingWeighted*. This class calculates the ranking of code smells based on a α parameter that weights the criteria. Specifically, given the three criteria presented before, the ranking of smells is calculated as:

$$Ranking = \alpha * (SR * CA) + (1 - \alpha) * SA$$

where $0 \leq \alpha \leq 1$.

Figure 10 shows the JSpIRIT smells view ranking code smells generated using this calculation strategy. The ranking has four columns: (1) the kind of code smell and the name of the 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. $\alpha * (SR * CA)$), and (4) the weight of the scenarios in the ranking value (i.e. $(1 - \alpha) * SA$).

V. Conclusions

In this article we presented JSpIRIT, a flexible tool to prioritize technical debt in the form of code smells. JSpIRIT allows the user to focus on those code smells that are critical for the system, thus making their analysis cost-effective. The design of JSpIRIT supports a prioritization schema based on multiple criteria, which can be provided and extended by developers easily. We argue that JSpIRIT is lightweight because it can already work with a small set of criteria and provide a useful ranking to the users. The flexibility of the tool architecture to

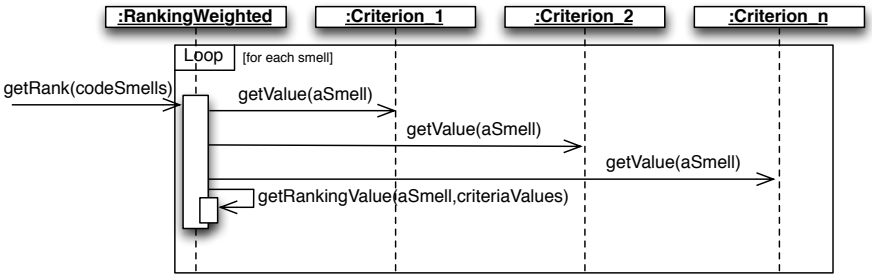


Fig. 8: Sequence diagram of ranking calculation

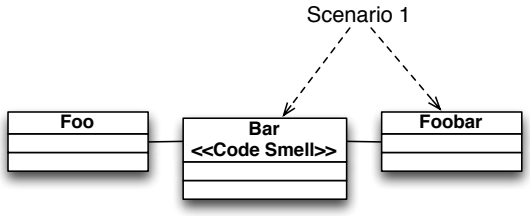


Fig. 9: Scenario Analysis example

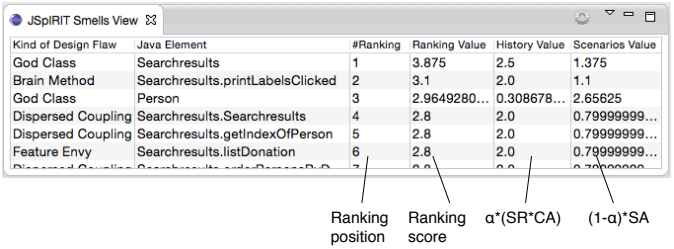


Fig. 10: JSpIRIT ranking

support new criteria was evaluated by instantiating JSpIRIT with three criteria. We found that JSpIRIT can be extended with a reasonable effort to prioritize smells with a broad kind of criteria.

As future work, we will extend our tool to support refactoring. Our intention is to suggest refactoring strategies for each kind of smells. This will help novice developers to analyze refactoring strategies for complex smells. Also, we plan to create visualizations to awareness developers of the components of the application most affected by smells.

REFERENCES

[1] W. Cunningham, "The wycash portfolio management system." *OOPS Messenger*, vol. 4, no. 2, pp. 29–30, 1993. [Online]. Available: http://dblp.uni-trier.de/db/journals/oopsm/oopsm4.html#Cunningham93

[2] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical debt: From metaphor to theory and practice." *IEEE Software*, vol. 29, no. 6, pp. 18–21, 2012. [Online]. Available: http://dblp.uni-trier.de/db/journals/software/software29.html#KruchtenNO12

[3] M. Fowler, *Refactoring: improving the design of existing code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[4] R. Arcoverde, E. Guimaraes, I. Macia, A. Garcia, and Y. Cai, "Prioritization of code anomalies based on architecture sensitiveness," in *27th Brazilian Symposium on Software Engineering (SBES)*, 2013.

[5] I. Macia, F. Dantas, A. Garcia, and A. von Staa, "Are code anomaly patterns relevant to architectural problems?" in *IEEE Transactions on Software Engineering*, 2014.

[6] W. Oizumi, A. Garcia, M. Ferreira, A. von Staa, and T. E. Colanzi, "When code-anomaly agglomerations represent architectural problems?" in *28th Brazilian Symposium on Software Engineering*, 2014.

[7] W. Oizumi, A. Garcia, L. Sousa, D. Albuquerque, and D. Cedrim, "Towards the synthesis of architecturally-relevant code anomalies," in *11th Workshop on Software Modularity (WMod'14)*, 2014.

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

[9] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. L. Meur, "Decor: A method for the specification and detection of code and design smells." *IEEE Trans. Software Eng.*, vol. 36, no. 1, pp. 20–36, 2010.

[10] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "Jdeodorant: Identification and removal of type-checking bad smells," in *12th European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 2008, pp. 329–331.

[11] P. Clements and R. Kazman, *Software Architecture in Practice*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[12] I. Ozkaya, J. A. D. Pace, A. Gurfinkel, and S. Chaki, "Using architecturally significant requirements for guiding system evolution." in *CSMR*, R. Capilla, R. Ferenc, and J. C. Dueñas, Eds. IEEE, 2010, pp. 127–136.

[13] N. Tsantalis and A. Chatzigeorgiou, "Ranking refactoring suggestions based on historical volatility." in *CSMR*, T. Mens, Y. Kanellopoulos, and A. Winter, Eds. IEEE Computer Society, 2011, pp. 25–34.

[14] H. Liu, Z. Ma, W. Shao, and Z. Niu, "Schedule of bad smell detection and resolution: A new way to save effort," *IEEE Transactions on Software Engineering*, vol. 99, no. RapidPosts, 2011.

[15] R. Marinescu, "Assessing technical debt by identifying design flaws in software systems." *IBM Journal of Research and Development*, vol. 56, no. 5, p. 9, 2012. [Online]. Available: http://dblp.uni-trier.de/db/journals/ibmrd/ibmrd56.html#Marinescu12

[16] I. Macia, R. Arcoverde, E. Cirilo, A. Garcia, and A. von Staa, "Supporting the identification of architecturally-relevant code anomalies," in *28th IEEE International Conference on Software Maintenance (ICSM)*, 2012, pp. 662–665.

[17] R. Peters and A. Zaidman, "Evaluating the lifespan of code smells using software repository mining," in *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*. IEEE, 2012, pp. 411–416.

[18] I. Macia, R. Arcoverde, A. Garcia, C. Chavez, and A. von Staa, "On the relevance of code anomalies for identifying architecture degradation symptoms," in *CSMR*, 2012.

[19] E. Gamma, R. Helm, and R. E. Johnson, *Design Patterns. Elements of Reusable Object-Oriented Software.*, 1st ed. Amsterdam: Addison-Wesley Longman, 1995.

[20] S. A. Vidal, C. Marcos, and J. A. Díaz-Pace, "An approach to prioritize code smells for refactoring," *Automated Software Engineering*, pp. 1–32, 2014.