Visualizing Feature Interaction in 3-D

Orla Greevy[1], Michele Lanza[2] and Christoph Wysseier[1]
[1] Software Composition Group - University of Berne - Switzerland
[2] Faculty of Informatics, University of Lugano - Switzerland

Abstract

Without a clear understanding of how features of a software system are implemented, a maintenance change in one part of the code may risk adversely affecting other features. Feature implementation and relationships between features are not explicit in the code. To address this problem, we propose an interactive 3D visualization technique based on a combination of static and dynamic analysis which enables the software developer to step through visual representations of execution traces. We visualize dynamic behaviors of execution traces in terms of object creations and interactions and represent this in the context of a static class-hierarchy view of a system. We describe how we apply our approach to a case study to visualize and identify common parts of the code that are active during feature execution.

1 Introduction

Many reverse engineering approaches to software analysis focus on static source code entities of a system, such as classes and methods [3, 13, 15]. A static perspective considers only the structure and implementation details of a system. Using static analysis alone we are unable to easily determine the roles of software entities play in the features of a system and how these features interact. Without explicit relationships between features and the entities that implement their functionality, it is difficult for software developers to determine if their maintenance changes cause undesirable side effects in other parts of the system.

Several works have shown that exercising the features of a system is a reliable means of correlating features and code [7, 20]. In previous works [9, 10], we described a feature-driven approach based on dynamic analysis, in which we extract execution traces to achieve an explicit mapping between features and software entities like classes and methods. Our definition of a feature is a unit of behavior of a system.

In this paper we concentrate on describing and understanding relationships between features. We define feature interaction relationships in terms of classes they share.

Dynamic analysis implies a vast amount of information, which makes interpretation difficult. We introduce an approach which we refer to as *controlled dynamic visualization*. Using our technique the software developer can selectively step through execution traces and drive a visualization engine which represents the events of the traces in a 3-D visualization. By selecting execution traces collected by executing the feature or features that require maintenance, we limit the amount of information to be processed to the relevant parts of the code.

Furthermore, our *controlled dynamic visualizations* are useful to observe which parts of the system are being used by more than one feature of the system. This information is useful to determine which other features may be affected by a change to the code during a maintenance activity.

Structure of the paper. We start by introducing the terminology we use to describe and interpret dynamic information. In Section 3 we then outline the visualization mechanisms of our technique and the trace generation process (Section 4). We discuss the advantages of our 3-D approach. In Section 5 we report on the case study conducted using our approach. Subsequently, in Section 6 we discuss our results. We summarize related work in Section 7. Section 8 outlines our conclusions and future work.

2 Feature Terminology

We establish the relationship between the features and software entites by exercising the features or *usage scenarios* and capturing their execution traces, which we refer to as *feature-traces*. A *feature-trace* is a sequence of runtime events (*e.g.,* object creation/deletion, method invocation) that describes the dynamic behavior of a feature.

We define the measurements $NOFRC$ to compute the # feature-traces that reference a class and FC to compute a characterization of a class in terms of how many features reference it and how many features are currently modeled.

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

model.

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

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

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

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

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

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

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

A *feature-fingerprint* (FP) is a set of sets of characterized software entities. In this paper we focus on feature-fingerprints of classes to determine feature-interaction in terms of how features share classes:

$FP_i = \{\{NC(classes)\}, \{SF(classes)\}, \{GF(classes)\}, \{I(classes)\}\}$

Figure 1 shows the relationships between features and classes.

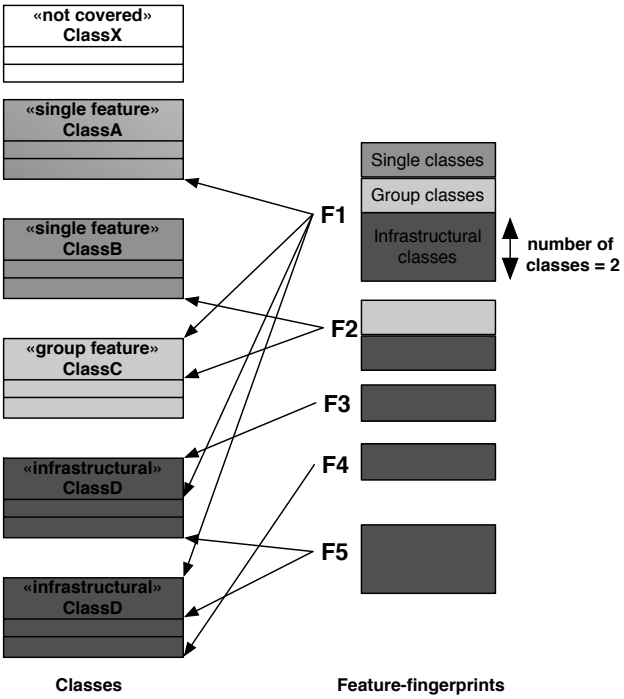

Figure 1. Feature-Fingerprints and Classes Relationships

3 Visualization of Dynamic Behavior

We use visualization to understand the dynamic behavior of features. The visualization we propose is based on polymetric views as described by [16] but extended to 3D. The visualizations we propose are (1) a feature interaction view and (2) a dynamic feature-trace view.

Feature Interaction View. In this view we display the class hierarchies of a system. The classes are displayed as 3-D nodes and the inheritance relationships are displayed as connecting edges. We map the FC *(a class characterization in terms of feature participation*) measurement to the color of the node. In previous work [10] we described simple views that show the distribution of class characterizations in terms of features based on Figure 1. The *feature interaction view* is more detailed, as it shows the actual classes that participate in features in the context of a class hierarchy. In Figure 5 we see a feature interaction view we generated for a small example system. The colors of the nodes indicate the feature characterization of the classes *(FC)*. We see from this view, where the *single-feature* (medium gray), *group-feature* (dark gray) and *inftrastructural* classes (black nodes) are located in the system. The light gray classes are *not-covered.* This view presents a summary of the relationships between features terms of the classes that they share. We query the nodes to obtain the details of the represented classes.

Dynamic Feature-trace View. This view is a representation of a system behavior during the execution of a feature in terms of classes, object-instantiations and message sends. In Figure 3 we see a schematic display of such a view. It is a 3D visualization which displays the static structure of the system on a plane ”floating” above the ground (the gray boxes are the classes connected by inheritance edges). When the trace is interpreted each instantiation of a class (the creation of an object) generates a blue box (like a ”floor” in a building) above the ground level. The more blue boxes are above a class, the more instances of this class have been created. The currently active objects are displayed in green, *i.e.*, each time an object sends a message to another object, a message edge is draw between the two object boxes.

The tool which implements these dynamic feature-trace views is called TraceCrawler, an extension of the CodeCrawler tool [14]. TraceCrawler permits the user to step through the traces and at each point in time of the trace to see the current state of the trace and also to backward and forward within the trace: In Figure 4 we see a small scenario generated from a simple test system, *i.e.*, three different points in time of the same trace. On the lefthand side

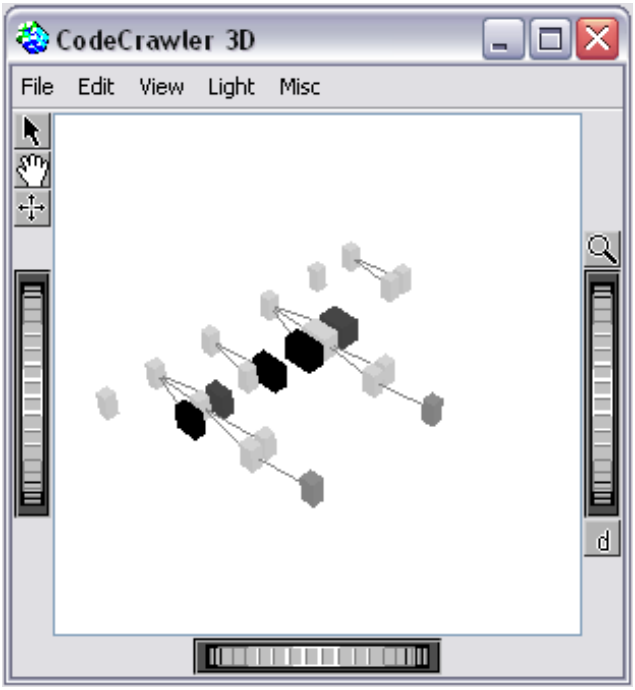

Figure 2. Example Feature-Interaction View.

we see the 3-D visualization and on the righthand side we see the corresponding position in the trace that is currently being processed. This allows for an in-depth and detailed inspection of the trace. The visualizations are interactive (the user can click on the nodes and edges and inspect them) and navigable (the user can ”fly through” the 3D space and position the point of view wherever he wants. This allows to observe certain parts of the visualization closely.

4 Extracting and Visualizing Feature Traces

We outline how we apply our technique to obtain and visualize feature-traces from a software system.

1. We apply static analysis and abstract a static model of the source code entities of the application (*e.g.*, classes, methods and hierarchy relationships).

2. We identify the features to include in our analysis. We do not require complete coverage of the system’s functionality (as discussed in Section 6). Our approach allows the developer to focus on a subset of features of interest.

3. Our dynamic analysis tool *TraceScraper* builds on method wrappers [2] to instrument the code. Our traces are extracted as a tree of method invocation calls. The unit of observable behavior that we identify as being a feature should be defined as small as possible [8]. This limits the volume of data generated by the feature capture. We implement a test case that 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. We abstract feature-traces for a set of features using dynamic analysis. To achieve this we instrument the code and execute the features.

4. We model the feature-traces as first class entities and incorporate them into the static model of the source code. By doing so we establish the relationships between the methods calls of the feature traces and the static model class and method entities. We compute the feature characterizations of the classes as described in Section 2

5. Our *TraceCrawler* tool generates a static view of the application in terms of class hierarchies. It then processes the feature-traces and converts the events into the previously presented visual representations.

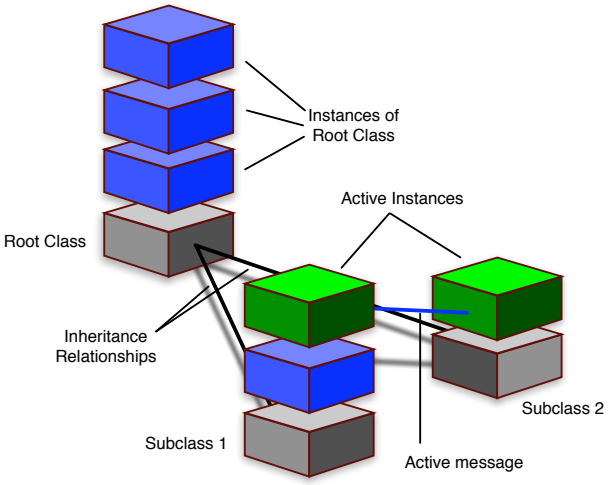

Figure 3. The Dynamic Feature-trace View

5 Validation: Moose Case Study

In this section we present the results of applying our approach to the *Moose* case study, a language-independent environment for reengineering object-oriented systems [5]. It provides features for storing models derived from souce code, navigating, querying and applying metrics to the model entites. The version we use for this experiment (3.0.25) consists of 782 classes.

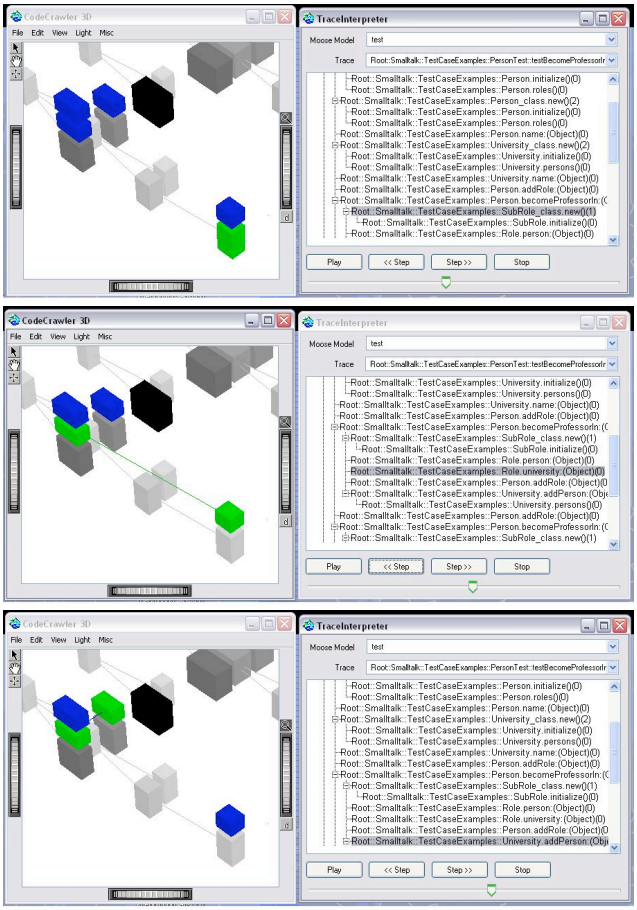

Figure 4. TraceCrawler allows the user to step through the traces.

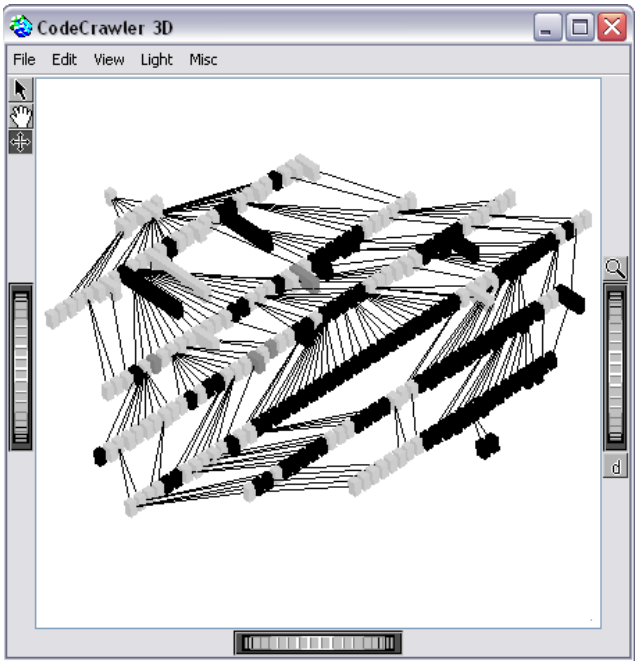

Figure 5. The Feature-Interaction View for Moose.

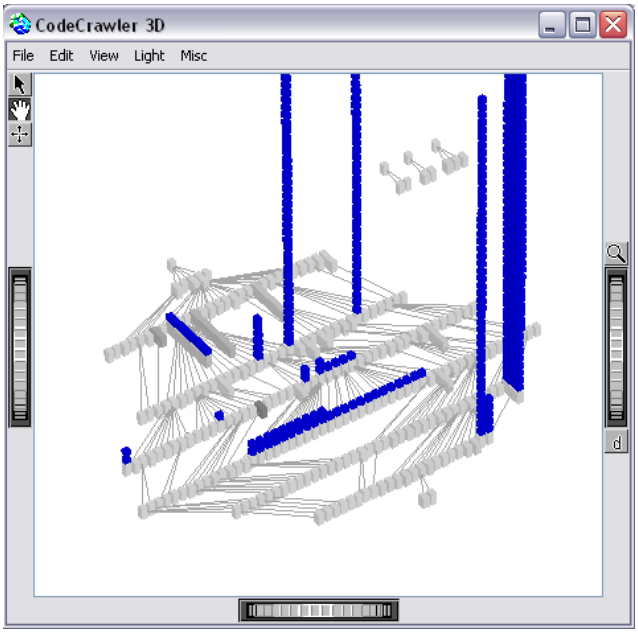

Figure 6. A Visualization of the Feature-Trace 'Load a Model From Source Code'.

For our feature analysis of *Moose* we focus on 3 features, which represent typical user interactions with the application (the numbers are the associated class characterization values):
1-Loading a model from source code (SF=0, GF=0, I=156)
2-Loading a model from a file (SF = 0, GF = 0, I = 156)
3-Saving a model to a file (SF = 24, GF = 0, I = 175)

In Figure 5 we show a part of the feature interaction view we generated for Moose based on the a class characterization in terms of feature participation (FC) measurment described in Section 2.

From the point of view of feature interaction, we are interested in the classes that are shared by all 3 features we traced (*i.e.*, the classes that are characterized as *infrastructural*). These are shown in black in Figure 5. We query the view to obtain names of the classes. For example we obtain that the classes *PropertyOperatorFactory* and *FAMIXClass* are characterized as infrastructural.

Figure 6 is a visualization that results from stepping though the feature-trace generated for the feature 'Load a Model from Source Code'. As Moose loads a model from source code, it instantiates the entities of the model it abstracts from the source code. This visualization shows which classes are active during the execution of the feature. We see instantiations and message sends between objects. By inspecting the class nodes of this view we see that the Moose classes *EntityType, MOFExtendedClass, MOFPackage MOFImport* are instantiated a large number of times. We validated this behavior makes sense in the context of this feature with the developers of Moose.

6 Discussion

The large volume of information and complexity of dynamic information makes it hard to infer higher level of information about the system. Our approach is based on visualizations to reveal key semantic information about the relationships features of a system.

In our Moose case study, we chose three conceptually related features - the functionality that deals with abstracting and modelling entities from source code. Our *feature interaction view* reveals, which classes participate in all the three features, and which classes are specific to just one feature. We use developer knowledge to validate our results.

Our *dynamic feature interaction views* provide us with an insight into how the classes and methods behave at runtime. We see which parts of the system are stressed during the execution of a feature.

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

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

Garbage Collection. Currently our traces are generated by instrumenting the code of the application. We do not instrument low level calls. Therefore for languages rely on garbage collection, we currently have no event in the trace.

Language Independence Obtaining the traces from the running application requires code instrumentation. The means of instrumenting the application is language dependent. We abstract a feature model of the traces which is the same for every language. Our analysis is performed on the feature model.

7 Related Work

Many researchers have identified the potential of feature-centric approaches in software engineering and in particular as a basis for reverse-engineering [7, 19, 20]. Our work is directly related to the field of dynamic analysis [1, 11, 21] and user-driven approaches [12] . Our main focus with this work is to use 3-D visualizations of feature-traces to analyze the dynamic behavior of features and how they interact in terms of the parts of the code that they share.

Feature location techniques such as *Software Reconnaissence* described by Wilde and Scully [19] , and that of Eisenbarth et al. [7] are closely related to our feature location approach. In contrast, our main focus is applying feature-driven analysis to object-oriented applications.

Salah and Mancoridis [18] describe a hierarchy of dynamic views that is constructed using tools that analyze program execution traces and the environment used to construct these views.

De Pauw *et al.* present two visualization techniques. In their tool Jinsight, they focused on interaction diagrams [17]. Thus all interactions between objects are visualized.

Ducasse *et al.* [6] present visualizations of dynamic behavior of a system based on metrics calculated for the number of invocations, number of object creations and the number of used classes/methods.

The main focus of our approach is to visualize of the dynamic behavior of features and how they interact. Our approach complements approaches [17] in that we by allowing the developer to interact with the visualization and and control the display of events in a feature trace. Our visualizations add semantic information to the software entities by showing how they participate in features. We use this semantic information to reason about the functional roles of software entities of a system in terms its features.

8 Conclusions and Future Work

Reverse engineering approaches that focus only on the implementation details and static structure of a system overlook its dynamic behavior at runtime. Our feature perspective establishes the semantic purpose of the individual software entities of a system. Our visualizations combine both a static view of a system in terms of class hierarchies with a dynamic information obtained from feature-traces. We cope with large amounts of dynamic data as the visualization tool steps sequentially through the trace and converts each event of the trace into a visual representation.

Our goal is to analyze the functional roles of classes from a feature perspective and to use visualization of dynamic behavior to gain an understanding of how software entities participate in features.

We applied our approach to the Moose case study and showed how our *feature interaction view* is useful for interpreting the functional roles of classes from a feature perspective. This view reveals valuable information about the interaction between features in terms of classes they share. This information supports the maintainer of a system by presenting visually which parts of the system are affected by changes to features.

Our *dynamic feature-trace* views provide us with in-depth views at the level of object interaction.

In the future, we would like to consider applying filtering mechanisms to the traces to improve the scalability of the approach and to reduce the volume of data without loss of information about the relationships between features and code. We intend to consider variations in how we present the dynamic behavior in the context of the static entities in the system. Furthermore, we plan to extend our feature representation within to include multiple paths of execution of a feature. We expect that as a result we achieve a higher coverage of classes and methods and increase the accuracy of our approach.

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

References

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

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

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

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

[6] 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.

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

[8] 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.

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

[10] O. Greevy, S. Ducasse, and T. Gîrba. Analyzing feature traces to incorporate the semantics of change in software evolution analysis. In *Proceedings of ICSM 2005 (21th International Conference on Software Maintenance)*, Sept. 2005. to appear.

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

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

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

[14] M. Lanza. Codecrawler — lessons learned in building a software visualization tool. In *Proceedings of CSMR 2003*, pages 409–418. IEEE Press, 2003.

[15] M. Lanza and S. Ducasse. A Categorization of Classes based on the Visualization of their Internal Structure: the Class Blueprint. In *Proceedings of OOPSLA '01 (International Conference on Object-Oriented Programming Systems, Languages and Applications)*, pages 300–311. ACM Press, 2001.

[16] M. Lanza and S. Ducasse. Polymetric views — a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, 29(9):782–795, Sept. 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] M. Salah and S. Mancoridis. A hierarchy of dynamic software views: from object-interactions to feature-interacions. In *Proceedings of The 20th IEEE International Conference on Software Maintenance (ICSM 2004)*, 2004.

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

[20] 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.

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