

On the Use of Software Visualization to Support Concern Modularization Analysis

Glauco de F. Carneiro¹, Claudio Sant’Anna¹, Alessandro Garcia², Christina Chavez¹,
Manoel Mendonça¹

¹Computer Science Department, Federal University of Bahia, Brazil
{glauco.carneiro, santanna}@dcc.ufba.br, {flach, manoel.mendonca}@ufba.br

²Informatics Department, Pontifical Catholic University of Rio de Janeiro, Brazil
afgarcia@inf.puc-rio.br

Abstract

Most of the methods for concern modularity analysis rely on the exploration of crosscutting concerns directly in source code artifacts. However, as both the code size and the number of concerns increase, analyses of concern properties become an even more cumbersome and time-consuming task. In this paper, we propose a visual representation of crosscutting concerns using a software visualization infrastructure called SourceMiner. The infrastructure facilitates concern modularization analysis based on the following concepts: inheritance tree and package-class-method structure. The analysis based on each of these concepts is supported by the views and its resources available in the infrastructure. An example of use illustrates the infrastructure support to visually analyze the concern modularity of an open source code.

Keywords: software visualization; concern modularity; software comprehension.

1. Introduction

The separation of concerns in software implementation was first mentioned by Dijkstra [1] and Parnas [2]. Since then, the term concern has been used to describe anything a developer might want to consider as a conceptual unit in a program. Ideally, concerns should be tidily encapsulated within modules with well-defined interfaces [2]. Much of the complexity of software design can be derived from the poor modularization of concerns. Hence, software engineers generally try to keep concerns well modularized as they implement a system.

However, in practice a number of concerns end up scattered across the source code and tangled with other concerns due to various reasons [3]. These concerns are called crosscutting concerns. A number of studies indicate that various forms of crosscutting concerns are detrimental to software modularity [4][5][6][7][8]. In this context, there is an increasing need for tools and methodologies to support continuous concern modularization analyses.

A number of different tools have been developed to help software engineers to identify the source code related to a concern as well as document, view and manage this information [3][9][10][11]. Measurement approaches and tools have also been defined for identifying harmful crosscutting concerns and supporting the detection of respective design anomalies [4][16][20][22]. These tools became more important with the emergence of new concern modularization techniques, such as aspect-oriented software development.

These approaches and tools contribute significantly to improve concern modularity assessment. However, they do not scale well to large systems. It is usually hard to have a big picture of the nature and degree of crosscutting of a concern using only such tools. The documented concerns are visualized directly on source code or by means of representations that do not make an efficient use of the screen space. In addition, in some of these tools, it is not trivial to identify and navigate through all the elements affected by a concern.

To tackle this problem, we developed a tool for performing concern modularization analysis with the use of visualization resources widely known as efficient to represent large systems [14][15]. We extended a software visualization infrastructure called SourceMiner [12][13] (Section 2) with the purpose of representing software modules with their respective concerns. Currently the infrastructure can represent concerns simultaneously in two different views (Section 3). These views can be used to interactively configure the best visualization scenarios together with filtering and searching facilities. We illustrate how the infrastructure can be used to execute the concern modularization analysis for an open source system (Section 4). The definition of the concern views were inspired in concern properties found to be good modularity indicators in previous empirical studies (e.g. [4][5][6][7][8]).

2. Visually Representing the Source Code with SourceMiner

We recently proposed SourceMiner [12][13] as an infrastructure to bring forward visual information from the source

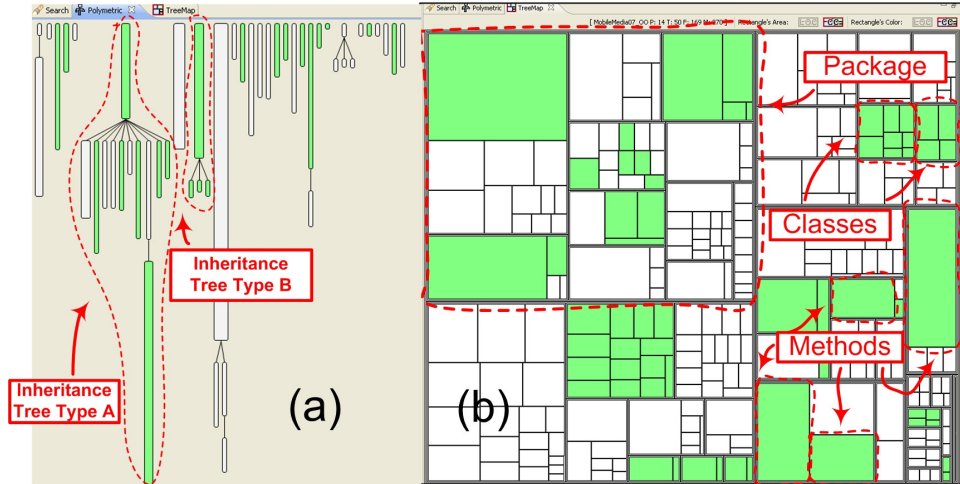


Figure 1. A concern represented as light green in (a) Polymetric and (b) in Treemap View

code. This paper presents how this environment has been augmented to represent concerns affecting the implementation of one or more modules. SourceMiner currently uses two views to represent the software from different perspectives. In the following, we briefly describe each of these views without the concern representation. Next section describes how these views were adapted to explicitly represent concerns across module structures.

The Polymetric View [14] is a two-dimensional display that uses rectangles to represent software entities, such as classes and interfaces, and edges to represent inheritance relationships between those entities - Figure 1(a). Polymetric is particularly efficient to represent inheritance trees that comprise the software system. The rectangles' dimensions are used to represent properties of the entities. In SourceMiner, the width corresponds to the number of methods while the height to the number of lines of code of classes or interfaces.

Treemap [15] is a space-filling method of visualizing large hierarchical data sets - Figure 1(b). SourceMiner represents packages, classes and methods in this view. They are represented as nested rectangles where the innermost rectangles represent methods. Using this metaphor, classes that are in a specific package are represented together in a rectangle. Likewise, all the methods declared in a class are represented in the rectangle area related to its class.

All views are scalable. They can represent hundreds or even thousands of software modules in a single screen. They are also browsable. Programmers can use widgets to filter and search for modules based on features, such as name, and measurable attributes such as size and complexity. Any filtering and searching affect all of them simultaneously. The views are also fully integrated with the IDE. Programmers can go back and forth from visual objects and the corresponding source code.

3. Analyzing Concern Modularization

In order to leverage concern modularization analysis, we augmented SourceMiner to represent concerns in the two views. Therefore, programmers can visually analyze concern modularization based on the following concepts: inheritance tree (polymetric) and package-class-method structure (treemap). Each concept deals with a different perspective regarding the modularity unit under analysis. Based on that assumption, we believe they provide favorable conditions to identify non trivial patterns and relationships between concerns and module units as well as between distinct concerns. Concerns are represented in the views as different colors. A visual element is filled with colors representing concerns. For instance, visual elements such as rectangles representing classes, interfaces or methods affected by a specific concern will be filled with the color associated to that concern. 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.

Concerns should be first assigned manually to code by expert programmers using the Eclipse plug-in ConcernMapper [11]. The mapping is then saved in xml-like files. SourceMiner reads these files in order to present the concern mapping visually as a color attribute in the views. By doing this way, programmers can identify concerns as an abstraction of the code visual representation.

In the following subsections, we describe in details how concerns are portrayed in each view. Besides, we discuss how programmers can use this information to assess concern modularization. This assessment can be based on the following attributes: (i) scattering – the degree to which a concern is spread over different modularity units, (ii) dedication – how much of each modularity unit is affected by a concern, and (iii) tangling – the degree to which concerns are intertwined to each other in the modularity units.



Figure 2. A concern represented as dark blue in (a) Polymetric and (b) in Treemap View

The use of visualization to analyze concern modularization differs from other approaches in that it allows programmers to assess simultaneously scattering, dedication and tangling from the views. These concepts are manifested originally altogether in the views. It is up to the programmers to configure the best scenario to analyze them.

Analyzing Concern Modularization with Polymetric View

Figures 1(a), 2(a) and 3(a) illustrate how concerns can be represented in the polymetric view. The rectangles colored in light green, dark blue and red respectively correspond to classes or interfaces that are affected by a specific concern selected by the programmer. Using this view, programmers can interactively reason about scattering, tangling and dedication in terms of inheritance trees.

The degree of scattering of a concern is related to the number of inheritance trees showing the color of that concern. A color manifested in several trees indicates that a concern affects at least one class or interface in each of these inheritance trees. The support of scattering analysis is particularly relevant as recent studies have found a strong correlation between highly-scattered concerns, faults [16] and design instabilities [4][6][7]. We can see from Figure 1(a) that the concern represented by the light green color scatters over 13 trees. A concern spread over a high number of inheritance trees is usually an indicator of modularity flaw [6][7]. To maintain the concern, the programmer may have to understand most of the affected trees.

The dedication of an inheritance tree to a concern is how much of the tree is filled with the color of the concern. A tree that has all of its rectangles filled with a specific concern's color means that all classes or interfaces in the tree have at least one method dedicated to the concern. On the other hand, trees with low dedication to a specific concern have few colored rectangles. This may be an indication that implementing this concern is not the main purpose of the tree [4]. We can see from Figures 1(a) and 3(a) that the two

trees marked with dotted lines have different degrees of dedication to the concern represented – inheritance trees of types A and B.

The number of distinct colors manifested in an inheritance tree reveals the degree of concern tangling. An inheritance tree that presents several colors means that it deals with many concerns. This may be an indication that this tree may be susceptible to different kinds of changes due to any of these concerns.

Analyzing Concern Modularization with Treemap View

Figures 1(b), 2(b) and 3(b) illustrate how concerns can be represented in the treemap view. The rectangles colored in light green, dark blue and red respectively correspond to methods that are affected by a specific concern selected by the programmer. Using this view, programmers can interactively reason about scattering, tangling and dedication in terms of the package-class-method structure.

Treemap enables the analysis of scattering in terms of packages and classes simultaneously. The degree of scattering of a concern is related to how many packages show the color of that concern. A color manifested in several packages indicates that a concern affects at least one class in each of these packages. A concern spread over a high number of packages may be an indication of modularity flaw. To maintain the concern, the programmer may have to understand most of the affected packages.

The dedication of a package to a concern is how much of the classes are filled with the color of the concern. A package that has all of its classes dedicated to a concern means that at least one method per class is filled with a specific concern's color. Similarly, a class that has all of its methods dedicated to a concern means that all its methods are filled with a specific concern's color. On the other hand, packages with low dedication to a specific concern have few colored classes presenting the concern's color. This may be an indication that implementing this concern is not the main purpose of the package.

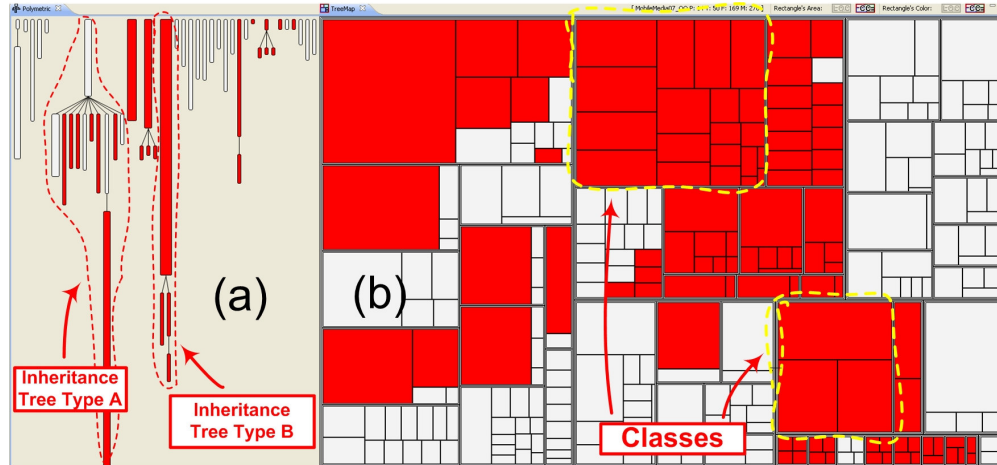


Figure 3. (a) Persistence Concern in Polymetric and (b) in Treemap View portrayed in Red

The number of distinct colors manifested in a package reveals the degree of concern tangling. A package that presents several colors means that it deals with many concerns. This may be an indication that this package may be susceptible to different kinds of changes due to any of these concerns. Similarly, a class that presents several colors means that it deals with many concerns. This may be an indication that this class may be also susceptible to different kinds of changes due to any of these concerns.

4. An Example of Use

We present here an example of use to illustrate how SourceMiner can support concern modularization analysis based on software visualization. The goal is to have a preliminary flavor on whether and how the visualization resources are useful to assess concern scattering, tangling and dedication.

The Target System

In the example of this paper, we partially analyze the concern modularization of an object-oriented system called MobileMedia [17]. It is a software product line with about 3 KLOC that manipulates photo, music, and video on mobile devices, such as mobile phones. It is based on a previous software product line called MobilePhoto [18], developed at the University of British Columbia. The concerns identified were photo, music, video, album, create/delete media, label media, view/play media, sort media, copy media and set favorites. Due to the fact that its design follows the Model-View-Controller (MVC) architectural pattern, the controller concern is also identified in the system. We also identified other concerns related to non-functional requirements, such as exception handling and persistence.

We selected MobileMedia due to several reasons. First, its Java implementation is available on the web. Second, this system has served as object of study on various works about assessment of contemporary modularization techniques, mainly aspect-oriented development [4][17][18]. Third, the

mapping of most of its concern to source code was previously done and is also available [19]. And, finally, the modularization of its concern was already assessed by means of concern-driven metrics and detection strategies [4][17].

Visual Analysis of MobileMedia Concerns

We analyzed six concerns of MobileMedia, namely photo, video, album, sorting, persistence and controller. In this section, we present the analysis of the controller, album and persistence concerns. These concerns are representative of the six because they represent different types of concerns: (i) album is related to the business domain of MobileMedia, (ii) persistence represents domain independent requirements, and (iii) controller is one of the three roles of the MVC architectural pattern, based on which the system was designed. Moreover, their visual analyses have two distinct outcomes: the controller concern is well modularized whereas the persistence and album concern does not present such behavior.

Figure 1 shows how the album concern is visualized with the Polymetric and Treemap views. We can see in the Treemap view (Figure 1b) that this concern is spread over various methods, classes and packages. Some classes have all methods dedicated to it. Checking the name of these classes directly from the view (placing the mouse over the visual elements shows the name of the represented software elements), we observe that they are classes whose main purpose is implementing the album feature. For instance, AlbumListScreen, AlbumData and AlbumController are the names of some of these classes. On the other hand, Treemap also highlights that other classes have just some of their methods dedicated to album. These are methods and classes that use the services of the album-specific classes. The album concern is tangled with other concerns in these classes and their packages.

Checking now the album concern in the Polymetric view, Figure 1(a), we observe that the information provided by this view corroborates somehow with the Treemap view: the

album concerns is spread over several different classes. But, it adds that these classes are elements of different inheritance trees. The class inheritance structure of MobileMedia was designed following the MVC pattern. As a consequence, there are three main inheritance trees, each for implementing each of the pattern roles. It was expected that the album concern was scattered over the different trees: the implementation of a business-related feature, as album, usually affects the three roles the MVC pattern. Note that there is a tree entirely dedicated to album. This tree is part of the MVC model role. This was also expected as all the three types of media supported by MobileMedia, namely photo, music and video, are organized in albums.

The visualization of the persistence concern is presented in Figure 3. Similarly to the album concern, the Treemap view shows that the persistence concern is scattered over several packages and classes - Figure 3(b). The difference is that there are more classes with all methods dedicated to persistence. This is because the design includes some classes for persistence in general and classes for persisting each type of media. When analyzing the Polymetric view, Figure 3(a), we observed that persistence affects various inheritance trees, including the three trees related to each role of the MVC pattern. This looked strange to us as the persistence concern should be restricted to the model inheritance tree. In order to check this out in more details we navigated from the Polymetric view directly to the source code of the classes that were not supposed to incorporate persistence. We observed, then, that these classes handle some persistence-specific exceptions, such as PersistenceMechanismException, which were propagated from the model classes to them. This information provided us an opportunity for refactoring the implementation of the system. Our suggestion is to change the classes in the model inheritance tree in order to not propagate the persistence-specific exceptions. Instead, these exceptions should be handled in the model classes, and other types of exceptions related to the business domain should be passed to the other classes.

Finally, Figure 2 shows the Treemap and Polymetric view for the controller concern. Differently from the other two concerns, the controller concern is well modularized in few packages and classes. The Polymetric view shows that this concern is almost totally restricted to one inheritance tree, which is the tree that implements the MVC controller role.

5. Related Work

The recognition that concern identification and analysis are important through software design activities is not new. Besides, the aspect-oriented paradigm has promoted a growing body of relevant work in the software engineering literature focusing on concern identification and documentation tools. The Feature Exploration and Analysis Tool (FEAT) [3] supports the documentation of implementation concerns

in a graph-based representation, called concern graphs. So-QueT [9] is another tool that supports the description and documentation of concerns in source code using queries. Concern Mapper [10] and Concern Tagger [11] 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 Source Miner, having a clear view of the overall influence of a concern on the inheritance trees and package structure is not straightforward with these tools.

Ducasse and colleagues [20] proposed a generic technique that can be used to visualize and analyze the association of concerns and modules. However, their visual approach only allows concern visualization based on only one view. Also, it does not have interaction features, such as filters, and does not allow the programmer to navigate from the visualization elements to source code.

The Aspect Browser [23] is a tool proposed to help developers to find concerns using lexical searches of the program text. In Aspect Browser the user can visualize how the concerns are spread over classes. The classes are represented by rectangles similar as in the polymetric view. However, there is no information about the inheritance hierarchy or package structure as in SourceMiner. Therefore, concern tangling and scattering analysis is limited if compared with the analysis supported by the polymetric and treemap views.

6. Final Remarks and Ongoing Work

In this paper we presented how SourceMiner can enhance the analysis of concern modularization based on software visualization. It provides interactive visual abstractions of software systems, allowing a concern-sensitive analysis by means of different views: package-class-methods structure and inheritance trees. An example of use of the tool to analyze the concern modularization of an open source system was presented. It is a first step towards the evaluation of the utility and efficacy of the proposed tool.

Despite the promising advantages of using SourceMiner, some limitations came up during the first example of use. Firstly, methods are the finer granular elements to which concerns are mapped. The visual analysis could be more detailed if the concerns were associated to lines of code. Another limitation is that an effective analysis with SourceMiner depends on an appropriate assignment of concerns to source code.

Currently, we are developing a module dependency view. It has the purpose to represent dependencies between classes or packages. In contrast to traditional dependency graphs, this dependency view do not represent all nodes of a software system, but rather the ones that are linked through afferent or efferent coupling [21]. The motivation behind the use of this view is to support programmers to analyze how concern modularization affects coupling between modules.

As future work, we intend to use SourceMiner to analyze industrial and larger systems. We plan to perform controlled experiments and case studies to assess the efficacy of using SourceMiner in comparison to other tools, such as Concern Mapper [10]. Our hypothesis is that SourceMiner is more efficacious to support the concern modularization analysis. We also aim at undertaking experimental studies to investigate how the tool facilitates maintenance tasks.

Acknowledgments

This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES), funded by CNPq and FACEPE, grants 573964/2008-4 and APQ-1037-1.03/08.

References

- [1] Dijkstra, E. A Discipline of Programming. Prentice Hall, Englewood Cliffs, NJ, USA, 1976.
- [2] Parnas, D. On the criteria to be used in decomposing systems into modules. Communications of the ACM, 15(12):1053–1058, December 1972.
- [3] Robillard, M.; Murphy, G. Representing Concerns in Source Code. ACM Transactions on Software Engineering and Methodology, 16(1), Article 3, February 2007.
- [4] Figueiredo, E. et al. Crosscutting Patterns and Design Stability: An Exploratory Analysis. Proc. of Int'l Conf. on Program Comprehension (ICPC). Vancouver, Canada, 2009.
- [5] Cacho, N. et al. Composing Design Patterns: A Scalability Study of Aspect-Oriented Programming. In AOSD'06, Bonn, Germany, 20-24 March 2006.
- [6] Conejero, J. et al. Early Crosscutting Metrics as Predictors of Software Instability. In ICOMCP – TOOLS-Europe 2009.
- [7] Greenwood, P. et al. On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. In ECOOP.07, July 2007, Germany.
- [8] Garcia, A. et al. Modularizing Design Patterns with Aspects: A Quantitative Study. 4th International Conference on Aspect-Oriented Software Development (AOSD'05), Chicago, USA, 14-18 March 2005.
- [9] Marin, M.; Moonen, L.; Deursen, A. SoQueT: Query-Based Documentation of Crosscutting Concerns, Proceedings of the 29th International Conference on Software Engineering (ICSE'07), pp. 758-761, 2007.
- [10] ConcernMapper - Simple Separation of Concerns for Eclipse. Available at <http://www.cs.mcgill.ca/~martin/cm/>.
- [11] Eaddy, M.; Aho, A.; Murphy, G. Identifying, Assigning, and Quantifying Crosscutting Concerns. ICSE Workshop on Assessment of Contemporary Modularization Techniques (ACoM 2007), Minneapolis, MN, May 22, 2007.
- [12] Carneiro, G.; Magnavita, R.; Mendonça, M. SourceMiner as an Experimental Platform to Characterize Software Comprehension Activities. In ICPC Tools Session. Vancouver, 2009.
- [13] Carneiro, G.; Magnavita, R.; Mendonça, M. An Experimental Platform to Characterize Software Comprehension Activities supported by Visualization. In ICSE Research Demo Paper (Poster). Vancouver, 2009.
- [14] Lanza, M.; Ducasse, S. Polymetric Views - A Lightweight Visual Approach to Reverse Engineering. In IEEE TSE, Vol. 29, No. 9, pp. 782 - 795, September. 2003.
- [15] Bederson, B.; Shneiderman, B.; and Wattenberg, M. Ordered and Quantum Treemaps: Making Effective Use of 2D Space to Display Hierarchies. ACM Transactions on Graphics (TOG), 21, (4), October 2002, 833-854.
- [16] Marc Eaddy et al. Do Crosscutting Concerns Cause Defects? IEEE Trans. Software Eng. 34(4): 497-515 (2008).
- [17] Figueiredo, E. et al. Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability. In proceedings of the 30th International Conference on Software Engineering (ICSE), pp. 261-270. Leipzig, Germany, 10-18 May 2008.
- [18] Young, T. Using AspectJ to Build a Software Product Line for Mobile Devices. MSc dissertation, Univ. of British Columbia, 2005.
- [19] Figueiredo, E. Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability. Available at <http://www.lancs.ac.uk/postgrad/figueire/spl/icse08/>.
- [20] Ducasse, S.; Girba, T.; Kuhn, A. Distribution Map, Proceedings of the Int'l Conference on Software Maintenance (ICSM), Philadelphia, pp. 203-212, 2006.
- [21] Martin, R. OO Design Quality Metrics An Analysis of Dependencies. Available at <http://www.objectmentor.com/resources/articles/oodmetric.pdf>.
- [22] Sant'Anna, C.; Figueiredo, E.; Garcia, A.; Lucena, C. On the Modularity of Software Architectures: A Concern-Driven Measurement Framework, Proceedings of the 1st European Conference on Software Architecture, September 24-26, Madrid, Spain, 2007.
- [23] Griswold, W.; Yuan, J.; Kato, Y. Exploiting The Map Metaphor In A Tool For Software Evolution, Proc. of the 23rd Int'l Conference on Software Engineering, pp. 265–274, 2001.