Visualizing Live Software Systems in 3D

Orla Greevy[1], Michele Lanza[2] and Christoph Wysseier[3]

[1] Software Composition Group - University of Berne, Switzerland

[2] Faculty of Informatics - University of Lugano, Switzerland

[3] netstyle.ch GmbH - Bern, Switzerland

Abstract

The analysis of the runtime behavior of a software system yields vast amounts of information, making accurate interpretations difficult. Filtering or compression techniques are often applied to reduce the volume of data without loss of key information vital for a specific analysis goal. Alternatively, visualization is generally accepted as a means of effectively representing large amounts of data. The challenge lies in creating effective and expressive visual representations that not only allows for a global picture, but also enables us to inspect the details of the large data sets. We define the focus of our analysis to be the runtime behavior of features. Static structural visualizations of a system are typically represented in two dimensions. We exploit a third dimension to visually represent the dynamic information, namely object instantiations and message sends. We introduce a novel 3D visualization technique that supports animation of feature behavior and integrates zooming, panning, rotating and on-demand details. As proof of concept, we apply our visualization technique to feature execution traces of an example system.

Keywords: 3D visualization, reverse engineering, dynamic analysis, feature traces

1 Introduction

Reverse engineering usually implies the abstraction of high level views that represent different aspects of a software system. Visual representations of software are widely accepted as comprehension aids [Fowler 2003], [Stasko et al. 1998]. The Unified Modeling Language for example is one of the most widely used visual language for object-oriented software development [Fowler 2003].

Due to the complexity of software, visualizations should be designed with a specific task in mind [Maletic et al. 2002]. UML provides distinct diagrams for static and dynamic information. The static visualizations of a system, (*e.g.,* UML class diagrams), describe the static structure of a system in terms of software entities and their relationships. To represent dynamic aspects of a system, UML provides object models and sequence diagrams. Their main problem is that they do not scale beyond a certain number of events because of physical constraints, which can however be partially overcome with the help of interactive exploration techniques [Sharp and Rountev 2005].

The context of our research [Greevy and Ducasse 2005], is feature-centric reverse engineering, (we exercise a systems' features on an instrumented software system and capture traces of their runtime behavior). We focus on how the static source artifacts contribute to the dynamic runtime behavior of features. A *feature trace* is a record of the steps a program takes during the execution of a feature. For this paper we adopt the definition of a feature as a user-triggerable functionality of a software system [Eisenbarth et al. 2003].

As a basis of our approach we combine static and dynamic information to focus on how the static source artifacts contribute to the dynamic runtime behavior of features. Therefore we need a combined visualization that captures both static structural information and dynamic runtime data.

The key question we address in this paper is: how do we visually represent and manipulate the large information space of feature traces in a way that is intuitive and useful for the reverse engineer?

As proof of concept of our 3D visualization technique we apply it to two software systems. We motivate our visual analysis by addressing the following feature-centric reverse engineering questions:

- *Which parts of the code (classes and objects) are active during the execution of a feature?* Our visualizations show which classes are instantiated and how they collaborate during the execution of a feature.
- *What are the patterns of activity that are common to features and which activities are specific to one feature?* Similar patterns of activity of feature behavior, (*i.e.,* similar sequences of communications between objects), may give insights into the architectural structure of a system.

Structure of the paper. In the next section we briefly describe our approach to feature analysis. We outline the purpose and scope of our visualization technique in Section 3. We describe the 3D visualizations and the capabilities of our technique to manipulate a large information space. In Section 4 we provide a proof of concept for our approach by applying it on two systems. We then evaluate our technique and discuss the advantages and limitations of our approach in section Section 5. We present our implementation in Section 6. In Section 7 we provide an overview of related work and compare our visualization techniques with other visualization-based approaches. Finally present our future work in this area.

2 Combined Static and Dynamic Analysis

Two main but distinct approaches to reverse engineering dominated the research field [Chikofsky and II 1990], namely dynamic analysis approaches and static analysis approaches. In recent years the synergies and dualities of these approaches have been recognized [Ernst 2003]. Stroulia and Systa [Stroulia and Systa 2002] argue that static analysis approaches, though valuable, are incomplete and do not meet reverse engineering goals of todays object-oriented systems. Object-oriented systems are difficult to understand by browsing the source code due to language features such as inheritance, dynamic binding and polymorphism. The behavior of the system can only be completely determined at runtime. The dynamics of the program in terms of object interactions, associations and collaborations enhance system comprehension [Jerding et al. 1997].

The basis of our feature analysis approach (see Figure 1) [Greevy et al. 2005a] is to combine both static and dynamic information. In this paper we describe how we visually represent the dynamic behavior of features in the context of a static structural view of the system. We apply static analysis to the source code of a system to extract a static model of the source code entities. We then apply dynamic analysis to obtain traces of feature executions. To

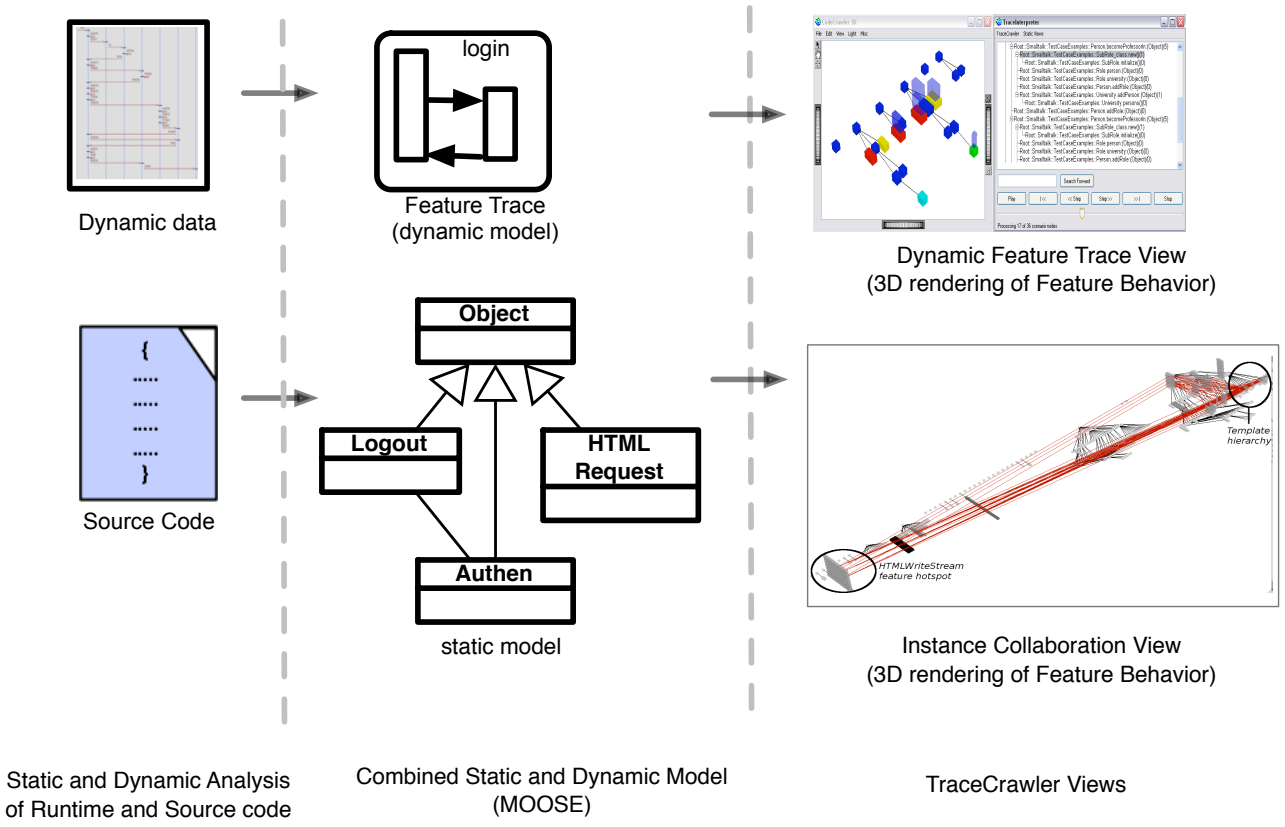

Figure 1: Our Approach to Rendering Feature Behavior in 3D

achieve this, we instrument a system, exercise a set of features and abstract and represent the execution data of each individual feature as a tree of method calls or *feature trace*. We resolve the methods and classes involved in the events of the trace in the context of the static model.

Interpretation of a *feature trace* is difficult due to its sheer size. Many dynamic analysis approaches apply filtering or compression to the dynamic data to abstract high level views and reason about the information [Greevy and Ducasse 2005; Hamou-Lhadj et al. 2005] by defining measurements to summarize the data. A purely metrics-based approach provides us with a quantitative impression of which objects are active during the execution of a feature-trace. However, we lose qualitative aspects like the notion of time, as the quantitative values do not provide information about when an object is active during the trace. Thus, trace summarization techniques may eliminate details that provide valuable insights into feature behavior of object-oriented systems.

We exploit visualization as a means of representing the dynamic runtime information of features. Our visualization technique renders both the static structure of the software system as a class hierarchy, and the dynamic behavior of a feature as towers of communicating instances. We describe our visualization technique in more detail in the next section.

3 3D Visualization of Dynamic Behavior

The visualizations we propose for feature-traces are based on polymetric views [Lanza and Ducasse 2003] extended to 3D[Greevy et al. 2005b]. We describe the purpose and goals of our visualizations in terms of the five dimensions of interest of software visualization defined by Maletic *et al.* [Maletic et al. 2002].

1. *Task.* Why is the visualization needed? The *task* of our visualizations is to support the understanding of the dynamic behavior of features. We aim to support feature-centric reverse engineering [Greevy and Ducasse 2005] by extracting high level views of feature behavior and visualizing this information in the context of the static structural information of a system.

2. *Audience.* Who will use the visualization? Our target audience is the software developer/reverse engineer who is faced with the task of understanding a complex software system. Our approach aims to support system comprehension through visualization of features. Bug reports and modification requests are usually expressed in language that reflects the features of a system. We argue that by understanding how the features are implemented in the system, we support maintenance activities. Our approach could also be extended to support debugging tasks, and we plan to pursue this path in our future work.

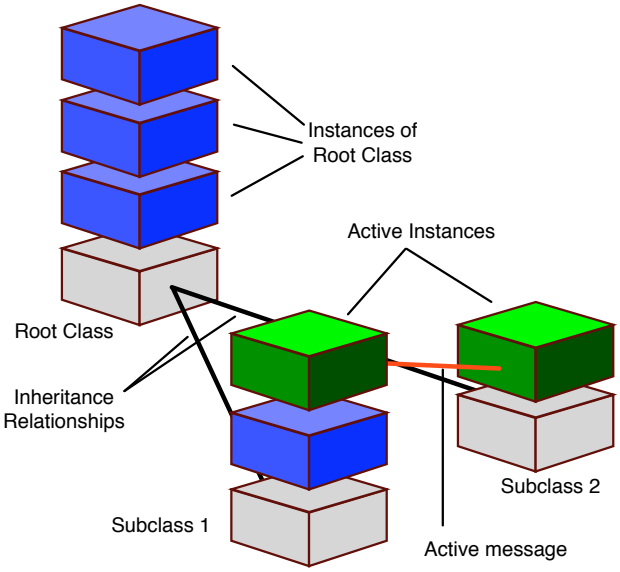

Figure 2: A schematic view of our 3D visualizations

3. *Target.* What low level aspects are visualized? We represent data extracted from both static and dynamic analysis of a system. We represent dynamic runtime behavior of features (*i.e.,* object creations and message sends) and a static structural view of the system (*i.e.,* classes and inheritance relationships).

4. *Representation.* What form of representation that best conveys the target information to the reverse engineer? We exploit the software developers' familiarity with UML class diagrams [Fowler 2003]. The basis of our visualizations is a graph representation of classes (nodes) and inheritance relationships (edges) similar to a UML class diagram of inheritance relationships. Our visualization engine renders the runtime events of a feature in 3D as object instantiations (boxes) and message sends between objects (connectors). The boxes are displayed as towers of instances on top of their defining classes in the class hierarchy view. We preserve information about order of occurrence, instantiation and frequency of the runtime events. We render the data using a 3D representation of the dynamic and static data.

   Our polymetric-based visualization is capable of representing quantitative values that affect the size of the nodes. The user can select 3 metrics from a range of metrics supported by our tool and map these to the width, length and color of the nodes. The metrics enhance the expressiveness of our visualizations. For example, in Figure 4 we see that the class `HtmlWriteStream` is represented with a longer box as we mapped (*NOM number of methods*) to the length of the node, and *NOA number of attributes* to the width of the node.

5. *Medium.* Where the visualizations are rendered? We built a tool called *TraceCrawler* [Wysseier 2005], a massive extension of the CodeCrawler tool[Lanza 2003] that provides capabilities to interact and navigate the visualizations (see Figure 3). It supports the animation of feature behavior (*i.e.,* the reverse engineer can step through a feature trace and the 3D visualization engine of TraceCrawler renders the events in the visualization).

According to the integrated model of comprehension [Storey et al. 1999], developers often switch between high level views of the software and low level source code. Hence to support program comprehension, a tool should support a wide variety of comprehension strategies through its visualization and navigation capabilities [Storey and Müller 1995]. With this philosophy in mind, we designed our TraceCrawler tool to generate interactive and navigable 3D visualizations, *i.e.,* the reverse engineer can view the 'big picture' of a feature trace and at the same time examine details of objects (*e.g.,* source code) on demand. Other capabilitiies include (1) changing the point of view in the 3D space with zooming, panning and rotating capabilities., (2) search mechanisms to navigate to a specific event of interest in the trace, and (3) replay mechanisms that support navigation forwards and backwards through the events of a feature trace.

Furthermore we tackle the problem of large amounts of data by providing filtering mechanisms to enable the developer to reconfigure the visualizations to contain only information about specific parts of interest of a system (*i.e.,* a specific class hierarchy).

In the following sections we describe in detail two of the 3D views that are possible with our tool.

3.1 Dynamic Feature-trace View

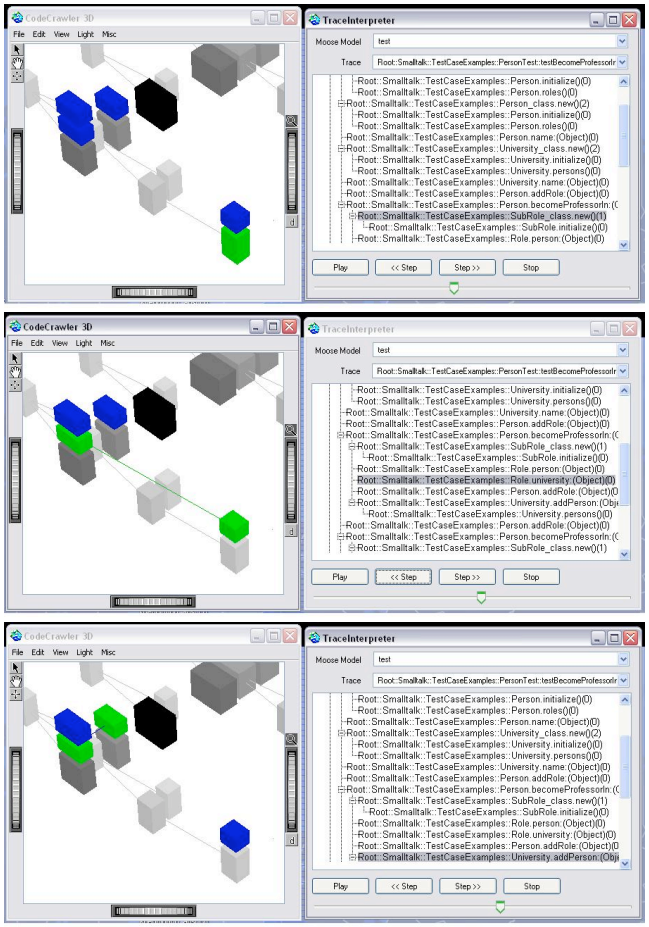

Figure 3: The Dynamic Feature View allows the user to step through and animate the traces.

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 2 we see a schematic display of such a view. The 3D visualization displays the static structure of the system (*i.e.,* the class hierarchy) on a plane "floating" above the ground (the white 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 box (like a "floor" in a building) above the ground level of its corresponding class representation. The more boxes that are above a class, the more instances of this class have been created. The currently active objects are highlighted in green. Each time an object sends a message to another object, a message edge is draw between the two object boxes. The message edges are colored in red to distinguish them from the inheritance edges.

It is fundamental to our 3D visualization technique that the feature trace, extracted during our dynamic analysis, contains unique object identifiers of the sender and receiver of a message.

We refer to our navigable visualization as the *dynamic feature interaction view*. This view allows the reverse engineer to visually step through the individual events of feature-traces. At each point in time of a feature-trace, we see a visual representation of the current state of the feature trace in terms of the active object and message send of the current event. The navigation facilities of the tool allow us to move backward and forward within the events of the feature-trace. Thus we control and delve into the visualization to gain a more fine-grained understanding of the behavior of a feature.

To illustrate the navigation of our visualization we apply our approach to a simple example system consisting of 22 classes that models people and their roles in a university. We generate simple feature-traces.

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 move backward and forward within the trace: In Figure 3 we see a small scenario generated from a simple test system, *i.e.,* three different points in time of the same trace. On the left hand side 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.

Thus the software developer can step backward and forward through the execution of a feature and visually observe the creation of object instances and messages sent between objects. The interactive nature of our visualization (the user can click on the nodes and edges and inspect them) allows the user to navigate to places of interest in the system and observe in detail a certain part of a feature. Our tool provides a search mechanism, which allows the developer to enter a search string (*e.g.,* the name of a method) and then navigate to the next occurrence of the method in the trace.

Moreover, we provide direct access to the source code under observation using a pop-up menu. This visualization does not replace, but perfectly complements, the existing practice of rendering traces as a tree of events (see the right side of Figure 3).

3.2 The Instance Collaboration View

This view is a retrospective view of how the system behaved during the execution of a feature in terms of classes, object-instantiations and message sends. This view conveys the 'big picture' of a feature at runtime. We can locate which parts of the system were actively participating in the feature's behavior at runtime and to which extent. Moreover we can visually compare the visualizations of features to identify parts of the system that appear to be common, and parts that are specific to a give feature. One again the interactive capabilities of our visualization allow us to zoom in on areas of interest and query them for more fine-grained details.

In Figure 4 we show the instance collaboration view of a Login feature from our *SmallWiki* case study. This is a 'big picture' view of SmallWiki as class hierarchies. We see at a glance where the runtime activity occurred when this feature was executed. On the left hand side we see instances of the `HtmlWriteStream` class. There are edges spanning the entire view that indicate that instances of the `HtmlWriteStream` class are communicating with classes in the `Template Hierarchy`.

4 Proof of Concept

4.1 Applying our Approach to Feature Traces of SmallWiki

In this section we present some of the results of applying our visualizations to features of the *SmallWiki* system [Ducasse et al. 2005b]. *SmallWiki* is an object-oriented wiki implementation written in Smalltalk. It provides typical wiki functionalities such as adding and editing pages and user authentication. The version we analyzed (1.297) consists of 288 classes. We assume that each link or button on a page triggers a distinct feature of the application. The features we chose represent typical user interactions with the application such as login, editing a page or searching a web site.

For our proof of concept we generated both dynamic feature views and instance collaboration views. We used the instance collaboration views for postmortem analysis of the feature behavior. We used dynamic feature traces to observe the sequence of events (when behavior occurred in the trace).

The 'big picture' and the detailed view. In Figure 5 we show an instance collaboration view of the *login* feature (4008 events). This represents the 'big picture' view of the dynamic behavior *after execution*. We see areas of activity in the system in terms of towers object instances and message sends, while this feature was executing. We refer to these areas of activities as *feature hotspots*. Using the zoom and rotate capabilities of our TraceCrawler tool, we zoom in to view and inspect the right hand side of the visualization in more detail (see Figure 5). From this perspective we see details of feature behavior. Our visualization reveals that instances of the `Login` class are created when this feature is exercised. One instance of `Login` communicates heavily with instances of the Template hierarchy (subclasses of `TemplateBody` and `TemplateHead`). The developers of SmallWiki confirm that templates are used for the composition of pages. That is why this instance renders the login form and is executed to perform the login functionality.

Patterns of common activity in the features. Due to the nature of SmallWIki as is a web based application, interacting with the web browser to resolve a URL or render a page represents behavior that is common to the user-initiated features of our analysis. Our instance collaboration views of features confirm this characteristic of the application by rendering visually similar execution patterns in all our features. For example in each of our views of features we detect instances of the `Response` class. This class is responsible for handling HTTP responses sent to the web browser.

Another common pattern is that each view contains instances of the `HTMLWriteStream` class. This class is responsible for the gen-

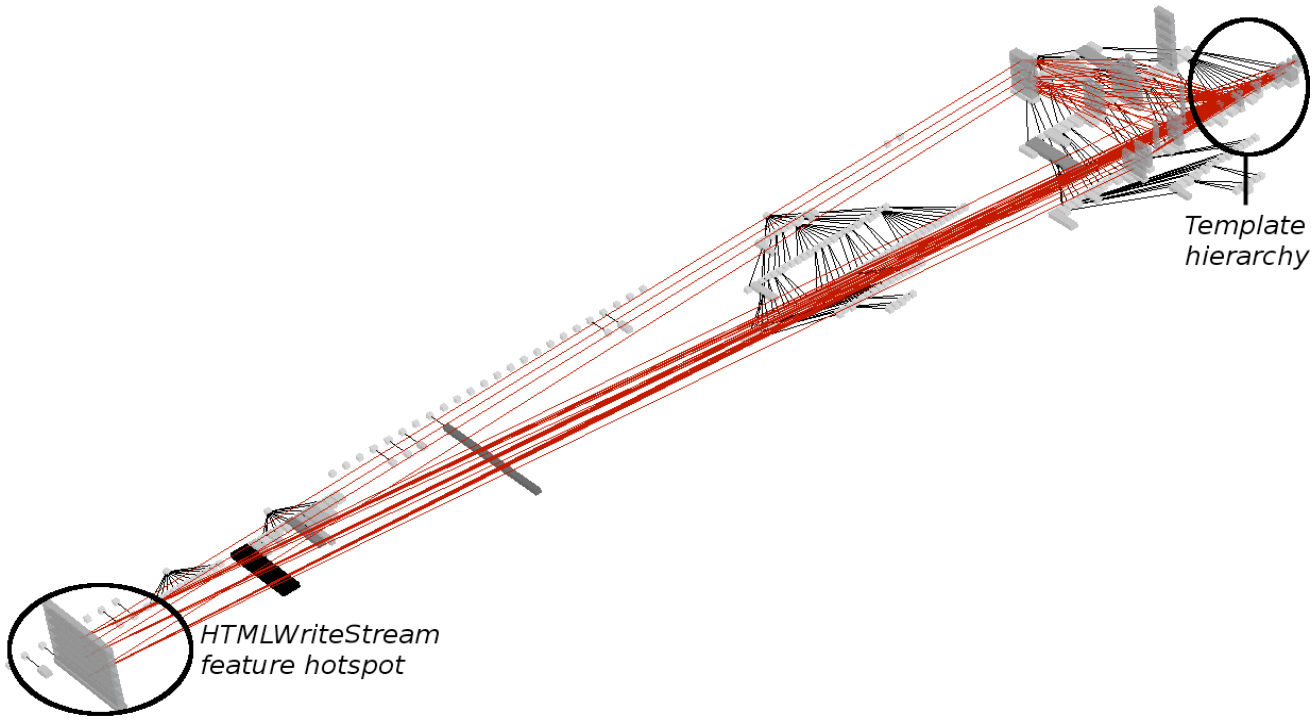

Figure 4: An Overview of the SmallWiki case study after the execution of the login feature.

eration of the necessary HTML code to be displayed in the web browser. Therefore this implements functionality that is generic or common to all features under analysis.

Our visualization reveals that an instance of the `PageView` class communicates heavily with instances of the `Template` hierarchy. The developers once again confirm that this recurring collaboration is due to the fact that SmallWiki pages are composed of templates.

In the zoomed-in view of the Login feature in Figure 5 we detect many of the common patterns of behavior.

Behavior that is unique to one Feature. We examine the feature-specific runtime behavior we detected in some of the other features of SmallWiki.

Edit Page Feature Trace (5608 events) allows the user to modify the page by entering an editing mode. Once the user is finished editing the page, the new version gets saved and displayed. The visualization shown in Figure 6 reveals that instances of the *PageEdit* class participate in this feature. This behavior is unique to this feature. We validate our findings with the developers and discover that the `PageEdit` class is responsible for rendering the form to edit a wiki page and to save the submitted content. As this feature is the only feature that exercises this class, the behavior is unique to it.

Search Feature Trace (7742 events) allows the user to search all the pages of a wiki for a specific string. Figure 7 reveals a tower of instances of the `Search` class. Two of the instances heavily communicate with other objects to perform this task. This collaboration pattern is unique to this feature.

4.2 Applying our Approach to Feature Traces of Moose

In this section we present the results of applying our approach to large feature traces captured by exercising features of the *Moose* system. Moose is a language-independent environment for reengineering object-oriented systems[Ducasse et al. 2005a]. It provides features for manipulating, navigating, querying and applying metrics to models derived from parsing source code. The version we use for this experiment (3.0.25) consists of 782 classes.

The reason we include this case study is to illustrate how our visualizations handle large traces.

Visualizing large traces. Figure 8 shows the *instance collaboration view* of the feature that loads a model of a software system from a file. As the information space is large we use the zooming and scrolling features of the TraceCrawler tool to view the trace. We highlight activities of interest. The most striking part of this view is the large number of object instantiations on the right side. The developers of Moose confirm this behavior is in line with the purpose of the feature as each model entity is instantiated as a model is loaded.

We also see that the class `CCEntityTypeFactory` has a large number of message edges. The developers confirm that this class is responsible for creating entities when a model is being loaded.

We used the dynamic feature trace view to observe the sequence of events (when behavior occured in the trace). For this feature we see that most of the instantiations of the `MooseEntity` occur at regular intervals in the trace. This suggests a loop in the feature's behavior. Also we see the order of message sends which reveals when instances of a class are communicating with other instances.

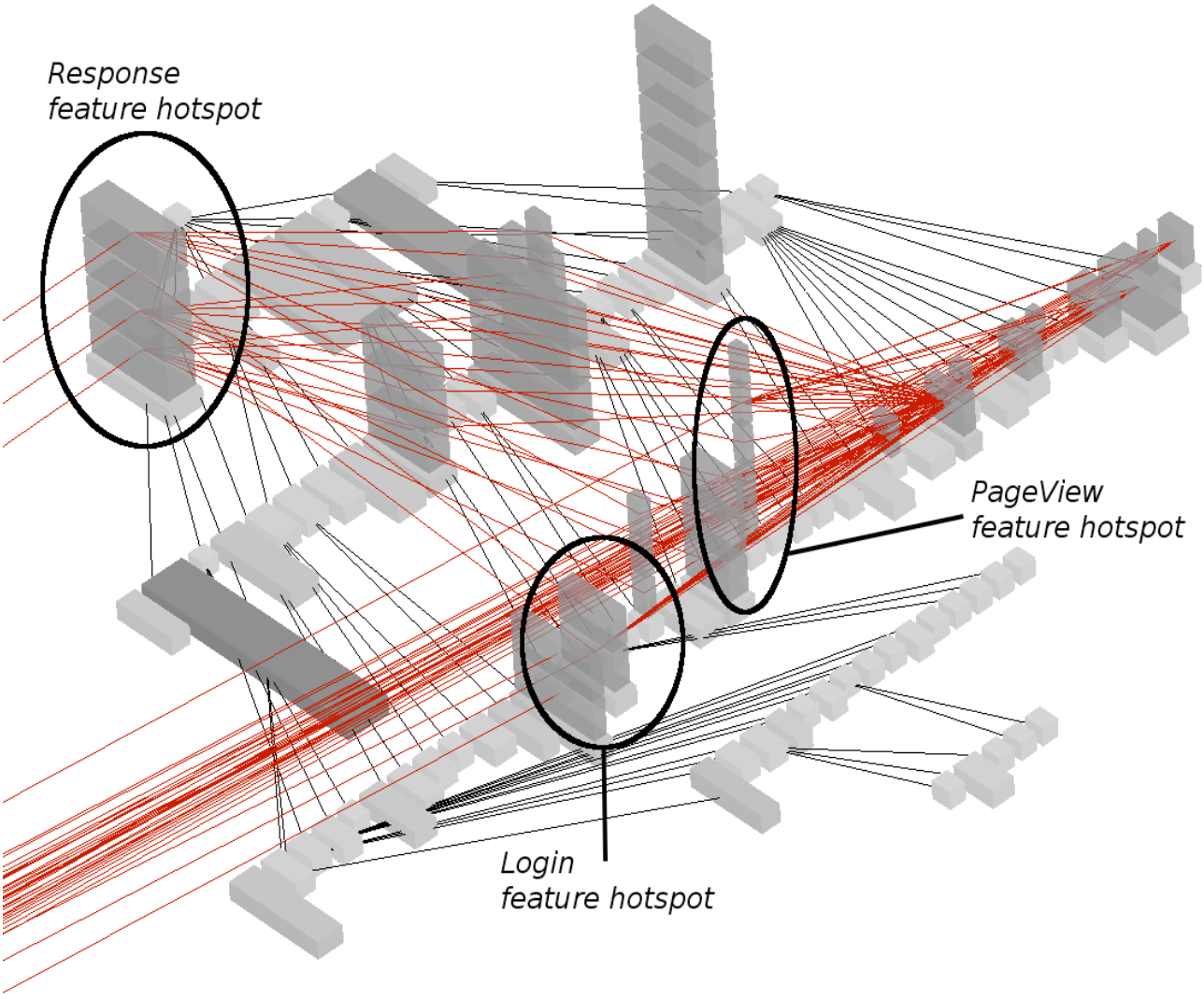

Figure 5: Zooming into the class hierarchy active during the login feature. We highlight common patterns of behavior

5 Discussion

In this section we discuss issues relevant to our approach and outline some of the constraints of our visualizations.

2D versus 3D. One major point of criticism is whether using a 2D visualization would not be better, because navigating 3D visualizations is problematic. We already visualized combined static and dynamic information previously[Ducasse et al. 2004] but the drawbacks are the loss of the notion of time and the condensed information: both dynamic and static information have to be rendered in only two dimensions, while in the present approach we use two dimensions for the static information and the third dimension for the dynamic information.

The Interactive Visualization. Our visualizations provide an overview of the entire collection of data that is represented. This can often be difficult to interpret in the case of large systems with a large number of classes. Therefore the interactive capabilities of our visualization is an integral part of this discussion. Our visualizations allow for zooming, pannning and rotation of the view. Thus we address problems such as occlusion. The interactive capabilities of our visualizations enable us to query the visualization to obtain more fine-grained information about the key entities of interest. For example, we query a node of the visualization to obtain the class name and view the source code. It is difficult to render such an interactive process on paper media. However, we show from the the results of applying our approach that we reveal valuable information to support the reverse engineering process merely viewing the visualizations. The fine-grained details of the features are obtained by manipulation and interaction with the visualizations. Moreover, the user can toggle the display the edges, where in one mode only the last message edge is displayed, while in the other mode all message edges sent up until the moment in time the view is in are displayed.

Naming. The evidence of analysis of the SmallWiki application reveals that the developers adhered to sound naming practices. The names of key classes identified by our approach reflect the intention of the corresponding features. Thus our visualizations reveal

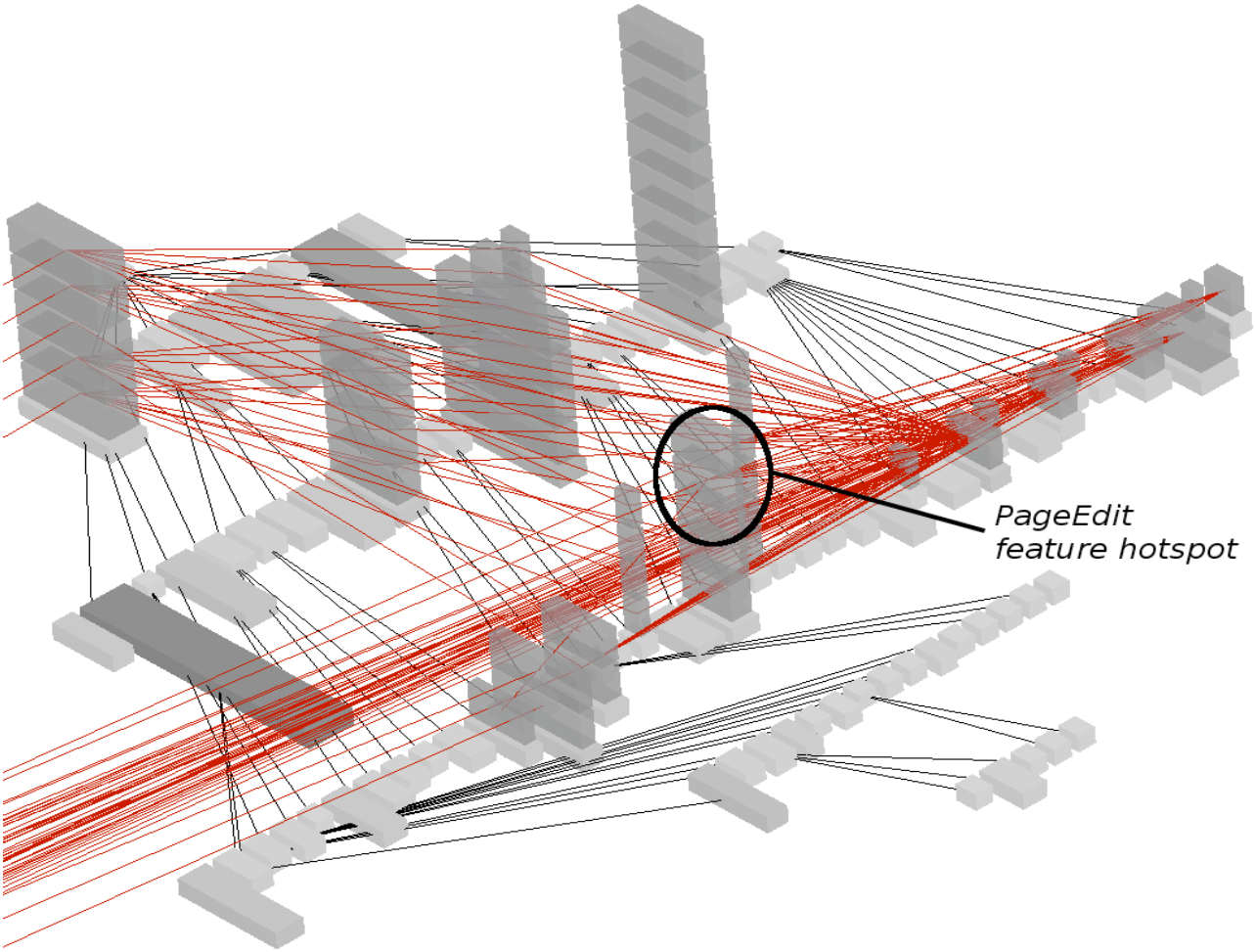

Figure 6: A detail of the ”Edit Page” feature revealing a unique behavioral pattern.

evidence that, in the case of the SmallWiki system, semantic analysis or regular expression matching techniques could be applied to uncover concepts in the code.

Scalability. As discussed above, the expressiveness of the visualization and the interactive capabilities of our technique support the representation and interpretation of large amounts of data. In our SmallWiki case study, we chose five relatively small features that involve over 8000 interactions among classes and objects. We have successfully applied our approach to feature traces of the Moose application which consist of over 70’000 events.

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. As long as the traces contain events of message sends and object instance information about sender and receiver instances, our visualizations will work for any object oriented language.

Selective Instrumentation and Garbage Collection. To limit the amount of dynamic information, we applied selective instrumentation of SmallWiki and Moose. In other words, we did not instrument the entire system. We only includes classes from the SmallWiki and Moose namespaces in the traces. This results in incomplete traces. Our approach is in the experimentation phase. We believe that with increased experience, an iterative approach to trace collection via selective instrumentation and post-filtering will improve the results. This implies that we do not model the fact that objects are garbage collected and removed from memory.

6 Implementation

The implementation of the discussed approach in based on three tools: Moose, TraceScraper and TraceCrawler. Moose is a language-independent environment for representing software models [Nierstrasz et al. 2005].

TraceScraper is our feature analysis tool. It is based on the Moose [Ducasse et al. 2005a] reengineering platform. It provides a means of instrumenting Smalltalk code and automatically executing features to abstract and model feature-traces as first-class entities in the Moose environment. Tracescraper also provides a means of importing dynamic information, captured by instrumenting systems written in other languages. We extend the FAMIX model

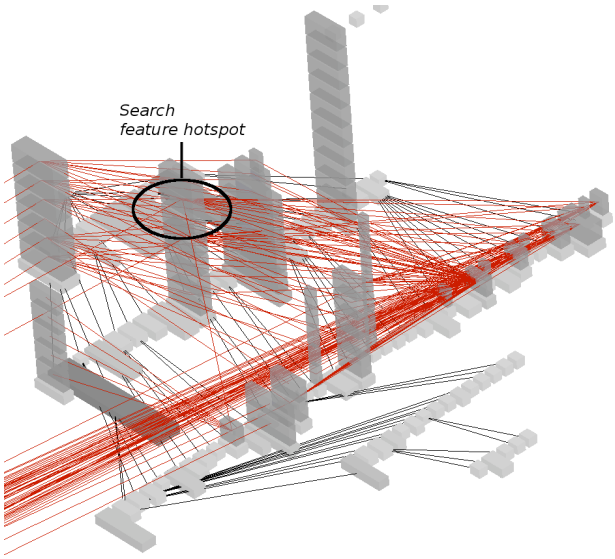

Figure 7: A detail of the "Search" feature revealing a unique behavioral pattern..

with *feature-trace* entities to relate the feature-trace information with the class and method entities of the model.

TraceCrawler [Wysseier 2005] interprets the trace information provided by TraceScraper and controls the visualization. It processes execution traces in the Moose model and represents the events of the trace as 3D visualizations. The visualization is created by CCJun[Wysseier 2004] which is an extension of CodeCrawler[Lanza 2003] and based on the 3D framework Jun.

TraceCrawler can easily be extended to interpret traces generated from any system as long as the required information is available in the event data, namely sender and recipient classes, sender and recipient instance identifiers, method and instance identifier of the method return value.

7 Related Work

The focus of this work is to show how our 3D visualization technique supports system comprehension in terms of its features. Feature location techniques such as *Software Reconnaissence* described by Wilde and Scully [Wilde and Scully 1995], and that of Eisenbarth et al. [Eisenbarth et al. 2003] are closely related to our feature driven analysis. We extend the focus beyond the task of feature location in the source code to consider the object-oriented dynamic behavior of features in terms of instantiation and message sends. This level of information is not addressed in previous feature location approaches.

Graphical representations of software have long been accepted as a comprehension aid [Stasko et al. 1998]. The work of Maletic *et al.* has provided important guidelines for motivating and defining our visualizations. In their work they defined levels of interest and the criteria of effectiveness and expressiveness of software visualization[Maletic et al. 2002]. Marcus *et al.* also use a 3D metaphor in their sv3D tool to represent a software system and analysis data [Marcus et al. 2003], but they do not make a specialized use of the 3rd dimension, but use all 3 dimensions to render static information. Our approach distinguishes clearly between the representation of static structural information and the dynamic information, which grows as "towers'" of object instances above the structural representation.

Substantial research has been conducted on runtime information visualization. Various tools and approaches make use of dynamic (trace-based) information such as Program Explorer [Lange and Nakamura 1995a], Jinsight and its ancestors [De Pauw et al. 1993], and Graphtrace [Kleyn and Gingrich 1988]. Vion and Drury [Vion-Dury and Santana 1994] use 3D to represent the runtime of objects in distributed and concurrent systems. De Pauw *et al.* present two visualization techniques. In their tool Jinsight, they focused on interaction diagrams [De Pauw et al. 1993]. Thus all interactions between objects are visualized. The focus of our visualizations is to address reverse engineering dynamic behavior of features. Thus we tackle the challenge of obtaining high level views from a large volume of information to support reasoning about the runtime behavior of features.

Kleyn and Gingrich [Kleyn and Gingrich 1988] and also Lange and Nakamura [Lange and Nakamura 1995b] chose a graph-based approach to visualize dynamic behaviour. Kleyn and Gingrich also animate their views by highlighting and annotating nodes and edges to represent activity in the code.

Walker *et al.* [Walker et al. 1998] use program animation techniques to display the number of objects involved in the execution and the interaction between them through user-defined high-level models. Their tool uses a summary strategy to show live objects as a histogram, and the reduction of the information space by allowing the user to cluster together code elemets to create a high level model. Our *TraceCrawler* tool provides a means of stepping through the trace of a feature and to render each event in the visualization.

Jerding *et al.* propose an approach to visualizing execution traces as Information Murals [Jerding et al. 1997]. They define a *Execution Mural* as a graphical depiction an entire execution trace of the messages sent during a program's execution. These murals provide a global overview of the behavior, They also define a *Pattern Mural* which visually represents a summary of a trace in terms of recurring execution patterns. Both views are interdependent.

Reiss [Reiss 2003] developed *Jive* to visualize the runtime activity of Java programs. The focus of this tool was to visually represent runtime activity in real time. The goal of this work is to support software development activities such as debugging and performance optimizations. Our focus is feature-centric reverse engineering. Feature trace data is captured from a running system and then modeled in the context of static source code entities. However our technique is non-restrictive and could easily be adapted to interpret real time trace information.

Our visualization metaphor is intuitive as it exploits the developers familiarity with graph visualizations. We emphasize the importance of ease of interpretation of a visualization to gain acceptance by the software developer. We preserve the sequence of events. Our approach complements the approach of De Pauw *et al.* [De Pauw et al. 1993] by allowing the developer to interact with the visualization and control the display of events in a feature-trace. Thus the reverse engineer exploits her feature understanding of a system and directly focus on the parts of dynamic data of interest.

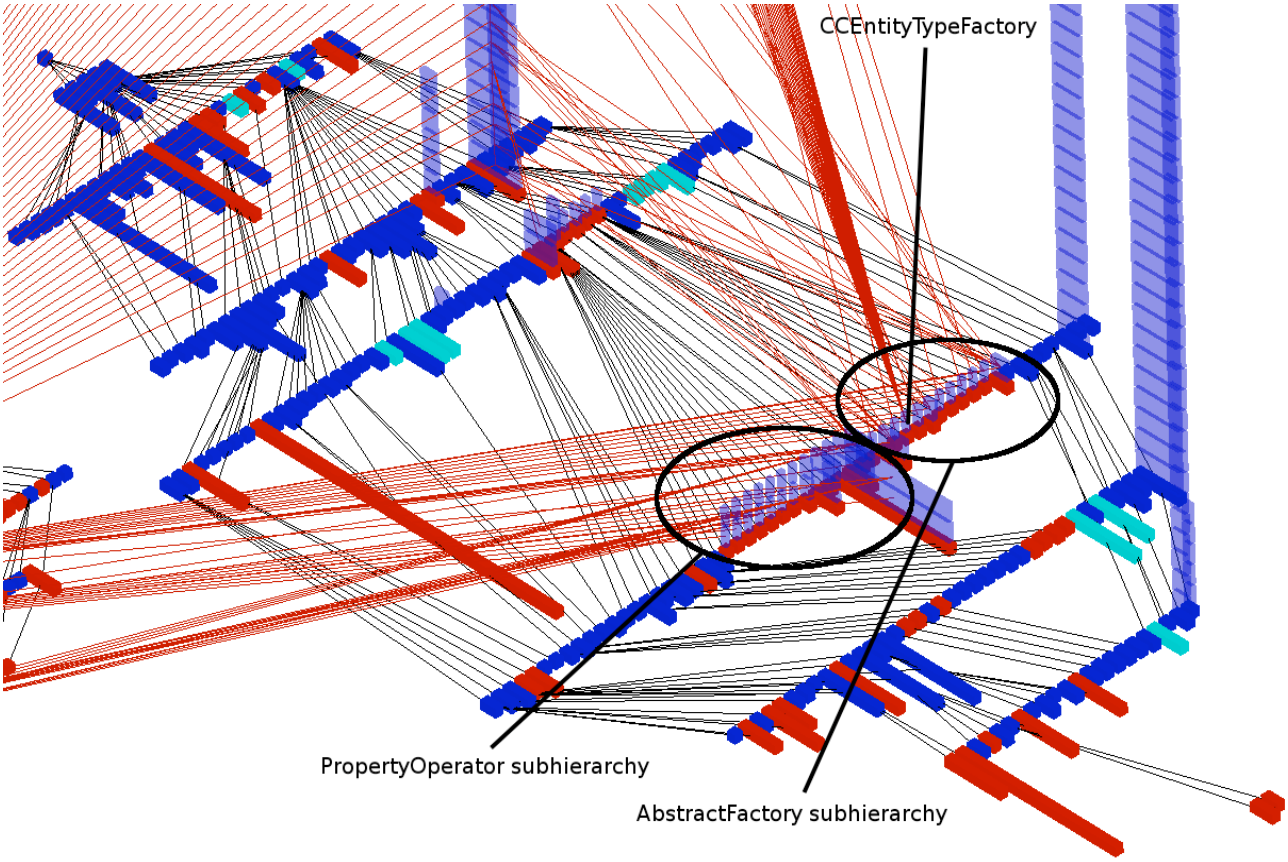

Figure 8: Part of the Instance Collaboration View of the Moose 'load model' feature.

8 Conclusions

The large volume and complexity of dynamic information makes it difficult to infer how a software system implements features. Our visualization metaphor of growing towers of instances represents large amounts of dynamic data effectively, while still maintaining its structural context. The developer quickly obtains an overview of the dynamic behavior of features. The trace of a whole feature can for example be displayed as a "movie" which starts with a static and flat visualization, and then as the trace is being executed, we can see towers of objects grow in the different parts of the system that are being activated. We also provide visual feedback of the currently happening event by means of a colorisation of the active object or sent message. We set out to show how our visualizations answer the following the reverse engineering questions:

Which classes and objects are most active during the execution of a feature? The classes that participate in feature behavior are easily identifiable in our visualizations. The interactive capabilities of the visualization allows to query the nodes to obtain more fine-grained information about the classes and instances involved.

What are the patterns of activity that are common in feature behavior and which are specific to one feature? The feature visualizations of our *SmallWiki* case study reveal which classes are active in more than one feature, and recurring collaborations between instances. We also detect behavior that is specific to one feature.

The main contribution of this paper is a novel approach to the visualization of large feature execution information in the context of static structural information. Our visualization allows the developer to navigate the large information space given by the analysis of run-time information.

Acknowledgments: We gratefully acknowledge the financial support of the Swiss National Science foundation for the projects "The Achievement and Validation of Evolution-Oriented Software Systems" (SNF Project No. PMCD2-102511), "COSE - Controlling Software Evolution" (SNF Project No. 200021-107584/1), and "NOREX - Network of Reengineering Expertise" (SNF SCOPES Project No. IB7320-110997), and the Hasler Foundation for the project "EvoSpaces - Multi-dimensional navigation spaces for software evolution" (Hasler Foundation Project No. MMI 1976). We also thank Tudor Girba for his constructive comments on this work.

References

CHIKOFSKY, E. J., AND II, J. H. C. 1990. Reverse engineering and design recovery: A taxonomy. *IEEE Software* (Jan.), 13–17.

DE PAUW, W., HELM, R., KIMELMAN, D., AND VLISSIDES, J. 1993. Visualizing the behavior of object-oriented systems. In *Proceedings OOPSLA '93*, 326–337.

DUCASSE, S., LANZA, M., AND BERTULI, R. 2004. High-level polymetric views of condensed run-time information. In *Proceedings of CSMR 2004 (Conference on Software Maintenance and Reengineering)*, 309–318.

DUCASSE, S., GÎRBA, T., LANZA, M., AND DEMEYER, S. 2005. Moose: a collaborative and extensible reengineering environment. In *Tools for Software Maintenance and Reengineering*, RCOST / Software Technology Series. Franco Angeli, Milano, 55–71.

DUCASSE, S., RENGGLI, L., AND WUYTS, R. 2005. SmallWiki — a meta-described collaborative content management system. In *International Symposium on Wikis (WikiSym'05)*, ACM Computer Society, New York, NY, USA, 75–82.

EISENBARTH, T., KOSCHKE, R., AND SIMON, D. 2003. Locating Features in Source Code. *IEEE Computer 29*, 3 (Mar.), 210–224.

ERNST, E. 2003. Higher-order hierarchies. In *Proceedins of the European Conference on Object-Oriented Programming*, Springer Verlag, Darmstadt, Germany, LNCS.

FOWLER, M. 2003. *UML Distilled*. Addison Wesley.

GREEVY, O., AND DUCASSE, S. 2005. 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, 314–323.

GREEVY, O., DUCASSE, S., AND GÎRBA, T. 2005. Analyzing feature traces to incorporate the semantics of change in software evolution analysis. In *Proceedings of ICSM 2005 (21th International Conference on Software Maintenance)*, IEEE Computer Society Press, 347–356.

GREEVY, O., LANZA, M., AND WYSSEIER, C. 2005. Visualizing feature interaction in 3-d. In *Proceedings of Vissoft 2005 (3th IEEE International Workshop on Visualizing Software for Understanding)*.

HAMOU-LHADJ, A., BRAUN, E., AMYOT, D., AND LETHBRIDGE, T. 2005. Recovering behavioral design models from execution traces. In *Proceedings of CSMR 2005 (9th European Conference on Software Maintenance and Reengineering*, IEEE Computer Society Press.

JERDING, D. J., STASKO, J. T., AND BALL, T. 1997. Visualizing interactions in program executions. In *Proceedings of ICSE '97*, 360–370.

KLEYN, M. F., AND GINGRICH, P. C. 1988. GraphTrace — understanding object-oriented systems using concurrently animated views. In *Proceedings OOPSLA '88 (International Conference on Object-Oriented Programming Systems, Languages, and Applications*, ACM Press, vol. 23, 191–205.

LANGE, D. B., AND NAKAMURA, Y. 1995. Interactive Visualization of Design Patterns can help in Framework Understanding. In *Proceedings of OOPSLA '95 (International Conference on Object-Oriented Programming Systems, Languages and Applications)*, ACM Press, 342–357.

LANGE, D., AND NAKAMURA, Y. 1995. Object-oriented program tracing and visualization. Research Report RT0111, IBM Research, Tokyo Research Laboratory.

LANZA, M., AND DUCASSE, S. 2003. Polymetric views—a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering 29*, 9 (Sept.), 782–795.

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

MALETIC, J. I., MARCUS, A., AND COLLARD, M. 2002. A task oriented view of software visualization. In *Proceedings of the 1st Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2002)*, IEEE, 32–40.

MARCUS, A., FENG, L., AND MALETIC, J. I. 2003. 3d representations for software visualization. In *Proceedings of the ACM Symposium on Software Visualization*, IEEE, 27–ff.

NIERSTRASZ, O., DUCASSE, S., AND GÎRBA, T. 2005. The story of Moose: an agile reengineering environment. In *Proceedings of the European Software Engineering Conference (ESEC/FSE 2005)*, ACM Press, New York NY, 1–10. Invited paper.

REISS, S. P. 2003. Visualizing Java in action. In *Proceedings of SoftVis 2003 (ACM Symposium on Software Visualization)*, 57–66.

SHARP, R., AND ROUNTEV, A. 2005. Interactive exploration of uml sequence diagrams. In *Proceedings of VISSOFT 2005 (3rd IEEE Workshop on Visualizing Software for Understanding and Analysis*, IEEE CS Press, 8–13.

STASKO, J. T., DOMINGUE, J., BROWN, M. H., AND PRICE, B. A., Eds. 1998. *Software Visualization — Programming as a Multimedia Experience*. The MIT Press.

STOREY, M.-A. D., AND MÜLLER, H. A. 1995. Manipulating and Documenting Software Structures using SHriMP Views. In *Proceedings of ICSM '95 (International Conference on Software Maintenance)*, IEEE Computer Society Press, 275–284.

STOREY, M.-A. D., FRACCHIA, F. D., AND MÜLLER, H. A. 1999. Cognitive Design Elements to Support the Construction of a Mental Model during Software Exploration. *Journal of Software Systems 44*, 171–185.

STROULIA, E., AND SYSTA, T. 2002. Dynamic analysis for reverse engineering and program understanding. *SIGAPP. Appl. Comput. Rev. 10*, 1, 8–17.

VION-DURY, J.-Y., AND SANTANA, M. 1994. Virtual images: Interactive visualization of distributed object-oriented systems. In *Proceedings of OOPSLA 1994*, A. Press, Ed., 65–84.

WALKER, R. J., MURPHY, G. C., FREEMAN-BENSON, B., WRIGHT, D., SWANSON, D., AND ISAAK, J. 1998. Visualizing dynamic software system information through high-level models. In *Proceedings OOPSLA '98*, ACM, 271–283.

WILDE, N., AND SCULLY, M. C. 1995. Software reconnaisance: Mapping program features to code. *Software Maintenance: Research and Practice 7*, 1, 49–62.

WYSSEIER, C. 2004. CCJun – polymetric views in three-dimensional space. Informatikprojekt, University of Berne, June.

WYSSEIER, C. 2005. *Interactive 3-D Visualization of Feature-Traces*. MSc. thesis, University of Berne, Switzerland.