Research

Analyzing Software Evolution through Feature Views

Orla Greevy, Stéphane Ducasse and Tudor Gîrba

Software Composition Group, Institute for Applied Mathematics and Computer Science, University of Berne, Neubrückstrasse 10, CH-3012 Berne, Switzerland
LISTIC, University of Savoie, France

SUMMARY

Features encapsulate the domain knowledge of a software system and thus are valuable sources of information for a reverse engineer. When analyzing the evolution of a system, we need to know how and which features were modified to recover both the change intention and extent, namely which source artifacts are affected. Typically, the implementation of a feature crosscuts a number of source artifacts. To obtain a mapping between features and the source artifacts, we exercise the features and capture their execution traces. However this results in large traces that are difficult to interpret. To tackle this issue we compact the traces into simple sets of source artifacts that participate in a feature's runtime behavior. We refer to these compacted traces as *feature views*. Within a feature view, we partition the source artifacts into disjoint sets of characterized software entities. The characterization defines the level of participation of a source entity in the features. We then analyze the features over several versions of a system and we plot their evolution to reveal *how* and *which* features were affected by code changes. We show the usefulness of our approach by applying it to a case study where we address the problem of merging parallel development tracks of the same system.

1. Introduction

Software evolution is driven predominantly by activities such as iterative development, bug reports and changing requirements. Typically, change requests and bug reports are expressed in terms of system *features*. Previous works on feature identification define a feature to be a unit of observable behavior of a system [1, 2, 3, 4]. As such, a feature represents a unit of domain knowledge, as it typically corresponds to a realized functional requirement of a system. Many researchers have highlighted the potential of exploiting features when reverse engineering a system [1, 5, 6, 7]. Popular object-oriented programming languages such as Java or C# do not provide a language construct that encapsulates the notion of a feature. The structuring unit of object-oriented languages is the class. Typically, features do not map directly on the classes of the system, but extend across several classes [6]. At the same time, a class may participate

in several features. In other words, there is a *n-to-n* relationship between classes and features [8].

Reverse engineering approaches based on static analysis focus mainly on the structural entities and implementation details. Thus, key semantic information about the roles of source artifacts in the features of a system is overlooked. Furthermore, source code analysis of object-oriented systems is difficult due to language features such as inheritance, dynamic binding and polymorphism. The behavior of the features and the parts of the system participating in their functionality can only be completely determined at runtime [9]. Recently researchers are advocating a combined approach to reverse engineering based on both static and dynamic analysis to achieve optimal results [10, 11, 12, 13]. By exploiting feature knowledge of a system, the reverse engineer obtains higher-level abstractions than those obtained by examining the static source code artifacts.

The development and maintenance phases of a software system typically involve several developers working in parallel on a common code base. This may often lead to branches in the code base representing parallel development tracks. For example, enhancements and additional features are made in one branch in preparation for the next release of a system, whereas bug fixes are made in a branch that corresponds to the release of the system in production. Inevitably branches need to be merged to reestablish a coherent code base. Merging branches is a nontrivial task as changes to one feature may conflict or break other features. Developers are faced with the task of understanding what motivated changes in the code and how the changes affect the system as a whole. Furthermore, without an explicit mapping between features and source artifacts, introducing new changes may result in undesired side effects such as increased complexity of the system due to unnecessary code duplication, or the introduction of bugs.

Much of the research effort in feature-centric approaches to date has focussed on feature identification, a well-known technique to identify subsets of a program source code activated when exercising a functionality [7]. The main contribution of this article is that we build on the work of previous established feature identification approaches [5, 1] by analyzing the mapping between features and source artifacts over several versions of a system. Our goal is to reason about the motivations behind changes in the code. A crucial element of our experimentation is that for each version of a system, we exercise the *same* set of features. We extract execution traces and compact them to *feature views* (*i.e.,* simple sets of source artifcacts referenced in the trace). Moreover, we choose a set of features that appear to behave in the same way for each version that we analyze.

In a previous work, we analyzed how the functional roles of source artifacts (*e.g.,* classes) changed over time [14]. We computed a *feature characterization* measurement for classes, based on the level of participation of a class in all the features we traced. Then we measured and interpreted changes of the feature characterizations of classes over time. In this work we extend this approach. The main difference is that in the present paper we treat our representation of a feature, namely the *feature view*, as the primary unit of analysis.

Our goal is to show how a feature-centric analysis of a software system supports evolution and software maintenance activities. In particular, we seek to answer to the following questions:

1. *Which features are affected by changes in the code?* By identifying which features have changed and how they are affected by changes gives an insight into *change intention*.

We characterize changes to determine their extent (*i.e.,* if a change affects one or more features). Thus, the extent of a change helps us to decide which tests (*i.e.,* unit tests, integration tests, acceptance tests) need to be performed after the change has been made. Moreover, we believe that a good understanding of intent and extent of changes supports the developer to tackle complex maintenance tasks such as merging two distinct development branches of a system.

2. *Are features becoming more complex over time?* We define complexity of a feature to be a function of the number of software artifacts (*e.g.,* classes) participating in its runtime behavior. An increase in the number of classes may be an indication that new functionality has been added. As our features appear to behave the same way from a user's perspective in each of the versions we analyze, changes imply the addition or removal of non observable behavior to a feature. Complexity of the features adversely affects the maintainability and comprehensibility of the system [15]. On the other hand, an increase or decrease in feature complexity may indicate that the developers have refactored the code to improve its design [16].
3. *Do similar patterns of change indicate relationships between features?* Similar patterns of increases or decreases in the number of source entities shared between features indicates that the functionality or purpose of certain features are related. Thus, by identifying patterns of change, we make the relationships between features explicit. This is important for reverse engineering as it supports maintenance activities such as regression testing.

To validate our claim that feature views provide semantic interpretation of the changes and support maintenance and software evolution, we apply our feature-centric analysis to four versions of a medium size software system. We show how we detect and interpret changes in the context of features. We perform experiments with two distinct development branches of the system consisting of three and two versions respectively. We address the problems of merging the changes from two development branches. We cross check our findings with the developers implementation knowledge.

The contributions of this article are:

- We describe a novel approach to analyze the evolution of a system in terms of features.
- We characterize changes in a way that reflects how the functional roles of software artifacts change.
- We introduce a simple visualization of the *feature views* as a grouping of participating software entities (*e.g.,* classes) for one version of a system. The goal of our visualizations is to support reasoning about the evolution of a system from a features perspective. Our visualizations are interactive, as they allow the software developer to query the visualization to discover the names of the classes that participate in a feature view. To represent changes in features over time, we describe three variations of the feature view visualizations: (1) the *feature history view* shows a summary view of changes in a feature over time, (2) the *feature additions view* shows feature views that show only the additional source entities that have been added to a feature (3) the *features intersection view* shows the source artifacts that have been added in both development tracks of the system (*i.e.,* the intersection of additions). To represent and quantify changes between the versions of feature views we use a *feature evolution chart*. This consists of four

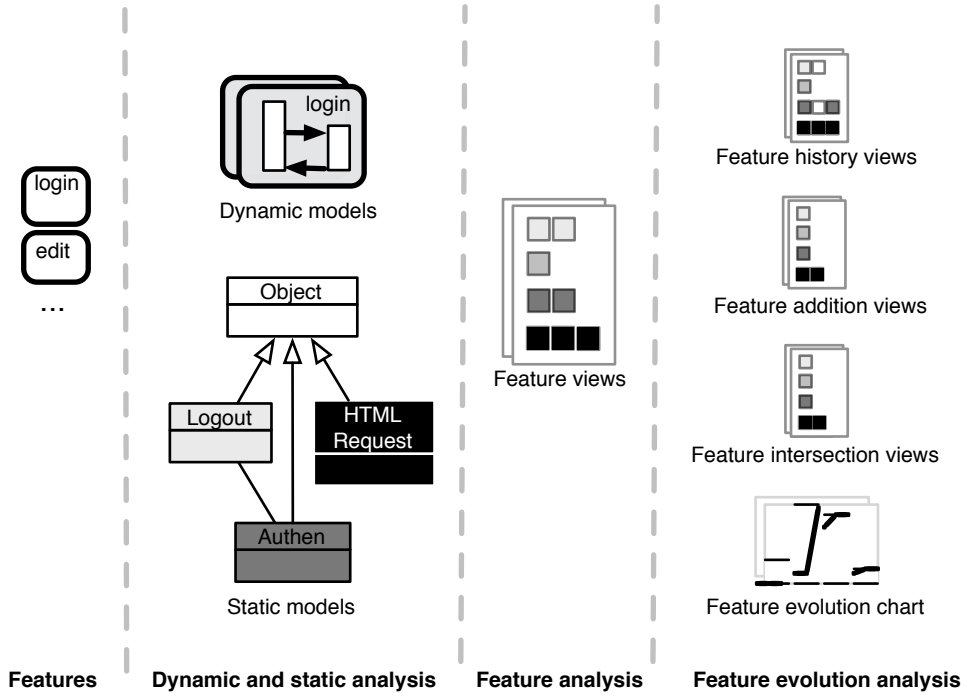

Figure 1. The Elements of our Approach To Evolution Analysis through Feature Views.

simple line graphs that show how four distinct properties of a feature view changes over a series of versions. Figure 1 shows an overview of our approach and how we derive the visualizations.

In the next Section we introduce the underlying terminology of our feature analysis approach. In Section 3 we outline our approach to evolution analysis and describe how we measure, characterize and visually represent changes. In Section 4 we describe in detail the results of our experimentation with a medium size application. We discuss and evaluate the results of our analysis in Section 5. In Section 6 we review related work. In particular, we review the current state of the art of feature-centric reverse engineering approaches such as feature location techniques, feature-based analysis techniques and software evolution analysis approaches. In Appendix A we provide details of the feature and history measurements, and a glossary of the feature related terms used throughout the article.

2. Feature Views

In this section we introduce our feature terminology and describe the underlying concepts of our feature analysis approach.

To map the features to source artifacts, we combine both static and dynamic analysis techniques [17, 14]. We analyze the source code to obtain a model of a software system in terms of static structural entities (*e.g.,* packages, classes and methods). We then extract *feature traces* by exercising a set of features on an instrumented system. A feature trace consists of runtime events represented as a tree of method invocations [18]. We establish relationships between feature traces and the static structural entities found in the traces. These relationships are the main focus of our analysis.

Interpretation of execution traces is difficult due to their sheer size [19, 20], thus filtering or compressing the data is a crucial step in the construction of high level views. As our focus is on establishing relationships between features and code, we reduce the volume of data by compacting the feature traces into sets of software entities (*e.g.,* sets of classes). We refer to the compacted sets as *feature views* [17]. They are simplified representations of a feature trace that do not preserve the sequence of execution or time information, as this is not relevant for our particular research focus. The advantage is that we reduce the volume of dynamic data while at the same time we preserve the information needed to establish the relationships between features and software entities. For the purpose of this explanation, we limit our discussion to classes. Our technique is equally applicable to other types of entities such as packages or methods. In a previous work, we applied our feature analysis technique to methods [17].

As with the feature location approaches of Wong *et al.* [6] and Eisenbarth *et al.* [1], we characterize software entities based on their level of participation in the features as either *general* or *specific*. The level of participation, or *characterization* of a software entity, is highly dependent on the feature definition and the choice of features.

Our approach defines a more fine-grained characterization of software entities. We distinguish between five mutually exclusive levels of participation of a software class with respect to the features under analysis. We define a simple measurement to calculate the usage level, or *feature characterization* of a class based on the number of features we analyze and the number of these features they participate in.

Let NOF be the number of features in our model and $NOFC$ the number of feature-traces referencing a class.

Not Covered (NC) is a class that does not participate to any of the features under analysis.

$$(NOFC = 0)$$

Single-Feature (SF) is a class that participates in only one feature of our analysis

$$(NOFC = 1)$$

Low Group Feature (LGF) is a class that participates in more than one but less than half of the features under analysis. In other words, a low group feature class provides common functionality to a subset of features.

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

High Group Feature (HGF) is a class that participates in half or more of the features of a feature model. In other words, high group feature classes provide functionality most but not all of the features under analysis.

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

Infrastructural (IF) is a class that participates in all of the features under analysis.

$$(NOFC = NOF)$$

The characterizations of classes may vary depending on which features chose to include in our analysis. We chose a threshold of 50% of the features under analysis. This enables us to distinguish between classes that appear to be common only to a small group of features and those that are common to most of the features. Our experimentation with a number of case studies have shown that by distinguishing between *low group* and *high group* classes, we obtain an accurate interpretation of the functional roles of the classes. *Single-feature* and *Low group* classes represent classes that provide functionality that is specific to one feature or a group of related features. The *high group* and infrastructural classes reveal classes that implement common or infrastructural functionality of the system as they participate in most, or all of the features.

Figure 2 shows the relationships between features and classes. On the left hand side we show classes and on the right hand side we show simple visual representations of feature views as groupings of classes. The visualization is composed of large rectangles thar represent features. Each feature contains four subgroups of characterized classes represented as small squares colored according to their characterization.

To quantify the feature views, we compute the cardinalities of the individual sets of characterized classes of a feature view as SF_c (number of single feature classes in a feature view), LGF_c (number of low group feature classes in a feature view), HGF_c (number of high group feature classes in a feature view) and IF_c (number of infrastructural feature classes in a feature view), and the cardinality of the set of all classes referenced in a feature view (CF). These cardinalities represent properties of a feature view.

3. Feature Views Evolution

In the previous Section we have described how we extract feature views. We now extend our focus to consider how the mapping between features and classes change over time. To achieve this we extract feature views for each version of the system and plot changes in their properties. The goal of our evolution analysis is twofold: (1) we want to detect which features are affected by modifications and (2) we want to interpret these modifications in the context of our feature views.

Approaches to analyzing system evolution can be characterized as *version-centered* or *history-centered* [21]. Version-centered approaches compare versions of a system with the aim of revealing *when* (*i.e.,* in which version) a particular change occurred. History-centered

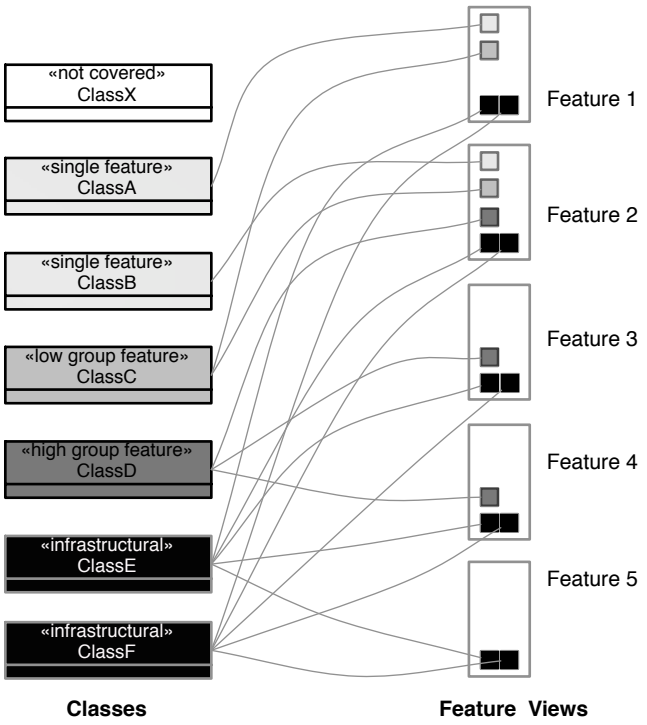

Figure 2. Example of the relationships between classes and features. A feature is represented by a feature view.

approaches on the other hand, are concerned with revealing *what* the changes were and *where* these occurred, by summarizing the evolution according to a particular point of view. For example, a graphic plotting the values of a property in time (*e.g.,* the line graph or our evolution chart as show in Figure 3) is a version-centered approach; on the other hand, a measure of how a property evolved over time is a history-centered approach.

Our evolution analysis combines both history-centered and version-centered evolution analysis approaches. We apply history-centered analysis to gain an overall impression of which features are affected by change and how. We summarize changes to focus on *where* (*i.e.,* in which feature views) changes occurred. Subsequently, we apply a version-centered analysis to obtain a more detailed view of the actual changes and *when* they occurred.

3.1. History Measurements for Feature View Properties

History-centered approaches to evolution analysis define history as an ordered set of versions of the same kind of entity (*e.g.,* a class history contains all the versions of the class) [21]. As we treat *feature views* as first class entities of our analysis, our primary focus is *feature view histories* (*i.e.,* it contains all the versions of a feature view and thus summarizes its evolution).

We measure changes in feature views over time by applying two history measurements (further details are given in Section 7) defined by the *Hismo* meta-model [22] to the properties of our feature views.

Number of Changes of P - This measurement counts in how many versions the property P has been changed with respect to the previous version. We apply this measurement to isolate which features have changed.

Additions of P - This measurement sums additions of a property P. We apply this measurement to detect an increase in a feature view property over time. We interpret additions in features to mean one of the following: (1) increased complexity: an increase in the number of classes participating in a feature may indicate the appearance of additional non-observable functionality in a feature, (2) refactorings or design improvements: these activities often lead to an increase in the number of classes to implement a functionality.

The *Number of Changes* gives us an overview of the amount of changes that occurred over a series of versions. The *Additions* history measurement gives an indication about how the features are growing. According to Lehman's second law of evolution, the increase in size of the code is a typical characteristic of an evolving system and effort is required to reduce complexity to ensure the system is still maintainable [15]. Thus increasingly complex features highlight places in the code where refactoring may be required.

3.2. Visualizing When Features Change

Figure 3 shows how a feature, named *editPage* in our case study, is changing over a series of three versions. For each version, we show its corresponding *feature view*. The views group classes by characterization and the classes are shown in different colors (grayscale). We represent the history of feature views as changes in the 4 sets or characterized classes. Thus, the chart shown below the feature views is a group of 4 evolution charts, each representing the evolution of a different property of the feature view (*i.e.,* how the cardinality of a class characterization set changes over time). For each evolution chart, a horizontal delimiter indicates the maximum value of a property when all the analyzed features of our case study experiment are considered (*e.g.,* max $LGF_c = 36$ classes). The values are indicated on the sides of the chart. The actual values of the properties (*i.e.,* the number of classes of each characterization) for each version are represented as points on the line graph. We use evolution charts to visually represent *when* (*i.e.,* in which version) a change in a property occurred.

In the case of the *editPage* feature, we detect that for LGF_c (number of low group feature classes in a feature view) the value increased from 10 to 36 classes.

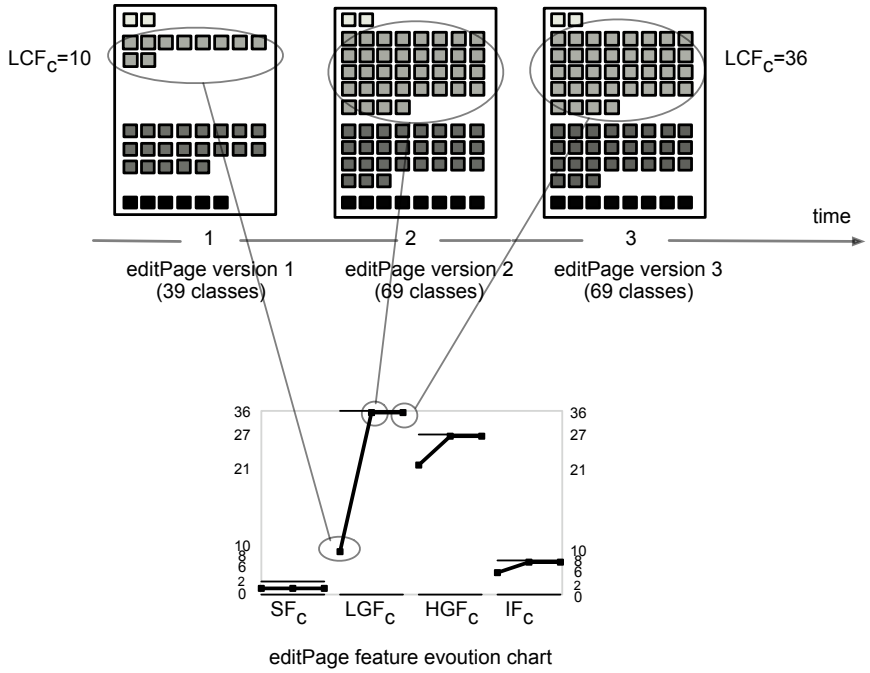

Figure 3. Version Analysis of the *editPage* Feature View (Branch development track).

A class characterization may change and disappear from one characterization set (*e.g.,* the single feature set) of a feature view. At the same time, the class may reappear in another set (*e.g.,* the infrastructural set) of the same feature view. This means that the class has a new characterization value and the evolution chart registers it as a removal and an addition in the two sets respectively. This represents a change in the system. Thus, it offers a good starting point for a developer for further investigation.

3.3. Visualizing How Features Change

We represent quantitative changes to the properties of feature views (*i.e.,* SF_c, LGF_c,HGF_c,IF_c) over a series of versions using evolution charts. Each chart consists of four simple line graphs, each plotting how the value of property changes over a series of versions. This representation is useful to provide the reverse engineer with a quantitative view of changes, but does not provide information about which classes have been added or removed from the

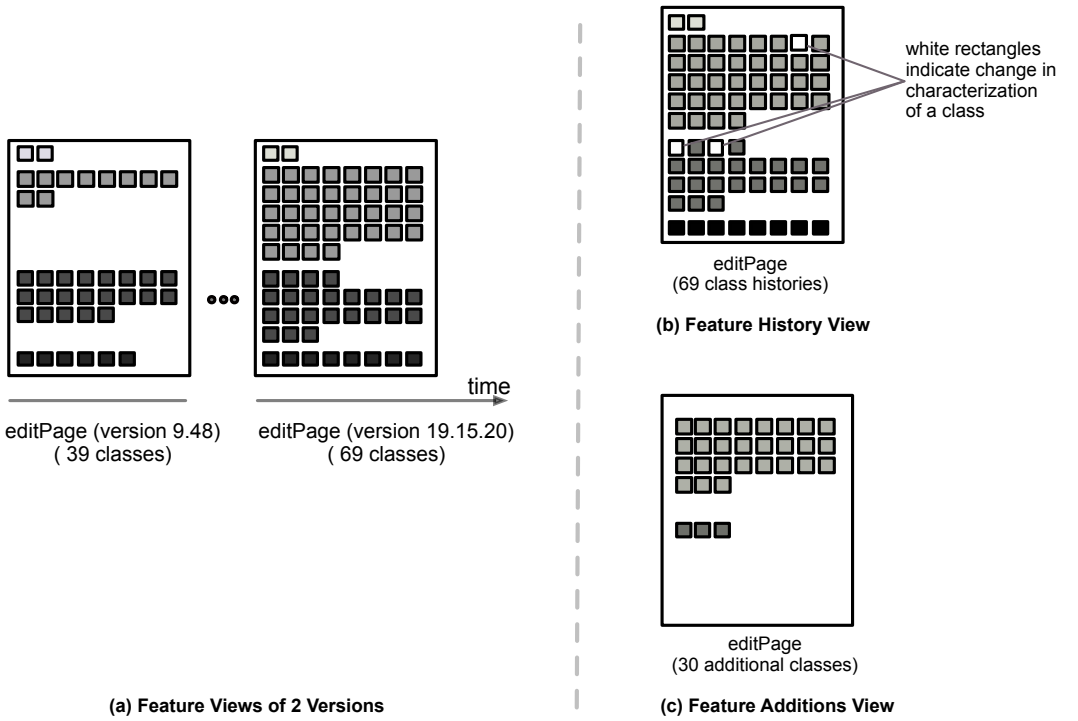

Figure 4. Feature History View (showing class histories, including the ones that were removed from the last version) and Feature Addition View (showing classes which have been added) of the *editPage* feature (Branch development track).

feature view as a whole. Nor do we see if a class characterization has changed over time, thus indicating a change in the functional role of the class with respect to the features.

The *feature history view* visualization provides this level of detail. In Figure 4 we show how each of our views is derived. Figure 4 (a) shows the first and last version of a feature view of the *editPage* feature. A close examination of the figure reveals that the number of classes has increased by 39 from the first to the last version. Figure 4 (b) shows the *feature history view* applied on the same example. As with the feature views of each individual version, (a) is represented as four characterized class groups. The small colored squares represent class histories (*i.e.,* a set of versions of a class [21]). The colors represent the characterization of a class in the last version. The order of the class histories in the sets reveals information about its history with respect to the feature view (*i.e.,* classes which were present since the early versions are shown first, newly added classes are shown last in the group). A white square represents a place holder for class that has disappeared from the grouping during the history

of a feature. The class may however still be participating in a feature, but has been assigned a new feature characterization. In the *high-group-feature* set of classes of this example, we see that the first class of the ordered set is shown as a white box. Thus, its characterization with respect to the features has changed over the evolution. Figure 4 (b) shows the additional classes participating in the feature. This provides an overview of added classes (*i.e.,* new classes particpating in the feature).

To conclusively determine whether a class has been added to a feature view, we need to apply the *addition* history measurements to the CF (number of classes referenced in a feature view) property. Figure 4 (b) shows the actual classes that have been added to the *editPage* feature. The feature characterization of the classes is computed with respect to the last version. The Feature Additions View Figure 4 (b) reveals that there are no additional infrastructural classes, whereas in the Feature History View Figure 4 (a) there appeared to be additions in this category. However they appear in this category due to a change of characterization of the classes over time. The place holders in the HGF_c category indicate these classes have possibly been re-characterized.

All our feature view visualizations are interactive; we can query each small rectangle to discover which class (or class history) is represented. Our visualizations are generated using Mondrian [23] which is integrated in our Moose reengineering environment [24].

4. Experimentation

For our experiments we chose *SmallWiki* [25], a fully object-oriented and extensible Wiki framework. SmallWiki provides features to create, edit and manage hypertext pages on the web. It is implemented in VisualWorks Smalltalk and is comprised of over 300 classes (considering only classes from the SmallWiki namespace). We decided to use SmallWiki for our case study for several reasons: (1) it is open source, thus its source code is freely available, (2) we have access to multiple versions of the system, (3) we are familiar with the features of the application from the user's perspective, and (4) we have access to developer knowledge to verify our findings.

Figure 5 shows the versions of SmallWiki we selected for our feature evolution analysis of two distinct development tracks originating from the same version. These are representative versions that reflect different phases of development in the lifecycle of SmallWiki.

Version 9.48 (22.03.2004). The original development of SmallWiki was done predominantly by two developers. The results of their work are represented by this version, a major release of the system.

Version 9.52 (17.09.2004). As SmallWiki system is an open source project, modifications and extensions are implemented by open source developers. 9.48 and 9.52 represent the main open source development track.

Versions 19.15.6 (30.08.2004) and 19.15.20 (08.09.2004). We selected this series of versions as it represents the work of a developer, who joined the development team at an advanced stage of development. He undertook the task to refactor and extend

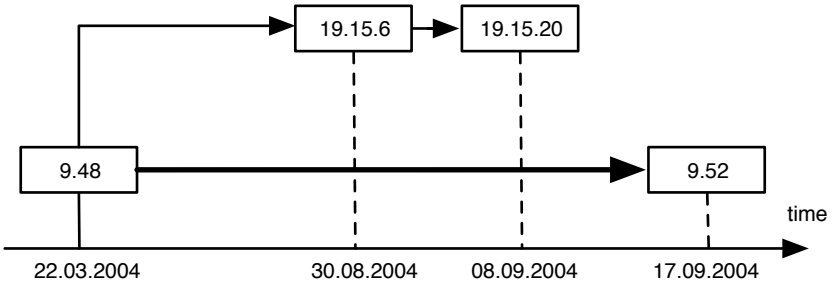

Figure 5. The order of the analyzed versions of Smallwiki.

SmallWiki with new features and new generic functionality which crosscuts the features of the application. These two versions represents a development branch of the system that is based on version 9.48. Changes to this version were *not* included in the 9.52 version of the system.

For our experimentation we want to analyze two distinct development tracks. In our first experiment we analyze the evolution of the branch development of the system in terms of how the modifications affected the existing features. In our second experiment we analyze the evolution of the same features from version 9.48 to 9.52 (the main open source development track). We want to see what changed in the branch (versions 19.15.6 and 19.15.20) and detect changes that could cause conflicts when merging the branch with the main development track.

In accordance with our definition of a feature as an observable unit of behavior [1], we identify features of SmallWiki by making the assumption that the elements of the user interface, namely the links, buttons and entry forms of the SmallWiki pages exercise distinct features. Based on this assumption, we selected 14 distinct interactive features (14 typical user interactions with the SmallWiki application such as login, editing a page or searching a web site). In addition, we also selected one non interactive feature (*s*tart SmallWiki) that initializes the application at startup. We exercised them on an instrumented system to capture 15 distinct execution traces.

Our dynamic analysis tool *TraceScraper* [17] allows us to define scripts to automate the execution of features. Thus, we ensure that the features are executed in the same way with the same inputs for each version of the system we analyze. We achieved 84 % coverage of the classes.

We assume a one-to-one mapping between features and traces. However, this assumption may not yield optimal results. Only by applying our feature analysis technique can we uncover similarities (*e.g.,* a generic approach to the way that they are implemented) in features. Thus, this suggests that an iterative approach to the selection and definition of features is required to obtain an optimal choice of features that execute distinct functionalities. We discuss feature definition in more detail in Section 5.

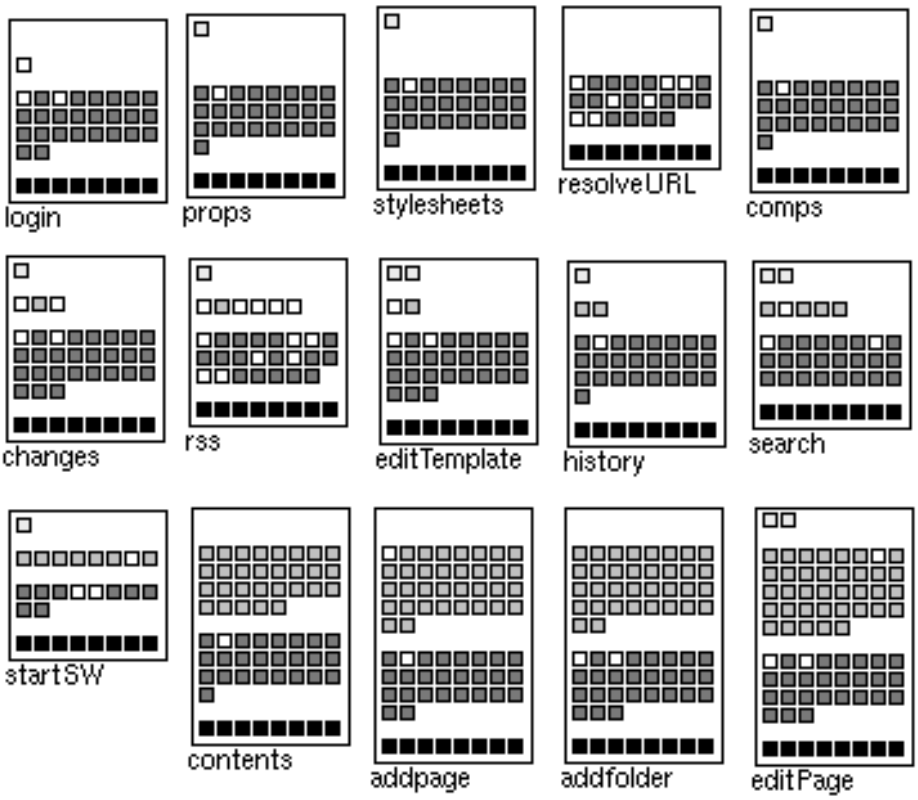

Figure 6. Feature History Views of 15 feature views considered in the branch of Smallwiki.

Figure 6 (*feature history view*) shows a summary of the evolution of the 15 feature views of SmallWiki in the branch.

4.1. Outline of our Evolution Analysis Approach

The goal of our analysis is to reveal the *extent* and *intent* of changes that are made to the system over time. We interpret these changes in the context of the features. We detect which features are affected by change and we interpret the extent of a change based on the characterization of the class where the change occurred. In other words, we describe the changes we detect in terms of the feature property that revealed that change. In general, our approach consists of the following steps:

1. We define a model for each version of our evolution analysis. We instrument each version and exercise the same set of features. For each feature we execute, we extract traces of runtime behavior. We resolve the references in the traces to the class and method entities of the model, which we derive by static analysis of the source code. Our feature views are generated for each feature trace and we apply our measurement to characterize the classes of feature views into four groups.
2. We apply history measurements to (1) the feature view properties (SF_c, LGF_c, HGF_c and IF_c), and (2) to the CF (number of classes referenced in a feature view) property. The interpretation of the history measurements depends heavily on the key aspect of our approach: we always analyze the *same* set of features, in the same way for each version. Furthermore, from a users perspective, the system appears to behave in the same way in each version.
3. To obtain a more fine-grained view of the changes, we plot the values of the feature view properties over the root version of our analysis (9.48) and the two versions of the SmallWiki branch development track (19.15.6 and 19.15.20) as simple line graphs, as shown in Figure 7. This visualization reveals when (*i.e.,* in which version) the changes occurred. Thus this visualization supports a *version-centered* approach to analyzing the evolution of feature views.
4. We analyze the *feature history view* visualization (*e.g.,* Figure 6 of the branch development track) which summarize the changes in feature views. The position of the class histories within the class characterization group of a feature view indicates of when they appeared in the feature view (*i.e.,* classes that have been present in early versions of the analysis appear first in the ordered set of characterized classes. Additions to the set appear at the end of the set).
5. We drive the analysis with the questions we asked in the introduction.
6. We summarize our findings and check them with the developers. Based on the developer knowledge, we document the context of the changes that our feature analysis reveals.

4.2. Experiment 1 - Analyzing the evolution of the branch development track

The branch development code base of SmallWiki consists of the evolution of the versions on the main axis as shown in Figure 5. Simply by applying history measurements to feature views properties, we detect *what* has changed in the system in the context of the features. By applying the history measurement to the individual properties, we qualify the changes and thus define the extent of their influence on the features of our model.

Which features are affected by changes in the code?

As a first step we isolate the features that have changed. Then we group the changes by applying the *number of changes* history measurement to each of the four properties (SF_c, LGF_c, HGF_c and IF_c) of feature views. Figure 7 shows the evolution charts for 3 versions of the 15 features.

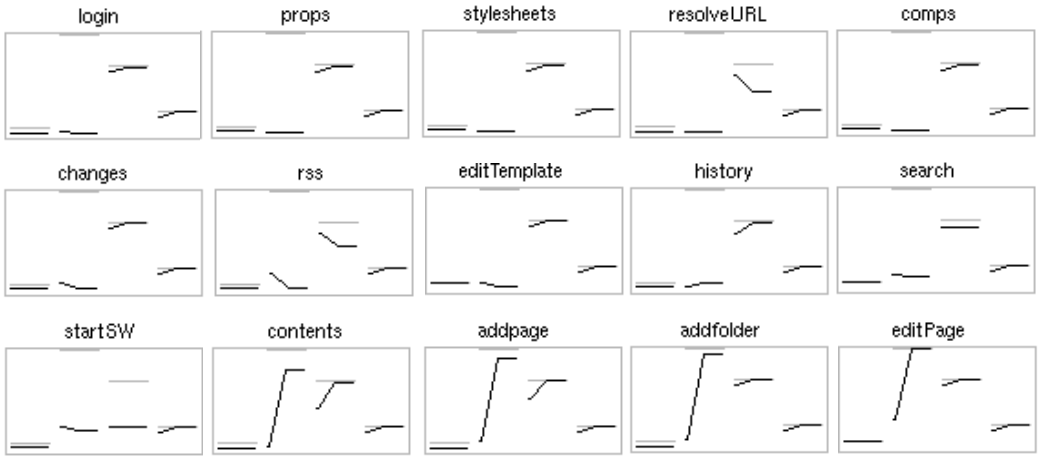

Figure 7. The evolution charts of the 15 features considered in Smallwiki (branch development).

Single Feature Changes. A change in the number of single feature classes (SF_c) is referred to as a *single feature change*. (*i.e.,* the extent the change is limited to the feature where the change was detected). This reveals that none of features of our analysis exhibit *single feature changes*. The evolution charts shown in Figure 7 reflect this result as the plot for the single feature classes ((SF_c) column) remains unchanged for each version.

Low Group Feature Changes. By definition low group feature change affects a subset of the features ($< 50\%$ of the features of our model). Most of the considered features are affected by this type of change. This is reflected in Figure 7. Only four features are *not* affected by this type of change, namely *properties*, *stylesheets*, *resolveURL* and *comps*.

High Group and Infrastructural Feature Changes. All features have been affected by these types of changes. Both high group feature change and infrastructural feature change imply changes to generic functionality of the application that is being used by all the features. The evolution charts (Figure 7) reveal that all changes were made in the second version analyzed.

Are features becoming more complex over time?

To determine if a feature is becoming more complex over time, we apply the *Additions* history measurement to the CF (number of classes referenced in a feature view) property.

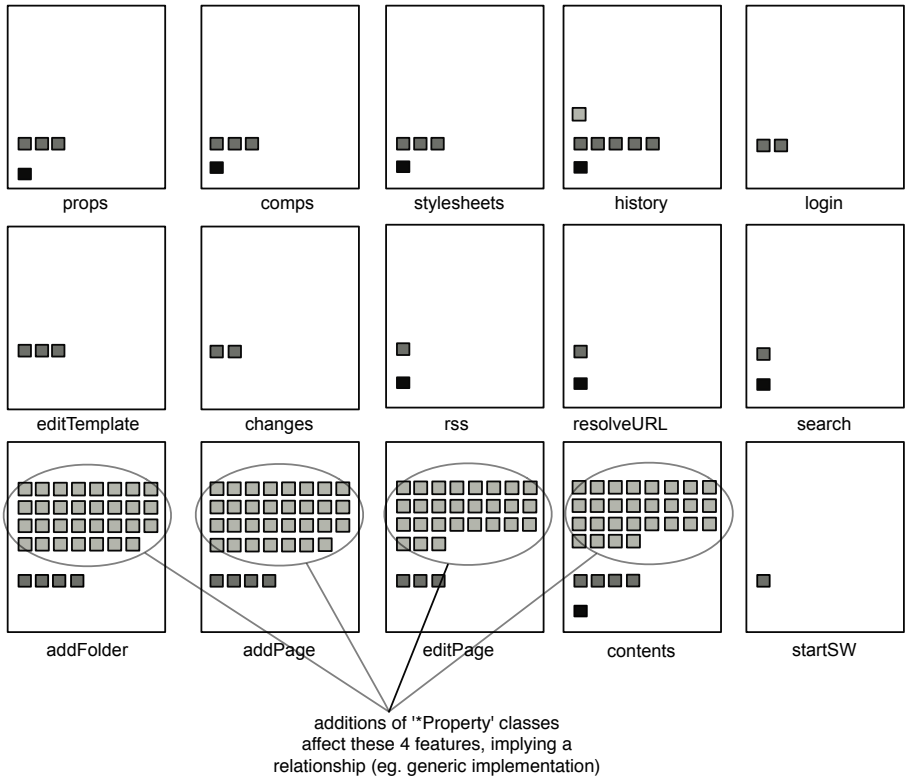

Figure 8. Feature Addtions Views (*i.e.,* showing only classes which represent additions to a feature view) of the Branch. The characterizations are then calculated based on the last version under analysis. The feature addition views are sorted by similarity to highlight patterns of change. For example, we emphasize the similar additions of low group classes.

In Figure 9 we plot the values of the CF measurement for both the main development track and the branch. The light colored bars represent additions to the features in the branch, thus revealing that all features show an increase in CF. Our graph shows that most of the additions occur in the 4 features *contents, addPage, addFolder and editPage*.

To differentiate between types of additions and determine the extent of their influence on the features, we then compute feature characterization for the added classes with respect to the last version. We discover that most of the changes are in the characterization LGF_c (number of low group feature classes in a feature view) and the features that are most affected are *contents, addPage, addFolder and editPage*. The feature views visualization of these features reveal similarities in classes and patterns of evolution. In Figure 6 we show the feature views of these features in the root version of our analysis (9.48) and the last version (19.15.20) of the

branch development track. We highlight the patterns of additions in the classes characterized as *low group*. By querying the classes, our visualizations reveal that theses additions represent the same classes. We discover that the added classes are named in a similar way *Property *e.g.,* AccessEditProperty, AccessRemoveProperty, AccessViewProperty, BrokenProperty.

Do similar patterns of change indicate relationships between the features?

An increase in the number of software entities shared between features suggests that the features may be related. For example, the implementation of these features may be realized using generic functionality. We see from the feature views that a large number of the classes participating in feature views are characterized as high group feature classes or infrastructural classes. This is due to the fact that SmallWiki is a web application and all features that are initiated by the user deal with the http request/response communication and page rendering.

We focus on changes to *low group feature* classes. By definition these are the classes that are shared by a subset of features. We identify patterns of change in these classes. One obvious pattern is shown in Figure 8. The features *editPage*, *addPage*, *addFolder* and *contents* are concerned with page rendering and storing of new pages or folders in SmallWiki. As these features change in a similar way over time, this indicates that they are closely related as they exercise generically implemented functionality of the system.

The line graph representations (see Figure 7) of the feature view evolution in terms of its properties reveal interesting patterns of evolution. We ordered the evolution charts in Figure 7 to emphasize patterns of change.

4.3. Developer Validation (Experiment 1)

As previously stated, for our analysis we chose features that, from a user's perspective, appear to behave in the same way for each version. Applying our history measurements to the classes reveals however, that for each feature, there is an increase in the number of classes that participate in the features. To obtain a contextual perspective of the additional classes appearing in the feature views we looked for *Additions* to each of the four characterized groups of classes. Our analysis reveals two main results:

- There are similar patterns of change (addition of low group feature classes) detected in the features *addPage* (31 classes), *addFolder* (31 classes) , *Contents* (28 classes) and *editPage* (27 classes).
- There is a small increase in high group classes and infrastructural classes (3 classes per feature on average), thus indicating the addition of functionality that affects most or all the features under analysis.

To verify our hypothesis that our feature views support understanding of the extent and intent of change, we asked the developers to state the purpose of the changes made during this development phase. In particular, with respect to the results, we asked if the changes were made to the four features listed above. The developer confirmed our findings by stating that a large proportion of the changes that he made were to reengineer how elements of the application (*e.g.,* form fields, labels, pages, folders) were manipulated and represented internally. This

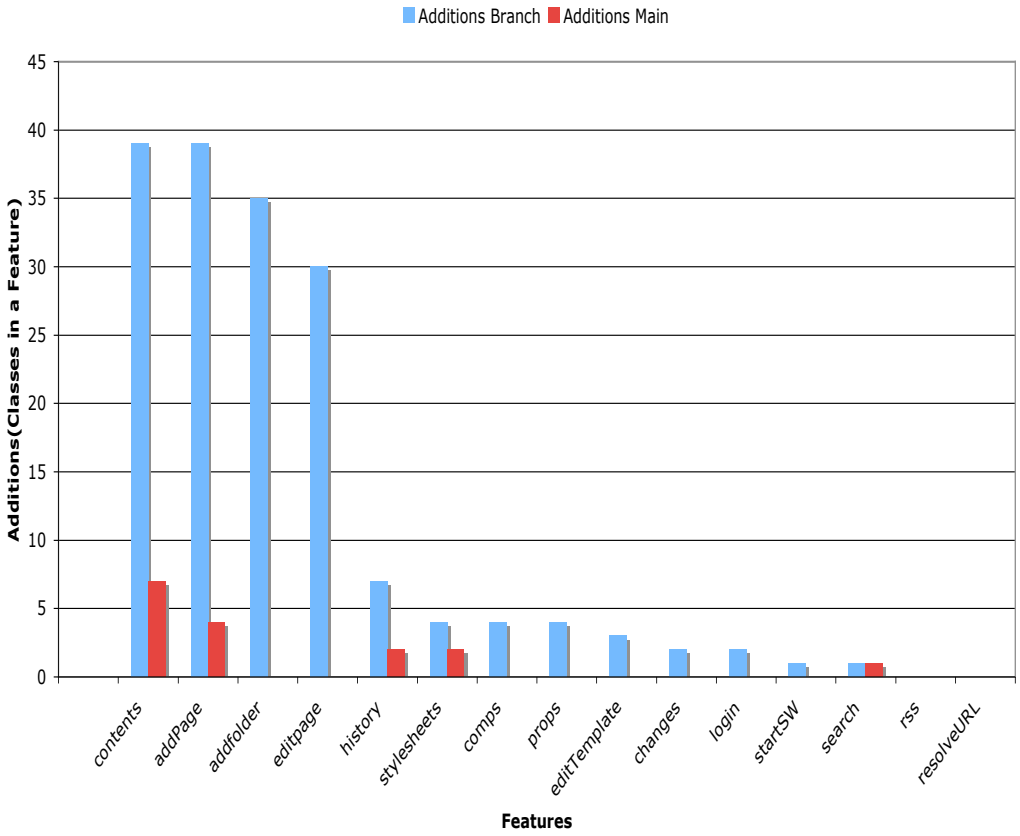

Figure 9. Additions History measurement applied to Number of Classes of a Feature (CF) for all features of the two (branch and main) development tracks.

reengineering effort accounts for the appearance of new classes, not specific to one feature but rather to a group of features concerned with page and folder manipulation.

The reengineering effort accounts for the appearance of new classes in the feature views over the versions of the branch, which our feature analysis characterized as *Low group additions*. Using the interactive capability of our feature view visualizations, we query to reveal the names of the additional classes. The new classes, for example AccessEditProperty, AccessRemoveProperty, AccessViewProperty, BrokenProperty participated in the features of the last two versions of the branch . Once again the developer confirms that these classes implement

a generic mechanism to define and add properties to SmallWiki pages. The classes are accessed by all the features that manipulate pages and folders. Thus our feature analysis reveals the addition of the classes and the context of these additions in terms of the features in which they participate. Furthermore, the developer confirmed that these classes do not affect their external observable behavior of the features.

The High group and Infrastructural Changes. The small increase we detected in our analysis was also identified by the developers. This reveals an the extension of the user role authentication functionality. The classes BasicRole and AdminRole are responsible for limiting access to administrator functionality. The developers confirm that this functionality has been incorporated into all the features included in our analysis. The integration of role-based authentication for all features was one of the defined goals of this development track.

4.4. Experiment 2: Analyzing the Evolution of the Main Development Track

The main development code base of SmallWiki consists of the evolution of the two versions on the main axis as shown in Figure 5. The focus of our second experiment was to apply our analysis technique to identify changes in the main development track that conflict with or duplicate the effort of changes in the branch.

Before we can merge two development branches, we first need to locate where changes in the system occurred and then determine if there are any conflicting changes. Our feature views provide us with the context of the changes.

The most striking result we obtained was when we compare the results of applying the *additions* history measurement to all the classes (CF) in the main development track and in the branch. The results are shown in the bar chart of Figure 9. The bars (light gray/blue) on the left to represent the number of additions in the branch track, and the bars (dark gray/red) on the right to represent the number of additions to the main development track. Only five features of the main development track additional classes appear (at most seven additional classes), whereas in the branch we see that all features have additional participating classes. Thus, we detect that the branch development exhibits more additions that affect more features.

As a next step we compute *number of changes* of the feature view properties to see which and how features changed:

numberOfChanges(F,SF_c): only the *startSW* (SmallWiki initialization) exhibits *single feature* change.

numberOfChanges(F,LGF_c): two features *Login* and *changes* exhibit *low group feature change*. In Figure 10 we indicate a white box in the low group position for the *login* feature view. This represents a class history of a class that is no longer characterized as low group in this version. Similarily we detect a white box in the low group class histories of the *changes* feature view.

numberOfChanges(F,HGF_c), *numberOfChanges*(F,IF_c): all features exhibit *high group feature* and *infrastructural feature change*.

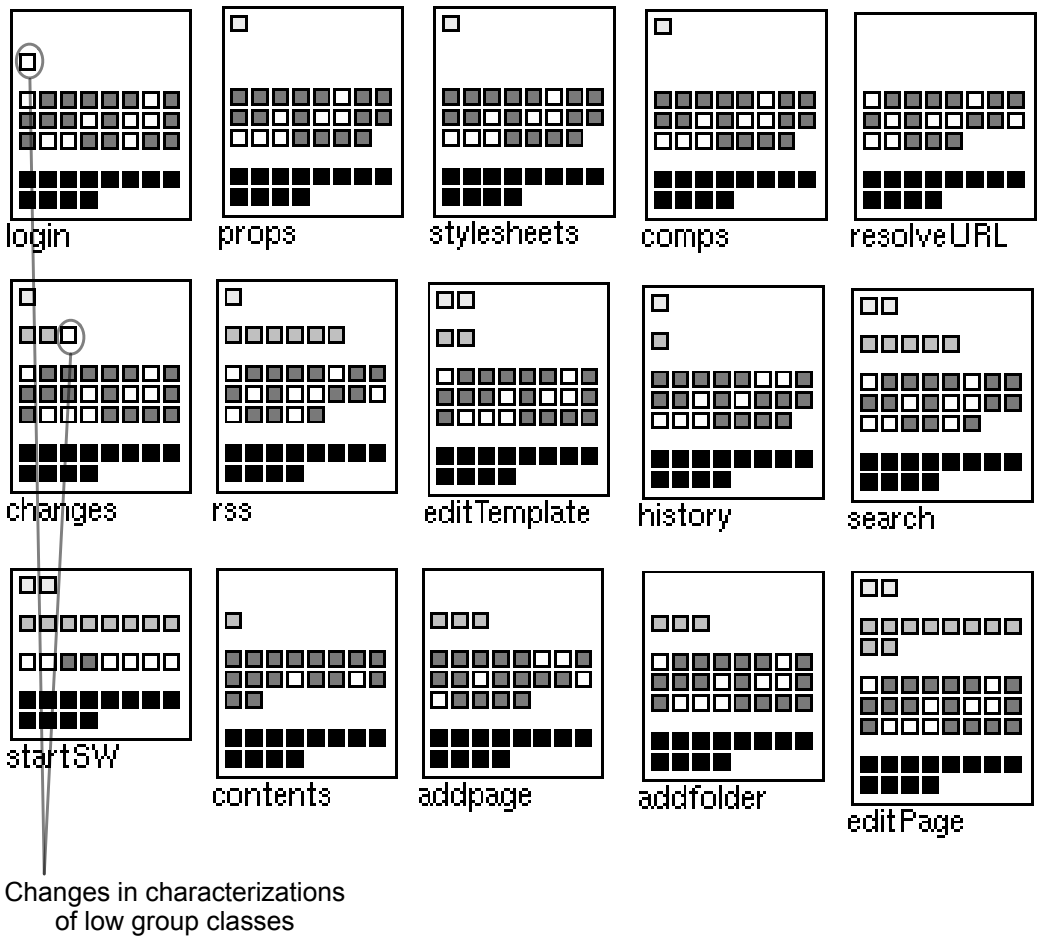

Figure 10. The Feature Change Charts of the 15 features considered in the main development track of Smallwiki.

We apply the *Additions* history measurement to the CF property and characterize the classes with the respect to version 9.52. Figure 11 shows the resulting *Feature addition views*. Our plot reveals that one additional class, namely ResourceStore, is participating in the *startSW* feature in version 9.52.

The class BasicRole is characterized as an infrastructural feature class. We also detected this change in the branch development track. This suggests that a similar change has been made to this class in both development tracks. This change represents the incorporation of the role

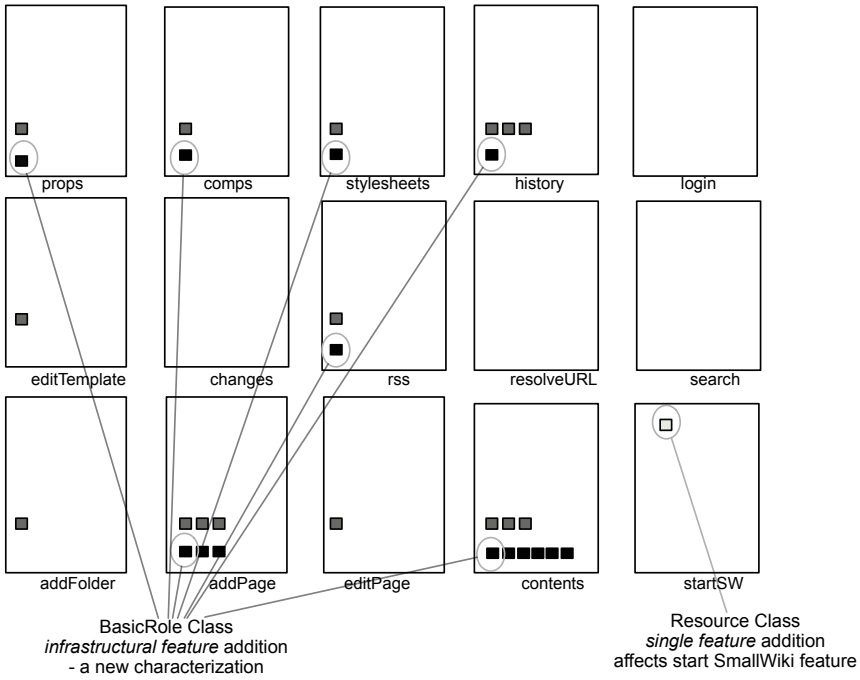

Figure 11. Feature Additions Views of the main development track. This shows only additional classes. The characterizations are calculated with respect to the last version of analysis.

checking functionality in all features of the system. The classes were characterized as *single feature* classes in version 9.48 of SmallWiki.

One goal of our experiment is to show how our technique supports developers when changes from two distinct development tracks need to be merged. We isolate and characterize the types of changes. Thus, we reduce the volume of information to be analyzed and we consider each type of change separately. For example, a *single feature* change is localized to one feature of our analysis. Furthermore, we distinguish between single feature additions and single feature removals. In both these cases we need to distinguish between classes that have been added to a feature view and those whose functional role has changed with respect to the feature model. Our feature change chart of class histories supports this.

Another important factor when merging two distinct development branches is to identify source artifacts in the code that have changed in both development tracks. These changes may be more difficult to merge as the two distinct development tracks may reveal conflicting changes, which if merged, would result in bugs and loss of functionalities. Classes that appear to have changed in both development tracks may also indicate that the same functionality

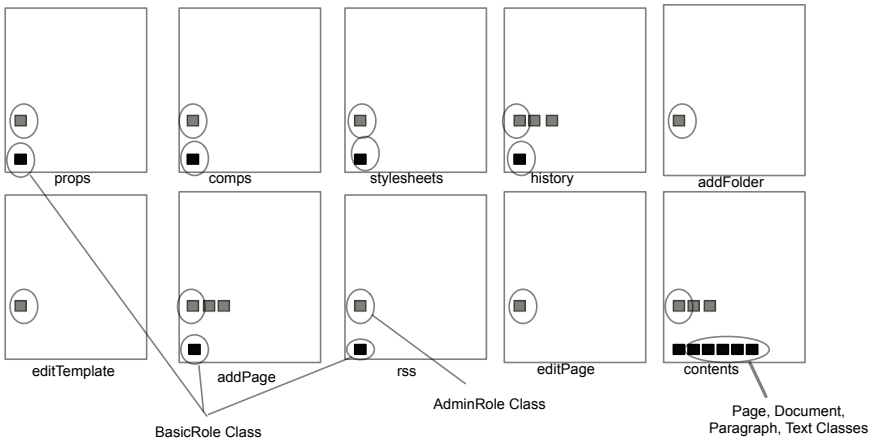

Figure 12. Feature Views showing only the conflicting additions (*i.e.,* classes that have been added to both main and branch development tracks).

has been added in both places. Our analysis of SmallWiki reveals an example of this. The BasicRole class appears to have changed in the same way in both development tracks (*i.e.,* it changed from being a single feature class to an *infrastructural feature* class. This is due to the fact that this functionality was reused by all the features in the later versions of the system.

The single feature addition changes that we detected in our two experiments affect different features and different classes. The changes we detected in the *startSW* feature of the branch version represent new functionality. These changes could therefore be merged back in the original development track without affecting the other features. A useful side effect of our approach is that we identify which features are affected by changes and thus require to be regression tested after the merge is complete.

4.5. Developer validation (Experiment 2 - Main development track)

The major findings of our feature analysis of the main development track are:

- We discover the addition of a class named ResourceStore as an addition to the *startSW* feature. As it is characterized as single-feature, this suggests that this class provides specific functionality to the system at initialization.
- We detect that the characterization of the classes BasicRole and AdminRole changes in the main development branch over time. In the initial version of our analysis, these classes were characterized as *low group feature*, whereas in the last version of our analysis, they are characterized as infrastructural feature classes.

- We detect additions of infrastructural classes, namely Page, Content, Text, Document in the *contents* feature.

Figure 12 isolates the conflicting changes and shows only additions that are common to both development tracks. For example the BasicRole class appears as an additional infrastructural class in both places. Similarily AdminRole appears as an additional high group feature class. In the root version of both tracks, these classes were characterized as *low group feature*. On further analysis, we discover that these additions occur in both tracks as a result of changed characterization.

The developers of the main development track confirm our first finding. They reveal that the class ResourceStore provides SmallWiki with a a cache implementation that has been added in the version 9.52 and is instantiated and populated by SmallWiki at startup.

Our second finding, namely the change of characterization of the BasicRole class, is confirmed by the developers of both main and branch developments. The integration of role-based authentication for all features represents a goal of both development tracks. The BasicRole and the AdminRole classes are responsible for role-based authentication that determines if a user has access rights to the features of the system. Typically access control cross cut all the features, as it is queried before each execution. The developers confirm that in the first version of our analysis, these classes were present but role-based authentication was only being checked by some of the features. This explains why they were initially characterized as *low group feature*.

The developers explain our third finding. New functionality has been added that results in the content feature registering all possible page contents. Thus, these classes are detected as new additions to this feature view.

5. Discussion

Our analysis of SmallWiki shows how we applied our approach to reason about the evolution of the system in terms of its features. We analyze changes in the context of feature views in two parallel development tracks and used the features to define the intent and extent of the changes. In particular, we highlight changes that may cause conflict when merging the two branches.

Feature definition. For this analysis, we adopted the definition of a feature as a unit of observable behavior triggered by the user [1]. However, our approach does not exclude other definitions such as features that carry out non-observable activities of a system like house-keeping tasks. Furthermore, for the case studies described in this paper, 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.

Furthermore, our analysis reveals that some features, for example *addFolder* and *addPage*, exercise exactly the same classes. Thus, these actions are closely related and could be treated as variations of the same feature, for example *addPageOrFolder*. High

degrees of similarities between features may suggest that we may need to consider a many-to-one mapping between user-triggerable actions and features. It is by performing feature analysis in the first place that we determine such similarities. Thus, obtaining the best feature definition for an analysis is based on the analysis itself. This clearly suggests an iterative approach to feature definition based on the findings of feature analysis. We plan to investigate this more in the future.

Coverage. Our feature analysis approach does not achieve 100% coverage of the system. For the purpose of feature location, complete coverage of a system is not necessary [5]. Wilde and Scully's *Software Reconnaissance* technique, and other approaches based on this technique, 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 [5].

As our focus is on detecting changes over time, we sought to achieve high coverage so as to obtain a characterization for a large proportion of the classes of the system. However, as our experiments do not exercise features that have been added in later versions of the system, 100% coverage is difficult to achieve.

Defining characterizations. In previous works [17, 14] we computed three distinct characterizations of a software artifact to describe its level of participation in a set of features, namely *single feature, group feature and infrastructural*. In addition, we defined the characterization *not covered* to describe classes that do not participate in the features of our model. For our feature view analysis, we refine the granularity of *group feature* software artifacts. We distinguish between software artifacts that are common to a small group of features (*low group feature*) and those that are common to a large group of features (*high group feature*).

Low group feature classes are shared by a small number of features and represent classes that implement a functionality that is shared by features that exercise similar functionality. Based on the presence of low group feature classes we detect similarities between features.

High group feature classes represent classes that provide functionality that is used by most of the features in our model. In the case of SmallWiki, all the features we exercised except for the *startSW* are initiated by the user. These features exercised the classes responsible for handling http requests (`HttpRequest`), responses (`HttpResponse`) and page rendering code. These classes provide *infrastructural* functionality to the features. However they are characterized as *high group feature* classes and not as infrastructural. This is because we included the *startSW* (start SmallWiki) feature in our analysis. This feature does not involve interaction with the end user, but it does require the system administrator to act.

The characterization of classes in feature views is therefore dependent on the set of features we chose to exercise. By distinguishing between low group and high group feature classes our characterizations are more reliable. We are better able to obtain a characterization of a class that reveals its role with respect to the features. However, we are in a position to assess our choice of features only after we have performed feature analysis.

Our feature characterization measurements currently define a threshold value of 50% to distinguish between low group and high group software artifacts. Perhaps the threshold value could also be defined by the reverse engineer, depending on the type of application to be analyzed. We plan to experiment with variable threshold values in the future.

Stability of characterizations. Our feature characterization provides a means of interpreting the role of a class based on the results of exercising a set of features. This feature characterization is dependent on the number of features exercised and the type of features traced. If two features with similar functionality are executed, classes that are specific to these features will be characterized as low group. In our experiment, we chose to exercise the same set of features for each version. Thus we expect that the characterizations under these circumstances should remain the same. Our technique detects these changes in characterization and thus supports the software developer to interpret the changes in the context of the features that have been exercised.

Changing roles of classes. In a previous work [14] we measured how the roles of classes changed with respect to features over time. We applied the feature characterization measurement (FC) to classes and analyzed changes in the characterizations of classes over a series of versions. In contrast to our previous work, where we focussed on changes in characterizations of the individual classes, in this article we treat the feature views as the first class entity of our analysis. The feature view defines a dynamic relationship between classes at runtime. We reason about how features are affected by changes. From our experimentation with SmallWiki we see that some of the changes we detect in feature views are due to changes in the characterizations of classes. A change in a feature view may not necessarily imply that an individual feature is directly affected. For example, in the case of the *components* feature of SmallWiki we detected that single feature classes were removed in the third version of our analysis. However, when we applied the *addition* history measurement to the low group feature classes we discovered that the number of low group feature classes had increased. On closer examination of our visualization, we discovered that the classes had not really been removed from the feature. Their roles with respect to all the features of our model had changed. We interpret this change to mean that the functionality provided by these classes is being used by other features in the later version. Based on our experience with the feature view approach described in ths paper and that of the previous work [14], we plan to perform more experiments based on a combination of both techniques. In this way we would isolate additions or removals that appear as a result of changing roles from new classes appearing in features over time.

Levels of granularity. In this article, we experimented with feature views of classes. For a more coarse-grained overview of a system features, for example when considering large systems, we could define feature views as sets of characterized packages. During our experimentation with SmallWiki, analysis of our static models revealed that the number of packages increased from 13 to 43 in the main development track. We were interested to see which features where now using functionality of the new packages. Thus we extracted feature views as groups of packages and applied history measurements to

compute additions in the number of characterized packages participating in a feature. Our results revealed that 7 of the new packages were referenced by the *components* feature. Thus by analyzing feature views of packages we obtained a coarse-grained view of which features are affected by the addition of new packages in the system and what type of functionaliity is provided by the packages (*i.e.,* single feature, low group, high group or infrastructural).

In the same way we obtain a more fine-grained analysis if we extract feature views as sets of characterized methods and we apply our measurements to reveal how a feature view is changing with respect to its participating methods.

Scaleability of feature views. The number of artifacts (*e.g.,* classes or methods) participating in a feature view may be large, depending on how the features are defined. This could lead to large visualizations with each feature view containing many entities. In such a case, an iterative approach to feature analysis could be adopted. Initially the reverse engineer would obtain a "big picture" perspective of the features. Packages or subsystems could be the chosen software artifact. Moreover, if the focus of the analysis is restricted to a particular part of the system (*e.g.,* system startup), the choice of features could be restricted to relate only to this part. Clearly the problems of scalability may be addressed by adopting an iterative approach to feature definition and by selecting a more coarse-grained feature view.

Obtaining feature traces. One of the difficulties of obtaining feature traces for a series of versions of a system, is that in some of the versions bugs may have been introduced which cause some or all of the features to be broken (*i.e.,* they do not function correctly). This problem could be overcome if it became an established *best practice* to associate usage scenario tests with each version of the system and incorporate them in the source code repository.

Language independence. Our technique is language independent as we work with a model of the system abstracted by static and dynamic analysis. Obtaining the traces from the running application typically requires code instrumentation. The means of instrumenting the application is language dependent. To obtain traces from our SmallWiki application, we use a code instrumenting technique for Smalltalk based on method wrappers [26]. In previous experiments with Java applications [27], we used the *Ejp (Extensible Java Profiler)* [28] based on the Java Virtual Machine Profiler Interface (JVMPI). As long as the traces obtained from the system under analysis contain message send events, our approach will work for any object oriented language.

6. Related Work

Our work relates to static and dynamic program analysis [29, 30, 31]. Our particular focus is on abstracting high level views of features to reason about the evolution of a system. The context of our work is reverse engineering. In the following subsections we outline the context and related research areas to our analysis.

6.1. Dynamic Analysis

Two main but distinct approaches to reverse engineering dominated reverse engineering research [32], namely dynamic analysis approaches and static analysis approaches. In recent years the synergies and dualities of these approaches have been recognized [33].

Many researchers emphasize the importance of incorporating dynamic analysis into the reverse engineering process. Stroulia and Systa [12] argue that static analysis approaches, though valuable are incomplete and do not meet reverse engineering goals of todays object-oriented systems. They highlight the importance of dynamic metrics as good indicators of external runtime behaviors. They define important considerations needed to achieve an optimal reverse engineering approach that combines static structural views and dynamic behavioral views of a system. A key aspect of our approach is that our model of dynamic feature behavior is placed in the context of static entities such as packages, classes and methods.

Our feature extraction technique is directly related to the field of dynamic analysis [34, 35]. Approaches based on dynamic analysis tend to be complex. The main reason is that it is difficult to design tools that process the huge volume of trace data and present the information in an understandable form [19, 36]. As a result much of the research in dynamic analysis focuses on this problem. Many compression and summarization approaches have been proposed to support the extraction of high level views to support system comprehension [37, 38, 39].

In the context of reverse engineering and system comprehension, Zaidman and Demeyer [39] propose an approach of managing trace volume through a heuristical clustering process based on event execution frequency. Their goal is to obtain an architectural insight into a program using dynamic analysis. They use a heuristic that divides a trace into recurring event clusters and show that these recurring event clusters represent interesting starting points for understanding the dynamic behavior of a system.

Zaidman *et al.* [20] define a dynamic analysis approach based on web-mining techniques that identifies key classes of a system. They show that well-designed object-oriented programs typically consist of key classes that work tightly together to provide the bulk of a systems functionality. In contrast to our feature-centric approach, this approach is based only on dynamic analysis. They do not partition the dynamic information into individual feature-traces. The advantage of analyzing individual feature traces is that we establish a mapping between features and code and exploit feature knowledge to reason about the higher level feature views.

6.2. A Review of Feature-Centered Approaches

Feature location in source code has been an active area of research in recent years. Many researchers have identified the potential of feature-centric approaches in software engineering and in particular as a basis for reverse-engineering [40, 41, 42, 43, 3].

Wilde and Scully [5] pioneered in locating features applying a purely dynamic approach which they call *Software Reconnaissance*. The goal of software reconaissance is to support maintenance programmers when they modify or extend functionality of legacy systems. Their approach deals with a single feature at a time and does not focus on the relationships between features. Their approach paved the way for subsequent research in feature location techniques.

Wong *et al.* [6] proposed three different metrics to determine quantitatively the binding of features to components or program code. Their technique captured 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. This technique represents a refinement of the Wilde technique. The underlying idea of this technique is the main inspiration for our feature characterization measurement of a software entity. In our experiments we define a characterization of classes based on their level of participation in the features under analysis. We exploit this information to determine the degree of relevance of a class for a given feature.

Chen and Rajlich identified requirements for an integrated support tool for feature location based on abstract system dependency graphs [40]. They proposed a semi-automatic approach as they emphasize the role of the software developer in the process of feature location.

Eisenbarth et al. [1] described a feature location technique which uses a combination of dynamic analysis and formal concept analysis to identify which computational units (*i.e.,* parts of the code) contribute to the behaviors of features. Similar to the software reconnaissance technique, they distinguished between general and specific computational units. They applied formal concept analysis to derive the correspondence between features and code. They used scenarios to invoke features and capture execution traces. Their technique identified computational units and is not concerned with the order of execution or the notion of time.

Eisenberg and de Volder [2] introduced a technique based on simple heuristics that uses ranking to determine the relevance of a software entity to a feature. They use test suites to generate *dynamic feature traces.* Their technique distinguished between parts of the code which are relevant for a feature and those which are not. Our four levels of characterization define a more fine-grained degrees of relevance of software entities to a feature.

Antoniol and Guéhéneuc [7] proposd an approach to feature location and feature comparison based on consolidated tools and techniques such as parsing and processor emulation. Their approach combines static and dynamic analysis.

Salah and Mancoridis [4] proposed a hierarchy of dynamic views based on execution traces of feature behavior. Their goal was to describe views that support program understanding by depicting low level interaction between objects of a trace and dependancies between features.

We build on the ideas of feature location approaches described above and we exploit the idea of feature views proposed by Salah and Mancoridis [4]. Our main focus is to exploit feature analysis to understand and interpret changes in the code over time. Thus, our approach complements and extends these existing approaches. In contrast to some of the previously mentioned approaches [1, 5], our main focus is applying feature analysis to object-oriented applications. The cornerstone of our approach is our definition a four layered characterization of features based on simple measurements. Thus we contribute to the state-of-the-art by exploiting feature analysis to understand the evolution of a software system.

6.3. Evolution Analysis of Features

Researchers have also considered evolution analysis of systems in terms of features. Fischer *et al.* [44, 45] modeled bug reports in relation to changes in a system. The purpose was to provide a link between bug reports and parts of the system that implement the code associated with the bug. They argue that parts of the system that change together are related.

Hsi *et al.* [46] 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 were derived from the user interface of an application. They compare models of an application while 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.

Licata *et al.* [47] assumed that unit tests of a system are partitioned into suites that are roughly aligned with the features of a system. The implementation of a feature "cross-cuts" the code base. They emphasized the value to new developers of a system of describing program changes in terms of features. Typically new developers run the program to form a mental model of the user-observable features of a system. Test suites describe a vocabulary that roughly corresponds to the user's, and thus the new developer's ontology of the program. Their approach focussed on the differences between two versions of the program.

Many approaches to evolution analysis are based on comparing two versions of a system to detect changes. The version-centered models allow for the comparison between two versions and they provide insights into when a particular event happened in the evolution. Xing and Stroulia detected server types of changes between two versions [48]. They represented each version of the system in an XMI format and then applied UML Diff to detect fine-grained changes like: addition/removal/moving and renaming of classes, methods and fields.

Our approach to evolution analysis through feature views is applicable to multiple versions of a system. We consider feature view histories as a set of versions and we apply history measurements to feature characterization properties. In this way we characterize and measure changes in features views. Our approach exploits the semantic knowledge of a feature view to interpret modifications to a system over time.

7. Conclusions

Our goal was to show that a features perspective of a system is a valuable asset when understanding the underlying reasons for changes. We combine static models of the source code with dynamic models of feature behavior to obtain a mapping between features and code. We then reason about a feature as a first class entity of our analysis. Our evolution analysis combines a history and a version anaylsis approach. We use simple visualizations to show features as groupings of participating structural entities, and we plot changes to the features to reveal the extent and type of changes in the code in the context of the features.

We describe the changes in a way that reveals how many and which features are affected by the change. Thus we describe change as *single feature*, *low group feature*, *high group feature* or *infrastructural*. In particular, we applied our approach on a case study consisting of versions from two distinct development tracks and we addressed the following questions:

1. *Which features are affected by changes in the code?* Our simple visualizations reveal which classes participate in the features at runtime. Our feature characterization of classes is computed based on the features we analyze. It categorizes the classes so that we can

reason about the types of changes we detect in the features over time, in particular changes that are localized to one feature (*single-feature* changes) and changes that affect most or all of the features. We apply history measurements and support evolution analysis of features with evolution charts and feature history views.

2. *Are features becoming more complex over time?* Our analysis reveals that in all cases, number of classes participating in features increases over time. This is a measure of increasing complexity. Our case study is a typical open source system that is constantly being maintained and extended with new functionality and features. Although we do not trace any new features in our experiments, our results reveal the appearance of classes in the system that indicate the addition of new functionalities or features.
3. *Do similar patterns of changes indicate relationships between features?* Due to the generic nature of SmallWiki, our analysis reveals that most of the classes in the feature views are shared by most or all of the features we analyze. The more classes are shared by features, the more features are affected by change to these classes. Moreover, our visualizations reveal changes to the characterizations of classes. For example, we detect classes that have changed from being *single-feature* to *infrastructural*. Although a feature where the change is detected may not be changed as a result of this type of change, the feature view perspective makes it possible to detect where changes have occurred and makes dependencies between features explicit.

In the future we want to apply our approach to more case studies. Furthermore, we want to extract feature views for different levels of granularity, namely packages and methods. We are interested to see if our analysis reveals similar patterns in the feature views and their evolution. Our approach is based on a combined model of static and dynamic information. We plan integrate this approach more with static analysis to correlate changes in the feature views with static changes.

Acknowledgments

We would like to thank the reviewers who gave valuable feedback. In particular we would like to thank Oscar Nierstrasz on his valuable comments on this work.

REFERENCES

1. Eisenbarth T, Koschke R, Simon D. Locating features in source code. *IEEE Computer*, 2003; **29**(3):210–224.
2. Eisenberg A, De Volder K. Dynamic feature traces: finding features in unfamiliar code. *Proceedings IEEE International Conference on Software Maintenance (ICSM 2004)*. IEEE Computer Society Press: Los Alamitos CA, 2005; 337–346.
3. Turner R, Wolf A, Fuggetta A, Lavazza L. Feature engineering. *Proceedings IEEE International Workshop on Software Specification and Design (WSSD 1998)*. IEEE Computer Society: Los Alamitos CA, 1998; 162.
4. Salah M, Mancoridis S. A hierarchy of dynamic software views: from object-interactions to feature-interacions. *Proceedings IEEE International Conference on Software Maintenance (ICSM 2004)*. IEEE Computer Society Press: Los Alamitos CA, 2004; 72–81.
5. Wilde N, Scully M. Software reconnaisance: mapping program features to code. *Software Maintenance: Research and Practice*, 1995; **7**(1):49–62.

6. Wong E, Gokhale S, Horgan J. Quantifying the closeness between program components and features. *Journal of Systems and Software*, 2000; **54**(2):87–98.
7. Antoniol G, Guéhéneuc Y.-G. Feature identification: a novel approach and a case study. *Proceedings IEEE International Conference on Software Maintenance (ICSM 2005)*. IEEE Computer Society Press: Los Alamitos CA, 2005; 357–366.
8. Jacobson I, Griss M, Jonsson P. *Software Reuse*. Addison Wesley/ACM Press, 1997.
9. Jerding D, Stasko J, Ball T. Visualizing message patterns in object-oriented program executions. *Technical Report GIT-GVU-96-15*. Georgia Institute of Technology, 1996.
10. Lange D, Nakamura Y. Interactive visualization of design patterns can help in framework understanding. *Proceedings ACM International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 1995)*. ACM Press: New York NY, 1995; 342–357.
11. Richner T, Ducasse S. Recovering high-level views of object-oriented applications from static and dynamic information. Yang H, White L, editors, *Proceedings IEEE International Conference on Software Maintenance (ICSM 1999)*. IEEE Computer Society Press: Los Alamitos CA, 1999; 13–22.
12. Stroulia E, Systä T. Dynamic analysis for reverse engineering and program understanding. *SIGAPP. Applied Computing Review*, 2002; **10**(1):8–17.
13. Systä T. Understanding the behavior of Java programs. *Proceedings IEEE International Working Conference in Reverse Engineering (WCRE 2000)*. IEEE Computer Society Press: Los Alamitos CA, 2000; 214–223.
14. Greevy O, Ducasse S, Gîrba T. Analyzing feature traces to incorporate the semantics of change in software evolution analysis. *Proceedings IEEE International Conference on Software Maintenance (ICSM 2005)*. IEEE Computer Society Press: Los Alamitos, 2005; 347–356.
15. Lehman M, Perry D, Ramil J, Turski W, Wernick P. Metrics and laws of software evolution–the nineties view. *Proceedings IEEE International Software Metrics Symposium (METRICS'97)*. IEEE Computer Society Press: Los Alamitos CA, 1997; 20–32.
16. Fowler M, Beck K, Brant J, Opdyke W, Roberts D. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.
17. Greevy O, Ducasse S. Correlating features and code using a compact two-sided trace analysis approach. *Proceedings IEEE European Conference on Software Maintenance and Reengineering (CSMR 2005)*. IEEE Computer Society Press: Los Alamitos CA, 2005; 314–323.
18. Kuhn A, Greevy O. Exploiting the analogy between traces and signal processing. *Proceedings IEEE International Confernce on Software Maintainance (ICSM 2006)*. IEEE Computer Society Press: Los Alamitos CA, 2006.
19. Ducasse S, Lanza M, Bertuli R. High-level polymetric views of condensed run-time information. *Proceedings of Conference on Software Maintenance and Reengineering (CSMR 2004)*. IEEE Computer Society Press: Los Alamitos CA, 2004; 309–318.
20. Zaidman A, Calders T, Demeyer S, Paredaens J. Applying webmining techniques to execution traces to support the program comprehension process. *Proceedings IEEE European Conference on Software Maintenance and Reengineering (CSMR 2005)*. IEEE Computer Society Press: Los Alamitos CA, 2005; 134–142.
21. Gîrba T, Ducasse S. Modeling history to analyze software evolution. *Journal of Software Maintenance: Research and Practice (JSME)*, 2006; **18**:207–236.
22. Gîrba T, Lanza M, Ducasse S. Characterizing the evolution of class hierarchies. *Proceedings IEEE European Conference on Software Maintenance and Reengineering (CSMR 2005)*. IEEE Computer Society: Los Alamitos CA, 2005; 2–11.
23. Meyer M, Gîrba T, Lungu M. Mondrian: an agile visualization framework. *ACM Symposium on Software Visualization (SoftVis 2006)*. ACM Press: New York, NY, USA, 2006; 135–144.
24. Nierstrasz O, Ducasse S. Moose–a language-independent reengineering environment. *European Research Consortim for Informatics and Mathematics (ERCIM) News*, 2004; **58**:24–25.
25. Ducasse S, Renggli L, Wuyts R. Smallwiki—a meta-described collaborative content management system. *Proceedings ACM International Symposium on Wikis (WikiSym'05)*. ACM Computer Society: New York, NY, USA, 2005; 75–82.
26. Brant J, Foote B, Johnson R, Roberts D. Wrappers to the rescue. *Proceedings European Conference on Object Oriented Programming (ECOOP 1998)*, volume 1445 of *LNCS*. Springer-Verlag, 1998; 396–417.
27. Kuhn A, Greevy O, Gîrba T. Applying semantic analysis to feature execution traces. *Proceedings IEEE Workshop on Program Comprehension through Dynamic Analysis (PCODA 2005)*. IEEE Computer Society Press: Los Alamitos CA, 2005; 48–53.
28. Vauclair S. Extensible Java profiler. Master's thesis, Ecole Polytechnique Fédérale de Lausanne, 2003.

29. Jacobson I. Use cases and aspects–working seamlessly together. *Journal of Object Technology*, 2003; **2**(4):7–28.
30. Jacobson I, Christerson M, Jonsson P, Overgaard G. *Object-Oriented Software Engineering — A Use Case Driven Approach*. Addison Wesley/ACM Press: Reading, Mass., 1992.
31. Memon A, Banerjee I, Nagarajan A. Gui ripping: Reverse engineering of graphical user interfaces for testing. *Proceedings IEEE Working Conference on Reverse Engineering (WCRE 2003)*. IEEE Computer Society Press: Los Alamitos CA, 2003; 260–269.
32. Chikofsky E, Cross II J. Reverse engineering and design recovery: a taxonomy. *IEEE Software*, 1990; **7**(1):13–17.
33. Ernst E. Higher-order hierarchies. *Proceedings European Conference on Object-Oriented Programming (ECOOP 2003)*, LNCS. Springer Verlag: Heidelberg, 2003; 303–329.
34. Ball T. The concept of dynamic analysis. *Proceedings European Software Engineering Conference and ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSC 1999)*, number 1687 in LNCS. Springer Verlag: Heidelberg, 1999; 216–234.
35. Winstead J, Evans D. Towards differential program analysis. *Proceedings ICSE International Workshop on Dynamic Analysis (WODA 2003)*. Portland, Oregon, 2003; 37–40.
36. Richner T, Ducasse S. Using dynamic information for the iterative recovery of collaborations and roles. *Proceedings IEEE International Conference on Software Maintenance (ICSM 2002)*. IEEE Computer Society Press: Los Alamitos CA, 2002; 34.
37. Hamou-Lhadj A, Braun E, Amyot D, Lethbridge T. Recovering behavioral design models from execution traces. *Proceedings IEEE European Conference on Software Maintenance and Reengineering (CSMR 2005)*. IEEE Computer Society Press: Los Alamitos CA, 2005; 112–121.
38. Hamou-Lhadj A, Lethbridge T. A survey of trace exploration tools and techniques. *Proceedings IBM Centers for Advanced Studies Conferences (CASON 2004)*. IBM Press: Indianapolis IN, 2004; 42–55.
39. Zaidman A, Demeyer S. Managing trace data volume through a heuristical clustering process based on event execution frequency. *Proceedings IEEE European Conference on Software Maintenance and Reengineering (CSMR 2004)*. IEEE Computer Society Press: Los Alamitos CA, 2004; 329–338.
40. Chen K, Rajlich V. Case study of feature location using dependence graph. *Proceedings IEEE International Conference on Software Maintenance (ICSM)*. IEEE Computer Society Press, 2000; 241–249.
41. El-Ramly M, Stroulia E, Sorenson P. Recovering software requirements from system-user interaction traces. *Proceedings ACM International Conference on Software Engineering and Knowledge Engineering*. ACM Press: New York NY, 2002; 447–454.
42. Hsi I, Potts C. Ontological excavation: unearthing the core concepts of an application. *Proceedings IEEE Working Conference on Reverse Engineering (WCRE 2003)*. IEEE Computer Society Press: Los Alamitos CA, 2003; 345–352.
43. Mehta A, Heineman G. Evolving legacy systems features using regression test cases and components. *Proceedings ACM International Workshop on Principles of Software Evolution*. ACM Press: New York NY, 2002; 190–193.
44. Fischer M, Gall H. Visualizing feature evolution of large-scale software based on problem and modification report data. *Journal of Software Maintenance and Evolution: Research and Practice*, 2004; **16**(6):385–403.
45. Fischer M, Pinzger M, Gall H. Analyzing and relating bug report data for feature tracking. *Proceedings IEEE Working Conference on Reverse Engineering (WCRE 2003)*. IEEE Computer Society Press: Los Alamitos CA, 2003; 90–99.
46. Hsi I, Potts C. Studying the evolution and enhancement of software features. *Proceedings IEEE International Conference on Software Maintenance (ICSM 2000)*. IEEE Computer Society Press: New York NY, 2000; 143–151.
47. Licata D, Harris C, Krishnamurthi S. The feature signatures of evolving programs. *Proceedings IEEE International Conference on Automated Software Engineering*. IEEE Computer Society Press: Los Alamitos CA, 2003; 281–285.
48. Xing Z, Stroulia E. Understanding class evolution in object-oriented software. *Proceedings 12th IEEE International Workshop on Program Comprehension (IWPC'04)*. IEEE Computer Society Press: Los Alamitos CA, 2004; 34–43.
49. Bézivin J, Gerbé O. Towards a precise definition of the OMG/MDA framework. *Proceedings Automated Software Engineering (ASE 2001)*. IEEE Computer Society: Los Alamitos CA, 2001; 273–282.
50. Seidewitz E. What models mean. *IEEE Software*, 2003; **20**(5):26–32.

APPENDIX A: HISTORY MEASUREMENTS

We define how we calculate the history measurements that we use for the evolution analysis of feature view histories and class histories.

Number of Changes (Num(P))

We define number of changes measurement applied on a feature view history F for a given property P.

$$(i > 1)$$
$$Num_i(F,P) = \begin{cases} 0, & P_i(F) - P_{i-1}(F) = 0 \\ 1, & P_i(F) - P_{i-1}(F) \neq 0 \end{cases}$$

$$(n > 2) \qquad Num_{1..n}(F,P) = \sum_{i=2}^{n} Num_i(F,P) \tag{1}$$

Additions (A(P))

This measurement sums the additions of a property P of a feature view history F.

$$(i > 1)$$
$$A_i(F,P) = \begin{cases} P_i(F) - P_{i-1}(F), & P_i(F) - P_{i-1}(F) > 0 \\ 0, & P_i(F) - P_{i-1}(F) \leq 0 \end{cases}$$

$$(n > 2) \qquad A_{1..n}(F,P) = \sum_{i=2}^{n} A_i(F,P) \tag{2}$$

Glossary

For entity, we use the definition as found in the Webster Dictionary:

> An *entity* is something that has separate and distinct existence in objective or conceptual reality.

For the general terms of model and meta-model we use the following definitions:

> A *model* is a simplification of a system built with an intended goal in mind. The model should be able to answer questions in place of the actual system [49].
>
> A *meta-model* is a specification model for a class of systems under study where each system under study in the class is itself a valid model expressed in a certain modeling language [50].

For features we use the following definitions:

> A *feature* is a realized functional requirement of a system. A feature is an observable unit of behavior of a system triggered by the user [1].
>
> A *feature-trace* is a sequence of runtime events (*e.g.,* object creation/deletion, method invocation) that describes the dynamic behavior of a feature.
>
> A *feature model* is the set of features of a analysis.
>
> A *feature characterization* describes the degree of usage of a software entity by the features of the feature model
>
> A *feature view* is a set of software entities abstracted from a feature-trace. A feature view can be a set of sets of characterized classes.

$$FeatureView = (C_{sf}, C_{lgf}, C_{hgf}, C_{if})$$