Characterizing the Functional Roles of Classes and Methods by Analyzing Feature Traces

Orla Greevy and Stéphane Ducasse
Software Composition Group
University of Bern, Switzerland
{greevy, ducasse}@iam.unibe.ch

Abstract

Software developers are often faced with the task of maintaining or extending large and complex applications, with which they are unfamiliar. Typically change requests and bug reports are expressed in terms of system features. Much of the maintenance effort is spent trying to identify which classes and methods provide functionality to individual features. To tackle this problem, we propose an approach based on dynamic analysis that exploits the relationships between features and software entities. Our definition of a feature is a unit of observable behavior of a software system. We apply our approach to large open source application and identify the key classes and methods which provide functionality for individual features.

Keywords: reverse engineering, features, feature-traces, feature model dynamic analysis, static analysis.

1 Introduction

Most reverse engineering approaches to software analysis focus on static source code entities of a system, such as classes and methods [3, 16, 17]. A static perspective considers only the structure and implementation details of a system. Thus, key semantic information about the roles of software entities in the features of a system is overlooked. Without explicit relationships between features and the entities that implement their functionality, it is difficult for software developers to maintain and extend the code.

Several works have shown that exercising the features of a system is a reliable means of correlating features and code [6, 24]. In a previous work [11], we describe a feature-driven approach based on dynamic analysis, in which we extract execution traces to achieve an explicit mapping between features and software entities like classes and methods. The focus of our approach is on object-oriented systems. We characterize features in terms of classes and methods that implement their functionality, and we characterize classes and methods based on how they participate in features.

One of the main problems of dynamic analysis is the large volume of data generated, thus making it difficult to reason about the data. In our approach we apply measurements to the feature traces and compact the data without loss of information about the relationships between features and classes or features and methods.

In this paper, we refine the approach of our previous work [11] and apply it to a large open source case study argoUML which is implemented in Java. Previously we applied our approach on medium sized applications implemented in Smalltalk and established a correlation between features and classes. Our goal with this paper is to (1) illustrate the scaleability of the approach, (2) its applicability to finer grained software entities such as methods and (3) to underline the language independence of the feature model we abstract.

We start by introducing the terminology we use to characterize software entities from a features perspective Section 2. In Section 3 we outline the mechanisms of our technique. In Section 4 we report on the open source case study argoUML conducted using our approach. Subsequently, in Section 5 we discuss our results. We summarize related work in Section 7. Section 8 outlines our conclusions and future work.

2 Feature Characterization

In this section we briefly outline the key background terminology of our approach to correlating features with code.

We adopt the definition of Eisenbarth *et al.* for *features* [6]. A *feature* is an observable unit of behavior of a system triggered by the user. We analyze the relationship between the features and classes by exercising the features and capturing their execution traces, which we refer to as *feature-traces*. We refer to the set of extracted feature-traces as a

feature model.

We focus on the relationships between both features and classes and features and methods. Moreover, our approach is generally applicable to coarser-grained units such as packages.

Our characterizations of classes and methods express their level of participation in a set of features under analysis. We define four distinct *class/method characterizations* as:

- *Not Covered (*NC*)* is a class/method that does not participate any of the features-traces of our current feature model.
- *Single-Feature (*SF*)* is a class/method that participates in only one feature-trace.
- *Group-Feature (*GF*)* is a class/method that participates in less than half of the features of a feature model. In other words, group-feature classes/methods provide functionality to a group of features, but not to all features.
- *Infrastructural (*I*)* is a class/method that participates in more than half of the features of a feature model.

We define a class characterization measurement *a class characterization in terms of feature participation* (FC) to compute the characterization of a class as NC, SF,GF or I as previously described. Similarily, we apply the *a method characterization in terms of feature participation* (FM) to compute the characterization of a method.

We compact a feature-trace into *feature-fingerprints* by reducing multiple references to the same class/method to one occurrence. In this way, we reduce the volume of data captured as a result of dynamic analysis without loss of information about the relationships between features and classes or features and methods.

A feature-fingerprint represents a set of sets characterized classes or a set of sets of characterized methods.

(FP_C) is a set of sets of characterized classes: $FP_i = \{\{NC(classes)\}, \{SF(classes)\}, \{GF(classes)\}, \{I(classes)\}\}$

Figure 1 shows a simple visualization of class characterizations and feature-fingerprints for 5 features. The arrows between the features (F1..F5) show which classes participate in features. The classes are color-coded according to their characterizations. Infrastructural classes are shown in dark gray. As previously explained, these participate in more than half of the features. On the right side we show the feature-fingerprints of color-coded parts, each representing the set of characterized classes that participate in the feature. The feature-fingerprint for F1, for example, consists of a set of one single-feature class, a set of one group feature-class and a set of two infrastructural classes. The cardinality of each set of characterized classes is represented by the height of the colored part.

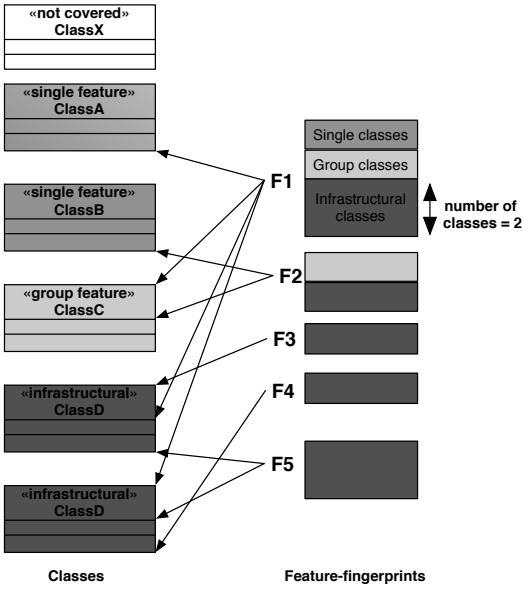

Figure 1. Feature-Fingerprints and Classes Relationships

3 Applying our Feature Analysis Technique

We outline how we apply our technique to obtain classes characterizations from a feature perspective.

- We apply static analysis and abstract a static model of the source code entities of the application.
- We abstract feature-traces for a set of features using dynamic analysis. To achieve this we instrument the code and execute the features.
- We model the feature-traces as first class entities and incorporate them into the static model of the source code. By doing so we establish the relationships between the methods calls of the feature traces and the static model class and method entities,
- We compact the feature-traces into *feature-fingerprints* for the class characterizations and *feature-fingerprints* for the method characteriztaions by applying feature characterization measurements *a class characterization in terms of feature participation* (FC and *a method characterization in terms of feature participation* (FM .

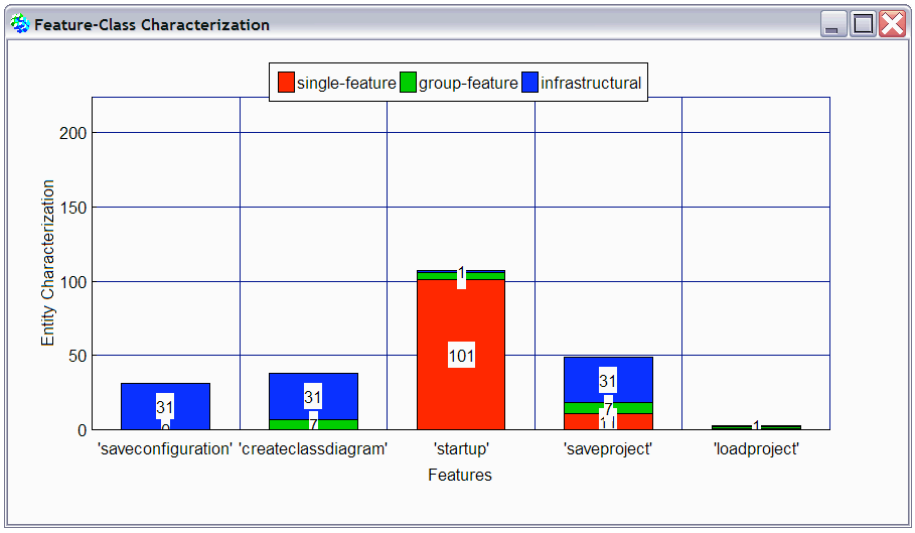

Figure 2. ArgoUML Class Characterization w.r.t. Features.

4 Feature Analysis of ArgoUML

In this section we present the results of applying our approach to the *ArgoUML* case study. *ArgoUML* is a UML modelling application implemented in Java. We applied static analysis to the java source and we extract 1735 Classes and 11762 methods.

ArgoUML provides us with a graphical user interface to create and manipulate UML diagrams. A set of UML diagrams are associated with a project. Projects can be created, saved to disk and reloaded in a later session. For our feature analysis of *ArgoUML* we consider 5 features, which represent typical user interactions with the application, namely:

- Save a Project
- Load a Project
- Save Configuration
- Create a Class Diagram
- Startup the application

Figure 2 shows the *feature fingerprints* of class characterizations. In the feature *Save a Project* we identify 11 single feature classes. The feature *Load a Project* we identify 1 single feature class. We summarize the class characterizations in Table 1

In the case of the features *Save a Project* and *Load a Project*, we discover the classes that manage persistency. The class *org.argouml.persistence.AbstractFilePersister* is characterized as a *group feature* class as it participates in these two features of our model. The class *org.argouml.persistence.ZargoFilePersister* is characterized as *single-feature* as it participates only in the feature *Save a Project*.

Feature	SF	GF	I
Save Configuration	0	0	31
Create a Class Diagram	0	7	31
Startup	101	1	1
Save a Project	11	7	31
Load a Project	1	1	1

Table 1. ArgoUML Class Characterizations w.r.t. Features

5 Discussion

The large volume of information and complexity of dynamic information makes it hard to infer higher level of information about the system. Our approach reduces the complexity of the information to reveal key semantical information about the system based on measuring how the relationships between classes and features.

Our feature perspective enables us to view semantic groupings of the classes and methods. The characterizations provides us with feature knowledge to reason about the design intent of the class and methods.

Feature definition. Not all features of a system satisfy our definition of a feature as a user-triggerable unit of observable behavior. System internal housekeeping tasks, for example, are not triggered directly by user interaction. For

the identification of features we limit the scope of our investigation to user-initiated features.

Coverage. We limit the scope of our investigation to focus on a set of features. Our feature model does not achieve 100% coverage of the system. We argue that for the purpose of feature location, complete coverage is not necessary. We use our feature model to focus on a specific set of features. The model is extensible and the approach to analysis is extensible to include more features if required.

Scaleability The results obtained show that our approach is applicable to large applications. The feature-traces are compacted and thus we can infer highlevel information from the feature-fingerprints of class and method characterizations.

Language Independence Obtaining the traces from the running application requires code instrumentation. The means of instrumenting the application is language dependent. For this experiment we used a Java profiler to extract traces. We abstract a feature model of the traces which is the same for every language. Our analysis is performed on the feature model.

6 Implementation - TraceScraper and Moose

TraceScraper is our feature analysis tool. It is based on the Moose [5] reengineering platform.For the purpose of this experiment we used a Java profiler tool to instrument the code. We manually activated a set of features by interacting with the GUI and thus extracted individual execution traces. We extended out TraceScraper tool to import these traces and model them as FAMIX [4] entites in Moose. The execution traces contain calls to java library classes. Only classes that exist in our static model are considered relevent for our analysis Figure 3 shows the relationship between trace entities and the FAMIX entities *Class* and *Method*. TraceScraper computes feature-fingerprints from the trace entities based on the relationships to *class* and *method* entities.

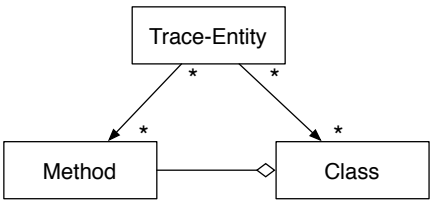

Figure 3. TraceMetaModel.

Moose is an implementation of the FAMIX [4] language independent meta-model. We extend the FAMIX model with *feature-trace* entities. In this way we can relate the feature-trace information with the class and method entities of the model.

7 Related Work

Many researchers have identified the potential of feature-centric approaches in software engineering and in particular as a basis for reverse-engineering [22, 19, 7, 20, 23, 24, 13, 6]. Our main focus with this work is define a reverse engineering approach that exploits history information of a systems features over a series of versions.

The basis of our work is directly related to the field of dynamic analysis [1, 25, 12], user-driven approaches [15, 14] and reverse engineering approaches that consider the evolution of a system [10, 26, 2, 18, 21] represent the groundwork on which we base our research.

Wilde and Scully [23] developed a method called *Software Reconnaissence*. They uses test cases to aid in locating product features. They have applied their methodology to legacy system case studies written in C.

Eisenbarth et al. [6] describe a methodology which combines dynamic, static and concept analysis. They collect execution traces and categorize the methods according to their degree of specificity to a given feature. The analysis automatically produces a set of concepts which are presented in a lattice. Using this technique they identify general and specific parts of the code.

Wong et al. [24] propose three different metrics for measuring the binding of features to components or program code. They quantitatively capture the disparity between a program component and a feature, the concentration of a feature in a program component, and the dedication of program component to a feature.

Hamou-Lhadj et al. [12] described an approach extracting behavioral views as use case models. They filter out utility methods. They using a an algorithm based on fan-in analysis to detect utilities.

Our approach complements these approaches. In contrast to the above approaches [13, 9, 8], our main focus is applying feature-driven analysis to object-oriented applications. We use execution traces to establish the link between features and software entities. Our characterizations add semantic information to the software entities and use this semantic information to reason about their functional roles in the system in terms its features.

8 Conclusions and Future Work

Reverse engineering approaches tend to focus on the implementation details and static structure of a system. By do-

ing so they overlook key knowledge about the system which establishes the semantic purpose of the individual software entities.

Our goal is to analyze the functional roles of classes and methods from a feature perspective and to obtain a feature-model of a system.

We applied our approach to a large case study and showed how a feature perspective of a system is useful for interpreting the functional roles of classes and methods in the system. We reduce the of a large volume of trace data so that we are able to reason about the information and infer high level information about the classes and methods of the system. The characterization for classes and methods as *single feature* reduces the volume of information a software developer needs to consider when performing a maintenance task for a specific feature.

Similarily our characterization of classes and methods as infrastructural identifies those software entities that provide general functionality to more that one feature of the system. These infrastructure classes and methods may correspond to the *utilities* identified by Hamou-Lhadj et al. [12].

In the future, we would like extend our definition of a feature to consider variations in the external behaviors of the system. In addition, we plan to extend our feature representation within the feature model to include multiple paths of execution of a feature. We expect that as a result we achieve a higher coverage of classes and methods and increase the accuracy of our approach.

Acknowledgments: We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "The Achievement and Validation of Evolution-Oriented Software Systems" (SNF Project No. PMCD2-102511).

References

[1] T. Ball. The Concept of Dynamic Analysis. In *Proceedings of ESEC/FSE '99 (7th European Software Engineering Conference and 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, number 1687 in LNCS, pages 216–234, sep 1999.

[2] E. Burd and M. Munro. An initial approach towards measuring and characterizing software evolution. In *Proceedings of the Working Conference on Reverse Engineering, WCRE '99*, pages 168–174, 1999.

[3] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *Proceedings of OOPSLA '2000 (International Conference on Object-Oriented Programming Systems, Languages and Applications)*, pages 166–178, 2000.

[4] S. Demeyer, S. Tichelaar, and S. Ducasse. FAMIX 2.1 — The FAMOOS Information Exchange Model. Technical report, University of Bern, 2001.

[5] S. Ducasse, T. Gîrba, M. Lanza, and S. Demeyer. Moose: a Collaborative and Extensible Reengineering Environment. In *Tools for Software Maintenance and Reengineering*, RCOST / Software Technology Series, pages 55 – 71. Franco Angeli, 2005.

[6] T. Eisenbarth, R. Koschke, and D. Simon. Locating Features in Source Code. *IEEE Computer*, 29(3):210–224, Mar. 2003.

[7] M. El-Ramly, E. Stroulia, and P. Sorenson. Recovering software requirements from system-user interaction traces. In *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 447–454, 2002.

[8] M. Fischer and H. Gall. Visualizing feature evolution of large-scale software based on problem and modification report data. *Journal of Software Maintenance and Evolution*, 2004.

[9] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Proceedings of the International Conference on Software Maintenance (ICSM 2003)*, pages 23–32, Sept. 2003.

[10] T. Gîrba, M. Lanza, and S. Ducasse. Characterizing the evolution of class hierarchies. In *Proceedings of European Conference on Software Maintenance (CSMR 2005)*, 2005.

[11] O. Greevy and S. Ducasse. Correlating features and code using a compact two-sided trace analysis approach. In *Proceedings of CSMR 2005 (9th European Conference on Software Maintenance and Reengineering*. IEEE Computer Society Press, 2005.

[12] A. Hamou-Lhadj, E. Braun, D. Amyot, and T. Lethbridge. Recovering behavioral design models from execution traces. In *Proceedings of CSMR 2005 (9th European Conference on Software Maintenance and Reengineering*. IEEE Computer Society Press, 2005.

[13] I. Hsi and C. Potts. Studying the evolution and enhancement of software features. In *Proceedings of the 2000 IEEE International Conference on Software Maintenance*, pages 143–151, 2000.

[14] I. Jacobson. Use cases and aspects—working seamlessly together. *Journal of Object Technology*, 2(4):7–28, July 2003.

[15] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. *Object-Oriented Software Engineering — A Use Case Driven Approach*. Addison Wesley/ACM Press, Reading, Mass., 1992.

[16] J. Krajewski. QCR - A methodology for software evolution analysis. Master's thesis, Information Systems Institute, Distributed Systems Group, Technical University of Vienna, Apr. 2003.

[17] M. Lanza and S. Ducasse. Understanding software evolution using a combination of software visualization and software metrics. In *Proceedings of LMO 2002 (Langages et Modèles à Objets*, pages 135–149, 2002.

[18] M. M. Lehman, D. E. Perry, and J. F. Ramil. Implications of evolution metrics on software maintenance. In *Proceedings of the International Conference on Software Maintenance (ICSM 1998)*, pages 208–217, 1998.

[19] D. Licata, C. Harris, and S. Krishnamurthi. The feature signatures of evolving programs. *Automated Software Engineering*, 2003.

[20] A. Mehta and G. T. Heineman. Evolving legacy systems features using regression test cases and components. In *Proceedings of the 4th international workshop on Principles of software evolution*, pages 190–193. ACM Press, 2002.

[21] T. Mens and M. Lanza. A graph-based metamodel for object-oriented software metrics. *Electronic Notes in Theoretical Computer Science*, 72(2), 2002.

[22] C. R. Turner, A. L. Wolf, A. Fuggetta, and L. Lavazza. Feature engineering. In *Proceedings of the 9th International Workshop on Software Specification and Design*, page 162. IEEE Computer Society, 1998.

[23] N. Wilde and M. C. Scully. Software reconnaisance: Mapping program features to code. *Software Maintenance: Research and Practice*, 7(1):49–62, 1995.

[24] W. E. Wong, S. S. Gokhale, and J. R. Horgan. Quantifying the closeness between program components and features. *J. Syst. Softw.*, 54(2):87–98, 2000.

[25] A. Zaidman, T. Calders, S. Demeyer, and J. Paredaens. Applying webmining techniques to execution traces to support the program comprehension process. In *Proceedings of CSMR 2005 (9th European Conference on Software Maintenance and Reengineering*. IEEE Computer Society Press, 2005.

[26] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *26th International Conference on Software Engineering (ICSE 2004)*, pages 563–572, 2004.