Correlating Features and Code Using A Compact Two-Sided Trace Analysis Approach

accepted to CSMR 2005

Orla Greevy and Stéphane Ducasse
Software Composition Group
Institut für Informatik und angewandte Mathematik
University of Bern, Switzerland
{greevy, ducasse}@iam.unibe.ch

Abstract

Software developers are constantly required to modify and adapt application features in response to changing requirements. The problem is that just by reading the source code, it is difficult to determine how classes and methods contribute to the runtime behavior of features. Moreover, dependencies between system features are not obvious, consequently software maintenance operations often result in unintended side effects. To tackle these problems, we propose a compact *feature-driven approach (i.e., summarized trace information) based on dynamic analysis to characterize features and computational units of an application. We extract execution traces to achieve an explicit mapping between features and classes using two complementary perspectives. We apply our approach to two case studies and we report our findings.*

Keywords: Feature, Feature-Traces, Dynamic Analysis, Software Evolution, Software Metrics.

1. Introduction

Developers who maintain and extend complex software systems are expected to translate change requests and bug reports into modifications in the code. This task is difficult because such requests are usually expressed in a language that reflects a feature perspective of the system [14]. As a result, developers spend lot of time locating relevant parts of the code before making the required modifications. If they are uncertain about how the features of a system interact, they risk adversely impacting other features as a result of their changes.

To address these problems we propose a *two-sided* approach to feature analysis. Essentially we aim to answer the following questions:

- *How do features relate to classes?* Understanding how a feature is implemented is essential for the software maintenance phase of a system. Moreover, developers tend to use their knowledge of how existing features are implemented as a basis for adding new features.
- *How do classes relate to features?* Knowledge of the role of a class in the behavior of a system is useful, when a developer needs to modify or adapt it.
- *How are features related to each other?* Knowing which features could be affected by modifications helps the developer estimate which parts of the system could be affected and need regression testing.

Several works [4, 25] have shown that dynamic analysis is a reliable means of associating behaviors of a system with the internal components of its implementation. However, dynamic analysis-based approaches tend to be complex. The main reason is it is diffcult to design tools that process the huge volume of trace data and present the information in an understandable form [20, 26, 3].

Considering the characteristics of previous works, the key contributions of our approach are: (1) an *easy-to-use* approach, (2) compactness of the trace information and (3) *two-sided* views.

Our approach is *easy-to-use* as we define a simple mapping between system behaviors and features. We automate the capture of individual traces of *features* that can be reproduced every time we need to experiment with them. Moreover, we compute some simple sets from the traces, which we use to characterize features and computational units of an application.

We tackle the problem of handling huge amount of information by *compacting* the traces to focus on key information. We refer to the compacted versions of the traces as *feature-fingerprints*.

Our *two-sided* approach characterizes both features and computational units. Firstly, we extract a feature perspective of a system by characterizing features in isolation and in combination with other features. Secondly, we extract a computational unit perspective which we use to characterize computational units based on how they participate in features.

The aim of our work is to define a technique that is easily integrated in the software life-cycle as means of supporting software developers when they have to modify or add features to an application.

Structure of the Paper. This paper is structured as follows: in Section 2 we introduce the terminology of our approach. In Sections 3 and 4 we introduce our two-sided approach and the high-level views we extract. In Section 5 we report on two case studies we conducted using our approach and present the results. Subsequently, in Section 6 we discuss our findings and outline the constraints and limitations of our approach. We summarize related work in Section 7. Finally, we outline our conclusions and future work in Section 8.

2. Terminology

In this paper we adopt the definitions of features and computational units proposed by Eisenbarth *et al.*[4] and refine them for the purpose of our approach.

Feature. *"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"* [4].

Not all features of an application satisfy this definition. System internal housekeeping tasks, for example, are not triggered directly as a result of user interaction. We limit the scope of our investigation to user-initiated features. We focus on the user-interface of an application to determine which user-observable features are most adequate to include in our analysis.

Computational units. *"A computational unit is an executable part of a system"* [4]. In our approach, we focus on characterizing features of object-oriented applications. We consider classes as computational units. However, our approach is also applicable to finer-grained computational units such as methods, or coarser-grained computational units such as packages.

Feature-Traces. We introduce the term *feature-trace* to refer to an individual execution trace captured as a result of triggering a feature. A feature-trace consists of performed method calls.

Feature-Fingerprints. A feature-fingerprint is a set of classes and/or methods which conform to a certain rule (described in Section 3.2). Due to iteration and recursion, feature-traces contain multiple references to the same classes and methods. Feature-fingerprints reduce the volume of information but at the same time preserve the necessary information to establish the relationships between features and computational units.

The Feature Model. In Figure 1 we show the relationship between feature-traces and *class* and *method* entities, we obtain by extracting a model of the source code.

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 that one path of execution is useful enough to reveal a mapping which directs the software developer to the relevant computational units (as discussed in Section 6).

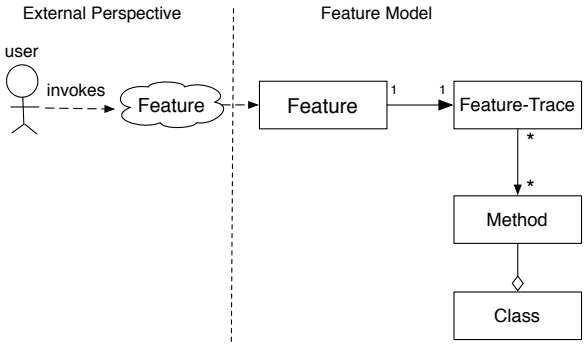

Figure 1. The Relationship between Features and Feature-Model (in UML notation). 'Class' and 'Method' are model entities extracted by static analysis of the source code.

3. Two-Sided Metric-Based Feature-Oriented Approach

The basis of our approach is to obtain a feature model, that combines dynamic information with a static model of the application under analysis as shown in Figure 1. Our feature model represents the relationship between the external viewpoint of a system in terms of features and the internal viewpoint in terms of computational units.

The focus of our approach is to apply a *two-sided* analysis of the feature model by extracting two distinct but related perspectives:

- *A Feature Perspective*, to identify the characteristics of feature-trace entities.

- *A Computational Unit Perspective*, to identify characteristics of class and method entities with respect to the features.

Figure 2 describes the overall process of our approach: first the system is instrumented, individual behaviors are triggered and for each behavior, a feature-trace is collected. The feature-fingerprints (described in Section 3.1) are computed for each feature-trace. Based on this information, different perspectives are developed supported by graphical views.

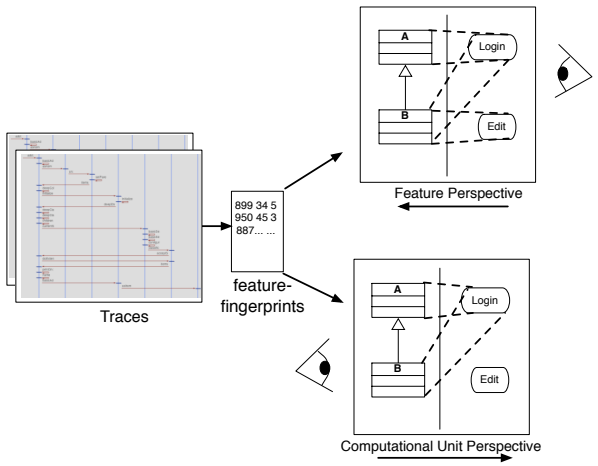

Figure 2. Our Two-Sided approach. We propose two views based on summarized traces.

3.1. Obtaining Reproducible Feature-Traces.

Our first step is to identify the features to include in our analysis. We do not require complete coverage of the system's functionality to obtain a feature model (as discussed in Section 6). Our approach allows the developer to focus on a subset of features of interest.

We build an analysis tool *TraceScraper* which uses method wrappers [2] to instrument the application code. The unit of observable behavior that we identify as being a feature should be defined as small as possible [6]. This limits the volume of data generated by the feature capture. Our tool simulates the user by scripting user actions and invokes each script separately. In this way it interacts with the application to trigger the feature behavior. The behavior is traced in a controlled environment. This means that no other system activity occurs at the time of capture. For each feature-trace captured, we start our case study application before triggering the individual feature. By doing so we ensure that our technique always yields the same feature-trace.

3.2. The Features Perspective

With the feature-perspective we focus on the characterization of the features of our feature model. We consider both the standalone feature, and the feature in relation to other features of the model. We obtain a characterization of features by applying feature-specific measurements to features-traces and their respective feature-fingerprints.

Feature-fingerprints. We extract 6 distinct *feature-fingerprints* from each feature-trace based on the following rules:

- C_f is the set of all classes which participate in a feature-trace.
- M_f is the set of all methods which participate in a feature-trace.
- CS_f is the set of all classes which are specific to a feature-trace.
- CC_f is the set of all classes which are common to more than one feature-trace.
- MS_f is the set of all methods which are specific to a feature-trace.
- MC_f is the set of all methods which are common to more than one feature-trace.

We group our measurements according to whether we are considering a standalone feature, or a feature in relation to other features.

Standalone feature measurements. These measurements enable us to characterize the features of an application in terms of their complexity. The complexity features is determined by the measurements shown in Table 1.

Metric	Description
NOM_f	# method calls in a feature-trace
NMR_f	# methods referenced in a feature-traces
NCR_f	# classes referenced in feature-trace

Table 1. Standalone Feature Measurements

Feature relationship measurements. We apply these measurements to obtain a characterization of a feature in relation to other features of our model. We measure the number of classes that are specific to a feature-trace (the cardinality of CS_f), and the number of classes that are referenced in other feature-traces (the cardinality of CC_f). These measurements are listed in Table 2.

Metric	Description
$NOCS_f$	# classes specific in feature-trace
$NOCC_f$	# classes common in other feature-traces
PCC_f	% classes common to other feature-traces
PCS_f	% classes specific to one feature-trace

Table 2. Feature Relationship Measurements

Based on the feature relationship measurements, we define a vocabulary to characterize feature-traces in terms of their participating classes:

- *Disjoint.* A disjoint feature-trace does not share classes with other feature-traces. ($NOCC_f$ = 0)
- *Completely related.* A completely related feature-trace shares all of its classes with other feature-traces. ($NOCS_f$ = 0)
- *Tightly related.* A tightly related feature-trace shares $>$ 50 % of its classes with other feature-traces. ($NOCC_f > (NCR_f \ / \ 2)$)
- *Loosely related.* A loosely related feature-trace shares $\leq$ 50% of its classes with other feature traces. ($NOCC_f \leq (NCR_f \ / \ 2)$)

3.3. The Computational Unit Perspective

We focus on characterizing the classes of an application based on their participation in the features of our model.

Computational Unit Measurements. We define two types of measurements: (1) we calculate the frequency with which a computational unit is referenced in a feature trace and (2) the number of feature-traces referencing the computational units.

The class related measurements are summarized in Table 3.

Metric	Description
$NOFRC$	# feature-traces referencing a class
$NORC_f$	# references to a class in a features-trace

Table 3. Computational unit measurements relating to feature-traces

The measurement *# references to a class in a features-trace* ($NORC_f$) counts all (including duplicate) references to classes in features. This measurement indicates how frequently a class is participating in a feature.

Computational Unit Characterization. We use the measurement *# feature-traces referencing a class* ($NOFRC$) for class characterization. As with feature characterization, we also define a vocabulary to characterize classes with respect to the feature model.

- *Infrastructural classes* participate in at least 50% of the features. These classes are providing common functionality, that is not specific to an individual feature ($NOFRC > (NCR_f \ / \ 2)$).
- *Single-feature classes* participate in only one feature-trace of the feature model ($NOFRC = 1$).
- *Group-feature classes* participate in a group feature-traces of the feature model. If group-feature classes are detected, this leads us to identify groups of related feature-traces ($1 < NOFRC \leq (NCR_f \ / \ 2)$).
- *Non-participating classes* do not participate in any feature-traces of the model ($NOFRC = 0$).

We use this characterization of classes to partition an application into *single-feature*, *group-feature* and *infrastructural* classes.

4. Feature and Computational Unit Perspective High-Level Views

We use a simple graph visualization to illustrate and convey the results of our two-sided approach to feature analysis. These visualizations represent both the feature perspective characterizations and the computational unit perspective. We generate all views using our *TraceScraper* tool.

4.1. Feature Perspective Views

We define two types of feature perspective views: *The Feature Characterization View* and *The Feature Class Correlation Views*.

Feature Characterization View. The purpose of this view is to represent a characterization of features in terms of the distribution of functionality in the participating computational units. We describe a *Feature Characterization View* based on the participating classes. We present a stacked bar chart, where each bar represents a distinct feature. We use color to represent the percentage of classes of the trace which are shared by other traces and those which are specific to one feature-trace. We calculate percentages for each feature using the measurements *% classes common to other feature-traces* (PCC_f) and *% classes specific to one feature-trace* (PCS_f). We represent each metric using a color on a bar chart as shown in Figure 3.

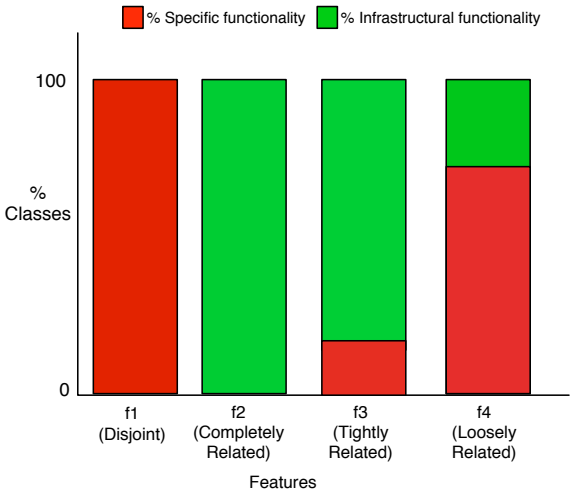

Figure 3. Feature Characterization View calculated using the metrics PCC_f and PCS_f.

If a feature is disjoint, it is displayed in red [1]. This means that with respect to the other features of the model, all its participating computational units are only referenced by this feature. If a feature is displayed in green it means that it is *completely related*. In other words all its computational units are referenced by other features of the model. (Figure 6 shows the results of applying this view to our SmallWiki case study.)

Feature Class Correlation Views. Correlation views reveal detailed information about which classes participate in a feature-trace. For applications with lots of classes, a grid view is difficult interpret due to its size. We address this problem by providing subviews which base the correlation between features and classes on subsets of the classes. We describe the following views:

- *Shared-Class-to-Feature-Correlation-View.* This view uses a grid to illustrate which classes of the feature-fingerprint CC_f participate in which features. Figure 5 shows the correlation of shared classes to features that we generated for our SmallWiki case study.
- *Exclusive-Class-to-Feature-Correlation-View.* Similarly this view displays a grid of the classes of the feature-fingerprint CS_f *(the set of specific classes)* participate in the features of our model. As this is similar in appearance to Figure 5, we do not include it in this paper.

1 On printed versions green will be displayed lighter than red

4.2. The Computational Unit Perspective View

The Class Characterization View represents graphically the types of classes of a system based on the characterization outlined in Section 3.3. We present a bar chart as show in Figure 4, where each color represents the number of classes that fall into each of the identified class characterizations.

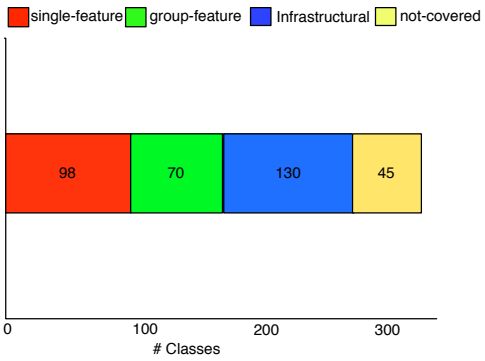

Figure 4. The Class Characterization View.

5. Case Studies

In this section we present the results of applying our feature-driven approach to two concrete case studies, For our experiments we chose two systems developed by our group: SmallWiki and BibOuter.

Table 4 gives an overview of the case studies.

Application	Size	# of Features Analyzed
SmallWiki	464 classes	11
BibOuter	126 classes	3

Table 4. Case Studies

5.1. Identifying Features for our Analysis.

SmallWiki [19] 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.

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

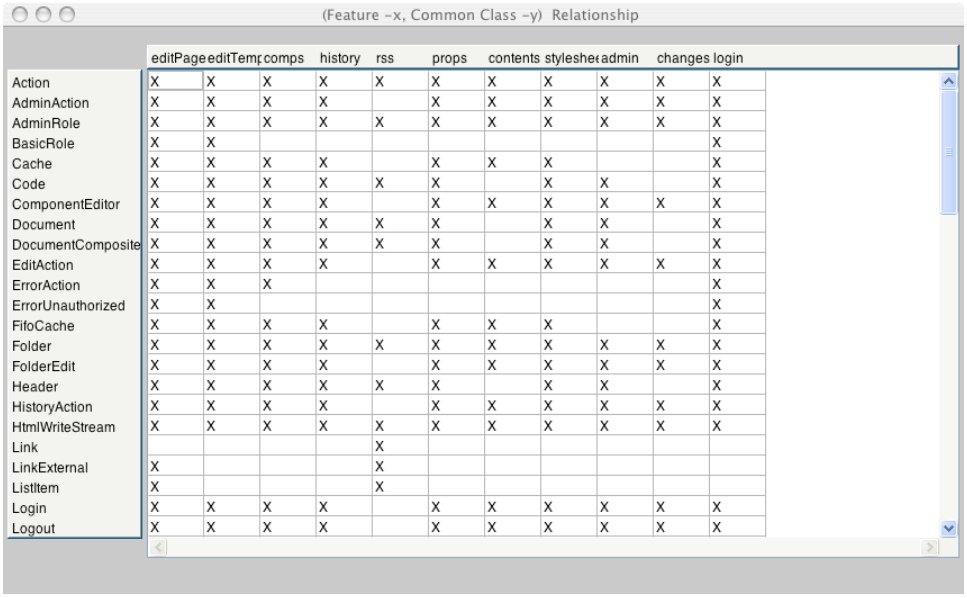

Figure 5. Shared-Class-To-Feature-Correlation-View. This shows the correlation between infrastructural classes and features.

of SmallWiki we associate features with the links and entry forms of the SmallWiki pages. We assume that each link on a page or button of an entry form triggers a distinct feature of the application.

To achieve a characterization of SmallWiki features, we selected 11 features for feature-driven analysis. Using or approach, we consider one possible path of execution for each feature. Table 5 lists the features of our case study and the results of applying standalone and relationship measurements.

For each feature we implemented scripts to simulate the user interactions and thus we trigger these features and capture a feature-trace in a controlled environment as described in Section 3. SmallWiki requires a user to login to the system before executing some of the features of the system. As a result our feature-traces initially contain the feature-trace method calls of the login feature. Therefore we filter out the login trace calls from our feature-traces so that our traces are not composed of other features.

BibOuter. This is a small application which is used to parse a library of references to generate documents in various formats, namely HTML, Latex and XML. We identify 3 features of the application by mapping each formatting capability of the application to a feature.

5.2. Case Study Results

Feature Perspective. We applied a feature perspective analysis to our SmallWiki feature model. Table 5 lists the features we selected for analysis and shows some of the measurements we obtained by applying *standalone feature measurements* and NOM_f *# method calls in a feature-trace* (NOM_f) and *# classes referenced in feature-trace* (NCR_f). These measurements give an indication of the size and complexity of the features.

We show a result of applying *# classes specific in feature-trace* ($NOCS_f$). This is a *feature relationship* metric. This provides the developer obtain with more precise information about the number of *single-feature* classes that participate in each feature.

Feature-trace	NOM_f	NCR_f	$NOCS_f$	**Characterization**
admin opt	2074	52	0	completely related
changes	2280	46	0	completely related
comps	7944	58	4	tightly related
contents	2615	43	1	tightly related
edit page	7251	67	11	tightly related
edit template	9377	66	10	tightly related
history	2402	54	2	tightly related
login	5273	60	0	completely related
properties	4320	54	1	tightly related
rss generate	2764	33	2	tightly related
stylesheet	6274	54	1	tightly related

Table 5. SmallWiki Feature Characterization.

The *# classes specific in feature-trace* ($NOCS_f$) measurement indicates how many classes are exclusive to one feature-trace. The feature-set CS_f contains ref-

erences to the actual classes. For example, for the feature-trace **rss generate** we discover that there are two single-feature classes, namely **RSSChangeFeed** and **Link**. Similarly, we for the feature-trace **edit template** we detect 10 single-feature classes.

Figure 6 shows our *Feature Characterization View* of the SmallWiki case study. We see these features are either *completely related* or *tightly related*.

Correlation Views. For space reasons, we have included only one correlation view, namely the *Shared-Class-To-Feature-Correlation-View* of our SmallWiki case study as shown in Figure 5,. We see which of the features are using the shared computational units. For example, the class **Action** is participating in all SmallWiki features.

Feature-trace	NOM_f	NCR_f	$NOCS_f$	Characterization
latex opt	11250	13	1	tightly related
html	11339	17	0	completely related
xml	11340	19	2	tightly related

Table 6. BibOuter Feature Characterization.

BibOuter Feature Characterization. Table 6 shows the results of our feature characterization of the *BibOuter* features. From the characterizations we see that these three features share most of their functionality. For Latex generation there is one single-feature class and for XML generation, there are 2 single-feature classes.

The Class Perspectives We apply our technique to obtain a characterization of classes and use our *TraceScraper* tool to generate class perspective views.

We calculate the measurements *# feature-traces referencing a class* $NOFRC$ and *# references to a class in a features-trace* ($NORC_f$) to discover the number of *infrastructural*, *single-feature* and *group-feature* classes with respect to the feature-model. Figure 7 shows the *Class Characterization View* obtained for SmallWiki. We summarize the results obtained for class characterization of both our case studies in Table 7.

Characterization	SmallWiki	BibOuter
# single-feature	34	4
# group-feature	25	0
# infrastructural	67	19
# not-participating	83	40

Table 7. Class Characterizations

6. Discussion and Evaluation

6.1. Results of the Experiment

The case studies show that by applying our *two-sided* approach, we obtain useful information about the way the features of a system are implemented and about how the classes are providing functionality to the features.

Our simple graphical visualizations of feature characteristics reveal how the functionality of the application is distributed among the features. Our *Feature Characterization View* shows the types of features in an application based on the characterizations we defined in Section 3.1. This view shows how features are related in terms of computational units. For both case studies, we obtained only *tightly related* and *completely related* features. This indicates that the features share most or all of their computational units to realize their functionality. We did not discover any *disjoint* or *loosely related* features. The existence of a *disjoint* feature would mean that the behavior of that feature is completely unique to it. In the case of SmallWiki, we do not expect to discover *disjoint* features. We expect the functionality that handles user interaction with the web server to be implemented once and shared by each feature.

Similarly in the case of the *BibOuter* application, we evaluate three features that generate documents. The only variation in these features is that they generate different formats. Therefore we expect that these features share large parts of their functionality and that the format specific parts are restricted to a small number of classes. This expectation corresponds with the results of our analysis.

Our approach promotes an understanding of how existing features are implemented. Using the high-level feature and computational unit views, the developer determines roles that classes and methods play in the features. This makes it easier for a developer to design and implement a new feature and reuse existing functionality. Moreover, the risk of code duplication is minimized, as the developer understands how existing classes and methods contribute to the dynamic behavior of the application. For example, if a developer has to add a new feature to SmallWiki, he can use his understanding of existing feature implementation to determine which parts of the code he can reuse. Moreover, he is more likely to adopt existing conventions of the application he is modifying to realize the feature.

6.2. Contributions

The main contributions of our approach are:

- We focus the description of our approach in the context of object-oriented applications.
- We propose to combine our feature model with a language independent meta-model to relate class, method

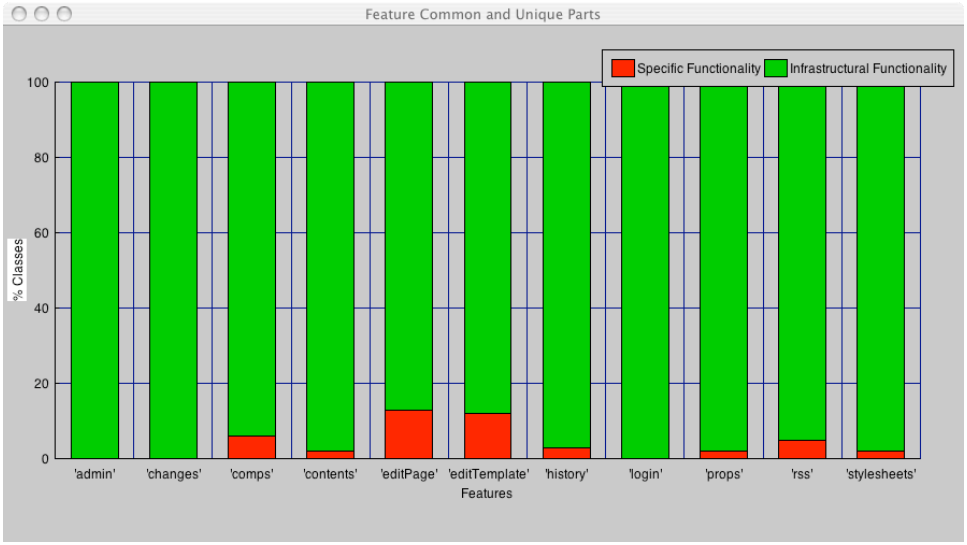

Figure 6. Feature-Characterization-View for SmallWiki.

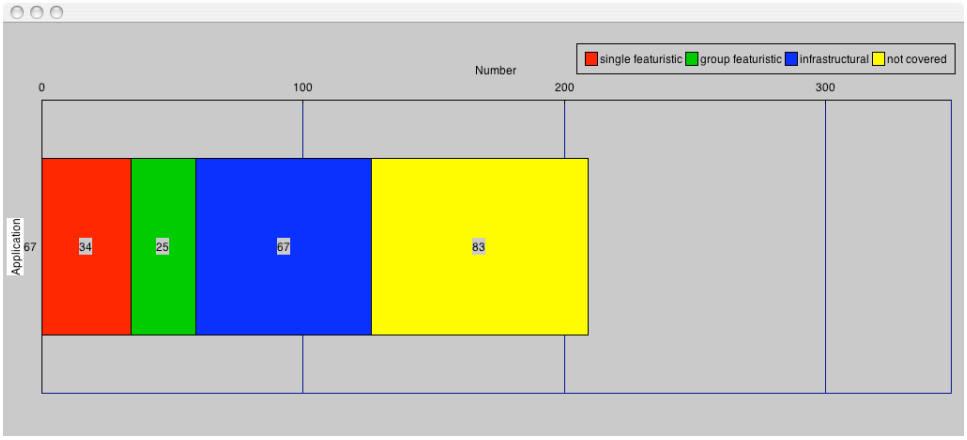

Figure 7. The Class Characterization View for SmallWiki.

and feature entities. This makes our approach generally applicable to systems written in different object-oriented programming languages.

- The emphasis of our approach is *ease-of-use* and compactness of dynamic information. We condense the information in the feature-traces to focus on key information to achieve the characterizations.
- Our approach proposes a *two-sided* approach: the feature perspective and computational unit perspective.

We have shown how our feature-oriented approach establishes the mapping and relationships between conceptual views of the system in terms of requirements and user views.

The results of our approach depend on the type of chosen case study application. The reuse of functionality for multiple features is more feasible in *infrastructural* classes than in *single-feature* classes.

6.3. Constraints and Limitations

Simplicity: One of arguments against dynamic analysis based approaches is that it is difficult to achieve completeness, as all possible paths of execution are not exercised [24]. We argue that for the purpose of feature location, completeness is not necessary. By triggering a feature in a controlled environment, we collect sufficient data to establish relationships between features and computational

units.

Mapping traces to features: In our approach we define a one-to-one mapping between an execution trace and a feature. Other approaches [4] combine traces from multiple scenario executions to obtain a feature mapping. At present, our approach does not consider exhaustive execution of individual features. Our experiments show that one path of execution is sufficient to establish the required mapping. However, our technique for collecting features is easily extensible and by adapting our model, we could define a one-to-many relationship between features and feature-traces.

Size of applications: The size of the system under analysis is an important factor when considering our approach. Our case studies are relatively small applications. For larger systems, it would be useful to apply an *iterative* approach to locate features by initially considering a coarser granularity of computational unit *e.g.,* a package or subsystem. Further iterations then focus on classes and methods.

Coverage: In general, coverage of the application by the feature model affects the characterizations of the features. If the model contains only one feature, the feature can only be considered in isolation. Only by executing all features of an application do we achieve a stable characterization of features and computational units. For example, a class is characterized as *single-feature*, if participates in one feature. However, this characterization may change as soon as the feature-model includes another feature which references this class. Then the characterization changes to *group-feature*.

User-triggerable features: Our approach is limited to user-triggerable features. However, it is easily extensible to encompass features such as housekeeping tasks of a system.

7. Related Work

Many researchers have identified the potential of feature-centric approaches in software engineering and in particular as a basis for reverse-engineering [21, 6, 12, 5, 14, 8]. The basis of our work is directly related to the field of dynamic analysis [1, 24, **?**], requirement and aspect mining extraction in legacy systems, and user-driven approaches [10, 9, 15]. A major research focus is in visualization of dynamic information [16, 3, 13, 17, 11, 18, 22]. These fields of research represent the groundwork on which our research is based.

The research efforts that are closely related to our approach for identifying features are Wilde and Scully [23], Eisenbarth et al. [4], Mehta and Heinemann [14], Fischer et al. [7] and Wong et al. [25]. All of them have developed techniques based on execution traces. We describe briefly the approaches that are most closely related to ours.

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

Eisenbarth et al. [4] describe a methodology which combines dynamic, static and concept analysis. They collect execution traces and categorize the methods according to their degree of specificity to a given feature. The analysis automatically produces a set of concepts which are presented in a lattice. Using this technique they identify general and specific parts of the code.

Wong et al. [25] propose three different metrics for measuring the binding of features to components or program code. They quantitatively capture the disparity between a program component and a feature, the concentration of a feature in a program component, and the dedication of program component to a feature.

Our approach complements these approaches. In contrast to the above approaches [4, 23], our main focus is applying feature-driven analysis to object-oriented applications. We also emphasize the *ease-of-use* of our technique. Moreover, we adopt a *two-sided* approach in that we consider both feature and class perspectives in our analysis and characterization. We discover relationships between features and classes. We define a finer characterization of features based on simple measurements. We also define measurements to characterize classes in the context of features.

8. Conclusions

Software developers handle change requests and bug reports that are expressed in a language that reflects a feature perspective of a system. We cope with this problem by considering a feature a first class entity of analysis. We abstract a feature model of feature-traces and feature-fingerprints to correlate classes and features. We adopt a *two-sided* approach to obtain a characterization of features and computational units. We outline a feature perspective, and a computational unit perspective. We define measurements for both perspectives. Based on our measurements, we defined a vocabulary to characterize features as *disjoint*, *loosely related*, *tightly related* and *completely related*.

In our computational unit perspective, we defined a vocabulary to characterize the classes with respect to features. We characterize the classes as *single-feature*, *group-feature* or *infrastructural*.

We used a simple graphical visualizations to convey the results of our two-sided approach. We display both characterization and correlation views from both perspectives.

8.1. Future Work

In the future work, we plan to extend the scope of our feature approach to consider applying compression and filtering techniques to the dynamic information. In addition, we plan to extend our feature representation within the feature model to include multiple paths of execution of a feature. We expect that as a result we achieve a higher coverage of classes and methods and increase the accuracy of our approach.

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

References

[1] T. Ball. The concept of dynamic analysis. In *Proceedings of ESEC/FSE '99*, number 1687 in LNCS, pages 216–234, 1999.

[2] J. Brant, B. Foote, R. Johnson, and D. Roberts. Wrappers to the Rescue. In *Proceedings ECOOP '98*, volume 1445 of *LNCS*, pages 396–417. Springer-Verlag, 1998.

[3] S. Ducasse, M. Lanza, and R. Bertuli. High-level polymetric views of condensed run-time information. In *Proceedings of CSMR 2004 (Conference on Software Maintenance and Reengineering)*, pages 309 – 318, 2004.

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

[5] 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.

[6] E.Pulvermüller, A. Speck, J.O.Coplien, M. D'Hondt, and W.DeMeuter. Position paper: Feature interaction in composed systems. In *Proceedings of the European Conference on Object-Oriented Programming, ECOOP 2001*, pages 1–6, 2001.

[7] 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.

[8] I. Hsi and C. Potts. Ontological excavation: Unearthing the core concepts of an application. In *Proceedings of the 2003 10th Working Conference on Reverse Engineering*, pages 345–352, Nov. 2003.

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

[10] 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.

[11] 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.

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

[13] L. M. A. G. J. Martin. Dynamic component program visualization. In *Proceedings of ninth Working Conference on Reverse Engineering*, 2002.

[14] 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.

[15] A. Memon, I. Banerjee, and A. Nagarajan. Gui ripping: Reverse engineering of graphical user interfaces for testing. In *Proceedings of the 2003 10 Working Conference on Reverse Engineering (WCRE'03)*, pages 260–269, Victoria, Canada, 2003.

[16] M. Pacione, M. Roper, and M. Wood. A Comparative Evaluation of Dynamic Visualization Tools. In *Proceedings of WCRE '03*, pages 80–89. IEEE Computer Society, Nov. 2003.

[17] W. D. Pauw, R. Helm, D. Kimelman, and J. Vlissides. Visualizing the behavior of object-oriented systems. In *Proceedings OOPSLA '93*, pages 326–337, Oct. 1993.

[18] S. P. Reiss, editor. *Visualizing Java in Action*, May 2003.

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

[20] T. Richner and S. Ducasse. Using dynamic information for the iterative recovery of collaborations and roles. In *Proceedings of ICSM '2002 (International Conference on Software Maintenance)*, Oct. 2002.

[21] 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.

[22] R. J. Walker, G. C. Murphy, B. Freeman-Benson, D. Wright, D. Swanson, and J. Isaak. Visualizing dynamic software system information through high-level models. In *Proceedings OOPSLA '98*, pages 271–283. ACM, Oct. 1998.

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

[24] J. Winstead and D. Evans. Towards differential program analysis. In *Proceedings WODA 2003 the Workshop on Dynamic Analysis*, pages 37–40, Portland, Oregon, May 2003.

[25] 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.

[26] A. Zaidman and S. Demeyer. Managing trace data volume through a heuristical clustering process based on event execution frequency. In *Proceedings of the 8th European Conference on Software Maintenance and Reengineering*, pages 329–338, Mar. 2004.