Identifying Code Smells with Multiple Concern Views

Glauco de F. Carneiro[1], Marcos Silva[2], Leandra Mara[2], Eduardo Figueiredo[3], Claudio Sant'Anna[1], Alessandro Garcia[2], Manoel Mendonça[1]
[1]Computer Science Department, Federal University of Bahia, Brazil
{glauco.carneiro, santanna, mgmendonca}@dcc.ufba.br
[2]Informatics Department, Pontifical Catholic University of Rio de Janeiro, Brazil
{mtsilva, lsilva, afgarcia}@inf.puc-rio.br
[3] Computer Science Department, Federal University of Minas Gerais, Brazil
figueiredo@dcc.ufmg.br

Abstract— **Code smells are anomalies often caused by the way concerns are realized in the source code. Their identification might depend on properties governing the structure of individual concerns and their inter-dependencies in the system implementation. Although code visualization tools are increasingly applied to support anomaly detection, they are mostly limited to represent modular structures, such as methods, classes and packages. This paper presents a multiple views approach that enriches four categories of code views with concern properties, namely: (i) concern's package-class-method structure, (ii) concern's inheritance-wise structure, (iii) concern dependency, and (iv) concern dependency weight. An exploratory study was conducted to assess the extent to which visual views support code smell detection. Developers identified a set of well-known code smells on five versions of an open-source system. Two important results came out of this study. First, the concern-driven views provided useful support to identify God Class and Divergent Change smells. Second, strategies for smell detection supported by the multiple concern views were uncovered.**

Keywords- software visualization; code smells; concerns; program comprehension.

I. INTRODUCTION

A concern is anything a stakeholder wants to consider as a conceptual unit, including domain-specific features, non-functional requirements, and design patterns [1]. Distribution, persistence, transaction management, security and caching are examples of concerns found in many programs. It has been recently claimed that many code smells are caused by the particular ways one or more stakeholders' concerns are structured in the source code [1]. Code smells are anomalies in the source code that contribute to the degeneration of software design maintainability [17]. Classical examples are God Class [14], Divergent Change and Feature Envy [17]. For instance, feature envy can be seen as a misplaced part of a concern, such as a method or a code block inside a method, which does not implement the main concern of its class.

However, the exhaustive inspection of code smells in the source code is impractical and cumbersome [26]. One of the reasons is that it is difficult to visualize the realization of concerns and their inter-dependencies in a program. They are often scattered across many modular structures, such as multiple methods and classes [2]. There is a growing number of software visualization tools that support identification of code smells (e.g., [11],[27]). However, they are mostly limited to visually represent module structures, such as packages, classes, and methods in one single view and with limited or no support for concern visualization.

The extent to which visual representation of concerns support code smells detection is still an open question. Primitive concern properties, such as tangling and scattering represent important information in the detection process. Tangling [18] is the degree to which concerns are intertwined to each other in the modularity units. Scattering [1] is the degree to which a concern is spread over different modularity units. However, according to recent empirical studies [19][22], tangling and scattering do not seem to be the only decisive factors for detecting code anomalies. In fact, contradicting previous research claims [36], not all forms of tangled or scattered concerns were found to be harmful to software maintainability in practice [2]. Other concern properties are likely to better characterize the manifestation of code smells, including: (i) how a concern traverses a particular hierarchical structure of a program [2], (ii) how a concern contributes to properties of a module, such as its interface size [33], coupling and cohesion [33], and (iii) how two or more concerns interact in the source code [8][33], such as their inter-dependencies and overlapping. Unfortunately, there is no investigation analyzing to which extent such concern properties support programmers on identifying code smells.

In this context, this paper presents a multiple views environment that adapts four categories of code views (Section II) to represent concern properties. The rationale for choosing these views is explained in Section II. The purpose of each view can be described as follows:

1. Concern's package-class-method structure: how a concern is realized through modularity units of a system, such as packages, classes, and methods;
2. Concern's inheritance-wise structure: how a concern is dispersed through one or more inheritance trees;
3. Concern dependency: how a concern affects the relationships among modules;
4. Concern dependency weight: how a concern can be perceived as affecting the weight with which modules are coupled to each other.

We developed an Eclipse plug-in[1], called SourceMiner [25], to support and evaluate the concern views of the Java

1 The Multiple Views Environment implemented as an Eclipse plug-in is available at http://www.sourceminer.org

source code. We analyzed how the tool supports the visual detection of recurring code smells (Section III - C). To this end, we conducted an exploratory study (Section III) with two main goals. First, we aimed at understand to which extent the multiple views environment supports programmers in the detection of certain code smells. Our analysis was driven by the computation of precision and recall in the answers given by programmers who participated in the study. The results show that the multiple views concern visualization provided useful support to identify the God Class and Divergent Change code smells. Second, we aimed to uncover successful detection strategies, based on the use of concern properties by the participants. The low number of false negatives is an indication that, even having no access to the source code, participants found useful strategies to spot the code smells.

II. A Multiple View Approach to Visually Represent Concerns in Source Code

According to [13], the characterization of an object-oriented system involves the analysis of size, complexity, inheritance and coupling. These properties comprise information that is important for supporting recurring software engineering tasks, such as detection of source code anomalies. Most of the information provided by these properties can be visually represented. Therefore, we chose graphical representations in order to describe these properties.

Currently, we defined and implemented four set of views to represent source code and the respective concerns that affect it: (i) concern's package-class-method structure (size and complexity), (ii) concern's inheritance-wise structure (size and inheritance), (iii) concern dependency (coupling), and (iv) concern dependency weight (coupling). The original versions of the first two sets of views have been described in a recent publication [25]. This paper describes improved versions of the first sets of views in Sections II.A and II.B. The last two sets of views are described in Sections II.C and II.D. Each set of views is comprised of one or more views that can be coordinated to facilitate comprehension, especially when effectively combined and cross-referenced.

We extended the views mentioned in the previous paragraph with the purpose of visually representing concerns. Concerns are stakeholders' interests of diverse nature, and range from domain-specific features to design patterns or non-functional requirements [1]. The definition of the concern views was inspired by concern properties which were found to be effective modularity indicators in previous empirical studies (e.g., [1], [22]). Although the conventional definitions of code smells [17] are not explicitly based on the notion of concerns, recent work has been arguing that their occurrences are directly correlated with poor concern modularization [2][23][24]. In this context, the purpose of our multiple view approach is to better support the identification of code smells that are likely to be sensitive to concern properties.

By combining these four sets of views, useful concern properties, beyond tangling and scattering can be better represented. For instance, the influence of a concern to the module's coupling and inheritance can be evaluated. These properties leverage the identification of concern properties considering that concerns can be represented in all views. Therefore, concern properties can also be analyzed in conjunction with module properties, such as size and cyclomatic complexity.

Concerns are represented as different colors in all views of our Eclipse plug-in SourceMiner [25]. Visual elements are filled with colors representing the concerns (Figures 1 and 2). For instance, visual shapes such as rectangles representing classes affected by a specific concern are filled with the color associated with that concern. For the cases where more than one concern affects a visual element, the number of concerns is displayed and the last color selected is conveyed (Figures 3 and 4). Programmers can select a color that corresponds to a concern, as well as the set of concerns that should be presented in a given moment. The concern-driven features of SourceMiner are included in all the sets of views, which are presented in the next subsections.

In order to allow the representation of concerns in SourceMiner, the concerns of a system must be first selected and mapped to the source code. This mapping consists on identifying the code fragments that are responsible for implementing each concern. For example, methods in an interface that include, remove and update elements in a storage are affected by the persistence concern. In our case, this mapping must be documented by means of a tool called ConcernMapper [1][4], which is also an Eclipse plug-in. ConcernMapper saves the mapping in XML files. SourceMiner reads these files in order to visually represent the concern mapping as colors in the views.

A. The Concern's Package-Class-Method Structure View

The concern's package-class-method structure view is related to structural representation. It deals with how modules are organized in packages, classes and methods. This view is based on Treemap [7] (Figure 1). A Treemap is a 2D visualization that maps a tree structure into rectangles with each rectangle representing a node. In SourceMiner, packages, classes, and methods are represented as nested rectangles, where the innermost rectangles are methods and the outermost rectangles are packages. Using this visual metaphor, methods that are together in the same class are represented in the same rectangle. Likewise, classes that are in a specific package are represented together in a rectangle. A single screen shot can show all methods, classes and interfaces in accordance with its position in the structure. Programmers can configure the size of each of the innermost rectangle to express the lines of code or complexity of the methods they represent.

We adapted the Treemap visual paradigm to use colors to represent methods that are affected by a specific concern. Figure 1 illustrates how concerns are represented in the Treemap view. The rectangles colored in dark blue correspond to methods that are affected by a specific concern selected by a programmer. In this example, we can see methods (dark blue rectangles), classes (rectangles that contain these dark blue innermost rectangles) and packages (the outermost rectangles as depicted in Figure 1) affected by the selected concern.

Using this view, programmers can interactively reason about how a concern is scattered in packages, classes and methods simultaneously. Also, selecting several concerns, programmers can reason about how tangled the concerns are. The rectangles' areas are also used in Treemap to represent software attributes. The area of each rectangle can represent, for example, the number of lines of code or the cyclomatic complexity of each method. The way Treemap was implemented allows the programmer to reason about association of the size or complexity of a method and the concerns implemented by it. They can reach the conclusion that methods affected by a given concern such as transaction management are at least 20% more complex than the average methods.

B. The Concern's Inheritance-Wise Structure View

The concern's inheritance-wise structure view is represented by the Polymetric view [6]. This view represents module hierarchy realized by the use of both class and interface inheritance (Figure 2). It is a two-dimensional display that uses rectangles to represent classes and interfaces and edges to represent inheritance relationships between them.

We extended the Polymetric view to use colors to represent classes and interfaces containing a concern. Figure 2 illustrates how concerns are represented in the Polymetric view. The rectangles colored in dark blue correspond to classes or interfaces that are affected by a specific concern selected by the programmer. In this example, we can see that the concern is spread over two inheritance trees. Moreover, we can see that two classes in the larger inheritance tree are not affected by the concern.

With this view, it is possible to reason about how a concern is spread over different inheritance trees. The programmer can also easily see which classes of an inheritance tree contain or not code related to the concern. In addition, selecting many concerns simultaneously, the programmer can check which inheritance trees address a high number of concerns.

The rectangles dimensions are used to represent other properties of classes and interfaces. In SourceMiner, the width corresponds to the number of methods while the height to the number of lines of code of a class or interface. Similarly to the Treemap view, the Polymetric view allows the programmer to analyze the association between the size of classes and interfaces with the presence of concerns.

C. The Concern Dependency View

The concern dependency view represents the dependencies among software packages and classes. It is based on a graph view. This view uses nodes (small circles) to represent packages and classes and arrows to represent dependency relationships between packages or classes (Figure 3). The programmer can interactively choose to see the dependencies among packages and/or classes. Currently, SourceMiner considers that class A depends on class B when: (i) a method of class B is called by a method of class A, (ii) class A declares an attribute or local variable as type of class B, (iii) class A is a subclass of class B, (iv) class A references attributes from class B. A package depends on another package when the former includes a class that depends on a class of the latter.

Programmers can select any combination of these types of dependencies to be displayed as arrows in the graph view. We use colors in the graph view to represent which packages and classes are affected by a specific concern. Figure 3 shows that each color represent a different concern in the graph view. In addition, the number in each node indicates how many concerns affect that node. This view allows the programmer to reason about the influence of a concern in the coupling of classes and packages. In this case, coupling is denoted by the amount of dependencies. The programmer can see, for instance, if classes containing a specific concern have a high number of dependencies and if they depend on classes with the same concern or not.

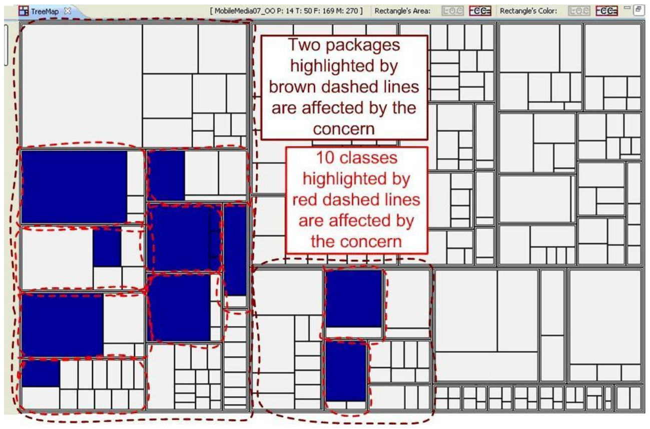
TreeMap
[MobileMedia07_OO P: 14 T: 50 F: 169 M: 270]
Rectangle's Area:
Rectangle's Color:
Two packages highlighted by brown dashed lines are affected by the concern
10 classes highlighted by red dashed lines are affected by the concern

Figure 1: The Treemap View in the concern's package-class-method structure view 2

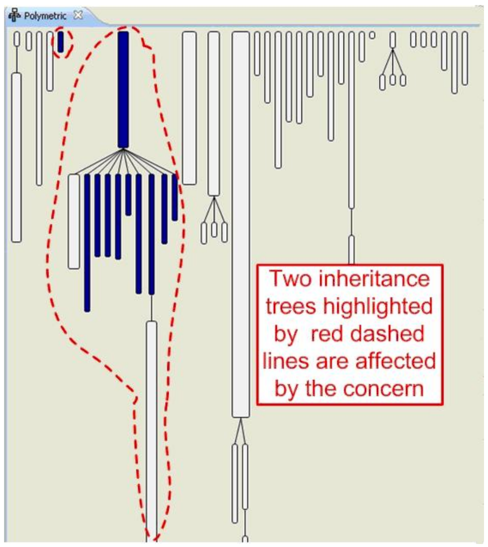
Polymetric
Two inheritance trees highlighted by red dashed lines are affected by the concern

Figure 2: The Polymetric represeting the Concern's Inheritance-wise View

[2] This paper contains colored figures. They are available at http://wiki.dcc.ufba.br/LES/GlaucoCarneiro.

D. *The Concern Dependency Weight Views*

The previously discussed dependency view shows the dependency among classes and packages. The dependency weight views complement this scenario. They display the weight of each dependency, i.e., the number of syntactic references to a module in the dependent module. These set of views therefore represent the dependency weight among classes. Differently from the concern dependency view, what is important here is the number of dependencies and not the number of dependent classes.

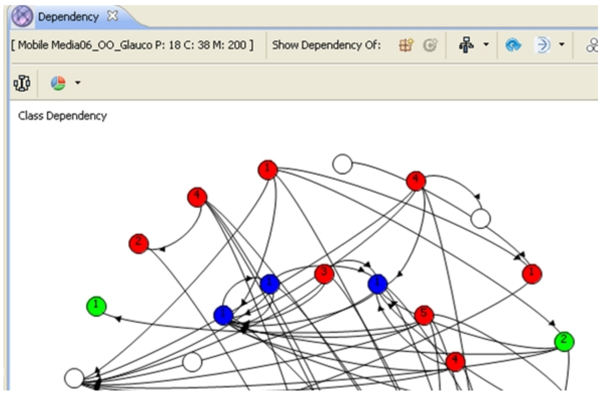

Figure 3: The Graph View representing the Concern Dependency View

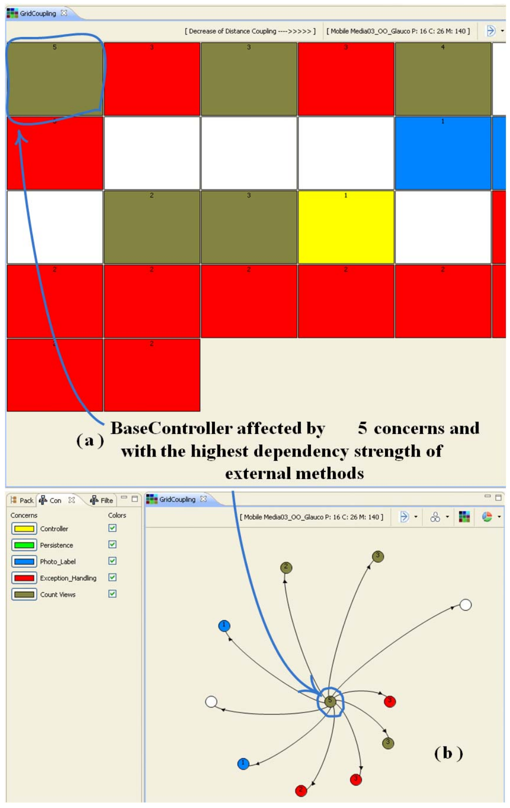
GridCoupling
[Decrease of Distance Coupling ---->>>>>]
[Mobile Media03_OO_Glauco P: 16 C: 26 M: 140]
(a)
BaseController affected by 5 concerns and with the highest dependency strength of external methods
Concerns
Colors
Controller
Persistence
Photo_Label
Exception_Handling
Count Views
(b)

Figure 4: The Chess Board-like and the Spiral Representations in the Concern Dependency Weight

For instance, consider that a class A declares an attribute with the class B as its type, declares a local variable with the class B as its type, and include two calls to methods of the class B. In this case, the value of the dependency weight between classes A and B is four.

The concern dependency weight is represented by two views: a chess board-like (Figure 4a) and a spiral view (Figure 4b). The chess board-like view plots classes from the whole system as rectangles in multiple rows arranged in decreasing order of the total dependency weight. The rectangle representing the class with the highest dependency weight value is placed on the top left corner. The total dependency weight of a class is the sum of the values of the dependency weight between this class and all the other classes.

The spiral view has the goal to complement the chess board-like view (Figure 4b). It represents the weight dependency of a class with other classes it depends on or are dependent on it. The spiral view is also a graph view: nodes (small circles) represent classes and arrows denote that a class depends on other classes. The spiral view is shown when a programmer selects and double clicks a given class on the chess board-like view. The node representing the selected class is presented in the center of the view. The nodes representing the classes with dependencies to the selected class are displayed around the central node forming a spiral. The node nearest to the central node represents the class with higher dependency weight with the selected class and so forth.

We use colors in both views to represent the classes affected by concerns. Figure 4a shows how colors are used in the chess board-like view. The rectangles in different colors correspond to classes responsible to implement or that are affected by different concerns. Figure 4b shows how colors are used in the spiral view. The nodes in different colors correspond to classes affected by different concerns. These views allow programmers to reason about the influence of a concern in the weight of dependency among classes. The programmer can identify, for instance, what concerns are addressed by the classes with the highest or lowest values of dependency weight. Also, the programmer can check, for instance, if a class is strongly coupled with classes implementing the same concern that it implements.

III. The Exploratory Study Design

This section describes the design and settings of an exploratory study with the goal of understanding how programmers identify code smells supported by a multiple view concern-driven approach. Exploratory studies are intended to lay the groundwork for further empirical work [15]. For this reason, there is no control group to compare to an experimental group in a test of a causal hypothesis, i.e., none of the participants attempted to identify code smells without the proposed visualization support. The goal is not to evaluate, but to characterize how participants performed the asked tasks. While the complete experimental material can be obtained elsewhere [37], this section aims to address the following research questions:

RQ1: To which extent the multiple views concern visualization supports the identification of code smells?

RQ2: What are the possible identification strategies for code smells using the multiple views approach?

To answer these questions, we defined the protocol of our study (A) and instantiated it in five study replications (B). This protocol is based on the identification of code smells (C) in five versions of a selected program (D). It also includes a tutorial session (E) and a reference list (F) of actual code smell instances.

A. *The Study Protocol*

Two experts manually identified all instances of three code smells in five versions of a program. These code smells instances represented what we called a reference list. The rationale for preparing the reference list containing the existing code smells is detailed in subsection III.F. Five participants took part in this study and attended a tutorial session on how to use the multiple views approach to analyze design characteristics of programs, such as code smells. We used a different system in the tutorial session to avoid biasing the study results. The participants were asked to follow written instructions in the code smell identification. Participants were not allowed to access the source code neither perform any search directly on it. They were also asked to record the code smells they found and the time they were identified in a questionnaire form. In addition to that information, they were also asked to describe their strategies, experiences, and impressions on the use of the views to accomplish each task. To answer RQ1, we computed the precision and recall (Section IV) for each participant. Additionally, we analyzed the questionnaires to know which views and respective resources were useful to identify each code smell and, therefore, to answer RQ2.

B. *Participants Selection*

The study involved five different developers recruited from the personal contacts list of the paper authors. This number of participants offered a reasonable tradeoff between the cost of the study and detailed qualitative analysis and the generalizability of the results [38][39]. Similarly to other exploratory studies [40], which used the same number of subjects, we were interested in making observations based on a detailed, qualitative analysis of program investigation behavior rather than testing causality hypotheses using statistical inference. To be eligible for inclusion, participants were required to have experience with object-oriented programming. This experience was verified by asking them to fill in questionnaire forms. No current member of our research groups took part in this study. The participants have varying backgrounds. For instance, some of them work in industry and others are computer science undergraduate students of graduate researchers. They were all volunteers and no compensation was provided for their participation in this study.

C. *Code Smells Identification*

Participants were asked to identify the following code smells using our tool: Feature Envy (FE), God Class (GC), and Divergent Change (DC). We selected these three code smells because previous work related them to crosscutting concerns [23]. Moreover, 42 occurrences of these code smells were indentified in the target program. The following descriptions of the code smells summarize the ones presented to the participants.

Feature Envy (FE) occurs when a piece of code seems more interested in a class other than the one it actually is in [17]. It is common that only some of the methods in a class suffer from this anomaly. This code smell can be seen as a misplaced piece of concern code, i.e., code which does not implement the main concern of its class. Hence, the concern realized by this misplaced code is probably located mainly in a different class.

God Class (GC) is characterized by non-cohesiveness of behavior and the tendency of a class to attract more and more features [14]. In a different perspective, we can look at GC as classes that implement too many concerns and, so, have too many responsibilities. It violates the idea that a class should capture only one key abstraction, and breaks the principle of separation of concerns.

Divergent Change (DC) occurs when one class commonly changes in different ways for different reasons [17]. Depending on the number of responsibilities of a given class, it can suffer unrelated changes. The fact that a class suffers many kinds of changes can be associated with a symptom of concern tangling. In other words, a class that presents mixed concerns is likely to be changed for different reasons.

D. *The Target System*

This study relies on five consecutive versions of a software product line, called MobileMedia (MM) [8] that manipulates photo, music, and video on mobile devices, such as mobile phones. Its largest version has about 4 KLOC distributed in 18 packages and 50 classes. We selected MobileMedia due to several reasons. First, its Java implementation is available and served as object in many studies [2][8][20]. Second, its key concerns were previously identified by the developers and mapped to the source code [8]. In this context, the concern selection was independently performed and not influenced by this experiment. In addition, the precision of concern selection and assignment is not the focus of our aforementioned research questions as explained later in Section V. Finally, code smells had already been detected in this system using a heuristic approach [32]. The reason to select versions 3 to 7 was that they contain a representative number of previously mapped concerns (minimum of 4), therefore providing enough information to analyze code smell occurrences. Another reason is that these versions contain the studied code smells. The other three versions of Mobile Media not analyzed in this study do not present the code smells analyzed.

E. *The Tutorial Session*

Prior to the study tasks, the participants were required to complete a tutorial session on how to use the multiple views approach implemented by SourceMiner. In this training session, the participants had 24 hours to familiarize themselves with the tool. They were asked to analyze a program, called Health Watcher [21], and to answer 28 basic questions regarding the tool functionalities. During the tutorial session, two experimenters were available online (email and chat) to provide complementary guidance and detailed explanation on how to use SourceMiner. After finishing with the tutorial tasks, the participants were asked to send their answers to one of the experimenters. In case of wrong answers, they received online instructions to clarify the tool usage.

F. The Code Smells Reference List

We relied on two experts to build a reference list for each analyzed code smell (FE, GC, and DC). The experts are researchers that participated in the development, maintenance, and assessment of the target system. The goal was to detect actual instances of each code smell in versions 3 to 7 of MobileMedia. Each expert was instructed to individually use their own strategy to detect "smelly" classes. As a result, the first expert focused on code inspection following more traditional code analysis. This expert also investigated suspected instances resulting from the application of detection strategies for code smell identification [10][13][31]. Relying on a different strategy, the second expert used a complementary set of concern metrics and concern-sensitive heuristic analysis [32] to identify instances of the three code smells. Not surprisingly, for each code smell, the two sets of potential instances – one set from each expert – were not exactly the same, although they have many classes in common (approx. 75%). In order to reach an agreement, the two experts discussed all classes which were in one set but not in the other, i.e., classes indicated by just one expert. The result of their discussion was recorded as a joint decision. Therefore, classes in the final reference list of each code smell, available at the experiment material web page [37], were detected by at least two experts.

IV. RESULTS AND DATA ANALYSIS

This section reports results in two steps. First, to answer RQ1 it presents the results of precision and recall in accordance with the participant and code smell oriented evaluation presented in section III. Second, to answer RQ2 it enriches RQ1 results with data gathered from questionnaires.

A. Collecting Data for Analysis

We collected direct and indirect data based on questionnaires answered by the participants and provided by an instrumentation system.

Direct data collection [28]: The questionnaires available at [37] described the MobileMedia main functionalities, the code smells with examples, and the tasks to be performed by the participants. Participants were asked to list classes suspected of manifesting each code smell as well as the strategies they use to identify them. They were also asked to describe which of the SourceMiner resources, such as views, concerns, filters, and colors, they found helpful to perform the task at hand. **Indirect data collection [28]:** A logging functionality of SourceMiner automatically records information about the environment usage at a fine-grained level of detail [25]. This functionality monitors how frequently a view or a feature of the tool is used, the transitions among views, and the time each action happened. The goal is a better understanding of the participants' strategies based on their recorded actions. A more detailed description and the logs from this study are available at [37].

B. Data Analysis

The direct and indirect data were analyzed aiming to answer the two research questions (Section III).

Analysis of Precision and Recall: We used two metrics, precision and recall, to analyze to which extent the multiple views concern visualization supported the code smell identification (RQ1). These metrics were adapted from previous work on information retrieval [29][30]. The precision metric quantifies the rate of correctly identified code smells by the number of detected code smell candidates. Recall quantifies the rate of correctly identified code smells by the totally number of actual code smells. These two metrics can be defined as follows.

$$\text{Precision} = \frac{|\{\text{existing code smells}\} \cap \{\text{detected code smells}\}|}{|\{\text{detected code smells}\}|}$$

$$\text{Recall} = \frac{|\{\text{existing code smells}\} \cap \{\text{detected code smells}\}|}{|\{\text{existing code smells}\}|}$$

Analysis of the Adopted Strategies. The questionnaires and recorded logs were used in a manual analysis to uncover the strategies adopted by the participants to accomplish each task (RQ2). That is, we analyzed how different views were commonly used together to accomplish each experimental task. The result is therefore a set of propositions supported by both our qualitative and quantitative findings.

C. RQ1: Multple Views Concern Visualization Support to Code Smell Detection

This section presents and discusses the results of a quantitative analysis addressing the first research question (RQ1). This analysis is supported by the two metrics, precision and recall, defined in the previous section.
Table 1 presents the values of Precision (p) and Recall (r) of each participant (P1 to P5) in the code smell detection. This table reports all the values of precision and recall considering all five MM versions (3-7).

As can be observed in Table 1, there were significant recall and precision variations both from one participant to another and from one code smell to another. For example, the recall value for God Class (GC) varies from 0.4 (P3 and P4) to 1.0 (P5). Despite of the high recall value, P5 had low rates for precision.

High values of recall and precision are desirable. For instance, P2 correctly identified 80% (r = 0.8) of the Divergent Change occurrences with a precision of 70% (p = 0.7). The results in Table 1 also give initial evidence that participants followed different strategies to identify the code smells. For example, we can speculate that an optimistic approach was adopted by P5. P5 identified all the GC, DC,

TABLE 1: PARTICIPANT-ORIENTED EVALUATION

	P1		P2		P3		P4		P5	
	Recall	*Precision*	*Recall*	*Precision*	*Recall*	*Precision*	*Recall*	*Precision*	*Recall*	*Precision*
GC	**0,7	**0,9	0,5	0,4	0,4	0,4	0,4	0,8	1	0,4
DC	0,4	0,8	**0,8	**0,7	0,4	0,8	0,2	0,4	1	0,4
FE	**0,5	**0,5	*Not Performed		0,0	0,0	0,6	0,2	1	0,2

* Participant 2 did not performed FE code smell detection
** Best precision-recall pairs for each code smell

TABLE 3: STRATEGIES ADOPTED TO IDENTIFY CODE SMELLS

	God Class	Divergent Change	Feature Envy
P1	** Inheritance + Concern	Dependency	** Dep. Weight (Grid) + Dep. Weight (Spiral)
P2	Inheritance + Pack-class-method + Concern	** Pack-class-method + Inheritance + Concern	Not Perfomed
P3	Dependency + Concern	Dependency + Grid + Concern	Pack-class-method + Concern
P4	Inheritance + Concern	Pack-class-method + Concern	Dep. Weight (Grid) + Dependency + Inheritance
P5	Inheritance + Grid + Concern	Grid	Dependency

** Best precision-recall pairs for each 1

TABLE 2: CODE SMELL ORIENTED EVALUATION

	Recall	Precision
GC	0,6	0,6
DC	0,5	0,5
FE	0,4	0,2

and FE elements from our reference list (recall = 1) but performed it with low precision (p =< 0.4), i.e., no more than 40% of the recommendations were correct.

On the other hand, P1 demonstrated a conservative approach and did not identify all occurrences of GC, DC, and FE (recall =< 0.7). However, among all participants, P1 achieved the highest precision in the identification of code smells. P2 reported that s/he was not enough confident to identify the Feature Envy code smell.

Table 2 presents the results of precision and recall considering each of the code smells of this study. Data from this table indicate that, in average, participants correctly identified 60% (r = 0.6) of all God Class instances with 60 % (p = 0.6) of correct recommendations. The corresponding values of recall and precision for Divergent Change were both 0.5. And the results for Feature Envy were (r = 0.4 and p = 0.2)

The results presented in Tables 1 and 2 bring initial evidence that the multiple views visual environment can support the identification of the God Class and Divergent Change code smells. These results answer RQ1: the use of the visual representation of concerns in the multiple views approach provided useful means to visually spot some kinds of code smells more than others.

Based on the analysis of RQ1, we present our first observations as follows:

Observation 1: Based on the values of recall and precision, the types of code smells that benefit most from the use of the multiple views approach were God Class (GC) and Divergent Change (DC). An example is the BaseController class identified as GC. The main reason for participants recognize BaseController as GC is that it is easily spotted as an outlier in terms of its size and the number of concerns it realizes. Moreover, methods such as showImageList and handleCommand are also outliers because they realize different concerns.

Observation 2: Varying values of precision and recall obtained by different participants preliminarily suggest that the multiple views approach provides enough support for different styles of code smells identification, ranging from optimistic to conservative.

Much of the information provided by the views could be complemented with source code inspection. Therefore, an interesting question would be if the access to the source code enables optimistic participants to raise their precision and conservative participants to raise their recall rates. In fact, we believe that the power of the multiple views approach is to help programmers to spot code smell candidates before delving into the source code.

D. RQ2: Uncovering Identification Strategies

From the results of the RQ1 analysis, a preliminary conclusion is that participants had different perceptions and judgments during the code smells identification. The differences in the precision and recall values presented in Table 1 support this conclusion. Moreover, we also compared the strategies adopted by participants when performing the tasks. For this comparison, we relied on information provided by questionnaires and registered in the log files of SourceMiner.

Table 3 portrays the views and resources participants used most in the detection of each code smell. For instance, to identify God Class, all participants, including P1 that achieved the best precision and recall, adopted similar strategies: they mostly made synergistic use of the inheritance view and the mapped concerns. This is a clear evidence of an effective strategy to spot God Class instances.

Based on the log files, available at [37], we can uncover the successful stepwise path as follows.
First, participants configured the views to visually represent all concerns. Second, they used the package-class-method structure to spot the classes and interfaces that were candidate outliers in terms of size and the realization of many concerns. Optionally, the inheritance-wise structure could also be used to identify outliers. An interesting result was that all participants successfully identified BaseController as God Class using this strategy. For example, Figure 5 portrays a possible scenario of MM version 3 where BaseController and ImageAccessor clearly stand out as God Class candidates. In Figure 5, BaseController is the largest rectangle as indicated by the arrows in Treemap and Polymetric View. Moreover, it contains methods with different concerns (colors). In the same Figure, ImageAccessor is also indicated by arrows in Treemap and Polymetric. Differently from other participants, in addition to use the package-class-method structure view, P5 also adopted the grid view to spot God Classes. P5 identified all the God Class instances from our reference list. This fact brings initial evidence that the concern dependency weight views may be useful to detect this code smell by providing the number of dependencies of God Class candidates.

Observation 3: A possible strategy to identify God Class candidates is the use of the concern's package-class-method structure together with the concern's inheritance-wise structure view. The concern dependency weight views can offer complementary information regarding the number of dependencies of God Class candidates.
In the case of Feature Envy, P1 also presented the best precision-recall pair result. This participant informed in the questionnaire that the concern dependency weight views (the grid and the spiral views) were used to spot Feature Envy. As already mentioned, these views present classes and interfaces in decreasing dependency order. That is, the grid view first presents classes with higher dependency weight. In this view, the participant selected the BaseController class (Figure 4) and then double clicked on it so that the spiral view could display its dependency relationships. The concern's inheritance-wise view was also used (P3 and P4) to confirm that a class realizes other secondary concerns (by moving the mouse over the rectangles or analyzing the number of concerns presented in each rectangle) in addition to its main one. In fact, using these two sets of views it is possible to easily spot BaseController as Feature Envy candidate (Figure 4). This class stands out due to its interest in other classes.

Observation 4: A possible strategy to identify Divergent Change candidates is the combined use of the concern's inheritance-wise and package-class-method sets of views to spot classes that may suffer many kinds of changes for different reasons. This was the strategy used by P2. None of the other participants used these two sets of views together. This reason may explain why only P2 reached 70% or more for both precision and recall.

A concern is tangled when it is mixed with other concerns within a module which can easily be observed in the package-class-method view. Moreover, if the ascendants of a given class realize different concerns, this class is change prone. The latter characteristic can be observed in the inheritance-wise view. This is again the case of the BaseController class in MM version 3.

Based on the Observations 3 and 4, Table 3 present the main strategies to identify each code smell marked with "**" in the cells. Data in this table also highlight the usefulness of concerns to spot the three code smells analyzed in this study. This is evidence that concern-based analysis plays an important role in software characterization and, in this particular case, to detect the studied code smells.

While analyzing the data provided by the questionnaires and the logs, it was possible to obtain the total time elapsed to perform the tasks as follows: P1 (55 min), P2 (01:51 hr), P3 (49 min), P4 (02:48 hr) and P5 (06:05 hr). Despite the second lower time to perform the study tasks, P1 demonstrated better results in terms of precision and recall to identify Feature Envy and God Class instances. On the other hand, P5 spent the highest amount of time to execute the assigned tasks using an optimistic style.

Observation 5: Despite having only one participant in this study with optimistic style, a possible pattern of behavior of participants like this is to investigate code smell candidates using more combinations of views when compared to participants with more conservative styles.

Observation 6: Further interesting evidence that came out of this study was that when a class or interface was identified in our reference list as a code smell in consecutive versions of MobileMedia, all participants that correctly indicated them in one version also identified them in the other versions. This implies that the knowledge acquired in one version effectively supported the identification of related code smells in other versions.

V. Threats to Validity

This section presents several factors that potentially affect the validity of our study. The applicability of our findings must be carefully established. All of the subjects were either students or recent graduates from a computer science department.

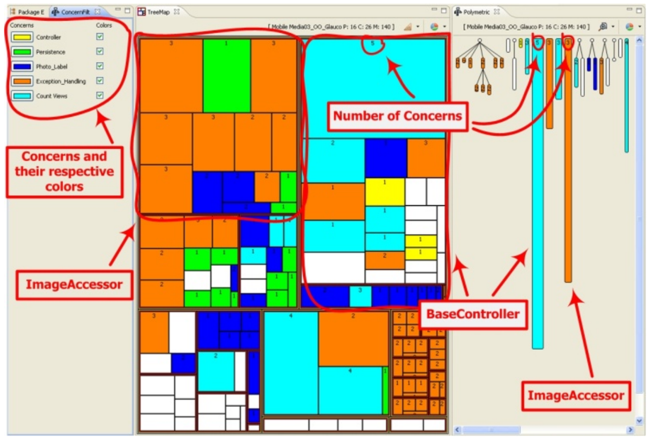

Figure 5: Identifying Outliers Classes with the Concern's Package-Class-Method Structure and the Concern's Inheritance-Wise Structure

Different results might be obtained from different populations, such as a population of senior developers. An important limit to the generalizability of our findings comes from the fact that we have based our observations on the analysis of the behavior of only five subjects. However, as already mentioned, the number of participants accepted in the study was based on a tradeoff between the cost of the study and qualitative analysis of the results to derive the observations.

The participants performed both the tutorial and the three selected code smells identification tasks remotely. We took two measures to address this limitation. First, we were available online (email and chat) to provide guidance and detailed explanation on how to use the tool or questions related to the study. Second, we emphasized to the participants that they were not allowed to use the source code. In addition, we checked in the log files and confirmed that none of the participants accessed the code. As already mentioned, they did not interact among themselves. The examples and questions presented in the tutorial were carefully selected in order to avoid bias. The examples were not related to code smells. They focused on resources provided by the tool.

There are possible limitations on the choices of the mapped concerns and code smells. For instance, the concerns mapped to MobileMedia versions could not be representative enough to spot the code smells. However, the same concerns were used in other experiments such as those reported in [2] and [8]. In addition, even though the three code smells selected are not many, they were the same used in studies reported in [10][13][32].

Participants could not have understood the meaning of God Class, Divergent Change and Feature Envy code smells and their relationship with concerns. The examples and illustrating source code snippets were provided in the questionnaires for each code smell. The reference list may not contain all code smells occurrences, but we tried to overcome this limitation by performing multiple reviews of it and involved two experts that adopted different strategies to its creation. Although we used a small-to-medium open source Java program and five of its versions, we cannot generalize the results to all programs. Future studies should replicate our study in different software systems to allow general results. We do not expect necessarily that identical results will be obtained for them. Nevertheless, we believe that our use of a real program and their respective code smells contributes to achieving an acceptable level of external quality.

VI. Related Work

The recognition that concern identification and analysis are important through software development activities is not new. A growing body of relevant work in this area has been already developed. The Feature Exploration and Analysis Tool (FEAT) [1], for instance, supports the documentation of implementation concerns in a graph-based representation, called concern graphs. SoQueT [3] is another tool that supports the description and documentation of concerns in source code using queries. ConcernMapper [4] and ConcernTagger [5] allow the manual association of concerns to Java code. Although useful, these approaches and tools do not provide efficient visualization support to have an overview of the concerns. Differently from SourceMiner, a clear view of the overall influence of a concern on different views of the source code is not straightforward with these tools. Also, to the best of our knowledge, there is no empirical study analyzing the support these tools provide for identifying code smells.

Regarding visualization support for concern analysis, Ducasse and colleagues proposed a generic technique, called Distribution Map [9], that can be used to visualize and analyze the association of concerns and modules. It uses large rectangles containing small squares filled with different colors. In another work, Ducasse and colleagues also proposed an approach for source code visualization in terms of Microprints [16]. This approach uses character-to-pixel representations of methods enriched with semantic information mapped on nominal colors. Both approaches limit the source code analyzed to only one view. Moreover, they were not used in the context of code smell detection.

Some works proposed the use of software visualization approaches to identify code smells. For instance, Dhambri et al [27] developed a 3D software visualization technique to detect design anomalies. It uses both quantitative data, based on metrics, and structural data, based on relationships among modules. Demeyer et al [12] proposed CodeCrawler to visualize code quality metrics in object-oriented programs. CodeCrawler is based on the Polymetric view. The Package Surface Blueprints approach [16] also supports the visual understanding of relationship among packages. The authors have been able to locate many conceptual bugs, such as some clearly unwanted dependencies. However, they do not explicitly define which types of dependencies are considered on their technique. Other approaches, such as the ones proposed by Marinescu [10] and Parnin et al [11], perform automatic detection of code smells based on metrics and use visualization techniques to present the detection results.

All these visualization techniques and tools differ from SourceMiner in the sense that they do not take into account the concerns realized by the source code. They are strictly based on structural information. In addition, all the work mentioned here supports only one type of visual resource. They are, therefore, limited to only detect code smells associated with the specific information provided by the supported view. For instance, the CodeCrawler tool [12] is based in a single view. Despite its effective support to characterize programs, there are code smells that require more views to be identified. SourceMiner relies on the multiple views approach that integrates different sets of views and enriches the code analyses with concern-driven information.

VII. Final Remarks and Ongoing Work

In this paper we proposed a multiple views approach based on concern-driven software visualization resources as a mean to effectively spot code smells. The approach relies on interactive visual abstractions of source code to support a concern-sensitive analysis based on different views. An exploratory study was conducted to analyze to which extent the multiples views concern visualization support the detection of three recurrent code smells. The outcome was a set of seven observations that can be used to derive hypotheses about the support of the multiple views approach to characterize programs. We intend to investigate these observations in further experiments to analyze industrial and larger programs. We also plan to evaluate the efficacy of the proposed approach against others. We also aim at undertaking experimental studies to investigate how the tool facilitates maintenance tasks. Despite the promising advantages, some limitations came up during the exploratory study. First, methods are the finer granular elements to which concerns are mapped in the current implementation of SourceMiner. The visual analysis could be more detailed if the concerns could be associated to lines of code. Another limitation is that an effective analysis of the proposed approach depends on the appropriate assignment of concerns to source code. We relied on concern assignment performed by others and evaluating this is out of the paper scope.

Acknowledgments. This work was partially supported by CNPq (grant 480374/2009-0) and by the National Institute of Science and Technology for Software Engineering (INES), funded by CNPq (grant 573964/2008-4).

REFERENCES

[1] Robillard, M; Murphy, G. Representing Concerns in Source Code. ACM TOSEM, 2007.

[2] Figueiredo, E. et al. Crosscutting Patterns and Design Stability: An Exploratory Analysis. ICPC, Vancouver, Canada, 2009.

[3] Marin, M; Moonen, L.; Deursen, A. SoQueT: Query-Based Documentation of Crosscutting Concerns. ICSE, 2007.

[4] ConcernMapper - Simple Separation of Concerns for Eclipse. Available at http://www.cs.mcgill.ca/~martin/cm/.

[5] Eaddy, M.; Aho, A.; Murphy, G. Identifying, Assigning, and Quantifying Crosscutting Concerns. ICSE (ACoM 2007), 2007.

[6] Lanza, M.; Ducasse, S. Polymetric Views - A Lightweight Visual Approach to Reverse Engineering. In IEEE TSE, September. 2003.

[7] Shneirderman, B. Tree Visualization with Tree-Maps: A 2-D Space-Filling Approach. ACM ToG 11, 1 (1992), 92–99.

[8] Figueiredo, E. et al. Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability. ICSE, May 2008.

[9] Ducasse, S.; Girba, T.; Kuhn, A. Distribution Map, ICSM, 2006.

[10] Marinescu, R. Detection Strategies: Metrics-Based Rules for Detecting Design Flaws. ICSM, IEEE Computer Society Press, pp 350 -359, 2004.

[11] Parnin, C.; Görg, C.; Nnadi, O. A Catalogue of Lightweight Visualizations to Support Code Smell Inspection. ACM SoftVIS 2008, Herrsching, Germany, 2008.

[12] Demeyer, S.; Ducasse, S.; Lanza, M.A Hybrid Reverse Engineering Approach Combining Metrics and Program Visualization. WCRE'99.

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

[14] Riel, A. Object-Oriented Design Heuristics. Addison-Wesley Professional, 1996.

[15] Seaman., C. Qualitative Methods in Empirical Studies of Software Engineering. IEEE TSE., vol. 25, no. 4, pp. 557-572, 1999.

[16] Ducasse, S.; Pollet, D.; Suen, M.; Abdeen, H.; Alloui, I. Package Surface Blueprints: Visually Supporting the Understanding of Package Relationships. In ICSM'2007

[17] Fowler, M. Refactoring: Improving the Design of Existing Code. Addison Wesley, 1999.

[18] Tarr, P.; Ossher, H.; Harrison, W.; Jr., N. Degrees of Separation: Multi-Dimensional Separation of Concerns. ICSE, 1999.

[19] Cacho, N. et al. Composing Design Patterns: A Scalability Study of Aspect-Oriented Programming. AOSD, March 2006.

[20] Conejero, J. et al. Early Crosscutting Metrics as Predictors of Software Instability. ICOMCP – TOOLS-Europe 2009.

[21] Greenwood, P. et al. On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. ECOOP, Germany, 2007.

[22] Garcia, A. et al. Modularizing Design Patterns with Aspects: A Quantitative Study. AOSD, Chicago, USA, 2005.

[23] Monteiro, M.; Fernandes, J. Towards a catalog of aspect-oriented refactorings. AOSD, March, 2005, Chicago, Illinois.

[24] Sant'Anna, C. et al. On the Modularity Assessment of Aspect-Oriented Multiagent Architectures: a Quantitative Study. International Journal of Agent-Oriented Software Engineering, v. 2, p. 34-61, 2008.

[25] Carneiro, G., Sant´Anna, C; Garcia, A ; Flach, C ;Mendonça, M. G. On the Use of Software Visualization to Support Concern Modularization Analysis. ACoM, 2009, Colocated with OOPSLA, 2009.

[26] Moha, N.; Gueheneuc, Y.; Leduc., P. Automatic Generation of Detection Algorithms for Design Defects. ASE, 2006.

[27] Dhambri, K.; Sahraoui, H.; Poulin, P. Visual Detection of Design Anomalies. CSMR, 2008.

[28] Lethbridge, T; Sim, S; Singer, J. Studying Software Engineers: Data Collection Techniques for Software Field Studies. Empirical Software Engineering 10(3):311– 341. 2005.

[29] Moha, N.; Guéhéneuc, Y.; Duchien, L.; Le Meur, A. Decor: A Method for the Specification and Detection of Code and Design Smells. IEEE TSE, vol. 99, no. 1, 5555. 2009.

[30] Rijsbergen, C. Information Retrieval, 2nd edition. Butterworths, London, 1979.

[31] Marinescu, R. Measurement and Quality in Object-Oriented Design. Ph.D. Thesis, Department of Computer Science, Politehnica University of Timisoara, 2002.

[32] Figueiredo, E.; Sant'Anna, C.; Garcia, A.; Lucena, C. Applying and Evaluating Concern-Sensitive Design Heuristics. SBES, 2009.

[33] Boulanger, J.; Robillard, M., Managing Concern Interfaces. ICSM 2006.

[34] Sant'Anna, C.; Garcia, A.; Lucena, C. Evaluating the Efficacy of Concern Driven Metrics: A Comparative Study. In: ACoM.08, colocated with OOPSLA 2008, 2008.

[35] Storey, M.-A. Theories, Tools and Research Methods in Program Comprehension: Past, Present and Future, Software Quality Journal, Springer, 2006.

[36] Kiczales, G. et al J Aspect-Oriented Programming. ECOOP, 1997.

[37] Experimental Package for the Multiple Views Exploratory Study Available at : http://wiki.dcc.ufba.br/LES/GlaucoCarneiro.

[38] S.L. Pfleeger. Experimental Design and Analysis in Software Engineering—Part 3: Types of Experimental Design. Software Eng. Notes, vol. 20, no. 2, pp. 14-16, Apr. 1995.

[39] R.K. Yin. Case Study Research: Design and Methods. Applied Social Research Methods Series, vol. 5, second ed. 1989.

[40] M. Robillard, W. Coelho, and G. Murphy. How effective developers investigate source code: an exploratory study. IEEE Transactions on Software Engineering, vol. 30, no. 12, pp. 889-903, 2004.

[41] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell and A. Wesslén, "Experimental Software Engineering - An Introduction", Kluwer Academic Publishers, ISBN 0-7923-8682-5, 2000.