Analyzing Feature Traces to Incorporate the Semantics of Change in Software Evolution Analysis

Published in ICSM 2005

Orla Greevy
Software Composition Group
University of Bern, Switzerland

Stéphane Ducasse
LISTIC - Université de Savoie, France and
SCG - University of Bern, Switzerland

Tudor Gîrba
Software Composition Group
University of Bern, Switzerland

Abstract

Many of the approaches that analyze software evolution consider a static perspective of a system. Static analysis approaches focus on the evolution of static software entities such as packages, classes and methods. Without knowledge of the roles software entities play in system features, it is difficult to interpret the motivation behind changes and extensions in the code. To tackle this problem, we propose an approach to software evolution 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 define history measurements that summarize the evolution of software entities from a feature perspective. We show how we use our feature perspective of software evolution to interpret modifications and extensions to the code. We apply our approach on two case studies and discuss our findings.

Keywords: reverse engineering, software evolution, feature history, transitions of characterizations, dynamic analysis.

1 Introduction

Most reverse engineering approaches to software evolution analysis focus on static source code entities of a system, such as classes and methods [3, 21, 22]. 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 maintainers to discover what motivated modifications and extensions to the code.

Software evolution is driven mostly by activities such as iterative development, bug reports and changing requirements. Typically change requests and bug reports are expressed in terms of system *features*. Several works have shown that exercising the features of a system is a reliable means of correlating features and code [7, 30]. In a previous work [16], 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. We characterize features in terms of classes that implement their functionality, and we characterize classes based on how they participate in features.

In this paper we describe an approach to software evolution analysis that is based on how software entities participate in features. In other words, we extend our *feature-driven* analysis approach with a time dimension.

Our approach combines dynamic analysis and evolution analysis of multiple versions of a system. We tackle the problem of manipulating the huge amounts of data generated by these techniques as (1) we apply measurements to compact the execution traces and (2) we apply *history measurements* to summarize changes in multiple versions. We compact the data to infer high level information about the evolution of a system from a feature perspective.

We focus on object-oriented systems, and as such, we analyze the relation between features and classes. Our goal is to detect the changes in the relationships between classes and features over time. In particular, we seek answers to the following questions:

- *Are classes becoming obsolete or less active with respect to the features over time?*
- *Can we detect the introduction of new classes in the features over time?*

- *Has the code been refactored?*[1]

We maintain the link between an external feature view of an application and its internal design and implementation details throughout its lifecycle. Our view provides semantic interpretation of modifications and extensions to the code.

We apply our approach to two medium size case studies. The results show that a feature perspective of classes is a useful technique for locating and interpreting changes to a system over time.

We describe simple graph visualizations that summarize the evolution of the relationships between classes and features.

Structure of the Paper. In Section 2 we introduce the terminology we use to characterize software entities from a features perspective. Section 3 describes changes to class characterization and history measurements to measure these changes over time. Section 4 introduces our feature-based evolution approach and the graph views we extract. In Section 5 we report on two case studies conducted using our approach. Subsequently, in Section 6 we discuss our results and outline the constraints and limitations of our approach. In Section 7 we briefly outline some implementation details of our approach. We summarize related work in Section 8. Section 9 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* [7]. 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*.

In this paper, we focus on the relationships between features and classes. Nevertheless, our approach is generally applicable to finer-grained units such as methods, or coarser-grained units such as packages.

Our characterizations of classes express their level of participation in a set of features under analysis. We define the measurements $NOFRC$ to compute the # feature-traces that reference a class and FC to compute four distinct characterizations of a class in terms of how many features reference it and how many features are currently modeled.

- *Not Covered (NC)* is a class that does not participate in any of the features-traces of our current feature model.

$$(NOFC = 0) \rightarrow FC = 0$$

- *Single-Feature (SF)* is a class that participates in only one feature-trace.

$$(NOFC = 1) \rightarrow FC = 1$$

- *Group-Feature (GF)* is a class that participates in less than half of the features of a feature model. In other words, group-feature classes provide functionality to a group of features, but not to all features.

$$(NOFC > 1) \wedge (NOFC < NOF/2) \rightarrow FC = 2$$

- *Infrastructural (I)* is a class that participates in more than half of the features of a feature model.

$$(NOFC >= NOF/2) \rightarrow FC = 3$$

[1]Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior [12]

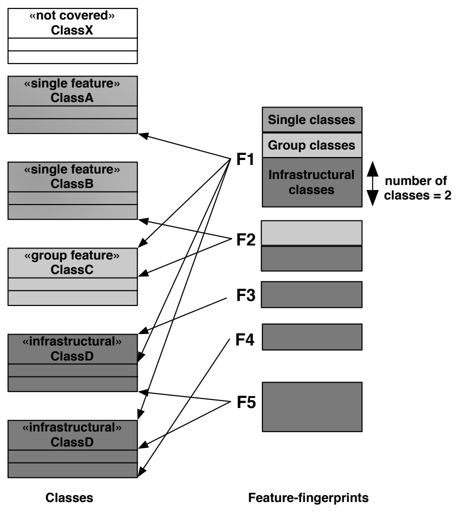

Figure 1. Feature-Fingerprints and Classes Relationships

We compact a feature-trace into *feature-fingerprints* by reducing multiple references to the same class 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. A feature-fingerprint (FP) 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.

3 Evolving Relationships between Features and Classes

Our approach to evolution analysis of a system is to focus on how the relationships between classes and features change over time. We are interested in changes to *characterizations* of classes. In Section 2 we define the *characterizations* of classes with respect to features. Now we extend these class characterizations with a time dimension. We measure changes in characterizations to detect how classes are affected by modifications to the code over time from a feature perspective. We refer to these as *characterization transitions* of classes.

In Figure 3 we show all possible characterization transitions and associate a numerical value with each of the 4 characterizations. We list the transitions in Table 1. We also associate an *activity indicator* to show if a class participates in more $(+)$, less $(-)$ or no (0) features as a result of a transition.

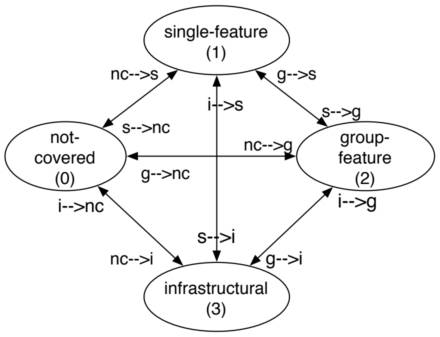

Figure 3. Characterization Transitions of Classes

For example, a *single-feature-to-group-feature* $(s \rightarrow g)$ transition indicates that a class that participated in a single feature entity in a version i has been changed in a subsequent version $i+1$ so that it participates in more than one feature.

Transition (T)	Detail	Activity Indicator
$(s \rightarrow g)$	single-feature-to-group-feature	$+$
$(s \rightarrow i)$	single-feature-to-infrastructural	$+$
$(s \rightarrow nc)$	single-feature-to-notcovered	0
$(g \rightarrow s)$	group-feature-to-single	$-$
$(g \rightarrow i)$	group-feature-to-infrastructural	$+$
$(g \rightarrow nc)$	group-feature-to-notcovered	0
$(i \rightarrow s)$	infrastructural-to-single-feature	$-$
$(i \rightarrow g)$	infrastructural-to-group-feature	$-$
$(i \rightarrow nc)$	infrastructural-to-notcovered	0
$(nc \rightarrow s)$	notcovered-to-single	$+$
$(nc \rightarrow g)$	notcovered-to-group	$+$
$(nc \rightarrow i)$	notcovered-to-infrastructural	$+$

Table 1. Characterization Transitions

We consider the extent of the class characterization transitions as relevant for our evolution analysis. The transition indicates a change in participation of a class in the features from one version to the next. For example, we consider a transition from *notcovered* to *infrastructural* $(nc \rightarrow i)$ to represent a more significant change than a transition from *single-feature* to *group feature* $(s \rightarrow g)$, as *infrastructural* classes affect more the features under analysis.

Activity Indicators 0,$+$ and $-$. The activity indicator of a class (as shown in column 3 of Table 1 represents an increase, a decrease or a non-participation of a class in the features under analysis over time.

3.1 Applying History Measurements to Class Characterizations

In this section we describe how we apply *history measurements* defined in previous works [14, 15] to class characterizations. *History* is defined as a sequence of versions of the same kind of entity (e.g., a class history). Four history measurements summarize the evolution of the relationships between classes and features over time. Our measurements are applied to the version property FC *(the characterization of a class with respect to features)* described in Section 2.

Evolution of Feature Characterization (EFC). We define the history measurement EFC as being the sum of the absolute difference of FC in subsequent versions from version 1 (the first version) to version n (the latest version) of a class history C:

$$(n > 1) \quad EFC_{1..n}(FC, C) = \sum_{i=2}^{n} |FC_i(C) - FC_{i-1}(C)|$$

History Average of a Feature Characterization ($HAFC$). The $HAFC$ history measurement calculates the average value of the FC of a class history C. We

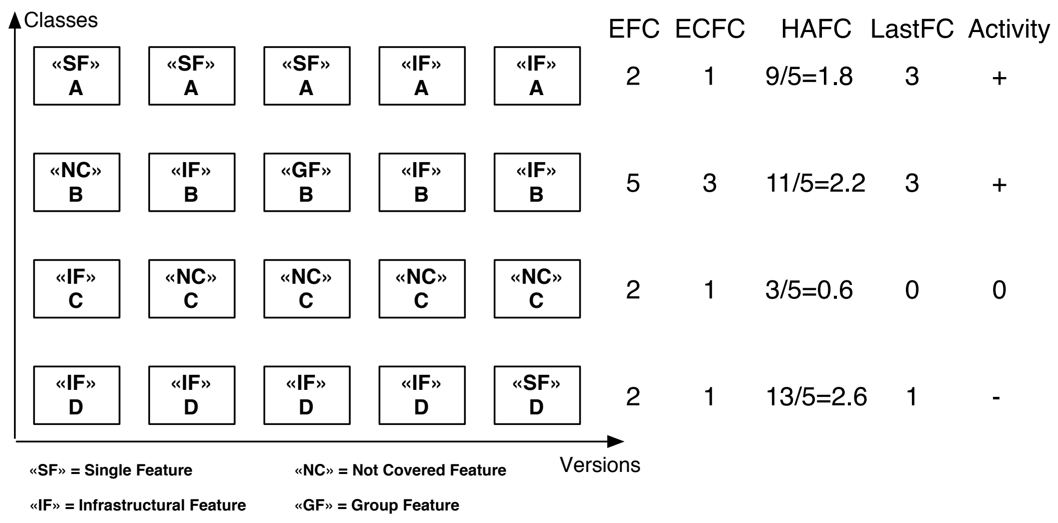

Figure 2. Characterization History Measurements.

use this $HAFC$ measurement to filter out *not-covered* classes to focus only classes that have been active in features at some point in its history.

$$(n > 0) \quad HAFC_{1..n}(FC, C) = \frac{\sum_{i=1}^{n} FC_i(C)}{n}$$

Evolution Count of Feature Characterization (ECFC). We define the $ECFC$ history measurement as sum of the changes of the value of FC for a class history C from one version to the next. A change occurs if $FC_i(C) \neq FC_{i-1}(C)$. In other words, we count the number of characterization transitions of a class over time.

$$EC_i(FC, C) = \begin{cases} 1, & |FC_i(C) - FC_{i-1}(C)| > 0 \\ 0, & FC_i(C) - FC_{i-1}(C) = 0 \end{cases}$$

$$(n > 1) \quad EC_{1..n}(FC, C) = \sum_{i=2}^{n} EC_i(FC, C)$$

Activity Indicator. We compute the activity indicator by comparing the *history average of a characterization* ($HAFC$) of a class with the value of *FC* in the last version of a system under analysis ($LastFC$).

3.2 Applying History Measurements to Changes in Class Characterizations

Figure 2 illustrates how the history measurements we defined for FC characterize the evolution of a class in terms of its changes in feature participation.

- EFC (*evolution of a characterization or weighted transition)* describes the sum of the *weighted transitions i.e.,* the extent of characterization changes that the class undergoes during its history. For example the class A is single-feature in version 1, 2 an 3, then becomes infrastructural in version 4 and 5. Therefore EFC *(A)=2 i.e.,* changing once from single-feature to infrastructural. Note that that even though class C changes from single-feature to not covered, we obtain a value 2 since we calculate the absolute value of the changes.

- $ECFC$ is the number of transitions. Classes A, C and D changed once their characterization from single to infrastructure for A, from infrastructural to not covered for C and from infrastructural to single-feature for D, while class B changed more: from a not covered class to infrastructural to group-feature back to infrastructural characterization.

- $HAFC$ is the average of class characterization according to changes weighted as in Figure 3. If we compare classes A and C which have the same EFC and $ECFC$ but a different $HAFC$, we see that class A changes less than class C.

- $LastFC$ represents the last characterization of a class. Here we see that class A and B are infrastructural classes while C is a not covered class and D a single-feature class.

- Activity is computed by comparing $HAFC$ and $LastFC$. Class A and Class B show $+$ activity, as $LastFC > HAFC$. Class C shows 0 activity as the value of $LastFC$ is 0. Class D shows $-$ activity as $HAFC > LastFC$.

3.3 Interpreting History Measurements.

The interpretation of the measurements depends heavily on the following context of our approach: we do not add new features and the features do not have observable behavior changes between the versions. This interpretation is from a class point of view.

Drop in Feature Participation. We refer to these classes flagged with $-$ *Indicator* as *suspect* classes where functionality has been removed from classes or is no longer being used by one or more features of our analysis. For example in Figure 2, the class D is suspect for further inspection since it changes it characterization from an infrastructural to a single-feature class.

Refactorings. We interpret the characterization transitions of the classes as indications of possible code refactorings, additions or removals of non-functional behavior. We use the term *non functional* to refer to code concerned with infrastructural aspects of the application like authentication, caching and persistence mechanisms. Due to the number of possible explanations for characterization transitions, we do not claim to use then to make conclusive statements about the types of changes in the system based on transitions. Our hypothesis is that by identifying classes that show frequent transitions, increase or decrease in activity with respect to the features we support maintenance activities as we incorporate a semantic interpretation of changes in the classes.

New Participation in Feature. We consider the classes flagged with $+$ *Indicator* as *suspect* classes where non-observable functionality has been added or existing functionality is being reused by more features over time. We apply the weighted transition measurement (EFC) to detect significant changes in classes. We assume that classes that show a single-to-infrastructural transition indicate places in the code where functionality has been added that affects multiple features.

Obsolete Code. Characterization transitions that have a 0 *Indicator* (see Table 1) show that class no longer participates in features. A class may be obsolete or contain candidate obsolete methods. As we do not obtain 100% coverage of the systems classes, we cannot claim conclusively that a class is obsolete, if it nolonger participates in the features of our model. Nevertheless, the reverse engineer can apply our approach to isolate these classes for further investigation.

4 Applying our Evolution Analysis Technique

We outline how we apply our technique to measure the evolution of class characterizations from a feature perspective.

- We apply static analysis and abstract static models of the source code entities (e.g. classes, methods, invocations) for multiple versions of an application.
- For each version, we abstract feature-traces for the same set of features using dynamic analysis. To achieve this we instrument each version of system. We automatically exercise the same set of features for each version. For the purposes of this experiment, we limit our scope to assume that the external observable behavior of each feature remains unchanged for all versions of our analysis.
- We compact the feature-traces into *feature-fingerprints* by applying feature characterization measurements.
- We combine the results of static and dynamic analysis and incorporate the feature-fingerprints as first class entities in our static model of the system. The implementation details of our model of static and dynamic information is described in detail in Section 7. We compute the characterization measurements for the classes of the static model with respect to the features by applying the *a class characterization in terms of feature participation* (FC) measurement. We obtain one model for each version under analysis.
- We apply version and history analysis to the multiple models of a system and compute history measurements (as outlined in Section 3.1) for classes.
- As the number of classes of an application is usually large, we define queries on the classes to filter out classes that have first never participated in the features of our analysis ($HAFC = 0$) and second never undergone any characterization transitions ($ECFC = 0$). In this way, we reduce the volume of information to be analyzed and focus on key classes of interest. Moreover, we apply queries to group classes that yield each of the activity indicators 0,$+$ and$-$.

4.1 Visualizing Class Characterization History

To illustrate and convey the results of our history analysis of class characterization transitions we use simple graph visualizations.

Class Characterization Evolution View. Figure 4 shows the Class Characterization Evolution view we generate for one of our case studies. The purpose of this view is to enable us to compare visually the features in terms of their class characterizations over a series of versions. We use a color-coded area chart to indicate the distribution of feature class characterizations. The versions of the system we analyze are listed on the x-axis and on the y-axis we show the number of participating classes. We use four colors to represent the possible characterizations of classes. The yellow (light gray) section represents the classes that are *not covered* in the features of our feature model. The height represents the number of classes. In this example, we see the changes in proportions of the characterizations in the features over time and in which versions these changes occurred.

5 Validation

In this section we present the results of applying our approach to two concrete case studies. For our experiments we chose two systems developed by our group: SmallWiki [27] and Moose [6]. *SmallWiki* [27] is a collaborative content management system used to create, edit and manage hypertext pages on the web. It is implemented predominately by two developers from our group. The application is used widely in the Smalltalk community.

Moose is an environment for reengineering object-oriented systems implemented in Smalltalk [6]. It provides an import/export framework responsible for importing source code from a system and represent the system in a model.

We chose the case studies for two main reasons. Firstly, our approach is a heuristic approach and we require developer knowledge to validate our results. As the case studies are developed in our group we had access to developer information. Secondly, the systems should be of a reasonable size ($>$ 500 classes). Table 2 gives an overview of the case studies.

Application	# Versions	# Class Histories	# Features
SmallWiki	11	522	6
Moose	12	708	5

Table 2. Case Studies Details

5.1 Case Study: SmallWiki

As it is a web-based application, user interaction with the features of SmallWiki is achieved by selecting the hyperlink and form options on its pages. To identify features, we associate features with the links and entry forms of its pages. We assume that each link on a page or button on an entry form triggers a distinct feature of the application.

Version	Date	Summary of Maintenance Comment
SW-13.00	08.10.2003	bugs fixed
SW-13.05.1	16.11.2003	-
SW-09.24	26.11.2003	-
SW-09.28	17.12.2003	-
SW-09.30	02.01.2004	refactor, new functionality
SW-09.36	06.01.2004	new feature, bug fix
SW-09.39	31.01.2004	refactor, new functionlity
SW-09.48	22.03.2004	new functionality
SW-13.11	06.04.2004	new functionality
SW-19.51.201	17.09.2004	new features

Table 3. SmallWiki Versions

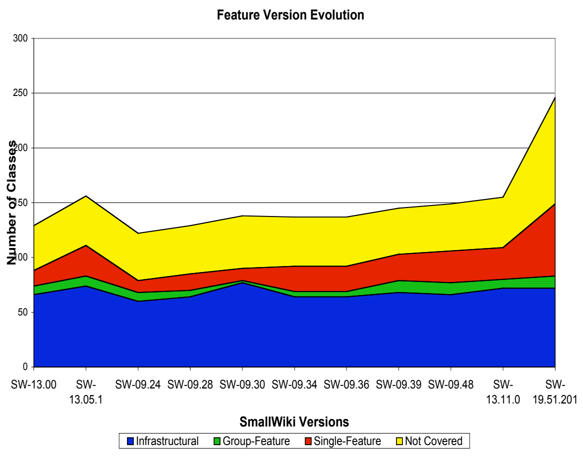

Figure 4. Class Characterization Evolution View calculated on the Smallwiki case study.

We select the same 6 features and 11 different versions of SmallWiki from the source code repository. The versions span an 11 month time period. Much of the changes in the code are as a result of iterative development and refactorings and restructurings in the code. Table 3 lists the versions we chose. In the third column, we note what type of maintenance activity was reported by the developer at the time the version was checked in to the repository for versions where it was available.

For each feature we implemented scripts to simulate the user interactions. We trigger these features and capture a feature-trace in a controlled environment. SmallWiki requires a user to login to the system before features can be triggered. The captured traces initially contain the login trace calls. Therefore we filter out login trace information from our features, so that traces are not composed of other traces.

Interpreting the Feature Characterization Evolution View. Figure 4 shows the feature version evolution view of our feature model for SmallWiki. This view differentiates the classes of the features by characterizations. Variation points in class transitions represent useful starting points for investigating the reasons for changes in the context of features. We see that the number of single-feature classes also increases, in particular in the last two versions analyzed. This result corresponds with the maintenance comment from Table 3 for the last two versions analyzed that states that new functionality has been added. From the second to the third version, the number of classes decreased. We verify with the developers that at this point, the system was indeed refactored.

Applying Class Characterization History Measurements. Our history model of SmallWiki contains 522 class histories [2]. We filter out the classes that were never covered by the six features of our analysis to obtain 166 classes ($HAFC > 0$). We apply a filter to obtain all classes whose characterizations have changed during the history of a class ($EFC > 0$) and obtain 63 classes.

We apply the indicator measurements $+$indicator, $-$ and 0 and we obtain 40 classes with a $+$indicator, no classes with a $-$indicator and 6 classes with a 0 indicator. This indicates that 67% of the classes that participate in features are more active and 9% of the classes detected are candidate obsolete classes or contain candidate obsolete methods.

To focus on the transitions that indicate introduction of infrastructural (I) functionality at some point in the evolution of the system, we apply a filter ($LastFC = 3$). We identify 5 classes where new or existing functionality is reused by more than half of the features of our analysis.

Indicator	Class	Validation by Developer
+	AdminAction	New I
+	ErrorUnauthorized_class	New I
+	ErrorUnauthorized	New I
+	ErrorAction	New I
+	FifoCache	New I
+	HistoryAction	New functionality
+	SearchAction	New functionality
+	EditAction	New functionality
0	VisitorCollectable	obsolete
0	VisitorRendererHtml	Refactored, class split
0	VisitorRendererHtml_class	Refactored, class split
0	Folder_class	Removed functionality

Table 4. SmallWiki Classes with Changing Functional Roles

[2]As a class history is a sequence of versions of a class entity, this gives an indication of the size of the application in terms of number of classes

5.2 Case Study: Moose

For this experiment, we selected features of the import/expo framework and model navigation features from 12 versions of Moose (696 class histories) spanning a four month time period of refactoring, bug fixing and addition of functionality. We summarize the versions and the comments entered by the developers in table Table 5.

Version	Date	Summary of Maintenance Comment
3.0.6	30.12.2004	bugs fixed
3.0.7	15.02.2005	republished
3.0.9	19.02.2005	republished
3.0.12	01.03.2005	fixed bug
3.0.13	01.03.2005	fixed small bug
3.0.14	01.03.2005	fixed small bug
3.0.15	01.03.2005	moved functionality Operators to sourceImporters
3.0.16	01.03.2005	removed functionality cfdetectionstrategy
3.0.17	02.03.2005	fixed bugs. Cleaned UI code
3.0.18.5	09.03.2005	removed functionality
3.0.21	13.03.2005	packaging fixed
3.0.22.3	16.03.2003	fixed bugs

Table 5. Moose Versions

Applying Activity Indicator History Measurements. We apply the activity indicator measurements to the classes to detect which classes became functionally more active, less active or inactive in in the feature-traces over time. We summarize the results in Table 6

Indicator	Class	Validation by Developer
+	DelegatorPropertyOperator	New functionality
+	CFCompositionOperator	New functionality
+	CFBlockOperator	New functionality
+	CFAbsoluteProperties	New functionality
+	CFExpression_class	New functionality
−	AbstractEntity	Removed functionality
0	MSEModelAttributeDescriptor	Removed functionality
0	FileIOFacade	Removed functionality
0	MSEAbstractSchemaSaver	Removed functionality
0	MSEModelAttributeDescriptor_class	Obsolete
0	CDIFSaver	Removed functionality
0	AbstractTool	Removed functionality

Table 6. Moose Classes with Changing Functional Roles

We take a closer look at the classes that are detected as having increased activity over time by analyzing the individual classes and the value of the characterizations of these classes w.r.t. the features for each version.

We detect the same pattern of transitions for the classes *CFCompositionOperator, CFBlockOperator, CFAbsoluteProperties and CFExpression*. We graph this in Figure 5:

We discover that these classes are part of the same hierarchy and that their characterization transitions occurred in

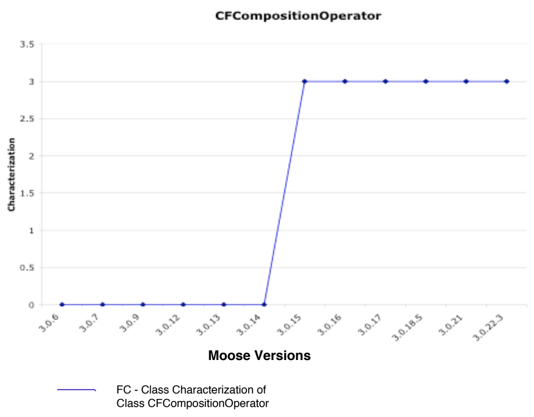

Figure 5. Detail of Characterization Transitions of Moose Class CFCompositionOperator

the Moose version 3.0.15. We verify the finding with the developers and they confirm that the hierarchy became more important to the system in that version.

Our results of our case studies show that our heuristic approach is useful to (1) locate classes where new functionality has been introduced, (2) refactoring and to (3) locate candidate obsolete code. We use developer knowledge to validate the results.

6 Discussion

The large volume of history information and complexity of dynamic information makes it hard to infer higher level of information about the evolution of a system. Our approach reduces the complexity of the information to reveal key semantic information about changes to the system based on measuring how the relationships between classes and features evolve over time.

We limit the scope of our investigation to focus on a constant set of features. Our goal is to apply feature-based evolution analysis to investigate the effects of maintenance on a specific set of features. Our feature perspective enables us to view semantic groupings of the classes. The characterization of these classes provides us with feature knowledge to reason about the design intent of the class. Using feature-based evolution analysis we determine the stability of features of a system by monitoring feature characterizations and the characterizations of the classes over a series of versions.

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.

Feature-Trace mapping. 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.

Coverage. 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.

Obtaining feature traces. One of the difficulties of obtaining feature traces of a system is that in a series of versions of a system, in some of the versions some or all of the functionality may be broken. This makes it difficult to trace the functionality.

Scalability of the approach. Method instrumentation effects the performance of the feature. For some of the features we traced in our Moose case study, the execution time of the instrumented code made experimentation difficult. To tackle this problem, we applied selective instrumentation for the Moose case study. We select which packages to instrument. Selective instrumentation of the packages requires prior knowledge of the application and the relationship between packages and features. The resulting traces and the values of our measurements are influenced by selective instrumentation.

Limitations of the approach. One limitation of the approach is that it cannot detect new functionality that is added to the system in a generic way, such that no new methods are invoked. Multiple calls to the same method of a class are compacted to one occurrence in a *feature-fingerprint*. This limitation was identified during the validation of Moose case study results.

Language independence. Our technique is language independent as we work with a model of the system abstracted by static and dynamic analysis.

7 Implementation - TraceScraper, Moose and Van

TraceScraper is our feature analysis tool. It is based on the Moose [6] reengineering platform. Using method wrappers, TraceScraper runs feature exercising scripts and captures individual traces of the executions. The traces are modeled as FAMIX [4] entites in Moose. Figure 6 shows the relationship between trace entities and the FAMIX entities *Class* and *Method*. The class and method referenced in the trace event is related to the static class and method entities. TraceScraper computes feature-fingerprints from the trace entities based on the relationships to *class* and *method* entities.

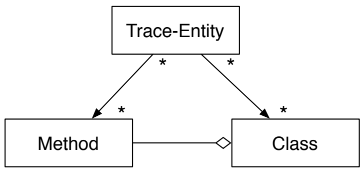

Figure 6. TraceMetaModel.

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.

Van is an implementation of the *Hismo* history meta-model [5] and provides features for analyzing versions of software systems. Our feature history entities and history measurements are implemented as extensions to *Van*. The FAMIX and Hismo metamodels are both language independent.

8 Related Work

Many researchers have identified the potential of feature-centric approaches in software engineering and in particular as a basis for reverse-engineering [7, 8, 18, 24, 25, 28, 29, 30]. 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, 17, 31], user-driven approaches [19] and reverse engineering approaches that consider the evolution of a system [2, 15, 23, 26, 32] represent the groundwork on which we base our research.

Hsi *et al.* [18] described an approach to studying the evolution of features by deriving three views of an application, a morphological, a functional and an object view, based on the domain knowledge of an application. Their models are derived from the user interface of an application. They compare models of an application as they evolve. The purpose of their approach is to depict the feature architecture of an application independently of the underlying software. They highlight the importance of studying the evolution of an feature perspective of a system.

Gall *et al.* [13] 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 semantic units of a system such as classes and methods.

Fischer *et al.* [9, 10] modeled bug reports in relation to changes in a system. The purpose is to provide a link between bug reports and parts of the system.

Our approach complements these approaches. In contrast to the above approaches [9, 11, 18], our main focus is applying feature-driven analysis to object-oriented applications. We use execution traces to establish the link between features and classes. Our characterizations add semantic information to the classes and use this semantic information to reason about the evolution of a system in terms its features.

9 Conclusions and Future Work

Reverse engineering approaches tend to focus on the implementation details and static structure of a system. By doing so they overlook key knowledge about the system which establishes the semantic purpose of the individual software entities.

In this paper, our goal was to analyze the way features of a system evolve and to reason about changes in the code from a feature perspective. We extract feature-models of a system over a series of versions and applied history measurements to determine *what* has happened to the features over time.

In particular we seek answer to three questions:

- *Are classes becoming obsolete or less active with respect to the features over time?*
- *Can we detect the introduction of new classes in the features over time?*
- *Has the code been refactored?*

We applied our approach to two case studies and showed how a feature perspective of a systems evolution is useful for interpreting changes in the code. We intend to perform empirical studies to assess our characterization thresholds for distinguishing between infrastructural and group features. As we had access to developer knowledge for these case studies, we validated our approach by verifying our findings against the developer information.

Our approach reduces a large volume of trace and version data so that we are able to reason about the information and infer high level information about the evolution of a system.

In the future, we would like extend our definition of a feature to consider variations in the external behaviors of the system. We would also like to investigate more correlations between transitions and modifications in the code.

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). We also want to thank Gabriela Arévalo and Andrew Black who reviewed earlier drafts of this paper.

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, and J.-M. Favre. Modeling software evolution by treating history as a first class entity. In *Workshop on Software Evolution Through Transformation (SETra 2004)*, pages 71–82, 2004.

[6] 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.

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

[8] 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.

[9] 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.

[10] M. Fischer, M. Pinzger, and H. Gall. Analyzing and relating bug report data for feature tracking. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE 2003)*, pages 90–99, Nov. 2003.

[11] 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.

[12] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[13] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Proceedings of the International Conference on Software Maintenance 1998 (ICSM '98)*, pages 190–198, 1998.

[14] T. Gîrba, S. Ducasse, and M. Lanza. Yesterday's Weather: Guiding Early Reverse Engineering Efforts by Summarizing the Evolution of Changes. In *Proceedings of ICSM '04 (International Conference on Software Maintenance)*, pages 40–49. IEEE Computer Society Press, 2004.

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

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.

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

[22] 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.

[23] 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.

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

[25] 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.

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

[27] L. Renggli. Smallwiki: Collaborative content management. Informatikprojekt, University of Bern, 2003.

[28] 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.

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

[30] 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.

[31] 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.

[32] 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.