An Approach for Graphical User Interface Bad Smells Detection

J.C. Silva[1] J.C Campos[2] J. Saraiva[2] J.L. Silva[3]
[1] Dep. Tecnologias, Instituto Politécnico do Cávado e do Ave, Barcelos, Portugal
[2] Dep. Informática, Universidade do Minho and HASLab / INESC TEC, Braga, Portugal
[3] M-ITI, Universidade da Madeira, Funchal, Portugal
jcsilva@ipca.pt, {jose.campos,jas}@di.uminho.pt, jose.l.silva@m-iti.org

Abstract. In the context of an effort to develop methodologies to support the evaluation of interactive system, this paper investigates an approach to detect graphical user interface bad smells. Our approach consists in detecting user interface bad smells through model-based reverse engineering from source code. Models are used to define which widgets are present in the interface, when can particular graphical user interface (GUI) events occur, under which conditions, which system actions are executed, and which GUI state is generated next.

Keywords: GUI, Reverse Engineering, Bad Smells.

1 Introduction

In the Software Engineering area, the use of reverse engineering approaches has been explored in order to derive models directly from existing interactive system using both static and dynamics analysis [13] [14] [15]. Static analysis is performed on the source code without executing the application. Static approaches are well suited for extracting information about the internal structure of the system, and about dependencies among structural elements. Classes, methods, and variables' information can be obtained from the analysis of the source code. On the contrary, dynamic analysis extracts information from the application by executing it. Within a dynamic approach the system is executed and its external behaviour is analysed. Reverse engineering is a process that helps understand a computer system. Similarly, user interface modelling helps designers and software engineers understand an interactive application from a user interface perspective. This includes identifying data entities and actions that are present in the user interface, as well as relationships between user interface objects. This paper makes use of user interface models to detect bad smells of interactive systems. This aims to be achieved through a reverse engineering approach from source code.

The first step of our approach enables to extract models containing GUI characteristics. Models allow to analyse systems and could be used to detect bad smells at reasonable cost [9]. Different types of models can be used for interactive systems, like user and task models. Models must specify which GUI components are present in the interface and their relationship, when a particular GUI event may occur and the associated conditions, which system actions are executed and which GUI state is generated next. The main goal of this paper is to describe an approach to detect bad smells presence.

This paper is structured into seven logical sections. The first one presents an introduction. An user interface modelling is exposed, and the aspects usually specified by graphical user interfaces are described. It is described over section 2. Section 3 describes bad smells in graphical user interfaces, and section 4 presents background information on bad smells. This paper presents the approach proposed in sections 5 and 6. Section 5 presents methodologies for internal bad smells detection. Section 6 presents relevant preliminary results. Finally the last section presentsconclusions concerning the present research.

2 User Interface Modelling

User interface models can describe the domain over which the user interface acts, the tasks that the user interface supports, and others aspects of the graphical view presented to the user. The use of interface models gives an abstract description of the user interface, potentially allowing to express the user interfaces at different levels of abstraction, thus enabling choice of the most appropriate abstraction level, re-use user interface specifications between projects, thus decreasing the cost of development, reason about the properties of the models, thus allowing validation of the user interface within its design, implementation and maintenance processes.

Previous research from authors enable to extract dialogue models [4]. On the one hand they are one of the more useful type of models to design or analyse the behaviour of the system. On the other hand, they are one of type of models that is closest to the implementation, thus reducing the gap to be filled by reverse engineering. Dialogue models describe the behaviour of the user interface. Unlike task models, where the main emphasis is the users, dialogue model focus on the device, defining which actions are made available to users via the user interface, and how it responds to them. These models capture all possible dialogues between users and the user interface. Dialog models express the interaction between human and computer. To this end, they stipulate all the widgets the user can interact with (e.g. buttons, commands, etc.) and the results of those interactions on the system.

Using dialogue models, we focus our attention on bad smells and on automatic processes for their detection. Code smells are structural or behavioural characteristics of software that may indicate a code or design problem. Code smell concept was introduced by Fowler [11] who defined 22 different kinds of smells. Code smells are usefull to enhance software's internal quality through refactoring process.

3 Bad Smells in Graphical User Interfaces

The analysis of source code can provide a means to guide the evaluation of the application. Bad smells detection aim to address software quality. For that purpose, bad smells must be defined and searched in source code. Bad smells can be divided into two groups: external and internal. External bad smells are defined in relation to running software. In what concerns GUIs, external bad smells can be used as usability indicators. However, external bad smells are not obtainable from source code analysis, rather through users feedback. In contrast, internal bad smells are obtained from the source code, and provide information to improve software development. With our approach, we aims to extract internal bad smells obtained from source code through a reverse engineering process.

Internal bad smells are structural characteristics of source code that may indicate a code or design problem. Internal bad smell concept was introduced by Fowler [11] who defined 22 different kinds of smells, being usefull to enhance software's internal quality through refactoring process. Different smells types were specified by Fowler, like:

- *Duplicated Code*: means that the same code structure appears in more than one place;
- *Feature Envy*: means that a method is in the wrong place since it is more tightly coupled to the other class than to the one where it is currently located;
- *God Class*: refers to class that tends to perform too much work;
- *Large Class*: referes to classes that have too many instance variables or methods;

Considering different types of bad smells, we aim to detect them and discuss some of the relevant problems which we have to face for their automatic detection in interactive systems. To achieve that purpose adequate metrics must be specified and calculated. In what concerns graphical user interfaces, external bad smells presence can be used as usability indicator, while internal bad smells presence provide information to improve software structure.

As an example, duplicates structures can be detected by measuring the percentage of duplicated code lines in the system. However our objective will be to find not only exact duplication, wich is simple to detect, but also other kinds of duplication, including entity renaming or aliasing and also duplicated code slightly modified and mixed with different code. The detection of Feature Envy can be achieved by measuring the connections that a method has to methods belonging to foreign classes. As example, from a graphical perspective, a method related to a particular widgets may need to belong to the form's class where the widget is defined. By calculating metrics over dialogue models, relevant knowledge may be acquired about the structural characteristics of interface that may indicate a code or design problem.

4 Bad Smells Background

As described above, Fowler created a list of 22 smells and defined a possible solution for each of them [11]. In the sequence of Fowler study, Mantyla *et al.* [12] presented a taxonomy to categorize similar bad smells. The taxonomy makes the smells more understandable and recognizes the relationships between smells. Mantyla *et al.* created five groups of smells, namely, the bloaters, the object-oriented abusers, the change preventers, the dispensables and the couplers. Both Fowler and Mantyla use metrics to bad smell detection. Stamelos et al. [6] used also metrics within the *Logiscope2* tool in order to study the quality of open source code. Ten different metrics were used. The results enable evaluation of each function against four basic criteria: testability, simplicity, readability and self-descriptiveness. While the GUI layer was not specifically targeted in the analysis, the results indicated a negative correlation between component size and user satisfaction with the software.

5 Metrics for Internal Bad Smells Detection

The approach described in this paper makes use of a fully functional reverse engineering prototype tool developped by authors. The tool makes use of static analysis as in [5] and is able to derive user interface models of interactive applications from source code.

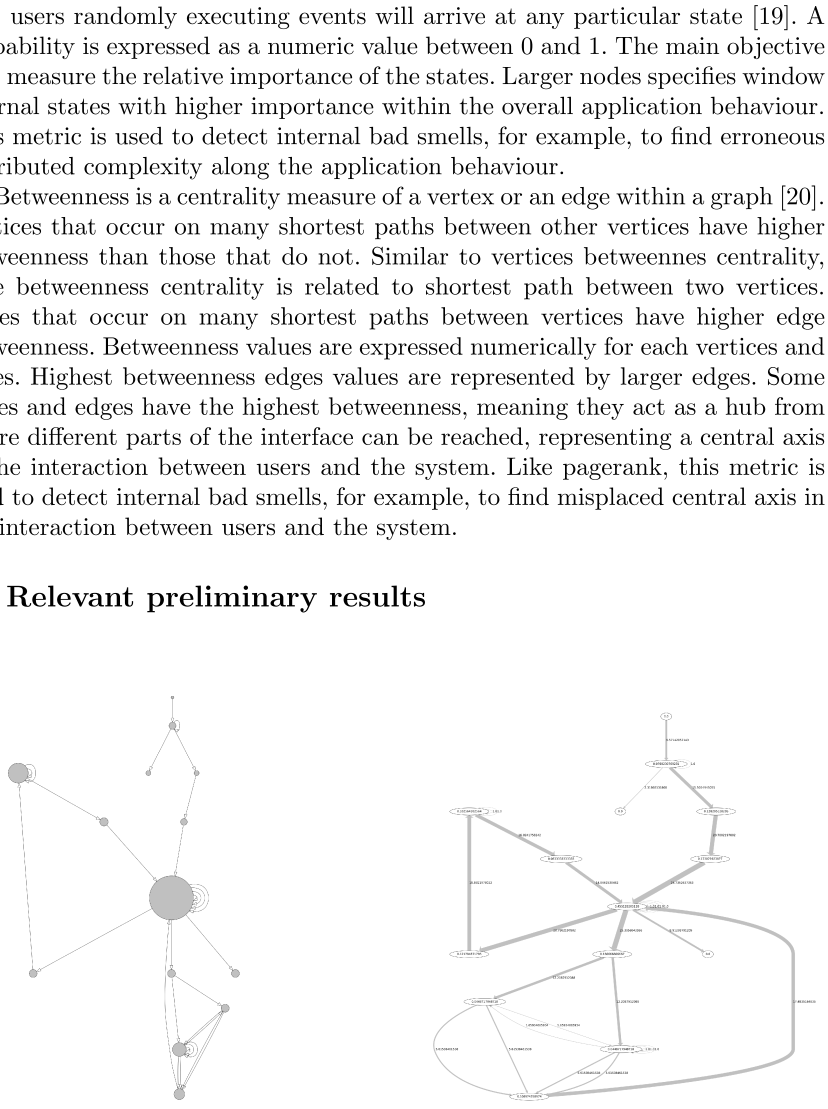
The interactive source code extraction process starts by defining/reusing a front-end for the programming language of the interactive applications source code. Modern parser generators automatically produce a parser and the construction of the Abstract Syntax Tree (AST) given the context-free grammar defining the programming language of the source code. Using this front-end, an AST is obtained from the source code of the system for which the user interface related code is to be analysed. Then, the process needs to identify all fragments in the AST that are members of the GUI layer. To achieve this a set of abstractions is used. In order to extract user interface relevant data from the AST, a slicing function was proposed which enables to isolate the GUI sub-program from the entire program. Behavioural models may be generated which capture graphical user interface behaviour by detecting components in the user interface through source code analysis. These components include user interface objects, events, actions and respective control flow. The technique will help in identifying graphical user interface abstractions from source code.

Different metrics have already been applied to the generated models. These metrics are used to detect bad smells like duplicated code, methods in wrong place (feature Envy), classes performing too much work (God Classes), classes with too many instance variables or methods (Large Class), methods too long (Long Method), parameter list too long (Long Parameter List). Another metrics have been applied to detect internal bad smells related to the interaction between users and the system, namely pagerank and betweenness.

Pagerank is a link analysis algorithm, that assigns a numerical weighting to each node [19]. Pagerank is a distribution used to represent the probability that users randomly executing events will arrive at any particular state [19]. A probability is expressed as a numeric value between 0 and 1. The main objective is to measure the relative importance of the states. Larger nodes specifies window internal states with higher importance within the overall application behaviour. This metric is used to detect internal bad smells, for example, to find erroneous distributed complexity along the application behaviour.

Betweenness is a centrality measure of a vertex or an edge within a graph [20]. Vertices that occur on many shortest paths between other vertices have higher betweenness than those that do not. Similar to vertices betweennes centrality, edge betweenness centrality is related to shortest path between two vertices. Edges that occur on many shortest paths between vertices have higher edge betweenness. Betweenness values are expressed numerically for each vertices and edges. Highest betweenness edges values are represented by larger edges. Some states and edges have the highest betweenness, meaning they act as a hub from where different parts of the interface can be reached, representing a central axis in the interaction between users and the system. Like pagerank, this metric is used to detect internal bad smells, for example, to find misplaced central axis in the interaction between users and the system.

6 Relevant preliminary results



(a) Pagerank result (b) Betweeness result

Fig. 1. HMS behavioral results

The application of our approach has been implemented to a large real interactive system written by third party: a Healthcare management system (HMS). This application is a proof of concept for the approach. The HMS system is implemented in Java/Swing and supports patients, doctors and bills management.

The implementation contains 66 classes, 29 windows forms (message box included) and 3588 lines of code. The approach enabled the extraction of different behavioural models. Methodologies have been also applied automating the activities involved in GUI model-based reasoning, such as, pagerank and betweenness algorithms. GUI behavioural metrics have been used as a way to detect bad smells. As example, through Figure 1 one can observe pagerank and betweeness HMS automatically generated results, which enable to detect visually internal bad smell related to overall HMS system complexity. For both models (Figures 1(a) and 1(b)) each node specify a particular window state and each edge specify a particular transition between windows states. The left model (Figure 1(a)) enables to visually detect erroneous distributed complexity along the application behaviour. The right model (Figure 1(b)) enables to find visually misplaced central axis in the interaction between users and the system. This case study demonstrate that the approach enables the analysis of real interactive applications written by third parties.

7 Conclusions and Future Work

Tools are currently available to developers that allow for the fast development of user interfaces with graphical components. However, the design of interactive systems does not seem to be much improved by the use of such tools. As described in this paper Fowler popularized the concept of program smells in the context of object-oriented programming. The presence of bad smells in software code can make programs harder to understand, maintain, and evolve. The detection of bad smells allows programmers to improve their programs by eliminating them.

In this paper we have discussed an approach for the detection of graphical user interface bad smells through model-based reverse engineering from source code. We have presented smells that could be considered from a GUI enhancement perspective. In the future, we intend to extend the approach presented in this paper enabling it to detect automatically a more significant amount of smells using our tool. To discover smells our approach is based on software metrics which aim to address software quality by measuring software aspects. Since the smells are usually associated with refactorings that can eliminate them, we plan to improve GUI application through refactoring. These are promising research directions that we are already exploring and whose results we plan to bring out in the near future.

References

1. Brad A. Myers, *Separating Application Code from Toolkits: Eliminating the Spaghetti of Call-backs*, School of Computer Science, Carnegie Mellon University, 1999.
2. A. M. Memon. *A Comprehensive Framework for Testing Graphical User Interfaces.* PhD thesis, Department of Computer Science, University of PittsBurgh, july 2001.
3. B. Berard. *Systems and Software Verification.* Springer edition, 2001.

4. João Carlos Silva, José Creissac Campos, and João Saraiva. *A generic library for GUI reasoning and testing.* 24th Annual ACM Symposium on Applied Computing, USA, March 2009.
5. M. M. Moore. *Rule-based detection for reverse engineering user interfaces.* Proceedings of the Third Working Conference on Reverse Engineering, pages 42-8, Monterey, CA, november 1996.
6. Ioannis Stamelos, Lefteris Angelis, Apostolos Oikonomou, and Georgios L. Bleris. *Code quality analysis in open source software development.* Information Systems Journal, 12:4360, 2002.
7. Young Sik Yoon and Wan Chul Yoon. *Development of quantitative metrics to support ui designer decision-making in the design process.* In Human-Computer Interaction. Interaction Design and Usability, pages 316324. Springer Berlin / Heidelberg, 2007.
8. Harold Thimbleby and Jeremy Gow. *Applying graph theory to interaction design.* pages 501519, 2008.
9. Steven P. Miller, Alan C. Tribble1, Michael W. Whalen1, and Mats P.E. Heimdahl. *Proving the shalls early validation of requirements through formal methods.* 2004.
10. J. Nielsen. *Usability Engineering.* Academic Press, San Diego, CA, 1993.
11. Fowler, M. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, Boston, MA, USA. 1999.
12. Mantyla, M., Vanhanen, J., Lassenius, C. *A taxonomy and an initial empirical study of bad smells in code.* In: Proceedings of the International Conference on Software Maintenance. pp. 381-384. ICSM'03, IEEE Computer Society, Washington, DC, USA. 2003.
13. Ana C. R. Paiva, João C. P. Faria, and Pedro M. C. Mendes, editors. *Reverse Engineered Formal Models for GUI Testing*, 10th International Workshop on Formal Methods for Industrial Critical Systems, 2007.
14. J. Chen and S. Subramaniam. *A GUI environment for testing guibased applications in Java.* Proceedings of the 34th Hawaii International Conferences on System Sciences, 2001.
15. T. Systa. *Dynamic reverse engineering of Java software.* Technical report, University of Tampere, Finland, 2001.
16. Yoshio Kataoka, Takeo Imai, Hiroki Andou, and Tetsuji Fukaya. *A quantitative evaluation of maintainability enhancement by refactoring.* In Proceedings of International Conference on Software Maintenance (ICSM 2002), pages 576585, Montral, Canada, October 2002. IEEE Computer Society. doi:10.1109/ICSM.2002.1167822
17. Ladan Tahvildari and Kostas Kontogiannis. *A metric-based approach to enhance design quality through meta-pattern transformations.* In Proceedings of the Seventh European Conference on Software Maintenance and Reengineering, pages 183192, Benevento, Italy, March 2003. doi:10.1109/CSMR.2003.1192426.
18. Francesca Arcelli Fontana and Stefano Spinelli. *Impact of refactoring on quality code evaluation.* In Proceeding of the 4th workshop on Refactoring tools, WRT 11, pages 3740, Waikiki, Honolulu, HI, USA, 2011. ACM. Workshop held in conjunction with ICSE 2011. doi: 10.1145/1984732.1984741.
19. Pavel Berkhin. *A survey on pagerank computing.* Internet Mathematics, 2:73120, 2005.
20. Shu Yan Shan and et al. *Fast centrality approximation in modular networks*, 2009.