How Developers Develop Features

In Proceedings of European Conference on Software Maintenance and Reengineering (CSMR 2007)

Orla Greevy[1], Tudor Gîrba[1] and Stéphane Ducasse[2]

[1] Software Composition Group - University of Berne, Switzerland
[2] LISTIC - University of Savoie, France

Abstract

Software systems are typically developed by teams of developers, with responsibilities for different parts of the code. Knowledge of how the developers collaborate, and how their responsibilities are distributed over the software artifacts is a valuable source of information when reverse engineering a system. Determining which developers are responsible for which software artifacts (e.g., packages or classes) is just one perspective. In this paper we complement the static perspective with the dynamic perspective of a system in terms of its features. We want to extract information about which developers are responsible for which features. To achieve these two perspectives, we correlate developer responsibilities both with a structural view of the system and with a feature view. We identify which developers are responsible for which features, and whether the responsibilities correspond with structural source code artifacts or with features. We apply our technique to two software projects developed by two teams of students as part of their course work, and to one large open source project.

Keywords: reverse engineering, software comprehension, features, dynamic analysis, development strategies.

1 Introduction

Many reverse engineering techniques consider source code as the most reliable source of information about a system. Few researchers in the field of reverse engineering have devoted much attention to the way developers interact with the system and the roles they play [17]. However, understanding how the developers build a software system represents a rich source of information for the reverse engineer. For example, it is useful to know who is responsible for which part of the system, or who developed which features [6].

Typically, software systems are built by teams of developers. It is a well known phenomena that the human factors such as collaborations and communication paths are often reflected in the structure of the code. According to Conway's law *"organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations"* [5].

The structure of a development team and the division of responsibilities has a major impact on how software system is structured and implemented. The problem is how to efficiently exploit and access knowledge about which developer developed which parts of the code, and how they collaborated during the implementation.

Often a discrepancy exists in the way developers and domain analysts see a system. Typically the mental model of the developer reflects the structural source code artifacts such as packages and classes. Domain analysts and users on the other hand, see the system in terms of features (*i.e.,* the capabilities of the system). Thus, understanding which developers are responsible for which software artifacts, and which developers are responsible for which features is the key to maintaining a traceability between the external perspective of a system and its internal structure.

In this paper, we focus on a features perspective of a system. We address the following questions:

- *Which developers or groups of developers are responsible for which features?* We seek to extract information that would not only lead to the relevant part of the code for a feature, but also to reveal the developers responsible for its implementation and maintenance. From a static perspective, developers who own the most classes in the system represent key developers. From a dynamic perspective, we assume that developers that are responsible for multiple features have a wider domain knowledge of the system than developers who contribute only to specific parts. Bug reports and change requests are expressed in a language that reflect the features of a system [18]. Thus, knowledge of which developer is responsible for which feature is useful especially when faced with the problem of assigning bug reports and change requests to the devel-

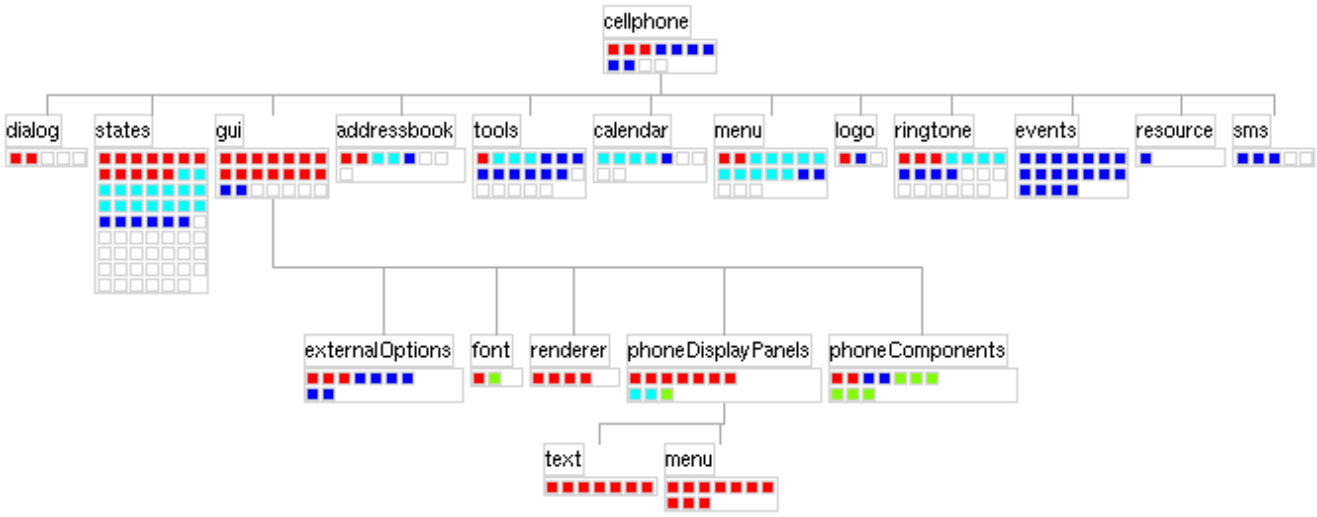

Figure 1. Package Owner View of the PhoneSimulator-1 case study showing classes arranged in packages and colored by owner

opers of large open source projects [2, 3].

- *Do developers develop features or do they develop functional blocks?* We aim to determine if the division of development responsibilities corresponded with structural source code artifacts or with feature concerns. We seek to reverse engineer which developers best understand parts of the system and whether the developers develop on static architectural boundary or on a feature boundary.

To address these questions, we analyze the correlation between developer behavior in a system in the contexts of its packages and of dynamic groups that reflect its features. To obtain the relationship between developers and classes we extract data from the source code repository of a project and use this to determine the *ownership* of a class (*i.e.,* the responsible developer) [11]. We then build on a feature analysis technique of our previous work [13] to establish the relationships between features and classes, and finally we link developers to features.

Structure of the Paper. In Section 2 we introduce the key elements of our analysis technique. In Section 3 we describe our structural and feature views and briefly explain how we apply our feature analysis technique in the context of developers. In Section 4 and Section 5 we report on three case studies conducted using our approach. Subsequently, in Section 6 we discuss our results and outline the constraints and limitations of our approach, and propose possible variation points. We summarize related work in Section 7, and we conclude in Section 8.

2 Extracting Developer Data from Work Artifacts

Recent research in the field of reverse engineering and system comprehension reveals a growing awareness in the role of the software developer in the software development process [17]. Lethbridge *et al.* define a taxonomy of techniques for collecting data about the developers involved in a software project. For our experimentation, we adopt what they refer to as a *third degree approach*. In other words we analyze work artifacts in an attempt to uncover information about the responsibilities of software developers of a system. The inputs of our analysis are: (1) source code repository log information (2) source code and (3) execution traces of features. Our approach makes use of the distribution map-like visualization [7] to represent the structural and feature perspectives of developer responsibilities.

In a previous work we describe a technique to define code ownership based on the data extracted from the CVS log of a project [11]. The technique is based on the assumption that the developer of a line of code is the most knowledgeable in that line of code. Based on this assumption, we determine the owner of a piece of code as being the developer that owns the most lines of that piece of code.

As CVS is a file-based repository, strictly speaking our ownership is calculated for files. However, for our experimentation we assume a one-to-one mapping between files and classes. We exploit the fact that the package structure in Java reflects the physical file structure and use this to determine which class maps to which file.

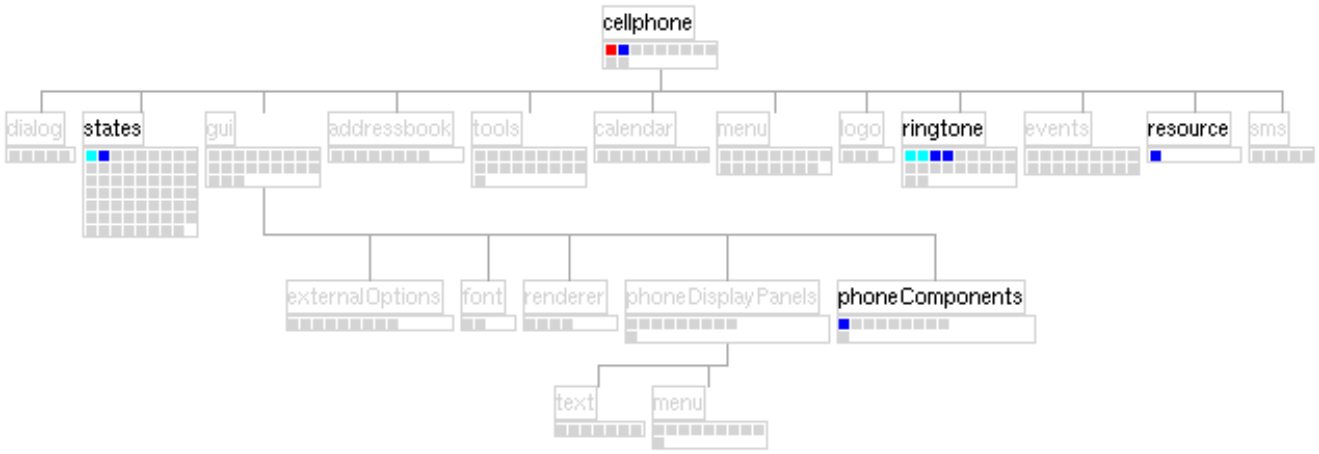

Figure 2. The Package Owner View of PhoneSimulator-1 showing classes organized in packages, where only the classes participating in the Play Ringtone feature are colored by owner.

We visualize who owns what parts by representing each owner with a unique color [7, 11]. Figure 1 shows the package hierarchy extracted from one of our case studies: the large boxes represent packages and are arranged in a tree, while the small boxes represent classes and are colored by the owner. This view reveals a developers perspective of the system structure. We detect packages, where there is only one responsible developer, or one main responsible developer (*i.e.,* a developer owns most of the classes in the package). We refer to these packages as *developer-focused packages*. In contrast, there are packages with classes owned by many developers.
With respect to the extraction of developer strategies within a project, our first hypothesis is:

A high-proportion of developer-focused packages *indicates that the developers adopted a strategy that reflected a structural division of responsibilities (on a package boundary) while developing the system.*

3 Feature Owner Analysis Technique

The main contribution of this paper is that we extend the focus of ownership analysis beyond ownership of code to consider the roles of developers with respect to features. We want to discover if the division of developer responsibilities in a team was influenced directly by *feature* requirements, hence we need to establish the relationships between features and developers. To achieve this, we apply a feature location technique from our previous work [12, 13]. Feature location is a recognized technique for revealing which parts of the code relate to a feature [1, 9, 20]. Typically feature location techniques define features to be user-triggerable actions of a system [9]. Essentially, we instrument the system and exercise a set of features and we characterize the classes based on their level of participation in the features traced. We define a *feature view* as a group of classes that participate in execution of the feature.

We differentiate between classes that implement *feature-specific* functionality (we named these *single-feature* and *low-group-feature*), and classes that provide functionality to a more than one of the features (we named these *high-group-feature* and *infrastructural*) [14]. For more details of how we detect the feature characterization of classes, we refer the reader to our previous works [12, 13].

3.1 Owners of Features

We establish developer responsibilities with respect to features by identifying the developers that own the *feature-specific* classes. For example, in Figure 2, we highlight the classes that participate in a feature by the owner of the class. All the classes that are not specific to the feature colored gray.
Our second hypothesis is:

A high-proportion of developer-focused feature views *indicates that the developers adopted a developer strategy that reflected a feature perspective of responsibilities (on a feature boundary) while developing the system.*

3.2 Modeling Developer Collaborations as Feature Teams

Features are can be built by one or more developers. To understand how developers collaborate to develop features,

we extract and explicitly model teams of developers and the relationships between teams. We define a *feature team* to be the set of owners of the classes of a *feature view*. A feature team models the collaborations between developers that exist during the development of a feature.

Typically developers or teams of developers are responsible for one or more features in the system. Once we have extracted which developers are responsible for features as teams, we can represent the relationship between the developer teams in a graph. Each node of the graph represents a team (one or more developers) and contains one or more feature views. A team of developers may represent a subset of another team. We model the *is a subset of* relationship between the teams as edges of the graph and perform transitive reduction to remove unnecessary edges.

For example Figure 3 shows a Team Collaboration View from one of our case studies. Each large rectangle represents a team formed by one or more developers. The graph shows the responsibilities of a developer and possible collaborations between developers with respect to a set of features we traced. Inside the developer collaboration rectangle, we represent each feature as a grouping of *feature-specific* classes. As with the package view, the classes are colored according to the owners. For example, the top box represents the team that includes the red, blue, cyan and green developers and the one on the bottom is a team formed by the red and blue developers. The bottom team represents two developers who collaborated to develop two of the features: *viewHelp* and *viewAbout*.

3.3 Developer-focused Features

As with our structural perspective, we define a feature F to be *developer-focused* if a high proportion of the classes are owned by one developer (*i.e.,* more than half of the classes). Thus, the existence of *developer-focused* features indicates that a feature-based development strategy was adopted during the development of the system. For this analysis we define a developer-focused feature to be one where more than 50% of the classes are owned by one developer.

4 Case Study 1: Student Team Projects

To validate our technique, we applied it on five student team projects (each of approximately 200 classes). Each team consisted of four students working over a time span of four months period as part of their course work. The applications were developed in Java and CVS was used as a source code repository. The goal of these projects was to implement a cell phone simulator.

In this section we report on our findings for two of these projects. We refer to the projects as PhoneSimulator-1 and PhoneSimulator-2. The requirements for the project were defined in terms of user stories.

Our motivation for choosing these projects were: (1) we have access to the CVS repository to obtain the information we need to calculate the ownership of files, (2) the resulting systems are the result of team effort, and (3) our approach is a heuristic-based approach and we require developer knowledge to validate our results.

4.1 Context Definition

We outline our approach to analyzing the team projects:

- For each system, we extract a static model from the source code and model it in the Moose reengineering environment [8].
- We identify the features of these systems by associating them with the *user-triggerable* actions accessible via their user interfaces. For each system, we instrument the application using the JIP profiler [16] and the capture traces when triggering the features. We resolve the traces and establish relationships between the method and class references within the trace with our static model using DynaMoose (a dynamic analysis and trace modeling tool integrated in Moose).
- We process the CVS log information for each application to determine the file ownership. We map files to classes, and we generate (1) a Package Owner View for the entire system and also just highlighting the *feature-specific* classes, and (2) a Team Collaboration View showing how developers collaborated to develop features (as in Figure 3).
- We drive our analysis by addressing the questions we identified in the introduction. We validate our findings with the development team members

4.2 PhoneSimulator-1

This system consists of 251 classes and 20 packages and was developed by 4 developers. We traced 14 features and obtained a 70% coverage of classes.

Figure 1 shows the package owner view of the system. We see from this view that there are packages with one responsible developer, packages with two responsible developers, and packages with multiple developers. (Note that the white classes of the visualization denote an unknown developer due to the initial import).

Our visualization reveals that the *red* developer is solely responsible for the packages *gui.PhoneDisplayPanels.text, gui.PhoneDisplayPanels.menu,dialog*, and *gui.renderer*. The *blue* developer is the unique responsible for the packages *events, resource* and *sms*.

We see 5 packages where the same three developers (red, cyan and blue) own classes. This pattern indicates a collaboration between these developers on a structural boundary.

The *green* developer owns only 8 classes in the system, distributed over three packages. Five of these classes are located in the *gui::phoneComponents* package.

Which developers or groups of developers are responsible for which features? Figure 2 shows an example of a feature perspectives of the system. Here we see the feature-specific classes of the *playRingtone* feature in the context of the package hierarchy. The *blue* developer is the main developer of this feature as he owns 60% of the *feature-specific* classes.

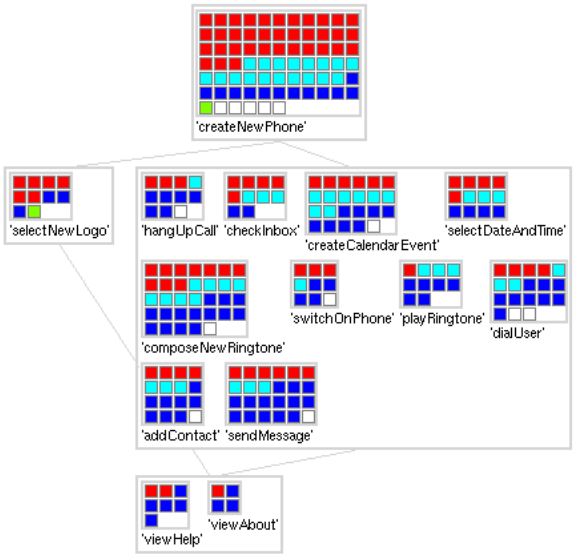

Figure 3. Team Collaboration View from PhoneSimulator-1: the small squares represent classes, the medium rectangles features and the large rectangle teams of developers

In Figure 3 we show a *Team Collaboration View* based on the features we traced. The view establishes the relationships between the features and their responsible developers. Our visualization reveals that the *red* developer is the main responsible for the classes of three features, namely *selectDateAndTime*, *checkInbox* and *selectNewLogo*. The blue developer is the main responsible for the classes of the features *switchOnPhone*, *dialUser* and *hangUpCall*.

Figure 3 also reveals that both the *red* and *blue* developers are active in all of the features. Thus our analysis suggests that from a domain knowledge perspective, the *blue* and *red* developers have a wider knowledge of the features than the *cyan* and *green* developers.

Do developers developer features or functional blocks? Our analysis of PhoneSimulator-1 indicates that the development strategy corresponds more closely with the structural divisions in the system, namely the package boundaries rather than the feature perspective. We checked our findings with the developers of PhoneSimulator-1. They confirmed that initially they adopted a development strategy based on the model-view-controller pattern. The *red* developer was responsible for the *view* classes and the *cyan* and *blue* developers were responsible for the model and controller classes. The *green* developer was responsible for the creation of images used by the application and the classes that manipulated these images. Thus the green developer *touches* only two packages of the system. Once the first iteration of the system was completed, the developers adopted a *user-story* or *feature* development strategy. In other words, the responsibility for developing new features (*e.g., playRingTone*) was typically assigned to one or two developers. They confirmed our finding that the *blue* developer was responsible for developing the *playRingTone* feature (Figure 2).

4.3 PhoneSimulator-2

This system consists of 196 classes and 25 packages and was developed by 4 developers. We traced 10 features and obtained a 68% coverage of classes.

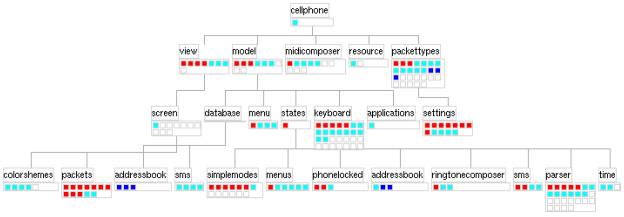

Figure 4. Package Owner View of PhoneSimulator-2

Figure 4 shows the package owner view of the system for PhoneSimulator-2. From this view we detect 5 packages where only the *cyan* developer is active. However, these packages contain only very few classes. Although the team consisted of 4 developers, the visualization reveals that according to our ownership analysis, only three of the developers actually own classes.

Our visualizations reveal that the *cyan* developer and the *red* developer are the key developers of this system. The *blue* developer is exclusively responsible for the package *model::database::addressbook*.

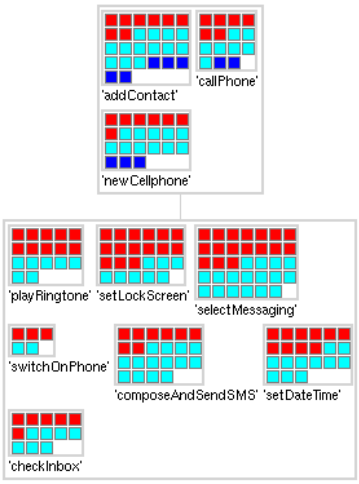

Figure 5. Team Collaboration View of PhoneSimulator-2

Which developers or groups of developers are responsible for which features? Figure 5 shows the team feature view of the system. We see that the *cyan* and *red* developers are responsible for all of the features that we traced. We also see that the blue user shares responsibility for the *addContact* and *newCellphone* features. This feature is related to the *addressbook subsystem* of the application, for which the blue developer is solely responsible.

Do developers develop features or functional blocks? We detect that there are structural divisions of responsibilities (*e.g.,* the addressbook package). However, the developers seem to be working in pairs to develop the individual packages. The main developer (cyan) is clearly the developer with most responsibility from a static and features perspective of the system.

The developers of the Phonesimulator-2 confirmed that our findings corresponded to the division of responsibilities within the project.

5 Case Study 2: ArgoUML Case Study

To test the scalability of our technique we applied it to ArgoUML, an open source UML modeling application implemented in Java. We chose ArgoUML because: (1) we have access to developer knowledge of the documentation of ArgoUML to validate our results, (2) it is an open source and we have access to the cvs repository to obtain the information we need to calculate the ownership measurement of files, and (3) it has been used by other researchers as a reverse engineering case study.

We focus on the core of the application (*i.e.,* we exclude library classes and plugin features). We parsed the source code and obtained a model consisting of 2075 classes. To narrow the scope of our investigation we filtered out the classes in the library *org.tigris* that provide GUI classes and java library classes. This resulted in 1501 classes.

We exercised 11 features by interacting with the user interface and traced each feature individually. We achieved a coverage of 58% of the classes.

Figure 6 shows the package owner view of the system. There are 83 packages in total. 13 packages are owned solely by the *red* developer. And four packages where the *cyan* developer is the sole owner. The remaining 66 packages are owned predominantly by the *red* developer. Thus our analysis reveals a that the system is predominantly developed by one developer. There is structural division of responsibilities between the *red* and *cyan* developers on a package boundary.

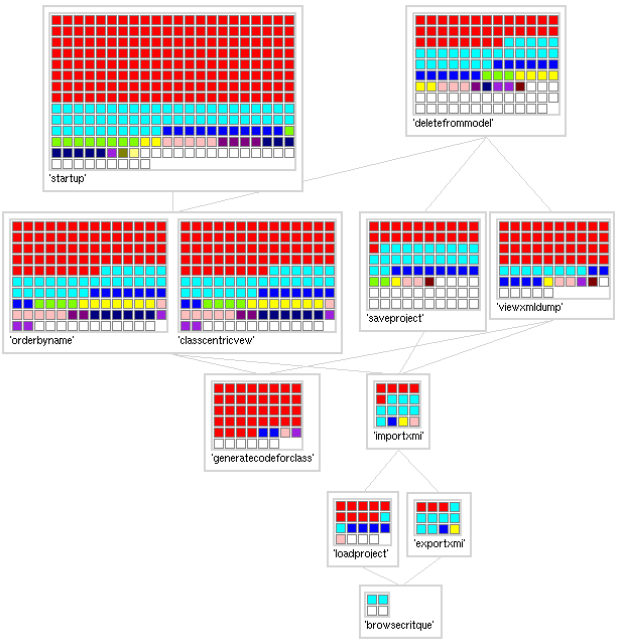

Figure 7. Team Collaboration View of ArgoUML showing relationships between teams of developers.

Which developers or groups of developers are responsible for which features? We extracted 11 teams of collaborating developers, shown in Figure 7. Each team is responsible for one or two features. There is only one feature *browsecritique* where only one developer (*cyan*) is responsible for all the classes of the feature. For all other features

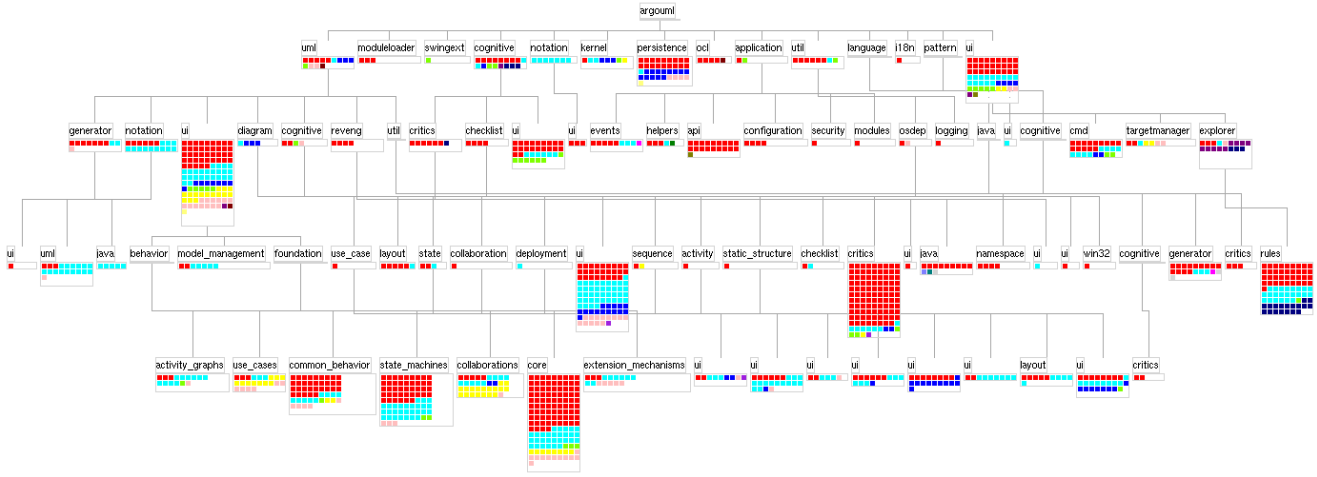

Figure 6. Package Owner View of argoUML

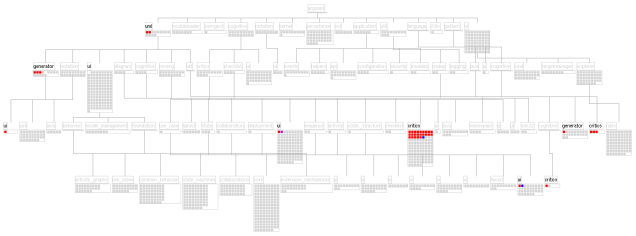

Figure 8. Package Owner View of ArgoUML highlighting *Generate Code for Class* feature

Figure 9. Package Owner View of ArgoUML highlighting the *Startup* Feature

we see that three or more developers are responsible for the classes that participate in the feature.

From Figure 7 we see that the *red* developer is predominantly responsible (*i.e.,* owns 80% of the classes) for the *generateCodeforClass* feature. Figure 8 shows this feature in the context of the package hierarchy of the system. The *feature-specific* classes crosscut 9 of the packages of ArgoUML.

Figure 7 also reveals that the *red* developer is also predominantly responsible (*i.e.,* owns 65% of the classes) for the *startup* feature. Figure 9 shows this feature in the context of the package hierarchy of the system. The *feature-specific* classes crosscut 37 of the packages of ArgoUML. The startup feature is responsible for initializing the application.

Our analysis reveals that the *cyan* developer is also predominantly responsible for the importXMI (*i.e.,* owns 50% of the classes). We obtained a similar result for the feature *exportXMI*. In both cases the *feature-specific* classes crosscut the same 5 of the packages of ArgoUML.

Do developers develop features or functional blocks? In ArgoUML we detect both structural and feature divisions of responsibilities. Of the 10 distinct developers our visual analysis reveals that there are two main developers (*red* and *cyan*), We verified that these two developers correspond with the main developers of ArgoUML. We detect that the *red* developer is responsible for most of the classes in the system and predominantly owns most of the packages and predominately responsible for most (80%) of the features we traced. The *cyan* developer responsibilities correspond to both package and feature boundaries. Our results show that in the case of ArgoUML it is difficult to deduce if the

developer strategy aligns with a structural or a features perspective as the responsibility for the classes of the application is dominated by two main developers. Despite this, our results reveal the existence of both development strategies.

6 Discussion

Limitations of the Ownership Measurement. We chose to use the ownership measurement calculated by analyzing the CVS log data. This solution is therefore coupled with the CVS tool and will not work for other repositories such as Subversion or Clearcase. However, we have encapsulated the repository dependent code, thus minimizing the required effort to adapt our solution to another repository if required.

Our validation revealed a flaw in our ownership measurement. In discussion with the developers of PhoneSumulator-1 project, we discovered that the *red* developer appears to own a large proportion of the classes. This is an incorrect representation as the *red* developer restructured a large proportion of the classes as a result of applying an automatic style checker that flagged long methods. As a result of this editing, our ownership calculation incorrectly assigns him as owner of these classes.

Feature definition and Feature Characterization. We define features as user-triggerable actions. However, not all features of a system satisfy this definition. System internal housekeeping tasks, for example, are not triggered directly by user interaction. Moreover, we assume a one-to-one mapping between feature-traces and features. This is a simplification of reality, as the execution path of a feature varies depending on the combination of user inputs when it is triggered. Exhaustive execution of a feature is costly. We see from our experiments that one path of execution is useful enough to reveal a mapping that directs the software developer to the relevant classes for a feature.

Our feature characterization measurements currently define a threshold value of 50% to distinguish between *feature-specific* and *infrastructural* classes. Varying this value will affect the results. One variation of our approach would be to allow the reverse engineer to define this value, depending on the type of application to be analyzed. We plan to experiment with variable threshold values in the future.

Team definition. For this analysis we extracted the definition of a team from the ownership information of the feature-specific classes of features. There are alternative ways to define a team based on structural collaborations. We plan to investigate the definition of teams based on extracting the developer collaborations in more detail in the future. Our visualization of the package hierarchy of a system reveals that developers tend to develop the system from a package perspective. The feature teams reveal how the developers collaborate when implementing code that is specific to one feature. In our case studies we see that typically one or two developers were responsible for the implementation of a feature.

The Roles of Software Developers. Our technique focuses exclusively on the software development of the software engineer. It excludes activities such as configuration management, creation of resources (*e.g.,* image files), build and release management. These are all relevant activities with a development project. Thus the picture obtained of the developer is incomplete in the general sense. We are interested in obtaining developer information in the context of structural views and feaure views of the system.

System Coverage. For the purpose of feature location, complete coverage is not necessary [19]. Wilde and Scully's *Software Reconnaissance* technique, and other approaches based on this technique [9], do not locate all the code associated with a feature, but provide good starting points for the software maintainer to understand the implementation of a feature [19].

7 Related Work

Researchers in the field of reverse engineering and system comprehension are becoming aware of the importance of analyzing the developer and exploiting new sources of data such as source code repositories to understand software systems [11, 15, 17, 21].

Lethbridge *et al.* define a taxonomy of data collection techniques to obtain information about the roles of software engineers during the development of a project. Their work highlights the growing awareness of this source of information in the field of reverse engineering and system comprehension.

Herbsleb and Mockus used data generated from a change management system to better understand how communication occurs in a globally distributed software development. They used several modeling techniques to understand the relationship between the modification request interval and other variables including the number of people involved, the size of the change, and the distributed nature of the group, working on the change.

Xiaomin Wu *et al.* describe a tool to visualize [21] the change log information to provide an overview of the active places in the system as well as of the author activities. They display measurements like the number of times an author changed a file, or the date of the last commitment.

Chuah and Eick proposed a three visualizations for comparing and correlating different evolution information like

the number of lines added, the errors recorded between versions, number of people working etc. [4].

Zimmerman *et al.* aimed to provide mechanism to warn developers about the correlation of changes between functions. The authors placed their analysis at the level of entities in the meta-model (*e.g.,* methods) [23]. The same authors defined a measurement of coupling based on co-changes [22].

Hassan *et al.* analyzed the types of data that are good predictors of change propagation, and came to the conclusion that historical co-change is a better mechanism than structural dependencies like call-graph [15].

Gall *et al.* [10] aimed to detect logical couplings between part of the system by identifying which parts of the system change together. They used this information to define coupling measurements. The more times modules were changed together, the more tightly coupled they are. This approach is based on files and folders of a system and does not consider the semantical units of a system such as classes and methods.

Anvik *et al.* [2] describe a semi-automatic technique to assign bug reports to developers. They base their analysis on data extracted from the bug repository of a software development project and use a machine learning algorithm to support the assignment of bugs to the appropriate developer. For two of the case studies they analyze the achieved a high precision (between 57% and 64%). One major contribution of their work is that they identify the problem of tracing the correct developer address a given bug report.

Canfora *et al.* [3] also consider the problem of assigning change requests to developers of open source projects. They design an approach to assign change requests to developers based on analyzing the previous assignment history of the change requests.

Our main focus in this paper was to define a reverse engineering approach that exploits developer information of a systems features. The work of Anvik *et al.* and Canfora *et al.* identifies a motivation for our technique of associating developers with features.

8 Conclusions and Future Work

In this paper, our goal was to analyze the roles of developers from both a structural and a features perspective or a system. Based on this analysis, we sought to extract development strategies based on the distribution of owners over the packages of the system and over the features of the system. In particular we addressed the questions:

- *Which groups of developers are responsible for which features?* We exploited our features perspective to extract teams of developers who are responsible for the features. We define a team collaboration based on a partial ordering of teams, and we reveal which developers and teams of developers are responsible for which features. Thus we assume that these developers are familiar with domain aspects (*i.e.,* the user-triggerable functionality of the system).
- *Do developers develop features or do they develop functional blocks?* Our visualizations of the package hierarchy showing the owners of classes shows which developers are responsible for which classes in the system. This view shows the structural groupings of classes as packages. Our analysis of the student projects revealed that the developers typically distribute responsibility on a package boundary. Different developers implement specialized functionalities such as xml handling or database interaction. The boundaries of model-view-controller are also split between different developers. We also discovered, that for some features, a developer strategy that reflects the feature boundaries has also been adopted in the case of new features that were added after the main functionality of the system was implemented.

The main contributions of our approach are:

- We identify a novel way of analyzing the roles of developers with respect to the features of a system.
- We describe a technique to extract and visualize static and dynamic views of the relationships between developers and structural packages and feature views.
- We extract and model the collaborations between developers and teams of developers based on their ownerships of classes of features.

In this paper we have presented an exploratory analysis that considers two complementary perspectives of the roles of developers in a system. Before we can make conclusive statements about our approach, we need to perform more case studies and refine the technique to define metrics to quantify more precisely the relationships between developers and features. We recognize the iterative nature of our approach. We need to experiment with variations of feature definition and threshold definition of feature-specific functionalities to achieve optimal results. We would like to apply our approach in an industrial setting with larger systems and teams.

Acknowledgments. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project “Analyzing, capturing and taming software change” (SNF Project No. 200020-113342, Oct. 2006 - Sept. 2008) and the Cook ANR project “COOK (JC05 42872): Réarchitecturisation des applications industrielles objets”.

References

[1] G. Antoniol and Y.-G. Guéhéneuc. Feature identification: a novel approach and a case study. In *Proceedings IEEE International Conference on Software Maintenance (ICSM 2005)*, pages 357–366, Los Alamitos CA, Sept. 2005. IEEE Computer Society Press.

[2] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proceedings of the 2006 ACM Conference on Software Engineering*, 2006.

[3] G. Canfora and L. Cerulo. Supporting change request assignment in open source development. In *Proceedings of 2006 ACM Symposium on Applied Computing*, pages 1767–1772. ACM, ACM Society Press, 2006.

[4] M. C. Chuah and S. G. Eick. Information rich glyphs for software management data. *IEEE Computer Graphics and Applications*, 18(4):24–29, July 1998.

[5] M. E. Conway. How do committees invent? *Datamation*, 14(4):28–31, Apr. 1968.

[6] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[7] S. Ducasse, T. Gîrba, and A. Kuhn. Distribution map. In *Proceedings International Conference on Software Maintainance (ICSM 2006)*, pages 203–212, Los Alamitos CA, 2006. IEEE Computer Society.

[8] 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, Milano, 2005.

[9] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Computer*, 29(3):210–224, Mar. 2003.

[10] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Proceedings International Conference on Software Maintenance (ICSM '98)*, pages 190–198, Los Alamitos CA, 1998. IEEE Computer Society Press.

[11] T. Gîrba, A. Kuhn, M. Seeberger, and S. Ducasse. How developers drive software evolution. In *Proceedings of International Workshop on Principles of Software Evolution (IWPSE 2005)*, pages 113–122. IEEE Computer Society Press, 2005.

[12] O. Greevy and S. Ducasse. Characterizing the functional roles of classes and methods by analyzing feature traces. In *Proceedings of WOOR 2005 (6th International Workshop on Object-Oriented Reengineering)*, July 2005.

[13] O. Greevy and S. Ducasse. Correlating features and code using a compact two-sided trace analysis approach. In *Proceedings IEEE European Conference on Software Maintenance and Reengineering (CSMR 2005)*, pages 314–323, Los Alamitos CA, 2005. IEEE Computer Society Press.

[14] O. Greevy, S. Ducasse, and T. Gîrba. Analyzing software evolution through feature views. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(6):425–456, 2006.

[15] A. Hassan and R. Holt. Predicting change propagation in software systems. In *Proceedings 20th IEEE International Conference on Software Maintenance (ICSM'04)*, pages 284–293, Los Alamitos CA, Sept. 2004. IEEE Computer Society Press.

[16] Java interactive profiler. http://sourceforge.net/projects/jiprof.

[17] T. C. Lethbridge, S. E. Sim, and J. Singer. Studying software engineers: Data collection techniques for software field studies. *Empirical Software Engineering, Springer Science and Business Media, Inc., The Netherlands*, 10(3):311–341, July 2005.

[18] A. Mehta and G. Heineman. Evolving legacy systems features using regression test cases and components. In *Proceedings ACM International Workshop on Principles of Software Evolution*, pages 190–193, New York NY, 2002. ACM Press.

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

[20] E. Wong, S. Gokhale, and J. Horgan. Quantifying the closeness between program components and features. *Journal of Systems and Software*, 54(2):87–98, 2000.

[21] X. Wu, A. Murray, M.-A. Storey, and R. Lintern. A reverse engineering approach to support software maintenance: Version control knowledge extraction. In *Proceedings of 11th Working Conference on Reverse Engineering (WCRE 2004)*, pages 90–99, Los Alamitos CA, Nov. 2004. IEEE Computer Society Press.

[22] T. Zimmermann, S. Diehl, and A. Zeller. How history justifies system architecture (or not). In *6th International Workshop on Principles of Software Evolution (IWPSE 2003)*, pages 73–83, Los Alamitos CA, 2003. IEEE Computer Society Press.

[23] 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, Los Alamitos CA, 2004. IEEE Computer Society Press.