Oizumi *et al. Journal of the Brazilian Computer Society* (2018) 24:13
https://doi.org/10.1186/s13173-018-0078-y

Journal of the Brazilian Computer Society

RESEARCH Open Access

On the identification of design problems in stinky code: experiences and tool support

Willian Oizumi[1,2*], Leonardo Sousa[1], Anderson Oliveira[1], Alessandro Garcia[1], Anne Benedicte Agbachi[1], Roberto Oliveira[1,3] and Carlos Lucena[1]

Abstract

Background: Developers often have to locate design problems in the source code. Several types of design problems may manifest as code smells in the program. A code smell is a source code structure that may reveal a partial hint about the manifestation of a design problem. Recent studies suggest that developers should ignore smells occurring in isolation in a program location. Instead, they should focus on analyzing stinkier code, i.e., program locations—e.g., a class or a hierarchy—affected by multiple smells. There is evidence that the stinkier a program location is, the more likely it contains a design problem. However, there is no empirical evidence on whether developers can effectively identify a design problem in stinkier code. Developers may struggle to make an analysis of inter-related smells affecting the same program location. Besides that, the analysis of stinkier code may require proper tool support due to its analysis complexity. However, there is little knowledge on what are the requirements for a tool that helps developers in revealing stinkier program locations. As a result, developers may not be able to identify design problems due to tool issues.

Method: To address this matter, we aimed at achieving three goals. In the first case, we proposed Organic—a tool supporting the analysis of stinky code. In the second case, we applied a mixed-method approach to analyze if and how developers can effectively find design problems when reflecting upon stinky code—i.e., a program location affected by multiple smells. We conducted a study with 11 software professionals. Finally, in the third case, we aimed at understanding if Organic could be used by developers to identify design problems. To achieve this goal, we used a method from the Semiotic Engineering theory. This method enabled us to evaluate what are the tool issues that may hinder the identification of design problems in stinky code.

Result: Our study revealed that only 36.36% of the developers found more design problems when explicitly reasoning about multiple smells as compared to single smells. Moreover, 63.63% of the developers reported much lesser false positives when using the first approach as compared to the latter. The second study, in its turn, showed that most developers may be unable to identify design problems in stinky code without proper tool support.

Conclusion: Our experiences, in particular the second study, helped us to refine the features of Organic for better supporting developers in reflecting upon stinkier code. For example, analyses of stinky code scattered in class hierarchies or packages is often difficult, time-consuming, and requires proper visualization support. Moreover, without effective support, it remains time-consuming to discard stinky program locations that do not represent design problems.

Keywords: Design problem, Software design, Code smell, Agglomeration

*Correspondence: woizumi@inf.puc-rio.br
[1]Opus Research Group, Informatics Department, PUC-Rio, Rio de Janeiro, RJ, Brazil
[2]IFPR, Campus Paranavai, Paranavai, PR, Brazil
Full list of author information is available at the end of the article

Introduction

The identification of design problems in the source code is not a trivial task [1, 2]. Developers usually need to find hints, as code smells, in the source code that can lead to a design problem. A code smell is a structure in the source code that may provide developers with a partial indication about the manifestation of a design problem [3]. A classical example of code smell is the *God Class*, which occurs when a class is long and complex, centralizing a considerable amount of intelligence of the system. However, the occurrence of a single smell in isolation in a program often does not represent a design problem [4, 5]. A design problem is a design characteristic that negatively impacts maintainability [2]. Recent studies reveal that design problems are much more often located in stinkier program locations (i.e., a class, a hierarchy, or a package) affected by multiple smells [4–8]. For instance, a *Fat Interface* [9] is a design problem that often manifests as multiple smells in a program, affecting various classes that implement, extend, and use the interface in a program [5].

The stinkier a program location is, the more likely it contains a design problem [5, 10]. In fact, developers tend to focus on refactoring program locations with a high density of code smells and ignore those locations affected by a single smell [11, 12]. However, there is limited understanding if developers can effectively identify design problems in stinkier code, i.e., program locations affected by multiple smells. Indeed, existing techniques tend to focus on the detection and visualization of each single smell [13–16]. They do not offer a summarized view of inter-related smells affecting a program location [5]. Moreover, previous studies focus on simply analyzing the correlation between design problems and code smells [5, 17]. They have not investigated if and how developers are indeed effective in the task of finding design problems in stinkier code.

Therefore, we do not know whether the analysis of multiple smells actually provides better precision for the identification of design problems. Developers may struggle to make a meaning out of inter-related smells affecting the same program location. Additionally, the analysis of stinkier code may require proper tool support due to its analytic complexity. However, there is limited knowledge on what are the requirements for a tool that supports the analysis of stinkier code. This is important because developers may not be able to identify design problems due to tool support issues. To address these matters, we defined three goals for our research: (1) provide proper support for the analysis of stinkier code, (2) assess to what extent developers are able to identify design problems in stinkier code, and (3) identify tool issues that may hinder the identification of design problems.

To achieve our first goal, we designed and implemented Organic—a tool supporting the analysis of stinky code. We used findings from previous studies [4, 5, 17–20] as a start point for defining the requirements of Organic. In a nutshell, Organic supports the analysis of multiple forms of stinkier code, provides detailed information about code smells, supports the analysis of dependencies between stinky elements, provides a visualization for stinkier code, provides historical information about stinkier code, and allows developers to specify the thresholds that should be considered when identifying stinkier code. In the context of Organic, the threshold defines the minimum number of smells that a program location should have to be considered stinkier. Those features will be presented in detail in the "Organic: a tool for the analysis of stinky code" section.

For achieving the second goal, we applied a mixed-method approach to analyze if and how developers can effectively find design problems when reflecting upon stinky code. This study comprised both quantitative and qualitative analyses. For the quantitative analysis, we compared the precision of the developers with a baseline, i.e., situations where only single smells were given to them. As we want to assess if multiple smells can help developers to reveal more design problems than single smells, we divided the developers into two groups. In the first group, we asked them to identify design problems through the analysis of stinky program locations. In the second group, we asked them to identify design problems with the analysis of single smells. After that, we inverted the groups, and we asked them to repeat the identification of design problems in a second system. In each identification task, we used the group that identified design problems with single smells as the control group. Thus, we could use the control group to measure if the analysis of stinkier program locations can improve the precision of design problem identification.

In the qualitative analysis, we performed a systematic evaluation through the careful observation of participants during the study execution and the application of a follow-up questionnaire. The objective of this analysis was to identify the main barriers of reflecting upon multiple smells along the task of identifying design problems. The outcomes of this analysis helped us to better understand ways to improve support for the identification of design problems in stinky code.

By triangulating the results of both analyses, we noticed that 36.36% of the developers found more design problems when explicitly reasoning about multiple smells. We found that the understanding of complex stinky code helped to confirm the occurrence of non-trivial design problems, such as *Scattered Concern* [21]. Furthermore, we found that 63.63% of the developers reported much less false positives when analyzing multiple smells than when analyzing single smells. Thus, developers that considered stinky program locations, instead of isolated smelly code, could identify design problems

with higher precision. However, this study also showed that developers need better support to analyze stinky program locations to reveal design problems. We observed that the analysis of stinky code may be difficult and time consuming. For instance, a prioritization algorithm is required so that developers do not waste time analyzing many stinky program locations not related to design problems.

Finally, to achieve our third goal, we evaluated Organic with the Communicability Evaluation Method (CEM) [22]. CEM is a method from the Semiotic Engineering theory, which is intended to reveal ruptures of communication when a user interacts a system, i.e., in our case when the developer interacts with the Organic tool. This method enabled us to identify issues in the Organic tool that may hinder the identification of design problems.

By conducting the communicability evaluation of Organic, we observed three major issues. First, although the tool detects stinkier program locations, it often fails to provide a concise message that facilitates the reasoning about the possible design problem (affecting the stinky program location). Second, the terms used in the tool are not adequate to certain software developers. Third, Organic uses ambiguous static symbols for representing different types of information. During the evaluation, we noted that, the aforementioned issues may often hinder the identification of design problems in stinky code.

Contextualization

This section, which is organized into two subsections, provides background information to support the understanding of this paper. The "Basic concepts" section outlines basic concepts. The "Identifying design problem in stinky code" section brings up an illustrative example of analyzing stinky code to identify design problems.

Basic concepts

Design problem

A design problem is a characteristic in the software design that leads to negative impact on maintainability [2]. Design problems affect program locations such as packages, interfaces, hierarchies, classes, and other structures that are relevant for the design of the system [23]. Examples of design problems include *Scattered Concern* [21] and *Fat Interface* [9]. The description of the eight types of design problems considered in our study is presented in Table 1. We opted by selecting these design problems since (i) they are often considered as critical in the systems [5] chosen in our study and (ii) other studies have shown the relation between such design problems and code smells [4, 5, 17, 18, 24].

Table 1 Description of design problems

Name	Description
Fat interface	Interface of a design component that offers only a general, ambiguous entry-point that provides non-cohesive services, thereby complicating the clients' logic.
Unwanted dependency	Dependency that violates an intended design rule.
Component overload	Design components that fulfill too many responsibilities.
Cyclic dependency	Two or more design components that directly or indirectly depend on each other.
Delegating abstraction	An abstraction that exists only for passing messages from one abstraction to another.
Scattered concern	Multiple components that are responsible for realizing a crosscutting concern.
Overused interface	Interface that is overloaded with many clients accessing it, that is, an interface with too many clients.
Unused abstraction	Design abstraction that is either unreachable or never used in the system.

Smelly code

Code smell is a recurring micro-structure in the source code that may indicate the manifestation of a design problem [3]. A design problem can manifest itself in a program by affecting multiple source code locations. Each of these locations are called here *smelly code*. Thus, the developers can analyze the smelly code to identify a design problem. There are several types of code smell, which may affect a method, a class, or a hierarchy. In this paper, we used nine types of code smell, which are *God Class*, *Brain Method*, *Data Class*, *Dispersed Coupling*, *Feature Envy*, *Intensive Coupling*, *Refused Bequest*, *Shotgun Surgery*, and *Tradition Breaker*. These types of smell were considered in this study as they occur in the target systems used in this work. The description of each type of smell is presented in Table 2.

Code smells and design problems

Developers can rely on the analysis of code smells to identify design problems [17, 25, 26]. The use of code smells to identify design problems is possible because some instances of code smells manifest in the source due to the presence of a design problem. Consequently, code smells tend to co-occur in elements affected by design problems [5, 8, 10, 18], which make them indicators of design problems. Unfortunately, not all (instance of) code smells are related to a design problem [17].

Usually, a code smell is related to a design problem when it occurs due to the presence of design problem. For instance, consider Scattered Concern [21], a design problem that occurs when multiple code elements implement a functionality that should have been implemented by only

Table 2 Types of code smell

Type	Description
God class	Long and complex class that centralizes the intelligence of the system
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	The case of an operation which is excessively tied to many other operations in the system, and additionally these provider methods that are dispersed among many classes
Feature envy	Method that calls more methods of a single external class than the internal methods of its own inner class
Intensive coupling	When a method is tied to many other operations in the system, whereby these provider operations are dispersed only into one or a few classes
Refused bequest	Subclass that does not use the protected methods of its superclass
Shotgun surgery	This smell is evident when you must change lots of pieces of code in different places simply to add a new or extended piece of behavior
Tradition breaker	Subclass that provides a large set of services that are unrelated to services provided by the superclass

a few elements. Often, elements that implement the scattered functionality contain code smells such as God Class, Feature Envy, Intensive Coupling, Divergent Change, and the like. As the code elements implement a scattered functionality, these elements are likely of realizing at least two functionalities: their predominant functionality and another one, in which the predominant functionality can be either the scattered one or not. Either way, the elements implement more than one functionality, which leads them to the appearance of a God Class. Additionally, the methods in the class have to communicate with other classes that also implement the scattered functionality. Thus, these methods can contain instances of Feature Envy, leading to the appearance of an Intensive Coupling smell. Furthermore, every chance in the functionality will impact the elements that implement it; thus, these elements will have the Shotgun Surgery and Divergent Change. In summary, these code smells could appear in the elements due to the scattered functionality, i.e., due to the Scattered Concern.

Stinky program location

Indeed, code smells can be indicators of design problems. In fact, recent studies [4–6, 8] suggest that the stinkier a program location is, the more likely it is to be affected by a design problem. Stinky code is the manifestation of multiple code smells in a program location. In this paper, we are especially interested in stinky code indicated by *smell agglomerations* [5]. A smell agglomeration is a group of inter-related code smells affecting the same program location, such as a method, a class, a hierarchy, or a package [5]. Thus, the agglomeration is determined in the program by the co-occurrence of two or more code smells in the same method, class, hierarchy, or package (or component). For code smells that co-occur in the last three cases, we only consider they are part of an agglomeration if they are syntactically related [5]. For instance, two classes can be related through structural relationships in the program, such as method calls and inheritance relationships. In this paper, we considered four categories of agglomeration, which are presented below.

An *intra-method smell agglomeration* consists of multiple code smells that are located in a single method. The minimum number of code smells required to characterize an intra-method smell agglomeration is arbitrarily defined by the developers through a threshold. Figure 1 presents an example of intra-method agglomeration extracted from the Apache OODT (Object Oriented Data Technology) system. OODT is a distributed system aimed at supporting the management and integration of processes, data, and metadata [27]. The agglomeration of Fig. 1 occurs in the *fromWorkflowInstance* method, which is implemented by the *WorkflowProcessorQueue* class. This method is affected by two code smells: Brain Method and Intensive Coupling. The smelly method is the source of a Brain Method because *fromWorkflowInstance* performs several operations related to pre-conditions, tasks, and pos-conditions. All these operations make the method difficult to read and, consequently, to maintain. Moreover, this method suffers from *Intensive Coupling* because it is tightly coupled to a few classes, namely *WorkflowInstance*,*WorkflowProcessor*, *WorkflowCondition*, and *WorkflowTask*. These two smells together indicate the method is complicated, addresses multiple responsibilities, and is intensively coupled to a few classes in the system.

An *intra-class smell agglomeration* consists of multiple code smells affecting a single class. Thus, a class C has an agglomeration whenever the number of code smells affecting C is higher than an arbitrary threshold defined by the developer. Figure 1 also shows an example of intra-class agglomeration extracted from the OODT system. This agglomeration occurs in the *WorkflowProcessorQueue* class and is composed by four code smells instances: Feature Envy in the *getLifeCycle* method, Dispersed Coupling in the *getProcessors* method, and Brain Method and Intensive Coupling in the *fromWorkflowInstance* method. Such a combination of smells suggests the *WorkflowProcessorQueue* class is tied to many other classes in the system and has more responsibilities than it should.

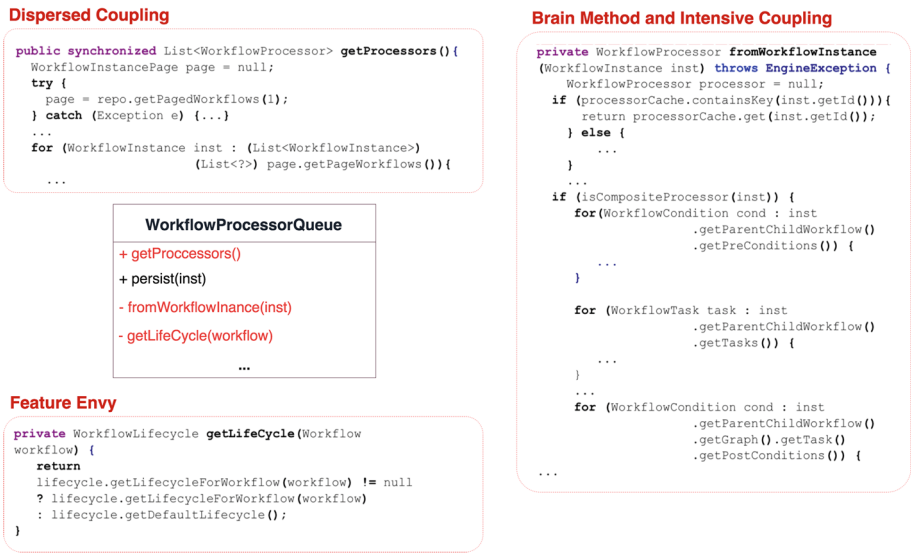

Fig. 1 Intra-method and intra-class agglomerations in the WorkflowProcessorQueue class

A *hierarchical agglomeration* follows two conditions. First, all code elements have to be affected by the same type of code smells. Second, these elements have to implement the same interface or inherit from the same code element. Figure 2 illustrates an example of hierarchical agglomeration affecting the Apache OODT system. The Versioner class is an abstraction affected by a Fat Interface instance due to the high number of responsibilities implemented into it, which realize different design concerns. Besides that, the Versioner class is inherited by other classes, namely *SingleFileBasicVersioner*, *BasicVersioner*, *DateTimeVersioner*, and *MetadataBasedFileVersioner*. All implementations are affected by Feature Envy, because they have too many dependencies with multiple classes of the system. Thus, all these Feature Envy instances together form an agglomeration that reifies the Fat Interface design problem affecting Versioner.

An *intra-component smell agglomeration* occurs inside of a single design component. This agglomeration comprises multiple code smells affecting different code elements that are located within the same component. The minimum number of code smells required to characterize an intra-component smell agglomeration is arbitrarily defined by the developer. Note that, for characterizing this type of smell agglomeration, all code elements have to (i) be affected by the same type of code smell and (ii) be connected by method calls or type references. The "Identifying design problem in stinky code" section presents a detailed example involving an intra-component smell agglomeration.

The concept of smell agglomeration is not limited to the categories presented above. There are other categories that we did not consider in this study. An example is the concern-overload category [5], which is provided by the Organic tool—as described in the "Organic: a tool for the analysis of stinky code" section. In this paper, our focus was to analyze the identification of design problems in stinky program locations. Therefore, we considered only categories that represent common program locations, such as methods and classes.

Identifying design problem in stinky code

As explained in the previous section, the identification of design problems can be based on code smells. For instance, let us consider the example illustrated in Fig. 3. This figure presents some classes that belong to the *Workflow Manager* subsystems–a component of the Apache OODT (Object Oriented Data Technology) system [27]. It is responsible for description, execution, and monitoring of workflows. Suppose that a developer is in charge of identifying design problems in the Workflow Manager subsystem, she can rely on the analysis of code smells to spot program locations that may contain a design problem. If she is analyzing the repository package, she will notice that this package contains several code smells as indicated by a smell agglomeration. This agglomeration is formed by four instances of the Feature Envy smell. As illustrated by Fig. 3, each of the Feature Envy occurrences affects a different class. In this case, three classes implement the *WorkflowRepository* interface. When the developer analyze these classes based on the Feature Envy smell, she will realize that these classes contain the smell because one of their methods is more interested in other classes than in its own hosting class. This happens because these methods are forced to implement a method that was defined in the *WorkflowRepository* interface, that

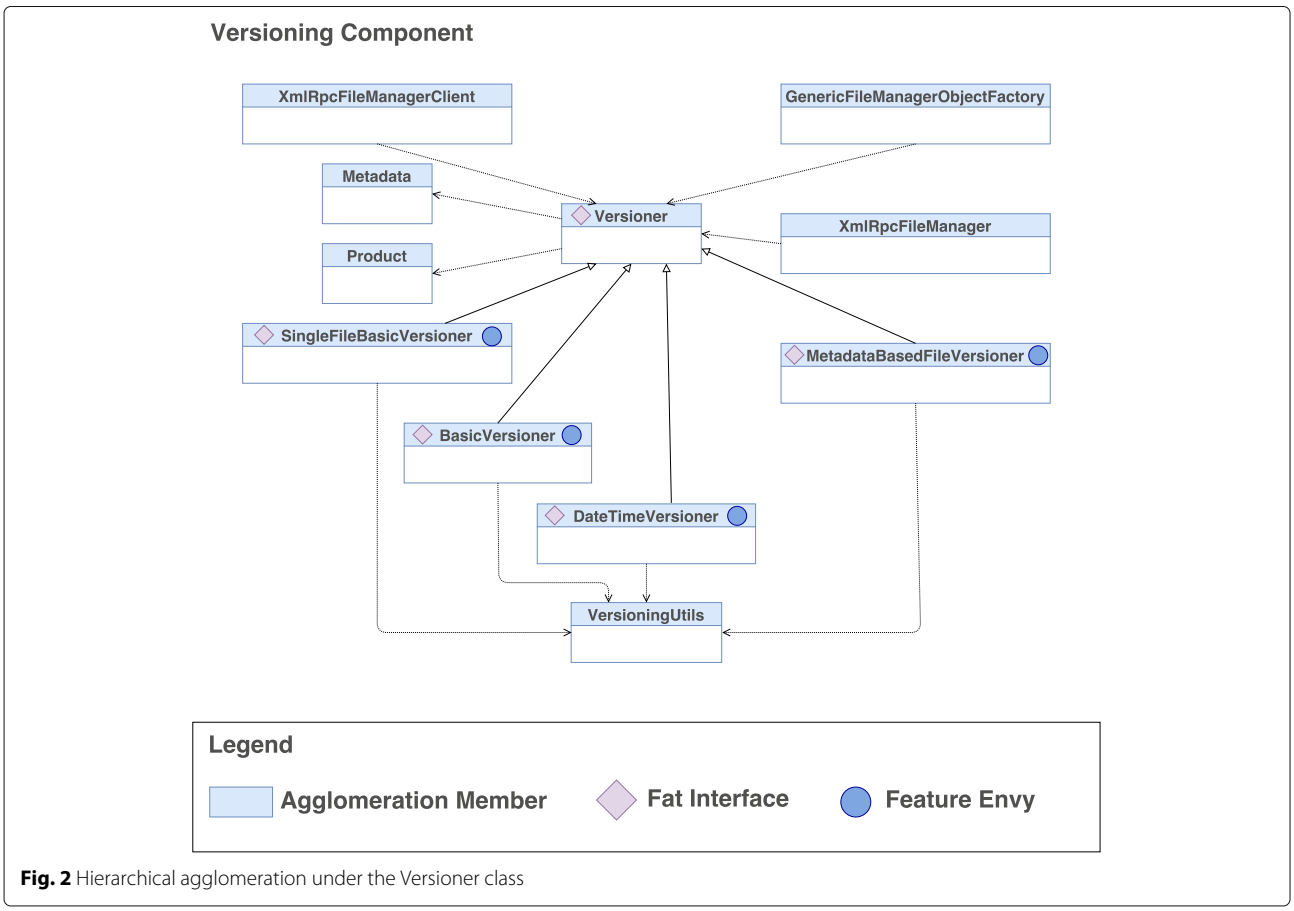

Fig. 2 Hierarchical agglomeration under the Versioner class

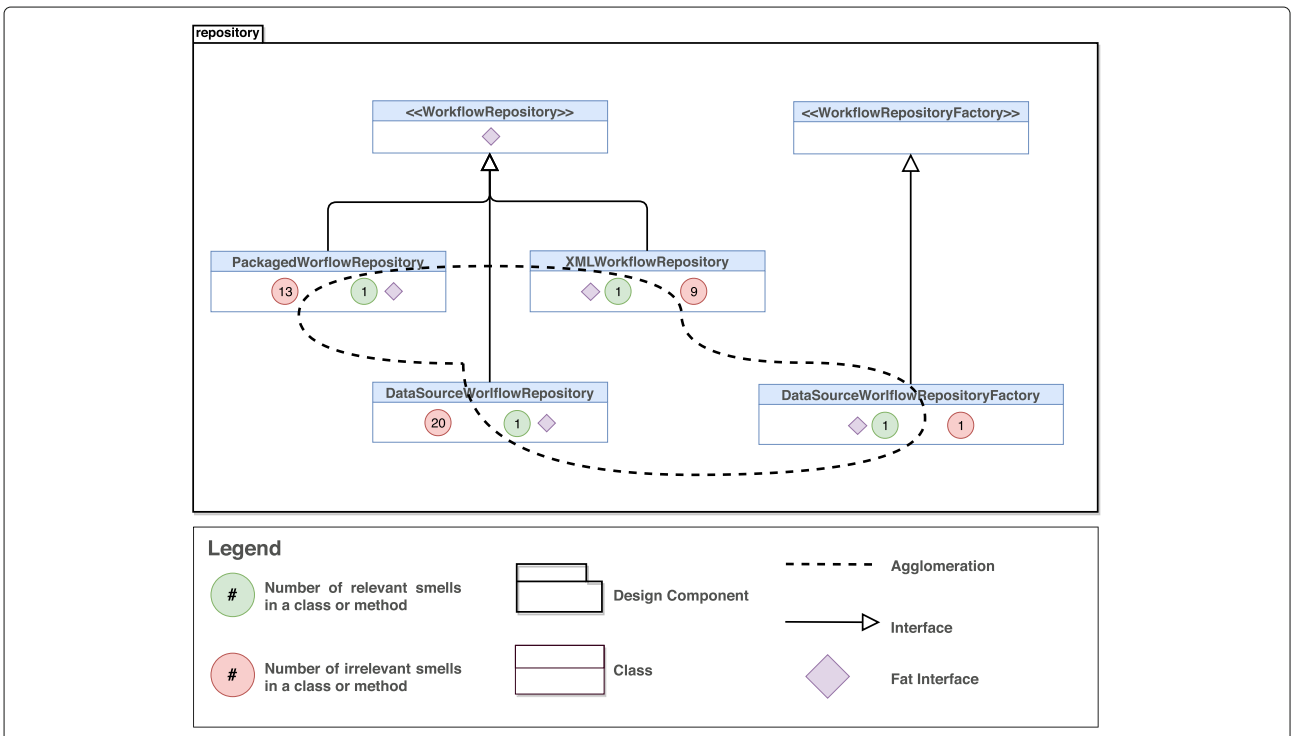

Fig. 3 Example of agglomeration in the workflow system

is, the smells in the agglomeration are indicating that (the corresponding method in) the interface may contain a design problem. In fact, this "forced implementation" becomes a problem because these methods are implementing a concern that should not have been implemented in their hosting classes. That happens because of the fact that the *WorkflowRepository* interface processes multiple services; thus, any class that implements this interface needs to handle more services than it actually should have.

In this example, the developer knows that the code smells in the agglomeration have the same type (Feature Envy). Also, she knows that three classes affected by the code smells implement the same interface, as reified in a *hierarchical* agglomeration. This interface, in its turn, seems to provide non-cohesive services. Thus, the developer can infer that a design problem, called Fat Interface, is affecting the *WorkflowRepository* interface. On the other hand, if she did not reflect upon the code smell agglomeration, it would be harder to her to identify the same design problem. One of the reasons is the number of code smells spread over the six classes and two interfaces within the package. Although the package contains only eight classes (Fig. 3 only shows some of them), it has more than 50 code smells, many of which are irrelevant for the identification of a Fat Interface. Thus, she has to analyze many smelly code snippets in order to discard, postpone, or further consider them in the identification of design problems.

Let us assume that the developer only reasons about each code smell in isolation to identify the design problem, i.e., without taking into consideration smell relationships in an agglomeration. Thus, she can choose to analyze the *DataSourceWorkflowRepository* class first because the class contains the highest number of smells in the package. Analyzing the 21 instances of code smells in the class, the developer will notice that the class has smells related to high coupling with other classes (Intensive Coupling and Dispersed Coupling), low cohesion (Feature Envy), and overload of responsibilities (God Class). However, all these smells may indicate different problems. Thus, she has to extend the analysis to other classes in order to gather more information that can potentially indicate a design problem. Unfortunately, the other classes also have different instances of code smells, and these instances may not be related to any design problem. Therefore, the developer can face difficulties to find the relevant code smells that can help her to identify a design problem. Thus, the analysis of stinky program locations, as revealed by agglomerations, seems to be a better strategy. However, there is little empirical understanding about this phenomenon. Existing studies are limited to investigating only if there is a correlation of design problems with stinky code.

Organic: a tool for the analysis of stinky code

In this section, we present the Organic tool[1]. Organic is a plug-in developed for the Eclipse IDE. The essential objective of Organic is to enable its users to identify and reason about design problems. To fulfill its role, Organic detects and groups code smells into agglomerations of code smells. The detection of smells is performed with the conventional detection strategies proposed by Marinescu [28]. Each conventional detection strategy is a heuristic that detects code elements that possibly suffer from a particular type of code smell. The heuristic of a detection strategy is based on a set of metrics and thresholds, which are combined into logical expressions [28]. After the detection of code smells, Organic explores different forms of relationship between smells in order to search for smell agglomerations. The smell agglomerations are identified through information extracted from different artifacts of the analyzed software. Finally, for each agglomeration, Organic extracts information that may be helpful for the identification of design problems.

We developed and evolved Organic's features based on findings from related work [4, 5, 17–20]. We used their findings as a start point for defining a preliminary set of requirements for supporting the identification of design problems in stinky code. Next, we present the requirements along with the corresponding features of Organic. The requirements are made explicit in the title of each paragraph below, followed by the description of Organic features that implement the corresponding requirement.

Supporting multiple categories of agglomeration. Our prior work [10, 19] provided evidence that design problems may be reified in the source code by different forms of agglomeration. Therefore, a tool for the analysis of stinky code must support multiple categories of agglomeration. Thus, to support this requirement, Organic provides five categories of agglomerations: (i) *intra-method*, (ii) *intra-class*, (iii) *intra-component*, (iv) *hierarchical*, and (v) *concern-overload*.

Before searching for agglomerations, Organic uses the source code model provided by Eclipse—through the *org.eclipse.jdt.core.dom* package—to compute metrics such as Access to Data, Number of Lines of Code, McCabe Complexity, and the like. After that, the metrics are combined into heuristics for the detection of code smells. Organic uses the heuristics defined by the conventional detection strategies of Marinescu [28]. Below, we present an example of detection strategy for Long Method smells:

> Long Method = Lines Of Code > VERY HIGH and Cyclomatic Complexity > HIGH

Detection strategy above determines that a Long Method occurs when the method (1) has more lines of code than the number defined by a given threshold (VERY HIGH) and (2) has cyclomatic complexity higher than a given threshold (HIGH).

After detecting all code smells, Organic uses different algorithms to search for different categories of agglomeration. Algorithm 1 shows a pseudo-code illustrating the algorithm used by Organic to search for intra-method agglomerations. For each method in the source code, Organic computes the number of smells. When a method has more code smells than a given threshold, Organic considers that there is an intra-method agglomeration.

Algorithm 1: Pseudo-code of the algorithm to search for intra-method agglomerations

```
Data: Set M of methods
Result: Set A of intra-method agglomerations
initialization;
for each method m in M do
    if numberOfSmells(m) > THRESHOLD then
        A.add(agglomerationOf(m))
    end
end
```

The different categories are shown by Organic in the *Agglomerations View*. This is the main view of the tool, and it provides features to support the identification of design problems. Figure 4 shows a screenshot of the Agglomerations View. As one can observe, this view is separated in two parts: the first part is called Agglomerations, which is shown on the left side; the second part is called Details, which is shown on the right side. The Agglomerations part shows the agglomerations found in one or more projects according to their category. For instance, Fig. 4 shows two categories of agglomerations detected in a project called *cas-pushpull*.

By clicking on an agglomeration category, one more subitem level is expanded. This new level displays all agglomerations in the selected category. For example, in Fig. 4, all the hierarchical agglomerations that were found in the *cas-pushpull* project are displayed. Thus, developers can use these Organic features to select the category and/or the specific agglomeration they want to focus.

Providing detailed information about code smells. Many developers have little to no knowledge about the concept of code smells. In a survey conducted by Yamashita and Moonen [29], for example, only 18% of participants reported a good or strong understanding about code smells. As a result, most developers may fail short in the analysis of stink code when using a tool that do not provide enough information. Hence, to overcome this limitation, Organic provides detailed information about each agglomeration of code smells.

When the user selects an agglomeration, Organic displays all the smells that compose the agglomeration on the right side of the screen (in the Anomalies tab). One can see in Fig. 4 that the first agglomeration in the *Hierarchical* category contains three *Intensive Coupling* smells. The second tab (Fig. 5) presents a textual description with information about the agglomeration according to the category. As one can observe in Fig. 5, the description of a *Hierarchical* agglomeration displays information about the number of smells that compose the agglomeration with a textual description of each type of smell.

Supporting the analysis of surrounding code elements. In our previous work [5, 10], we observed that a design problem may involve the surrounding code elements of an agglomeration. For instance, we found agglomerations in which one or more surrounding elements were the main cause for the design problem manifestation. Thus, to support the analysis of surrounding code elements, Organic provides a tab called References. This tab displays all references involving smelly code elements. For instance, in Fig. 6, the *getSite* method of the *RemoteSiteFile* class is referenced by 10 other methods. This Organic feature enables developers to reason about the external impact of an agglomeration and further help for the search of a design problem associated with the agglomeration. For example, a high number of references involving agglomerated element(s) and their surrounding elements may suggest the occurrence of a scattered functionality and/or an overly coupled component.

Providing a visual representation of stinkier code. Based on findings from our first (mixed-method) study (the "Study I: Quasi-experiment" section), we also incorporated a graph-based view into Organic. This view is displayed in the fourth tab. Figure 7 shows an example of the graph-based visualization for a selected agglomeration. The visualization is not intended to provide a dependency graph of the agglomeration's code elements. Instead, the objective is to provide an abstract representation of an agglomeration, an overview of the composition of the agglomeration, and to help the analysis and understanding of the agglomeration. Figure 7 illustrates the graphic representation of an *hierarchical* agglomeration. In this graph, the blue nodes represent smelly classes, while the red arrows represent inheritance relationships. Figure 8 illustrates an *intra-component* agglomeration graph. In the graph, a rectangle with a red outline represents the affected component while the nodes represented within the component are the smelly classes. In the same way, the *concern-overload* agglomeration graph (Fig. 9) also illustrates the component and smelly classes. In this graph, concerns are shown by hovering over the nodes

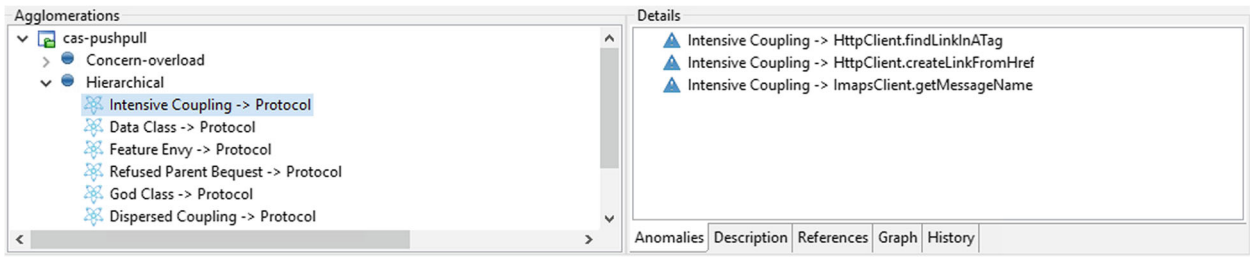

Fig. 4 Smells that compose an Agglomeration

representing the classes. The *intra-class* agglomeration graph (Fig. 10) shows a blue node that represents the agglomerated class, while the gray nodes represent the smelly methods of the class. Finally, the *intra-method* agglomeration graph (Fig. 11) shows a blue node representing the agglomerated method and red nodes to represent the name of smells affecting the method.

Providing historical information. During the analysis of an agglomeration, developers may benefit from information about the evolution of an agglomeration across the source code history [30]. Therefore, the fifth tab of Organic displays historical information about the selected agglomeration. By historical information, we mean information about the agglomeration in previous versions of the software. Thus, this historical information is organized by versions. For example, in the Fig. 12, information about the agglomeration is displayed in two previous versions: "0.2" and "0.5." Each version shows the code smells that were members of the agglomeration.

The objective of this tab is to assist the analysis of the evolution of each agglomeration throughout the different versions of the system. This tab shows the history of code smells that progressively composed the agglomeration along the versions of the software. As one can be observe in Fig. 12, in version 0.2, the agglomeration was composed by four smells (1 Dispersed Coupling, 2 Feature Envy, and 1 Intensive Coupling). On the other hand, the same agglomeration was composed of three smells (1 Dispersed Coupling, 1 Feature Envy, and 1 Intensive Coupling) in version 0.5. Using this feature, developers can identify agglomerations that are growing or shrinking along the system's evolution. The analysis of this phenomenon can help developers identify certain design problems. For example, a developer can check (i) if the number of smelly clients of a specific interface (all taking part in the agglomeration) is growing (or not) along the project history and (ii) if the agglomeration started from a smell affecting an interface in the program. These observations will help the developer to confirm if the design problem is located in that interface.

Allowing flexible thresholds. Similarly to conventional detection strategies for code smells [28], the detection of agglomerations also requires flexible thresholds. Therefore, to satisfy this requirement, Organic has a configuration screen (Fig. 13) that can be accessed through the *Window* → *Preferences* menu. The purpose of this screen is to allow users to define the threshold

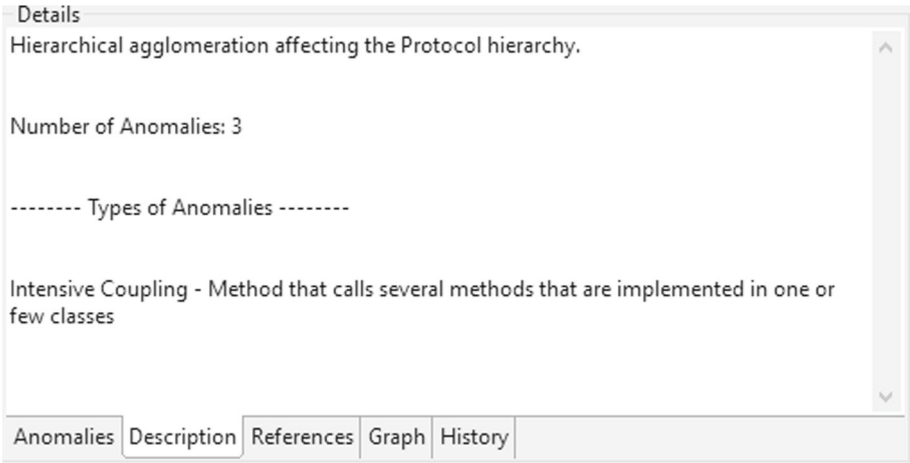

Fig. 5 Description of an Agglomeration

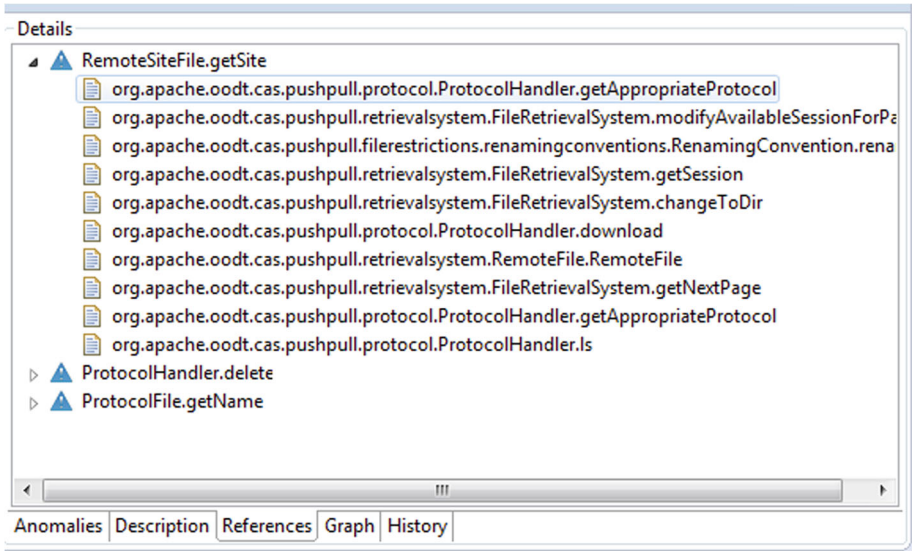

Fig. 6 References of an Agglomeration

for each agglomeration category. The threshold defines the minimum number of code smell that should be in a program location before being considered as an agglomeration. For example, if we configure the intra-method category with threshold 2, Organic will only find agglomerations in methods that contain 3 or more code smells.

Study I: Quasi-experiment

This work is intended to address three main goals, which are to (1) provide proper support for the analysis of stinkier code, (2) assess to what extent developers are able to identify design problems in stinkier code, and (3) identify tool issues that may hinder the identification of design problems.

In the previous section, we proposed the Organic tool, attending to our first goal. The "Study II: Communicability evaluation of Organic" section presents a study that addresses our third goal. Thus, aiming at achieving our second goal, in this section, we present a *quasi-experiment* with professional software developers. Quasi-experiment is an empirical interventional study in which the subjects are not randomly assigned to certain conditions [31]. In this study, we investigated whether the use of smell agglomerations improves the precision of developers in identifying design problems.

Study design

A previous study suggests that code smell agglomerations are related to occurrences of design problems [5].

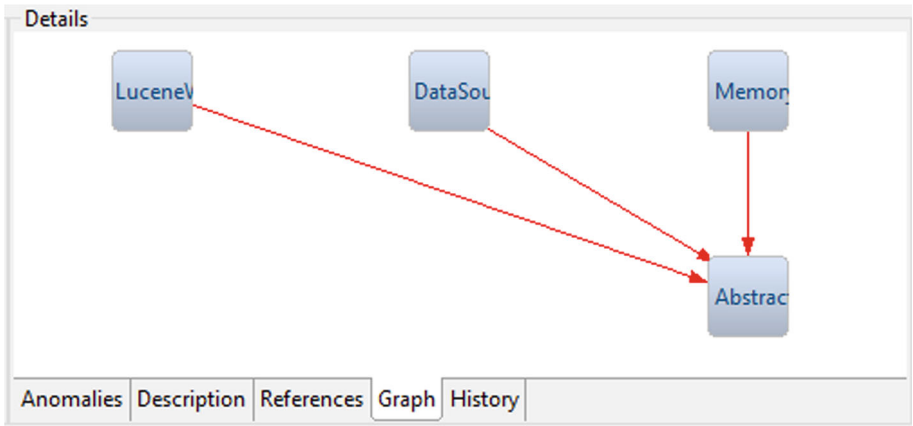

Fig. 7 Graph representation of a hierarchical agglomeration

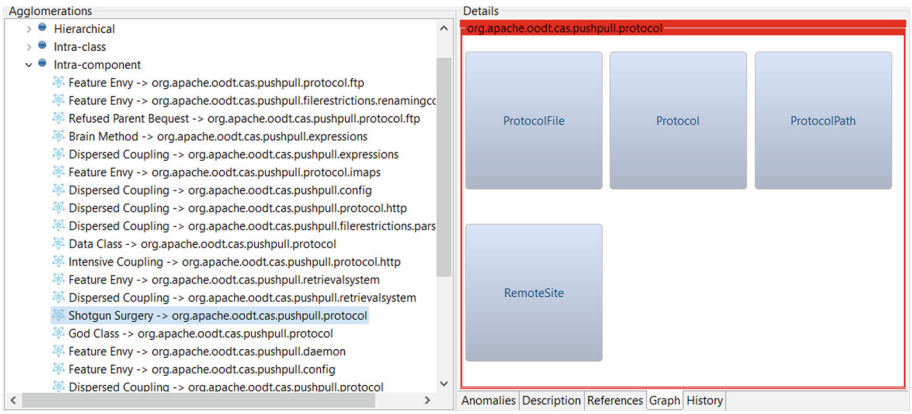

Fig. 8 Graph representation of an intra-component agglomeration

This study only analyzed the correlation between agglomerated smelly elements and code elements affected by design problems. However, such study did not show evidence that developers indeed identify design problems when exploring information about agglomerations. Thus, there is a need to investigate whether developers can, by themselves, identify design problems when exploring smell agglomerations. In order to address this matter, we defined two research questions. The first one is presented as follow:

> RQ1. Does the use of agglomerations improve the precision of developers in identifying design problems?

Research question RQ1 allows us to analyze whether code smell agglomerations help developers to identify design problems. To answer this question, we conducted a *quasi-experiment* with 11 professional developers. In this quasi-experiment, we measured the precision of developers using agglomerations to identify design problems. Precision in our context is measured based on the percentage of true positives indicated by the developers— i.e., the percentage of correctly identified design problems (against a ground truth, as explained later). We have used precision since it is an important aspect of the identification task.

Through the precise identification of design problems, developers are able to optimize their work by solving problems that really impact design. On the other hand, the lack of precision would lead software development teams to spend time and budget with irrelevant refactoring tasks. Refactoring is a transformation used for improving the structural quality of a system while preserving its observable behavior [3]. In companies adept of code review practices [32], the lack of precision in identifying design problems can lead developers to waste time on refactorings that do not contribute to improving software maintainability, or even refactorings that are harmful to the software design [12]. The precise identification of design problems is also important in open source projects. For instance, the contributions of eventual collaborators are often rejected by core developers due to the presence of design problems [33]. Therefore, in this case, lack

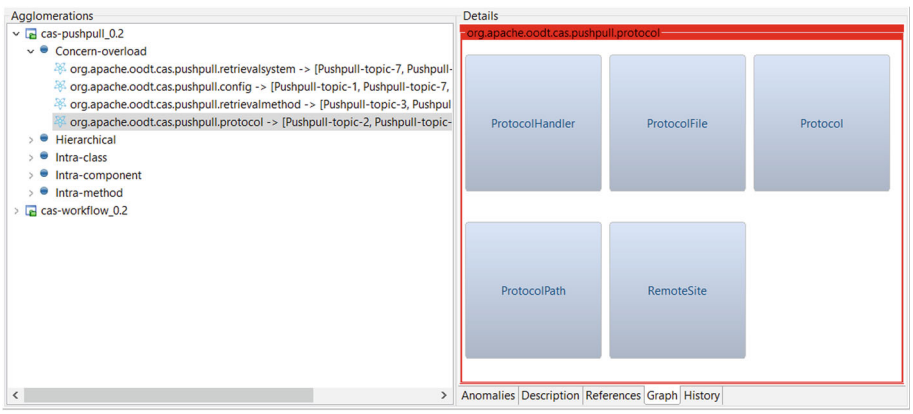

Fig. 9 Graph representation of a concern-overload agglomeration

Fig. 10 Graph representation of an intra-class agglomeration

of precision could lead core developers to reject relevant contributions due to "false design problems."

Someone could wonder why we have not measured recall. Although we agree to the relevance of measuring recall, we did not measure it. The reason is because the analyzed systems have a high number of design problems unfeasible to be identified during the quasi-experiment, which should not last (in total) for more than 90 min. Therefore, the consideration of recall would not be feasible in a quasi-experiment as the developers have limited time to search for some design problems only. Together with the system's original developers, we created a ground truth of design problems with more than 150 instances of design problems. Hence, it would be impracticable for participants to find all the design problems in the system due to the time constraints in the study (45 min). Consequently, they were expected to reach a low recall value. Therefore, we focused on the precision.

In order to measure whether precision improved or not, we compared the participants using agglomerations with a control group. The control group comprises of participants identifying design problems with a flat list of smells, i.e., code smells presented individually without showing their connection with other smells in the program. In the comparison, we used a ground truth to confirm or refute each design problem indicated by participants. Then, we compared the number of false positives and true positives produced with the code smell agglomerations against the number of false and true positives produced by the control group. In the context of this study, a false positive occurs when a participant reports a design problem that is not confirmed by our ground truth analysis. On the other hand, a true positive occurs when the design problem is confirmed during the ground truth analysis.

Someone could assume that developers would often benefit from the use of agglomerations in their quest for design problems. However, we do not have evidence about such benefit. Hence, we need to address RQ1 to verify if developers can correctly identify more design problems using smell agglomerations. Regardless of RQ1 results, another question involves the understanding of how to better support developers in exploring smell agglomerations. The success of developers on identifying design problems through agglomerations may strongly depend

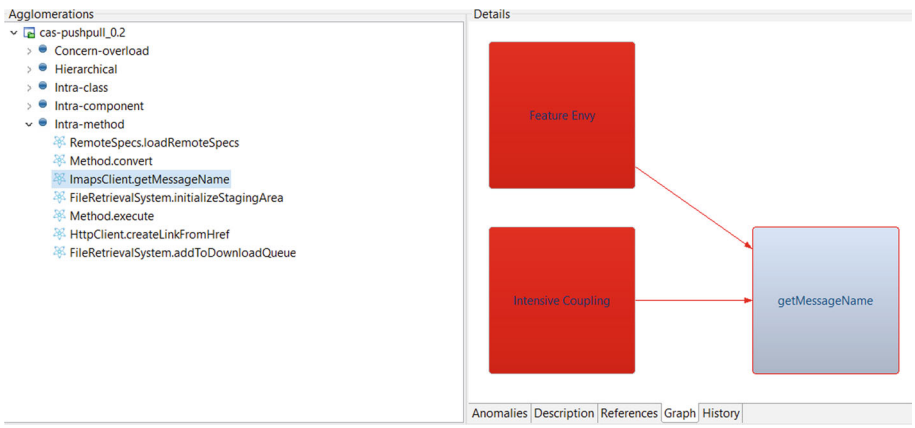

Fig. 11 Graph representation of an intra-method agglomeration

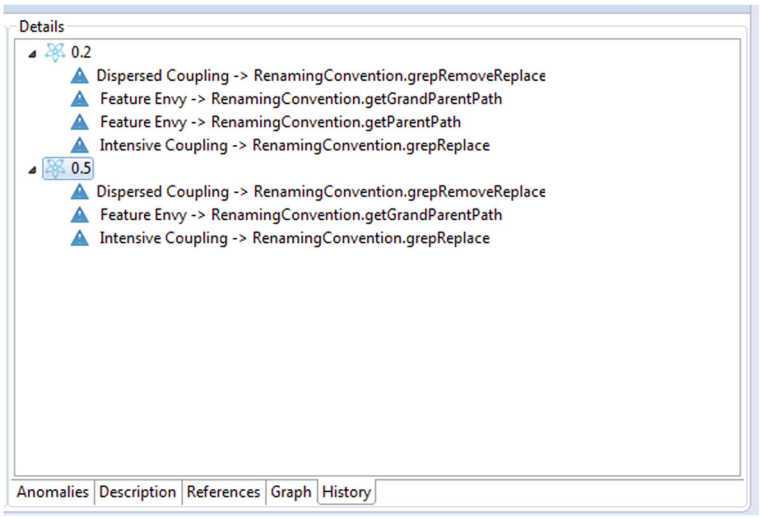

Fig. 12 Historical information of an agglomeration

on additional support for this task. Even though a previous study [5] has shown the strong relation between design problems and code smells within an agglomeration, we do not know whether and how the identification of design problems with agglomerations can be improved with additional support. The following question addresses this matter:

RQ2. How can the identification of design problems with code smell agglomerations be improved?

We conducted a qualitative analysis to address RQ2. This analysis was based on the observation of participants during the quasi-experiment and a follow-up questionnaire. As reported in the "Results and analysis"

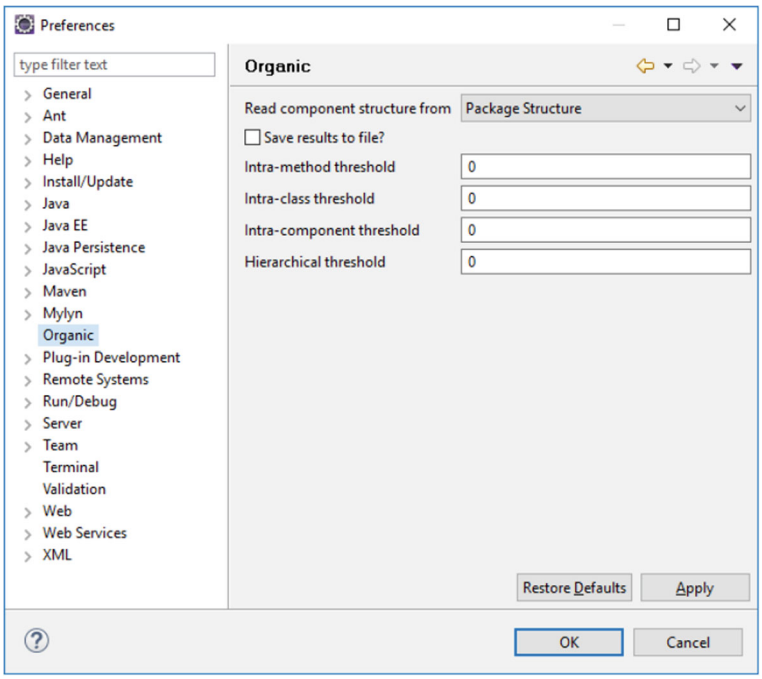

Fig. 13 Configuration of the tool

section, the combination of quantitative and qualitative analyses helped us to draw more well grounded conclusions. Thus, RQ1 can inform us if developers become (or not) more precise on identifying design problems when they use agglomeration, while RQ2 can provide a complementary perspective to explain why developers succedded or struggled to precisely identify design problems. For instance, RQ2 can reveal the benefits and barriers associated with the use of smell agglomerations.

Experimental procedure

We had to define a set of requirements in order to answer our research questions. Thus, we opted for conducting a quasi-experiment [31]. A quasi-experiment is an empirical study in which the units or groups are not assigned to conditions randomly. This allowed us to assign each participant to different treatments during the experimental steps. The experimental procedure was conducted individually with each participant. They had to perform the quasi-experiment in two steps with four tasks in each one. Both steps comprise the same set of tasks the only difference between them was regarding the treatment, i.e., usage of agglomerations or non-agglomerated smells.

As explained before, we compare developers using agglomeration with a control group using non-agglomerated smells. Thus, we divided the participants into two groups. The first group would identify design problems using agglomerations in the first step. After that, they would identify design problems using non-agglomerated smells in the second step. The second group of participants would make the identification inversely: using the non-agglomerated smells in the first step and, then, using the agglomerations in the second step. Thus, in each step, we have two groups of participants: a group using agglomerations and a control group.

As each participant identifies design problems twice (first and second step), we had to select two software projects. Thus, each participant could identify design problems using a different project in both steps. Another reason for providing two software projects is to avoid bias with the learning curve. For example, supposing that the participant uses the same project in both steps, she could find more problems in the second step than in the first step. That could happen because she can identify in the second step the same problems that she identified in the first step, plus other design problems identified only in the second step. This increase in the number of design problems found in the second step would not be due to the use of agglomerations, but rather due to the knowledge acquired by the participant.

There are four possible combinations with the participants based on the distribution between steps and software projects. Therefore, all participants were divided into four mutually exclusive arranges. Table 3 presents the cross design for the four arranges. The agglomeration group represents the group of participants that identified design problems using the agglomerations, and the control group comprises the participants that identified design problems using the list of non-agglomerated code smells.

Table 3 Combinations of groups, projects and steps

	Step 1		Step 2	
Arrange	Group	Project	Group	Project
1	Agglomeration	Project 1	Control	Project 2
2	Agglomeration	Project 2	Control	Project 1
3	Control	Project 1	Agglomeration	Project 2
4	Control	Project 2	Agglomeration	Project 1

The study was composed by a set of six activities distributed into three phases, as represented in Fig. 14 described as follows.

Activity 1: Apply the questionnaire for subjects' characterization. The subjects' characterization questionnaire is composed of questions to characterize each participant, including academic degree, professional experience with Java programming, background on code smells, and Eclipse IDE.

Activity 2: Training session. After defining the order of execution of each step, the next step was to provide a training session for the participants. The main objective of the training session was to level the participant at the same background required to understand and properly execute the experimental tasks. Thus, they received training about basic concepts and terminologies. This training was given only once for each participant before the first steps of the quasi-experiment. The training consisted of a 15-min presentation that covered the following topics: software design, code smells, and design problems. The training session took approximately 15 min, and the participants could make any question throughout it.

After the training, subjects received some artifacts that could be used during the study. They received a list with a brief description of the types of design problems presented in the training session. They also received a list with the description of basic principles of object-oriented programming and software design. They received a document containing (i) a brief description of both project systems and (ii) a very high level description of their design blueprint. We gave these documents because when they have to conduct perfective maintenance tasks, they need to have some minimal information about the systems to be maintained. The design blueprint represented the high-level design in the view of the project managers, but it was not detailed enough to support the identification of

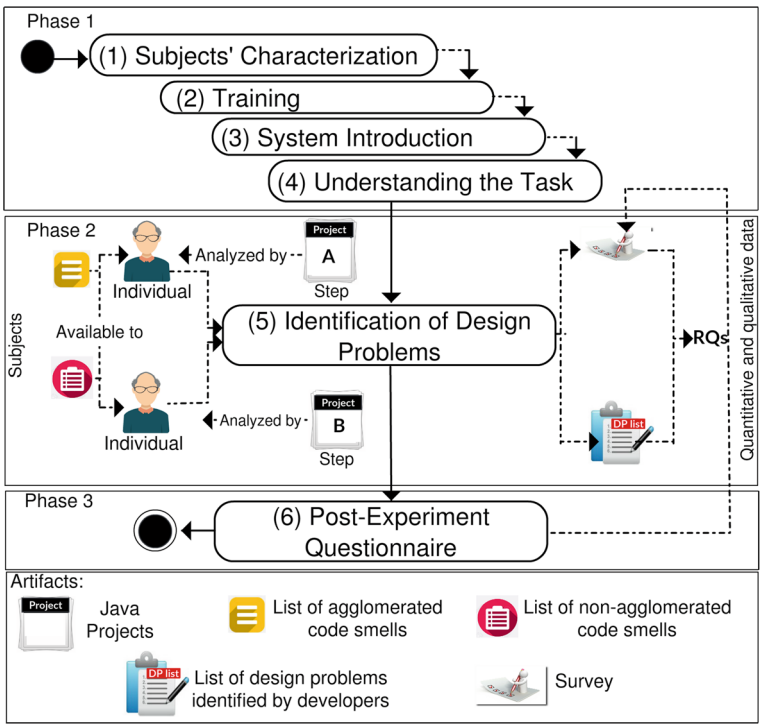

Fig. 14 The experimental design

design problems. As it often occurs in practice, the analysis of the source code is inevitably required to identify a design problem.

Activity 3: System introduction. We asked the participants to read the document containing the description of the project in which they would identify design problems. They had 20 min to read the description and the design blueprint of the system. Thus, they could start the identification with a certain level of familiarity with the software project.

Activity 4: Understanding the task. In this activity, we explained how the participant could use the Organic tool to collect either the list of agglomerations or the list of (non-agglomerated) code smells. As the Organic tool was developed as an Eclipse plug-in, we explained each one of the sections displayed in the Eclipse IDE and that was related to the Organic tool. This activity lasted approximately 10 min.

Activity 5: Identification of design problems. In this activity, the participant had 45 min to identify design problems in the project. We emphasized to the participant the importance of achieving the key goal of finding design problems. For each identified design problem, the participant was asked to provide the following information: (i) short description of the problem, (ii) possible consequences caused by the problem, (iii) classes, methods, or packages realizing the design problem in the source code, and (iv) the category(s) of agglomerations—described in "Basic concepts" section—that helped him to identify the design problems. If the participant was identifying design problems as part of the control group, she needed to provide almost the same information; the difference was that instead of providing the agglomeration (and its category), she needed to provide the code smells that she used to identify the design problem. For conducting this task, participants were instructed to use only the information provided by Organic in the current phase. This means that, neither the control group had access to the list of agglomerations, nor the agglomeration group had access to the list of non-agglomerated smells. This was guaranteed by providing different versions of Organic for each group—that is, one version for agglomerations and another version for non-agglomerated code smells. Nevertheless, both the project source code and the information provided by Organic (agglomerated or non-agglomerated smells) could be freely explored and analyzed during the design problem identification.

Activity 6: Follow-up questionnaire. In this activity, the participant received a feedback form. This form provides a list of questions, which enables the participant to expose her opinion on the identification of design problems. More details about this activity are provided in the "Qualitative analysis procedure" section.

After the sixth activity had been completed, we asked the same participant to repeat all tasks in the second phase.

Software projects and participant selection

In order to conduct the quasi-experiment as explained in the previous section, we selected two software systems in which developers had to identify design problems. We selected two programs that represent components of the Apache OODT project [27]: *Push Pull* and *Workflow Manager*. We selected subsystems of the OODT project since it is a large heterogeneous system; then, we could choose subsystems based on their diversity. Also, the Apache OODT project has a well-defined set of design problems previously identified by developers who actually implemented the systems [5], thus avoiding the introduction of false positive design problems in the ground truth. In addition, the OODT project was developed for NASA, used in other studies [4, 5, 17, 18, 24] and with a global community involved in its development. Table 4 presents the characteristics of each project. The columns of this table are organized as follows. The second column shows the project size in Source Lines of Code (SLOC), the third column presents the number of classes, and the fourth column contains the number of agglomerations in the project. A brief description of the project systems is presented as follows:

- Push Pull: it is the OODT component responsible for downloading remote content (pull) or accepting the delivery of remote content (push) to a local staging area.
- Workflow Manager: it is a component that is part of the OODT client-server system. It is responsible for describing, executing, and monitoring workflows.

After choosing the projects, our next step was to recruit developers for the study. Thus, we sent a characterization questionnaire for a group of developers of our network. Their answers were analyzed to determine which of them were eligible to participate in the study based on the following requirements:

- Four years or more of experience with software development and maintenance. We have chosen four years because this is the average time used by companies such as Yahoo [34] and Twitter [35] to classify a developer as experienced.

Table 4 Characteristics of software projects

Project	Size (SLOC)	# Classes	# Agglomerations
Push pull	11213	133	49
Workflow	18505	150	111

Table 5 Knowledge classification

Classification	Description
None	I have never heard about it
Minimum	I have heard about it, but I do not use it
Basic	I have a general understanding, but almost never use it
Intermediary	I have a good understanding, and use basic features sometimes
Advanced	I have a deep understanding, and often use advanced features
Expert	I am a specialist in this topic, and I use many features almost every day

- No previous knowledge about Push Pull and Workflow Manager.
- At least with basic knowledge about code smells.
- At least with intermediary knowledge on Java programming and Eclipse IDE.

We defined the knowledge in each topic based on a scale composed of six levels: none, minimum, basic, intermediary, advanced, and expert. Table 5 presents the description of such classification. We included in the questionnaire a description of each level, allowing the subjects to have a similar interpretation of the answers. Table 6 summarizes the characteristics of each selected developer.

Quantitative analysis procedure

In order to answer research question RQ1, we asked the study participants to analyze two systems with the aim of identifying design problems as described above. For each

Table 6 Characterization of the participants

ID	Experience in years	Education level	Knowledge		
			Java	Code smells	Eclipse
P1	5	PhD	Advanced	Advanced	Advanced
P2	6	Graduate	Advanced	Basic	Advanced
P3	8	Master	Advanced	Intermediary	Advanced
P4	4	Graduate	Intermediary	Basic	Basic
P5	5	Master	Advanced	Intermediary	Intermediary
P6	5	Graduate	Intermediary	Intermediary	Intermediary
P7	12	Graduate	Expert	Advanced	Expert
P8	5	Graduate	Advanced	Advanced	Advanced
P9	10	Graduate	Intermediary	Intermediary	Intermediary
P10	4	PhD	Advanced	Intermediary	Advanced
P11	5	PhD	Advanced	Intermediary	Advanced

system, we analyzed the precision of participants regarding the identification of design problems. The precision of participants was measured based on *true positives* (TP) and *false positives* (FP). In this context, a true positive is a candidate of design problem, as indicated by the participant, that was confirmed by a ground truth analysis. On the other hand, a false positive is a candidate of design problem that was not confirmed in the ground truth analysis. Thus, the precision is calculated using the following formula:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \quad (1)$$

We had to validate the identified design problems as true positive or false positive for each one of the analyzed systems. However, we could not argue that a design problem was correct or not since we were not involved with the design of each system. Thus, we relied on the knowledge of the systems' original designers and developers to help us in validating the design problems. We certified they were the people who had the deepest knowledge of the design of the investigated projects. We highlight these designers and developers were not subjects of this study.

We performed two steps to incrementally develop the ground truth. First, we asked original OODT designers and developers to provide us a list of design problems affecting the systems. They listed the problems and explained the relevance of each one through a questionnaire. They also described which code elements were contributing to the realization of each design problem. Second, we identified some design problems using a suite of design recovery tools [36]. We asked the developers of the systems to validate and combine our additional design problems with their list. The procedure for the additional identification was the following: (i) an initial list of design problems was identified using a method presented by Macia and colleagues [18], (ii) the developers had to confirm, refute, or expand the list, (iii) the developers provided a brief explanation of the relevance of each design problem, and (iv) when we suspected there was still inaccuracies in the list of design problems, we discussed with them. In the end, we had the ground truth of design problems validated by the original designers and developers.

Qualitative analysis procedure

Our first research question aims to investigate the precision of developers in the identification of design problems with agglomerations. Answer for such question will indicate whether developers benefit or not of using agglomerations. However, answering this questions is not enough for revealing the reasons why agglomerations may benefit developers. Moreover, we will not know how to improve the use of agglomerations to identify design problems. Therefore, we conducted a qualitative analysis to investigate what should be improved from the perspective of professional software developers. Besides identifying possible improvements, this analysis also helped us to understand what are the main strengths of exploring agglomerations for design problem identification.

As described in the "Experimental procedure" section, we asked the participants to provide us feedback about the identification of design problems. They answered a follow-up questionnaire, and we use their answers to conduct a qualitative analysis. The objective of the questionnaire was to gather participant's opinion regarding (i) the (dis)advantages of using the agglomerations or code smells to identify design problems, (ii) whether the provided information could be easily understood, (iii) which types of information were fundamental to identify design problems, (iv) what she believes that should be done to improve the identification of design problems, (v) what she thought about the use of the code smells for the identification of design problems, (vi) how the visualization mechanism provided by the Organic tool affected her performance, and (vii) which types of code smell and categories of agglomeration were the most useful for identifying design problems. The results of this questionnaire helped us to answer research question RQ2.

By applying the questionnaire, we were able to gather the opinion of developers regarding the use of code smell agglomerations. However, as reported by Easterbrook and colleagues [37], what is reported in the questionnaire may not be what actually happens in practice. Therefore, to obtain more reliable results, we also observed the participants of our study during the identification of design problems. This observation was performed during the study and also in analyses after it, through video and audio recorded during the quasi-experiement. This analysis allowed us to look at code smell agglomerations from the standpoint of professional software developers. It is important to note that the observation of participants during the quasi-experiment does not replace nor invalidate the questionnaire responses. In fact, the combination of observations and responses helped us to obtain a deeper understanding and interpretation on the results observed in the study.

Results and analysis

The results of this study are organized in two sub-sections. The "Do agglomerations improve precision?" section presents the results of our quantitative analysis regarding research question RQ1. The "How to improve design problem identification?" section provides the results of our qualitative analysis to answer research question RQ2.

Table 7 Precision

ID	Agglomeration group			Control group		
	TP	FP	Precision (%)	TP	FP	Precision (%)
1	2	1	66.67	1	1	50
2	0	3	0	1	4	20
3	3	2	60	1	4	20
4	2	0	100	1	3	25
5	4	0	100	3	1	75
6	1	0	100	1	0	100
7	1	1	50	1	1	50
8	3	0	100	3	0	100
9	0	1	0	0	6	0
10	0	0	–	1	1	50
11	0	1	0	0	0	–
All	16	9	64	13	21	38.24

Do agglomerations improve precision?

As described in "Quantitative analysis procedure" section, we conducted a quantitative analysis to answer our first research question: *Does the use of agglomerations improve the precision of developers in identifying design problems?* Table 7 presents the precision results for each participant (rows). The first column (ID) shows the identification number of each participant. The second column (Agglomeration group) presents the true positives (TP), false positives (FP), and precision for the participants when they were provided with agglomerations to identify design problems. Similarly, the third column (Control group) presents the true positives (TP), false positives (FP), and precision for the participants in the control group, i.e., when they were provided with non-agglomerated smells.

Agglomeration led to a slight increase of true positives. We can see in Table 7 that the developers identified a few more design problems (TPs) when they were in the agglomeration group (16 TP design problems) than when they were in the control group (13 TP design problems). As far as the per-subject analysis is concerned, four developers (light gray rows) identified more true positives when they used agglomerations than when they used the list of code smells in the control group. The use of agglomerations outperformed the use of smells in these four cases. On the other hand, two participants (2, 10) did not identify any true positive using the agglomerations, but they identified a true positive each in the control group. The other participants (6, 7, 8, 9, and 11) identified the same number of true positives (5 TP design problems) regardless the group.

Upon qualitative analysis, we were able to reveal the main reason why the four developers in the light gray rows identified more true positive design problems in the agglomeration group than in the control group. As illustrated in the example in Fig. 3 ("Identifying design problem in stinky code" section), these four participants systematically used each agglomeration's smell as an indicator of the presence of a design problem. They analyzed each one of the code smells as a complementary symptom of the presence of a design problem, which gave them increasing confidence to confirm the occurrence of the design problem. Surprisingly, we noticed the same behavior for the participant 8 even when she was in the control group. She was capable of agglomerating the code smells on her own, starting from the individual smells given in the flat list. Then, she used such agglomerations to identify design problems in the control group. This may be the reason why she reached a precision value of 100% in both groups.

Agglomerations help developers to avoid false positives. In general, developers identified less false positives when they used agglomerations (9 FP design problems) than when they used the list of code smells (21 FP design problems). As presented in our qualitative analysis ("How to improve design problem identification?" section), with the exception of participant 11, who analyzed several irrelevant agglomerations—i.e., agglomerations that do not reveal any design problem—all others identified either less or equal number of false positives when they were in the agglomeration group than when they were in the control group. When we analyze the control group, we can notice that more than half of the identified design problems are false positives (61.76%) while the agglomeration group identified only 36% of false positives.

After observing how developers identify design problems in the control group, we noticed that they did not go further with the analysis of the elements. Usually, a developer needs to analyze other classes in order to gather more information that can potentially indicate a design problem as discussed in the "Identifying design problem in stinky code" section. When the participants used the agglomerations, they analyzed multiple elements because they analyzed each code smell within the agglomeration even when the smells were in different elements. This behavior did not happen when participants were in the control group. In most of the cases, the participants in the control group analyzed only one code smell, which increased the likelihood of reporting false positives. Then, they reported a design problem in the class due to the presence of the code smell. However, some code smells are not related to any design problem; thus, the developer can report a false positive if she mistakenly considers a code smell that is not related to a design problem. That explains why developers in the control group found so many false positives. As developers tend to look at all agglomeration smells before reporting a design problem, the likelihood of reporting a false positive decreases, even when there is a code smell that is a false positive by itself.

Agglomerations improve the precision. Even though we cannot claim a statistical significance in our results due to the sample size of this study, we can notice that developers achieve a higher precision (64%) when they use agglomerations than when they use code smells (38.24%). Therefore, this result suggests that agglomerations may improve the precision of developers in identifying design problems, answering our first research question. However, someone could expect that *all* developers using agglomerations would significantly outperform the control group. As a matter of fact, we noticed some factors that explain, at least partially, why developers did not find much more design problems when they were in the agglomeration group than when they were in the control group. These factors are presented next, and they are useful to discover improvements for the identification of design problem with the analysis of stinky program locations.

How to improve design problem identification?

This section presents the answer for our second research question: *How can the identification of design problems with code smell agglomerations be improved?* We conducted a qualitative analysis to answer this question. As described in the "Qualitative analysis procedure" section, this analysis was based on the observation of participants during the identification of design problems as well as the analysis of responses to our follow-up questionnaire.

Where to start from? As discussed in the previous section, the participants identified few more true positives using agglomerations. Someone could expect that *all* developers using agglomerations would significantly outperform the control group. However, we observed that participants spent much more time analyzing the agglomerations than analyzing the smells in the control group. That happened because they analyzed each code smell in the stinky program location as previously explained in the "Do agglomerations improve precision?" section. Furthermore, sometimes the participants analyzed agglomerations that were not related to any design problem, which is a key factor that explains the almost same number of true positives between both groups.

Unfortunately, almost all the participants analyzed irrelevant agglomerations. Participants 6, 9, 10, and 11 were the ones that suffered the most from the analysis of irrelevant agglomerations. Since these four participants faced this issue, they suggested in our follow-up questionnaire that the Organic tool should provide means to prioritize (or, at least, rank) potentially relevant agglomerations. This feature would help to further reduce the time spent with the analysis of irrelevant stinky code. Thus, this issue also helps us to explain why they fell short in identifying additional design problems through the analysis of agglomerations.

Need for prioritizing agglomerations. The aforementioned need for agglomeration prioritization shows that the time and effort required to identify design problems is a key factor for developers; thus, prioritization should be taken into consideration. As a matter of fact, the prioritization of smelly code has been the focus of recent research [38–40]. The work of Vidal et al. [40], for example, proposed and assessed a set of criteria for prioritizing smell agglomerations. As they have observed, the prioritized list of agglomerations would help a developer to progressively analyze the agglomerations that are more likely to represent design problems, discarding the irrelevant ones. This would be especially useful in large legacy systems, in which thousands of agglomerations may be detected. Nevertheless, they observed there is no prioritization criterion that is always the most effective one for all the analyzed systems [40]. They provided some hints on how a developer could choose a promising prioritization criterion for her project.

Based on our qualitative analysis, we noticed that existing criteria for prioritization should select agglomerations that are cohesive. In our context, an agglomeration is considered to be cohesive whenever all its code smells are related to the same design problem. If there is one code smell that is not related to the design problem (i) in the best case, the developer will spend time analyzing a code smell that is useless to identify the design problem or (ii) in the worst case, such smell may direct the developer away from the design problem. This fact suggests that developers need accurate algorithms to find cohesive agglomerations and to discard the less cohesive ones. However, prioritization algorithms based on existing criteria are unable to do this as far we are concerned. Consequently, the prioritization of stinky program locations still poses as a challenging research topic. Therefore, after this study, we decided to not incorporate existing prioritization criteria into Organic. Before including any prioritization feature into Organic, we intend to propose and evaluate improvements for the existing prioritization criteria.

Stinky code analysis is challenging. Besides the prioritization issue, participants also suffered to analyze large agglomerations. As reported in the "Do agglomerations improve precision?" section, this problem was even worse for agglomerations affecting larger program scopes, i.e., agglomerations crosscutting implementation packages or class hierarchies. We noticed that a large agglomeration requires that developers reason about a wide range of scattered code smells. As they tend to use each code smell as a symptom of design problem, they have difficulties to correlate the multiple symptoms of an agglomeration. This is a challenging task because the higher the number of code elements involved in an agglomeration, the greater is the volume of code that must be analyzed.

Consequently, developers will have more code to analyze, which increases the complexity of the analysis.

Need for proper visualization mechanisms. In order to alleviate the analysis of stinky code, some participants suggested the adoption of better visualization mechanisms. For instance, participant number 8 suggested the visualization of agglomerations through a graph-based representation [41]. She mentioned that such visualization would provide an abstract and general view of agglomerations. The main advantage of this form of visualization is that the more abstract a representation is, the less details will be displayed for analysis. Consequently, the developers would not be overloaded with details. At the same time, an abstract representation like the graph-based visualization would help developers to see the full extent of an agglomeration. After providing an abstract view, a visualization mechanism could allow developers to progressively explore the agglomeration details such as the types of smells, location of stinky code, and relationships among smells. In order to address this matter, we incorporated a graph-based visualization mechanism into Organic.

Identification of the design problem type. The difficulty in analyzing agglomerations also raised the need for recommendations on which types of design problem each smell agglomeration is more likely to indicate. These recommendations would reduce the effort required to decide whether the elements are affected by a specific design problem. For example, the agglomeration of Fig. 3 occurs in classes of the same hierarchy that are implementing the *WorkflowRepository* interface. All smelly elements of this stinky program location manifest the same type of smell, which is the Feature Envy. The occurrence of multiple Feature Envies in a single hierarchy suggests that there is a problem, in a root abstract class or a root interface, which is spreading through all the subclasses of the hierarchy. Therefore, to help developers to decide whether there is a problem or not, the Organic tool could suggest the analysis of this hierarchical agglomeration trying to identify problems like Ambiguous Interface [21] and Fat Interface [9], for example.

Suggestions of design problem types can help developers to focus their attention in specific characteristics of the suggested design problems. However, this kind of recommendation algorithm requires multiple empirical studies to understand how and when each form of agglomeration may consistently represent specific types of design problem. This is indeed a challenging research topic to be addressed in the future and, therefore, we are unable to provide this recommendation feature in the Organic tool.

Threats to validity

This section presents some threats that could limit the validity of our main findings. For each threat, we present the actions taken to mitigate its impact on the research results.

The first two threats to validity are related to the number of participants in the study and to the convenience approach used to find participants. We have selected a sample of 11 participants, which may not be enough to achieve conclusive results. However, instead of drawing conclusions based on the quantitative results, we complemented our analysis with a qualitative analysis. In addition, we defined a set of requirements to selecting participants suitable for this study. Also, we conducted training sessions with all participants. Such sections aimed to resolve any gaps in the participants' knowledge and any terminology conflicts, allowing us to increase our confidence in the results.

The third threat is related to possible misunderstandings during the study. As we asked developers to conduct a specific software engineering task and to answer a questionnaire, they could have conducted the study different from what we asked. To mitigate this threat, we assisted the participants during the entire study, and we make sure of helping them to understand the experimental tasks and the questionnaire. We highlighted that our help was limited to only clarify the study in order to avoid some bias on our results.

Next threat concerns the ground truth used to confirm or to refute the design problems reported by participants. Since our ground truth was built manually, it is possible that some design problems are missing in the ground truth, which would impact the precision measure. To mitigate this threat, we built the ground truth with the help of original OODT designers and developers. Moreover, we relied on a suite of design recovery tools to identify possible design problems that were not reported by the original designers and developers of OODT.

There is another threat that is related to the amount of information we asked participants to provide for each design problem reported. Providing all information during the experiment may slow down the participants, and as a consequence, some participants may report fewer design problems than they would be able to do during the 45-min time frame. We mitigated this threat by asking the same amount of information for both the agglomeration group and the control group.

Finally, there are two threats concerning the selected projects. The first one is about the difficulty of the participants in understanding the source code used in the experimental tasks. This difficulty appears due to the complexity of the source code and time constraints to complete each task. The second threat is related to one software project that could be easier to identify design problem than the other. We minimized the first threat by running a pilot study to define an experimental time reasonable to perform the tasks. To minimize the second

threat, we selected projects with similar size, complexity, and number of known design problems. We also have trained all participants about each project. In addition, the cross design of our experiment allowed different combinations of project and technique. Finally, our results suggest no variation in difficulty for identifying design problems in the two projects.

Study II: Communicability evaluation of Organic

In the previous section, we presented a quasi-experiment, which allowed us to gather quantitative and qualitative results regarding design problem identification. In addition, we collected the opinion of developers regarding the use of code smell agglomerations. Nevertheless, the previous study did not evaluate Organic, which is the tool proposed in this paper for helping in the identification of design problems. Thus, we did not have shreds of evidence that indicate to what extent Organic affects the users during the identification of design problems. In order to understand these effects, we conducted a qualitative evaluation of the Organic tool. We opted for proposing and evaluating Organic because, to the extent of our knowledge, it is the only tool that meets the requirements ("Organic: a tool for the analysis of stinky code" section) for helping developers in the analysis of stinkier code.

To evaluate Organic, we applied the Communicability Evaluation Method (CEM) [42]. CEM is a qualitative evaluation method developed to capture communicability issues, which are problems that appear due to poor communication between users and a system, usually when users interact with a system. An example of communicability issue is when the user mistakenly believes that she performed a certain task on the system successfully. Another example is when the user does not understand the answers provided by the system. In our case, we are interested in communicability issues that happen when developers interact with the Organic tool. We have to investigate these communicability issues because they may hinder the identification of design problems when developers use the Organic tool.

CEM has been widely used in HCI (human-computer interaction) research to evaluate the communicability of software systems. This method is based on the theory of Semiotic Engineering [22] and is intended to find ruptures of communication when a user interacts with a system. Thus, in order to use CEM, we have to characterize the system and users in the context of our study. As explained in the Organic: a tool for the analysis of stinky code section, developers use the Organic tool to identify design problems. Hence, in the context of this study, the system is the Organic tool and the user is the software developer that uses Organic to identify design problems in stinky code. Therefore, the objective of this study is to use the CEM to find communicability issues in the Organic tool.

In the subsection below, we present details about this study. The "Study design" section presents the study design. The "Data analysis and evaluation procedure" section contains an overview of the evaluation procedure followed in this study. The "Results and interpretation" section presents the results. Finally, the "Threats to validity" section outlines the threats to validity of this study.

Study design

We defined our third research question to evaluate the Organic tool as follows:

> RQ3. Which are the communicability issues of Organic that hinder the identification of design problems?

To answer RQ3, we followed the procedure defined by the CEM. For being a method focused on user experience, CEM allowed us to look at the Organic tool from the standpoint of potential users, which are professional software developers. In this way, we can observe the aspects of the tool that affect the identification of design problems as if we were the users ourselves. Moreover, such observation was not fully accomplished by our previous study ("Study I: Quasi-experiment" section), since the primary goal there was to evaluate precision of developers when using the technique (code smell agglomerations) rather than the tool itself.

CEM requires the participation of potential users of the system under evaluation. Therefore, similarly to the previous study ("Software projects and participant selection" section), we selected participants for this study according to the following requirements:

- Minimum of 4 years experience with software development
- Intermediary knowledge about software design
- Advanced knowledge about the Java programming language
- Basic knowledge about the Eclipse development environment
- Basic knowledge about code smells

Requirements above are justified by the fact that Organic is part of a complex domain, which is the identification of design problems. Therefore, participants

Table 8 Profile of selected participants

Participant	Software design	Java programming language	Eclipse IDE	Code smells
1	Advanced	Advanced	Advanced	Advanced
2	Intermediary	Advanced	Advanced	Basic
3	Advanced	Advanced	Basic	Intermediary

that have a minimum knowledge about basic concepts have more chance of revealing communicability problems when using the tool. This happens since the influence of the domain complexity is mitigated by the experience and knowledge of participants. Based on the aforementioned requirements, we selected three participants for this study. On Table 8, we summarize the profile of each participant, presenting their experience with software design, Java programming language, Eclipse IDE, and code smells. Participant 1 is a developer with vast experience both in academic field and software industry. He has advanced knowledge about software design and code smells. Participant 2 is a professor from the computing field with 4 years of experience in the industry. He has intermediary knowledge about software design and basic knowledge about code smells. Finally, participant 3 had moderate experience in the industry, and he was also undergoing his postgraduate studies at the time of this evaluation. He has advanced knowledge about software design and intermediary knowledge about code smells. Table 8 summarizes the profile of all participants.

Test scenario

After selecting the participants, the definition of a test scenario is the next CEM procedure. Since Organic is designed to be applied to a single task, which is the identification of design problems, our test scenario is composed of one task as well. The task consists in:

> Using the Organic tool to search for design problems in the source code of a given software project.

In the context of this study, Apache OODT [27] is the selected software project. We selected Apache OODT due to the same reason explained at the "Software projects and participant selection" section. As defined by the CEM, this task was designed to last at most 30 min. During the execution of this task, for each design problem found, the participant should give the following information:

- Brief description of the problem.
- Classes and methods participating in the design problem.
- Tool resources that were useful to identify the problem.

During the identification task, besides using the Organic tool, the participants could consult three documents: (1) Apache OODT documentation, (2) a reference document about basic concepts (design problems, code smells, and the like), and (3) manual of the Organic tool. We provided these documents to help users to gather an understanding of the system. Consequently, they were able to focus on the task instead of wasting time trying to understand, for example, the system and basic concepts. This initial preparation was not part of the time frame of 30 min.

Environment and infrastructure

To perform this study, we used an individual room equipped with a computer containing the following hardware configuration: 8GB of RAM, CPU Intel Core i5 2.7GHz, GPU GeForce GT 740M, and built-in microphone in the notebook. In addition, we used the following softwares: Organic tool, Windows 10 Operational System, Java Development Kit 1.7, Eclipse Luna IDE, Rabbit Eclipse Plugin, and Screen record tool Active Presenter. The Eclipse Luna chosen as the Organic tool only works with this version at the moment. The Rabbit plugin registers information regarding the time spent using the resources of Eclipse (e.g., files, perspectives, and views). This information is useful to the analysis and interpretation of videos recorded using the Active Presenter.

Post-study interview

Using the interview pre-test, we collected data regarding the participant's profile. The questions of the interview post-test were developed individually for each participant. Thus, the evaluators, based on their observation, could explore the participants' answers. Additionally, the following questions were asked to all participants:

- What were the main difficulties to perform the task?
- What were the most useful tool resources?

Data analysis and evaluation procedure

In order to conduct our evaluation following the CEM, we collected the following data: (1) video from the computer screen with audio from the microphone, (2) report collected with the Rabbit Plugin, (3) annotations done during the execution of the tests, and (4) answers given during the interview.

After data collection, we followed three main steps, which are defined by the CEM [22]. They are (1) tagging, (2) interpretation, and (3) semiotic profile. On tagging, the researcher analyzes the recording of the task being performed, after that, she identifies the evidence of communicability failures. To each of these failures, she associates with one of the 13 tags defined by CEM [22]. On the interpretation, the researcher works with the tagged data, trying to identify the main communicability issues. The researcher then analyzes and organizes the collected evidence, according to some perspectives. Finally, in the semiotic profiling step, an in-depth characterization of metacommunication is achieved. The idea of these steps is to achieve higher levels of abstraction in our analysis and interpretation

of how the developers receive communication from the Organic tool [22].

Tagging. In this step, we analyzed and tagged the communicability failures that occurred during the interaction between the software developers and the Organic tool. A communicability failure is the result of a communicability issue. In our case, if the Organic tool contains a communicability issue, this issue will lead to a communicability failure observed when the user interacts with the tool. Thus, tagging was made according to what happened when the communicability failures were observed. To perform this step, we observed each participant during the task of identifying design problems, taking notes of possible communicability failures. After that, we analyzed the video and audio recorded during the task registering communicability issues according to the 13 tags defined by the CEM [22]. Whenever necessary, we consulted the Rabbit reports to confirm or to change the tags. For a detailed description of the tagging step, we refer to [22, 42]. Table 9 presents a brief description of the tags that occurred in this study.

Interpretation. In this step, we analyzed the tagged material aiming to identify the main communicability issues in the Organic tool. Based on the CEM [22], we analyzed and organized collected evidence based on three perspectives:

- Frequency and context of each communicability failure.
- Recurrent sequences of communicability failures.
- Identification of communicability issues that have caused the observed failures.

The analysis of frequency and context of communicability failures was helpful to discover the most frequent failures in the communication between the software developers and the Organic tool. Identifying recurrent sequences of failures helped us to discover the origins of communicability issues in Organic. Finally, the identification of communicability issues in the Organic tool is the main objective of this study, as defined by our main research question. To identify communicability issues, we classified tags as complete, partial, or temporary failures (first column of Table 9), following theoretical tag categorizations from Semiotic Engineering [22].

Complete failures occur when the developers are unable to identify any design problem with Organic and do not try again. Partial failures occur when the developer gives up from using Organic's functionalities before identifying any design problem and tries to achieve this in another way. Finally, temporary failures occur when the developer temporarily interrupts the identification of design problems with Organic due to some communicability issue. According to the CEM, there are three types of temporary failures: (1) trying to communicate, (2) trying to fix an error, and (3) trying to understand the rules.

Table 9 Description of CEM tags that occurred in this study

Classification	Tag	Description
Complete failure	I give up	The developer is unable to identify design problems with Organic either because he does not know how to or because he does not have enough time, or will, or patience for it.
Partial failure	I can do otherwise	The developer manages to identify design problems in a way that is not optimal. For example, without using the most functionalities provided by Organic.
Temporary failure—communicate	What now?	The developer searches for a clue of what to do next and not searching for a specific functionality that will help in the identification of design problems.
Temporary failure—understand the rules	What is this?	The developer seems to be exploring the tool to gain more (or some) understanding of what a specific functionality achieves.
	Help!	The developer deliberately calls a help function, using menus, dragging question marks, or asking for help.
	Why doesn't it?	The developer expects some sort of outcome from Organic, but does not achieve it. She steps through the path, again and again, to check that it is not working.

Semiotic profiling. In this last step, we conducted an analysis to understand the communication between the software developers and the Organic tool. After executing all steps defined by the CEM, we were able to look at Organic as if we were the users ourselves. This helped us to acquire a deeper understanding of Organic's communicability issues. In addition, looking from the perspective of potential users, we were able to identify the requirements for a tool that supports the identification of design problems in stinky code.

The steps defined by the CEM were fundamental for achieving the goal of this study: discover communicability issues in Organic that hinder the identification of design problems. The first step (tagging) provided a guideline for the identification of communicability failures that may occur when a user interacts with Organic. After that, with the interpretation step, CEM provided us with a systematic method for the analysis and classification of communicability failures. This classification was fundamental for organizing the data collected during this study. Finally, in the last step, we analyzed the data collected in previous steps to consolidate our results. During this step, we identified the main communicability issues of Organic based on the recurrent failures observed during the interaction of software developers with Organic. Next, we present the results and our interpretation of this study.

Results and interpretation

Table 10 presents the frequency of occurrence of each tag (rows) by participant (columns). Also, the total frequency, considering all participants, is summarized in the last column. Table 11 presents the frequency of communicability failures, categorized by the type of failure. We did not observe any recurring pattern of failures among the participant. Nevertheless, as seen on Table 11, all participants suffered from the "I give up" failure, which is a complete failure. For all of them, the complete failure occurred after successive temporary failures. As exposed in Table 11, most of the temporary failures were of type 3—that occur when the developer is trying to understand the communication rules of Organic. This sequence of failures indicates that developers tried to identify design problems with Organic. However, due to successive failures, the developers gave up on the task.

Communicability issues of Organic

Answering research question RQ3, we observed three main communicability issues in the Organic tool, which are (1) lack of a precise message, (2) inadequate terminology, and (3) ambiguity in static signs. Next, we present details about each of them.

Table 10 Frequency of occurrence total and by participant

Tag	Participant			Total
	P1	P2	P3	
I give up	1	1	1	3
Go another way	1	0	1	2
And now?	0	1	0	1
What is this?	1	4	2	7
Help!	2	8	2	12
Why doesn't it?	4	0	0	4

Table 11 Frequency of occurrence categorized by type of failure

Type of Failure	Participant			Total
	P1	P2	P3	
Complete failures	1	1	1	3
Partial failures	1	0	1	2
Temporary failures—type 1 communicate	0	1	0	1
Temporary failures—type 2 fix an error	0	0	0	0
Temporary failures—type 3 understand the rules	7	12	14	23

Lack of a precise message. Although the tool identifies and groups the symptoms that are interrelated (the agglomerated code smells), it does not provide a message that facilitates concise reasoning about the possible design problem. Hence, the developer needs, by himself, to explore and synthesize all the information needed to analyze a design problem. The tool gives the necessary information, but the analysis of those information requires a significant effort from the developer. Besides being a communicability issue, it also has relation with the domain complexity in which Organic is designed for. We list next some changes in Organic that can contribute for building and delivering a more precise message.

Following our findings from the previous study ("How to improve design problem identification?" section), for this study, we incorporated a graph-based view into Organic. However, we observed that developers did not use the graph-based view of Organic. In the post-study interviews, we noticed that this happened because the graph-based view did not prove to be useful for identifying design problems. Nevertheless, we believe that, after some improvements, this type of view can indeed contribute for transmitting a more concise message about each agglomeration. For that, a better integration of this view with other information is required. For example, instead of using a separated tab, the description of code smells could be provided in the graph nodes. Moreover, the graph could show the relationships of the agglomeration with external classes. This information could be provided on demand, when required by the developer. We believe that such improvements would help developers to conduct an integrated, smoother analysis of an agglomeration.

Inadequate terminology. The terms used in the tool are not adequate to the target public, which are common software developers. Terms such as "anomalies" and "agglomeration," for instance, are unknown by most developers. Two participants (P2 and P3) who have the least knowledge about software engineering mentioned the fact that the concepts embedded in the tool were too hard to

understand. The participant (P1) that did not have difficulties with the terms was a postgraduate student, acting on the software developer field.

To improve the communicability of Organic, participants suggested maximizing the use of terms from popular books like the books of Fowler [3] and Martin [43], which are widely known in the software development community. In addition, as observed in the quasi-experiment ("Study I: Quasi-experiment" section), for being a complex domain, developers would benefit from interactive help content. This kind of aid should be available, at least, in the first interaction between the developer and Organic.

Ambiguity in static signs. The last problem of communicability occurs due to the inadequate use of static symbols. As presented in Fig. 15, different types of information are presented with the same static symbols. This mixture confused all the participants, leading to situations in which, for instance, the participant believes that he is interacting with the tab "Anomalies," when in fact he was interacting with the tab "References." This is the least harmful communicability issue, but also affects the identification of design problems.

The direct solution for resolving this ambiguity consists on the use of different static symbols for each type of information. Also, the improvement on the graph visualization—mentioned to solve the first communicability issue—would be an excellent alternative to solve this problem. The graph would integrate the different information in a single view, removing existing ambiguities.

Communicability strengths of Organic

Despite presenting some communicability issues, Organic also has its strengths revealed in this study. In fact, there was no previous study evaluating Organic. Therefore, looking from the perspective of Semiotic Engineering, besides identifying communicability issues, we also identified some communicability strengths of Organic. This provided us with evidence on what is working well in Organic. It is important to note that the strengths presented here were not reported by participants. Instead, they are the result of our own observations.

Next, we present the main strengths found in Organic during this study:

Multiple analyses of stinky code. The identification of a design problem in stinky code requires multiple complementary analyses. Organic provides the opportunity to analyze stinky code based on different agglomeration categories ("Basic concepts" section). As reported by Oizumi and colleagues [5], each agglomeration category provides a different perspective to the analysis of source code. In addition, Organic provides multiple information about each agglomeration: (1) list of code smells, (2) description of code smells, (3) dependencies of the agglomeration with external classes, (4) a high level visualization, and (5) information about the agglomeration across different versions of the source code. We observed, in this study and in the quasi-experiment ("Study I: Quasi-experiment" section), that developers were able to identify design problems when they managed to explore and synthesize the multiple information provided by Organic for each agglomeration.

Integration with IDE. The analysis of stinky code requires the developer to constantly navigate from Organic to the source code, and vice-versa. This is necessary because most code smells can only be fully understood in the source code. Moreover, the analysis of source code is required to verify if a code smell is a false positive. As an Eclipse plugin, Organic promotes a smooth integration of its views with the source code. In addition, the developer can open the source code affected by a code smell with a double-click in the code smell. Without this integration, the developer would have to constantly shift between programs to analyze stinky code. For example, without this resource, the analysis of an agglomeration of five code smells would require, at least, nine shifts between programs.

Information about code smells. Most developers benefit from reading information about code smells during the analysis of stinky code. We observed in this study that, during the analysis of an agglomeration, even experienced developers usually consult the definition of each smell. As explained in the "Organic: a tool for the analysis of stinky code" section, Organic provides this information in a tab called "Description." We noted that developers used the "Description" tab (back and forth) as a guide for analyzing agglomerated code smells. Providing this information is important because even experienced developers do not know or remember the definition of all code smells. Thus, without this resource, they would spend more time analyzing an agglomeration and searching for information about code smells via external resources.

Threats to validity

This section presents threats that could impact the validity of this study. For each threat, we present the actions we took to mitigate its impact on the study.

First, one could claim it would be beneficial to have more participants in the study in order to achieve representative results. However, according to Yin [44], qualitative research is, by nature, particularistic. Thus, the analysis and understanding of qualitative results requires the study of specific situations and people, complemented by considering specific contextual conditions. We selected three software professionals, which are representative individuals of our target population. Thus, we

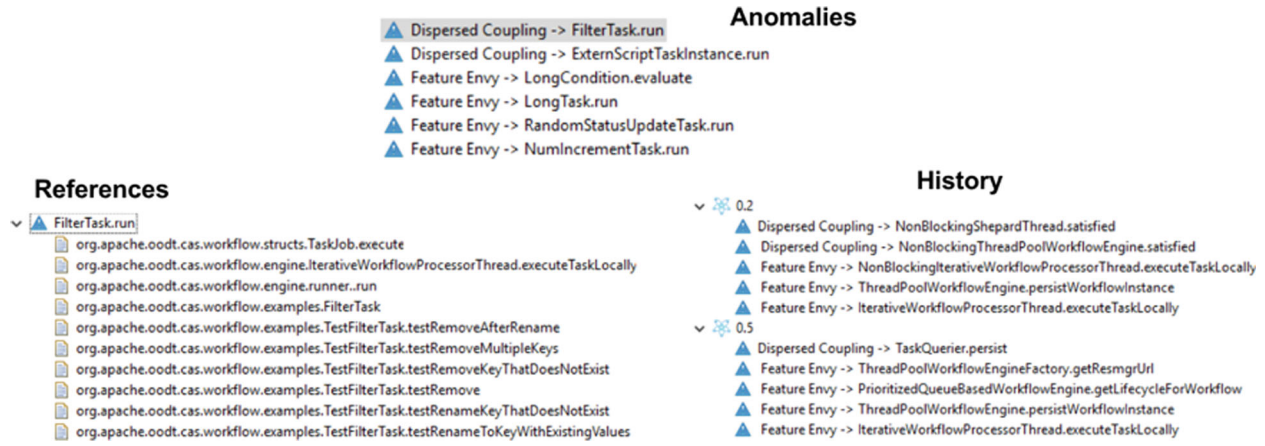

Fig. 15 Example of ambiguity in the static symbols

consider that this threat was properly mitigated. However, we plan to perform other studies in the future in order to analyze the behavior of professionals with other types of background and using future versions of Organic.

The second threat is related to possible misunderstandings during the study. To mitigate this threat, we prepared the participants before the study, explaining how the study would proceed. Moreover, in order to comply with the recommendations of CEM, two researchers assisted the participants during the entire study.

Finally, there is a threat related to the complexity of the system used in this study. We mitigated this threat by selecting system from a widely known computer science domain—Apache OODT is a middleware system that provides infrastructure services, such as file management and networking communication. In addition, as presented in the "Test scenario" section, we provided participants with the required documentation of Apache OODT.

Related work

In this section, we present the literature that is related to this paper. The "Identification of design problems with code smells" section presents studies investigating the occurrence of design problems in stinky code. The "Other approaches to identify design problems" section outlines studies that investigate the use of information other than code smells to identify design problems. The "Detection and visualization of stinky code" section provides a brief overview of the literature about detection and visualization of stinky code. Finally, the "Semiotic Engineering" section presents studies on the use of Semiotic Engineering in different research contexts.

Identification of design problems with code smells

In this paper, we have investigated if developers can identify design problems in stinkier code. In this investigation, we observed developers reasoning about code smell agglomerations to identify design problems. Oizumi et al. stated the concept of agglomeration [5]. In their study, they investigated to what extent code smells could "flock together" to realize a design problem. They found that these code smells flocked together because they were somehow related to each other. The groups of smells that flock together are called agglomerations. After analyzing more than 2200 agglomerations of code smells from seven software systems with different sizes and different domains, they concluded that certain forms of agglomerations are consistent indicators of design problems. Despite such result, Oizumi et al. did not evaluate if developers would identify most design problems when they use agglomerations.

Oizumi et al. study was not the only one to investigate inter-related code smells. Abbes et al. [6] brought up the notion of code smells that interact to each in the source code. They also investigated the effects of such interacting. They concluded that classes and methods identified as God Classes and God Methods in isolation did not affect maintenance effort; however, when these two smells appear together, they led to a statistically significant increase in maintenance effort. Yamashita and Moonen [7] also investigate the relationship between code smells. They observed that inter-smell relationships negatively affect the maintenance. Posteriorly, Yamashita et al. [8] studied collocated smells—code smells that interact in the same source code file—and coupled smells—code smells that interact across different source code files. Regarding software design, they observed that limiting the analysis to collocated smells would reduce their capability to reveal

design problems, as coupled smells may reveal critical design problems.

Macia et al. [17] analyzed the relevance of code smells to identify design problems. Their research revealed that none of the studied code smell types was consistently a strong indicator of design problems. Their results also revealed that a higher proportion of individual code smells did not impact software design. Macia [24], then, cataloged a set of patterns of inter-related code smells. Nevertheless, none of these authors studied the impact of agglomerations (i.e., inter-related code smells) on identifying design problems from the point of view of developers, and none of them characterized how to explore agglomerations to improve the identification of design problems.

Nevertheless, none of the aforementioned papers present evidence on whether developers can indeed find more design problems when they focus on inspecting stinky program location. We addressed this gap in our previous work [20]. We conducted a mixed-method study to investigate whether the analysis of stinky program locations help developers in revealing more design problems than the analysis of single smells. In this paper, we extended our previous work [20] by investigating what are the requirements for a tool that supports the analysis of stinky code. This investigation also complements the results of our previous work, showing that communicability issues in the tool may prejudice the identification of design problems in stinky code.

Other approaches to identify design problems

We also found studies that have investigated other approaches to identify design problems [45, 46]. Mo et al. [45] proposed and evaluated a suite of hotspot patterns, which are recurring design problems that lead to high maintenance cost. These patterns are detected by the combination of structural, history, and design information to identify potential design problems. In their study, they showed that these patterns might be the causes of bug-proneness and change-proneness. Xiao et al. [46] introduced an approach that uses a history coupling probability matrix to identify and quantify design problems. The proposed approach uses four patterns to show the correlation between design problems and the decrease of software quality. These studies did not also evaluate the identification of design problems from the perspective of software developers. Besides, these studies rely on history and design information, which may not be available for many software systems.

Other studies have focused on investigating the identification of design problems from the perspective of developers as well [47, 48]. Sousa et al. [47] conducted a qualitative study to investigate how developers identify design problems in practice. In their study, the authors observed the actions that developers apply to identify design problems. They noticed that some actions were applied to locate program elements to analyze, while other actions were applied to confirm if the location contains or not a design problem. They also observed that developers often search for multiple symptoms of design problems, such as code smells, during the identification. In a subsequent study, Sousa et al. [48] proposed a theory to describe how the identification of design problems happens in practice. Their theory explains factors that influence developers during the identification of design problems. For instance, the relation among symptoms is a recurring factor that developers take into account. Despite investigating the identification of design problems from the perspective of developers, these studies focused on observing developers identifying design problems. Conversely, they did not investigate whether developers can indeed find more design problems when they focus on inspecting stinky program location. Also, they did not provide any evaluation on the requirements for a tool that supports the analysis of stinky code.

Detection and visualization of stinky code

Related works propose techniques for supporting the detection and visualization of both single smells and inter-related smells. There are several studies that investigated detection and visualization of single code smells [13–16]. Van Emden and Moonen [13], for example, presented a tool that detects and visualizes code smells in source code. Their tool displays the code structures as a graph and maps code smells onto the graph attributes. Although this tool can represent the relation among elements, it fails on representing code smells. The tool is built upon the assumption that code smells concentrate in a particular location on the source and that software metrics will point developers to these locations. However, this assumption does not always hold since some code smells need the understanding of how the smelly elements interact to which other. For instance, let us consider the Dispersed Coupling, which happens when a code element is excessively tied to many other elements. In this case, a visualization should not only represent the element that contains the Dispersed Coupling, but also the elements to which it is coupled. These interactions cannot be represented by a simple mapping between code structures and colors.

Regarding the visualization of inter-related smells, we found few studies that investigated the detection and visualization of inter-related smells affecting a program location [39, 49, 50]. Palomba et al. [50] analyzed the co-occurrence of 13 code smell types detected in 395 releases of 30 software systems. Their results show that 59% of smelly classes are affected by more than one smell. They also observed that method-level smells may be the cause

of some class-level smells. Finally, they found six pairs of smell types that frequently co-occur together. Based on these findings, they suggest that code smell detectors and refactoring tools should be aware of co-related code smells to provide better results. Nevertheless, the study of Palomba et al. only considered the co-occurrence of code smells at the same class. On the other hand, in this paper, we are investigating four forms of co-occurrence ("Basic concepts" section). Moreover, they did not investigate the identification of design problems with co-located code smells.

Vidal et. al. [39] presented a tool for detecting code smells and agglomerations of a (Java-based) system and ranking them according to different criteria [39]. The main benefit of using their tool is that developers can configure and extend it by providing different strategies to identify and rank the smells and groups of smells (i.e., agglomerations). However, this tool cannot present all the characteristics of agglomerations. For instance, it cannot represents the relations that exist between the code smells within an agglomeration. Therefore, the developer has to reason about the agglomeration to first understand the relation among the smells, to then identify a design problem.

Semiotic Engineering

Semiotic Engineering is a semiotic theory on of human-computer interaction (HCI) [22]. It views a system as a computer-mediated communication between the programmer (system designer) and the user. Semiotic Engineering proposes two qualitative methods to evaluate the communicability in HCI, which are the Semiotic Inspection Method (SIM) and the Communicability Evaluation Method (CEM). These methods have been applied in technical contexts, focusing on how to to improve the communicability of specific systems [22], but also in scientific research. For example, based on a literature review, Reis and Prates [51] observed 21 technical applications and 10 scientific applications of the SIM. Overall, they observed that SIM have been frequently applied in the educational domain and in the evaluation of collaborative systems. Semiotic Engineering has also been applied in the evaluation of programming APIs. The work of Bastos et al. [52] applied a semiotic engineering method called *SigniFYIng APIs*. In this work, they evaluated a date and time API of the Java programming language. Although being applied in different research and technical contexts, Semiotic Engineering was never used to evaluate the impact of communicability issues in the identification of design problems. In this paper, we conducted a qualitative evaluation using the CEM. This analysis helped us to understand how communicability issues in the Organic tool can hinder the identification of design problems. In addition, this evaluation helped us to identify what are the communicability requirements for a tool that helps in the analysis of stinky code.

Concluding remarks

In this paper, we presented Organic—a tool supporting the analysis of stinky code. Organic is a tool to help developers to identify design problems through the analysis of code smells in the source code. Organic supports the analysis of multiple forms of stinkier code, provides detailed information about code smells, supports the analysis of dependencies involving stinky code, provides a graph-based visualization for stinkier code, and provides historical information about stinkier code. These features were designed and implemented based on findings from previous studies about the relation between design problems and code smells. We believe these features can provide the basic support to support developers on identifying design problems.

In addition to proposing Organic, we also conducted two studies to assess if developers are effective in revealing design problems when they reason about agglomerated code smells and to identify tool issues that may hinder the identification of design problems. These studies were important because they revealed that (i) developers find more design problems (and report less false positives) with agglomerations than with a flat list of smells and (ii) developers sometimes may not be able to identify design problems either because they cannot properly reason about multiple code smells or because limited support tool is hindering the identification. Thus, we address these two aspects not explored by previous studies.

In the first study, we conducted a multi-method study with 11 developers. We asked participants to identify design problems in stinkier program locations. After that, we compared their results with the results of when they analyzed a flat list of single code smells to identify design problems. Our analysis showed that developers found more design problems when they reasoned about stinkier code (i.e., agglomerated smells). In addition, we noticed that, when developers were aware of multiple smells in a program location, they reported less false positives. Therefore, our results suggest that reasoning about stinkier code may improve the precision of developers in identifying design problems. Based on the qualitative analysis, we observed that developers indeed tend to have higher confidence to identify the occurrence of non-trivial design problems when using information about multiple smells. That happens because developers usually analyze all smells before reporting a design problem. Consequently, the likelihood of reporting a false positive decreases.

Additionally to these results, this first study also helped us to identify opportunities to improve the tool support for developers. For instance, we observed that developers

need to prioritize stinkier program locations that are most likely to indicate a design problem. This need should be addressed because the analysis of stinky code is difficult and time-consuming. Furthermore, a system may contain several stinkier locations, which choosing which location to analyze can be a cumbersome task for developers. Thus, developers should focus on those locations that are most likely to embody a design problem. In addition, we also noticed that developers need proper visualization mechanisms to support the analyses of stinky code scattered across wider program locations, such as hierarchies or packages.

In the second study, we evaluated Organic with the Communicability Evaluation Method (CEM) [22]. This method enabled us to identify communicability issues in the Organic tool that may hinder the identification of design problems. For example, we observed that, although detecting stinkier program locations, Organic does not provide a message containing concise reasoning about the possible design problem occurring in the stinkier code. As a result, the developer may struggle to make a meaning out of multiple smells.

The second study also revealed some strengths of Organic. For instance, we observed that Organic provides useful information about code smells and about dependencies. Such information was considered useful by most participants. We also observed that Organic promotes a smooth integration of its views with the source code. This characteristic is important because most code smells can only be understood through source code analysis.

In a nutshell, we conclude that both studies encourage the analysis of stinky code to identify design problems. However, there are issues that should be addressed before developers can more effectively explore multiple code smells in a time-effective manner. As discussed above, there is a need to provide mechanisms for better prioritizing and visualizing stinkier code. As a future work, we plan to improve these mechanisms in the Organic tool ("Organic: a tool for the analysis of stinky code" section) and evaluate their impact on developers' effectiveness and efficacy.

Endnote

[1] Organic. Available at https://wnoizumi.github.io/organic/.

Abbreviations

CEM: Communicability evaluation method; OODT: Object oriented data technology; TP: True positives; FP: False positives; ID: Identification; HCI: Human-computer interaction; SIM: Semiotic inspection method

Acknowledgements

We would like to thank Clarisse Sieckenius de Souza for the valuable contributions to this work.

Funding

This work is funded by CNPq (309884/2012-8, 483425/2013-3, 477943/2013-6, 465614/2014-0, 308380/2016-9), CAPES/Procad (175956, 117875, 153363/2018-5), and FAPERJ (22520 7/2016, 102166/2013).

Availability of data and materials

Please contact the authors for data requests.

Authors' contributions

AG, LS, and WO conceived the research and coordinated the research activities. AG, LS, and WO designed and implemented the Organic tool. All authors participated in the design, data collection, and analysis of the mixed-method study. All authors participated in the design, data analysis, and interpretation of the communicability evaluation. All authors helped to draft the manuscript. All authors read and approved the final manuscript.

Competing interests

The authors declare that they have no competing interests.

Publisher's Note

Author details

[1] Opus Research Group, Informatics Department, PUC-Rio, Rio de Janeiro, RJ, Brazil. [2] IFPR, Campus Paranavai, Paranavai, PR, Brazil. [3] UEG, Campus Posse, Posse, GO, Brazil.

Received: 21 March 2018 Accepted: 4 October 2018
Published online: 22 October 2018

References

1. Ciupke O (1999) Automatic detection of design problems in object-oriented reengineering. In: Proceedings of technology of object-oriented languages and systems - TOOLS 30 (Cat. No.PR00278). IEEE, Washington, DC. pp 18–32
2. Trifu A, Marinescu R (2005) Diagnosing design problems in object oriented systems. In: WCRE'05. IEEE, Washington, DC. p 10
3. Fowler M (1999) Refactoring: improving the design of existing code. Addison-Wesley Professional, Boston
4. Macia I, Garcia J, Popescu D, Garcia A, Medvidovic N, von Staa A (2012) Are automatically-detected code anomalies relevant to architectural modularity?: an exploratory analysis of evolving systems. In: AOSD '12. ACM, New York. pp 167–178
5. Oizumi W, Garcia A, Sousa L, Cafeo B, Zhao Y (2016) Code anomalies flock together: exploring code anomaly agglomerations for locating design problems. In: The 38th International Conference on Software Engineering. ACM, New York
6. Abbes M, Khomh F, Gueheneuc Y, Antoniol G (2011) An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In: Proceedings of the 15th European Software Engineering Conference. IEEE, Washington, DC. pp 181–190
7. Yamashita A, Moonen L (2013) Exploring the impact of inter-smell relations on software maintainability: an empirical study. In: Proceedings of the 35th International Conference on Software Engineering. IEEE Press, Piscataway. pp 682–691
8. Yamashita A, Zanoni M, Fontana FA, Walter B (2015) Inter-smell relations in industrial and open source systems: a replication and comparative analysis. In: ICSME 2015. IEEE, Washington, DC. pp 121–130
9. Martin RC, Martin M (2006) Agile principles, patterns, and practices in C# (Robert C. Martin). Prentice Hall PTR, Upper Saddle River
10. Oizumi W, Garcia A, Colanzi T, Staa A, Ferreira M (2015) On the relationship of code-anomaly agglomerations and architectural problems. J Softw Eng Res Dev 3(1):1–22
11. Cedrim D, Sousa L, Garcia A, Gheyi R (2016) Does refactoring improve software structural quality? A longitudinal study of 25 projects. In: Proceedings of the 30th Brazilian Symposium on Software Engineering. SBES '16. ACM, New York. pp 73–82
12. Cedrim D, Garcia A, Mongiovi M, Gheyi R, Sousa L, Mello R, Fonseca B, Ribeiro M, Chávez A (2017) Understanding the impact of refactoring on smells. In: 11th Joint Meeting of the European Software Engineering

Conference and the ACM Sigsoft Symposium on the Foundations of Software. ESEC/FSE'17. ACM, New York
13. Emden E, Moonen L (2002) Java quality assurance by detecting code smells. In: Proceedings of the 9th Working Conference on Reverse Engineering. IEEE, Washington, DC. p 97
14. Ratzinger J, Fischer M, Gall H (2005) Improving evolvability through refactoring, Vol. 30. ACM, New York. https://doi.org/10.1145/1082983.1083155
15. Murphy-Hill E, Black AP (2010) An interactive ambient visualization for code smells. In: Proceedings of the 5th International Symposium on Software Visualization. ACM, Salt Lake City. pp 5–14
16. Wettel R, Lanza M (2008) Visually localizing design problems with disharmony maps. In: Proceedings of the 4th ACM Symposium on Software Visualization. ACM, New York. pp 155–164
17. Macia I, Arcoverde R, Garcia A, Chavez C, von Staa A (2012) On the relevance of code anomalies for identifying architecture degradation symptoms. In: CSMR12. IEEE, Washington, DC. pp 277–286
18. Macia I, Arcoverde R, Cirilo E, Garcia A, von Staa A (2012) Supporting the identification of architecturally-relevant code anomalies. In: ICSM12. IEEE, Washington, DC. pp 662–665
19. Oizumi W, Garcia A, Colanzi T, Ferreira M, Staa A (2014) When code-anomaly agglomerations represent architectural problems? An exploratory study. In: Proceedings of the 2014 Brazilian Symposium on Software Engineering (SBES). IEEE, Washington, DC. pp 91–100
20. Oizumi W, Sousa L, Garcia A, Oliveira R, Oliveira A, Agbachi OIAB, Lucena C (2017) Revealing design problems in stinky code: a mixed-method study. In: Proceedings of the 11th Brazilian Symposium on Software Components, Architectures, and Reuse. SBCARS '17. ACM, New York. pp 5–1510. https://doi.org/10.1145/3132498.3132514
21. Garcia J, Popescu D, Edwards G, Medvidovic N (2009) Identifying architectural bad smells. In: CSMR09. IEEE, Washington, DC
22. De Souza CS, Leitão CF (2009) Semiotic engineering methods for scientific research in HCI. Synthesis Lectures on Human-Centered Informatics 2(1):1–122
23. Bass L, Clements P, Kazman R (2003) Software architecture in practice. 2nd edn. Addison-Wesley Longman Publishing Co., Inc, Boston
24. Macia I (2013) On the detection of architecturally-relevant code anomalies in software systems. PhD thesis, Pontifical Catholic University of Rio de Janeiro, Informatics Department
25. Lanza M, Marinescu R (2006) Object-oriented metrics in practice. Springer, Heidelberg
26. Suryanarayana G, Samarthyam G, Sharma T (2014) Refactoring for software design smells: managing technical debt. 1st edn. Morgan Kaufmann Publishers Inc, San Francisco
27. Mattmann C, Crichton D, Medvidovic N, Hughes S (2006) A software architecture-based framework for highly distributed and data intensive scientific applications. In: Proceedings of the 28th ICSE: Software Engineering Achievements Track. ACM, New York. pp 721–730
28. Marinescu R (2004) Detection strategies: metrics-based rules for detecting design flaws. In: Proceedings of 20th IEEE International Conference on Software Maintenance (ICSM). IEEE, Washington, DC. pp 350–359
29. Yamashita A, Moonen L (2013) Do developers care about code smells? An exploratory survey. In: 2013 20th Working Conference on Reverse Engineering (WCRE). IEEE, Washington, DC. pp 242–251. https://doi.org/10.1109/WCRE.2013.6671299
30. Oizumi W, Garcia A, Sousa L, Albuquerque D, Cedrim D (2014) Towards the synthesis of architecturally-relevant code anomalies. In: Proceedings of the 11th Workshop on Software Modularity. SBC, Porto Alegre. pp 39–52
31. Shadish WR, Cook TD, Campbell DT (2002) Experimental and quasi-experimental designs for generalized causal inference. Houghton Mifflin, Michigan. https://books.google.com.br/books?id=o7jaAAAAMAAJ
32. McIntosh S, Kamei Y, Adams B, Hassan AE (2014) The impact of code review coverage and code review participation on software quality: a case study of the qt, vtk, and itk projects. In: Proceedings of the 11th Working Conference on Mining Software Repositories. ACM, New York. pp 192–201
33. Silva MCO, Valente MT, Terra R (2016) Does technical debt lead to the rejection of pull requests? In: Proceedings of the 12th Brazilian Symposium on Information Systems. SBSI '16. SBC, Porto Alegre. pp 248–254
34. Yahoo! Explore Career Opportunities. Available at https://careers.yahoo.com/us/buildyourcareer. Accessed 21 Mar 2018
35. Twitter Working at Twitter. Available at https://about.twitter.com/careers. Accessed 21 Mar 2018
36. Garcia J, Ivkovic I, Medvidovic N (2013) A comparative analysis of software architecture recovery techniques. In: Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering. IEEE Press, Piscataway
37. Easterbrook S, Singer J, Storey M-A, Damian D (2008) Selecting empirical methods for software engineering research(Shull F, Singer J, Sjøberg DIK, eds.). Springer, London
38. Arcoverde R, Guimares E, Macía I, Garcia A, Cai Y (2013) Prioritization of code anomalies based on architecture sensitiveness. In: 2013 27th Brazilian Symposium on Software Engineering. IEEE, Washington, DC. pp 69–78. https://doi.org/10.1109/SBES.2013.14
39. Vidal SA, Marcos C, Díaz-Pace JA (2016) An approach to prioritize code smells for refactoring. Automated Software Engg 23(3):501–532. https://doi.org/10.1007/s10515-014-0175-x
40. Vidal S, Guimaraes E, Oizumi W, Garcia A, Pace AD, Marcos C (2016) Identifying architectural problems through prioritization of code smells. In: 2016 X Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS). IEEE, Washington, DC. pp 41–50. https://doi.org/10.1109/SBCARS.2016.11
41. Herman I, Melancon G, Marshall MS (2000) Graph visualization and navigation in information visualization: a survey. IEEE Transactions on Visualization and Computer Graphics 6(1):24–43
42. Prates RO, de Souza CS, Barbosa SDJ (2000) Methods and tools: A method for evaluating the communicability of user interfaces. Interactions 7(1):31–38. https://doi.org/10.1145/328595.328608
43. Martin RC (2008) Clean code: a handbook of agile software craftsmanship. 1st edn. Prentice Hall PTR, Upper Saddle River
44. Yin RK (2015) Qualitative research from start to finish. Guilford Publications, New York
45. Mo R, Cai Y, Kazman R, Xiao L (2015) Hotspot patterns: the formal definition and automatic detection of architecture smells. In: Software Architecture (WICSA), 2015 12th Working IEEE/IFIP Conference On. IEEE, Washington, DC. pp 51–60
46. Xiao L, Cai Y, Kazman R, Mo R, Feng Q (2016) Identifying and quantifying architectural debt. In: Proceedings of the 38th International Conference on Software Engineering. ICSE '16. ACM, New York
47. Sousa L, Oliveira R, Garcia A, Lee J, Conte T, Oizumi W, de Mello R, Lopes A, Valentim N, Oliveira E, Lucena C (2017) How do software developers identify design problems?: a qualitative analysis. In: Proceedings of the 31st Brazilian Symposium on Software Engineering. SBES'17. ACM, New York. pp 54–63. https://doi.org/10.1145/3131151.3131168. http://doi.acm.org/10.1145/3131151.3131168
48. Sousa L, Oliveira A, Oizumi W, Barbosa S, Garcia A, Lee J, Kalinowski M, de Mello R, Neto B, Oliveira R, Lucena C, Paes R (2018) Identifying design problems in the source code: a grounded theory. In: Proceedings of the 40th International Conference on Software Engineering. ICSE'18. ACM, New York
49. Oizumi W, Garcia A Organic: a prototype tool for the synthesis of code anomalies. https://wnoizumi.github.io/organic/
50. Palomba F, Bavota G, Penta MD, Fasano F, Oliveto R, Lucia AD (2018) A large-scale empirical study on the lifecycle of code smell co-occurrences. Information and Software Technology. https://doi.org/10.1016/j.infsof.2018.02.004
51. De S. Reis S, Prates RO (2011) Applicability of the semiotic inspection method: a systematic literature review. In: Proceedings of the 10th Brazilian Symposium on Human Factors in Computing Systems and the 5th Latin American Conference on Human-Computer Interaction. IHC+CLIHC '11. Brazilian Computer Society, Porto Alegre. pp 177–186. https://dl.acm.org/citation.cfm?id=2254436.2254468
52. Bastos JADM, Afonso LM, de Souza CS (2017) Metacommunication between programmers through an application programming interface: a semiotic analysis of date and time APIs. In: 2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). IEEE, Washington, DC. pp 213–221. https://doi.org/10.1109/VLHCC.2017.8103470