



# Softwareonaut: A Tool for Collaborative Architecture Recovery

Mircea Lungu

*Software Composition Group  
University of Bern, Switzerland*

Michele Lanza

*REVEAL  
University of Lugano, Switzerland*

---

## Abstract

When the initial architecture of the system has eroded the only solution is architecture recovery. However, when the system under discussion is large, the user must use dedicated tools to support the recovery process. In this article we present Softwareonaut – a collaborative architecture recovery tool. Softwareonaut is built around the concept of interactive exploration and visualization and supports the recovery of multiple architectural views which can be shared with other users and tools.

*Keywords:* Architecture Recovery, Visualization, Reverse Engineering

---

## 1. Introduction

No software system is an island entire of itself. Instead the system exists and functions in an environment, and when the environment changes the system must change too or become obsolete[1]. As a result, maintaining a software system implies a continuous effort to keep it up to date with the unanticipated changes in its environment. Having a clear and up to date understanding of the architecture of the system is critical to maintaining and evolving that system [2].

As the system evolves, the architecture erodes [3] and an architectural mismatch appears between the *as-defined* and *as-is* architecture [4]. One accompanying property of this continuous drift between the actual architecture and the defined architecture of the system is an increasing brittleness





of the system [3]. The main reason for the architectural erosion and drift is widespread lack of programming language support for expressing the architecture as well as the lack of tools that associate architectural decisions with the source code. The problem is an instance of the documentation problem: it is well known that the documentation of the system becomes quickly obsolete unless developers dedicate conscientious effort towards keeping it up to date [5].

Having a clear and up to date understanding of the architecture is critical to maintaining and evolving a system [6]. When the drift and erosion have brought the architecture too far from the initial, the solution is recovering the architecture of the system from the source code. Jazayeri defines architecture recovery as “the techniques and processes used to uncover a systems architecture from available information” [7].



In the case of large software systems the architecture is specified through multiple architectural views that correspond to a set of given architectural viewpoints. An architectural viewpoint is a pattern or template from which to develop individual views by establishing the purposes and audience for a view and the techniques for its creation and analysis. Even if different authors propose different viewpoints [8, 9, 10] the consensus is that multiple viewpoints are necessary for capturing all the various facets of a system.

Usually, architecture recovery tools focus on recovering module views through visualization and interaction [11, 12, 13]. While some steps of the process are usually automated (e.g., fact extraction, view generation), none of the tools works completely without human intervention. In some cases the user has to group related artifacts together based on their similarity of purpose [12]. In others the user has to compare the architecture as-extracted with the architecture as-predicted [11]. Yet in others the user has to decide which exploration paths to follow [13].

In this article we present Softwarent - an architecture recovery tool that we have developed. Softwarent provides interactive exploration mechanisms that support the semi-automated discovery of architectural views of any object-oriented system and allows the sharing of such architectural views.

**Structure of the article.** We start with an overview of Softwarent in Section 2. We then dedicate three sections to discuss the important phases in the workflow of any architecture recovery tool: fact-extraction (Section 3), information aggregation (Section 4) and interactive exploration (Section 5). We present the support for collaboration in Section 6. In Section 7 we discuss aspects relevant to the tool-building experience and in Section 8 we compare our work with the state of the art. We conclude in Section 9.



## 2. A Quick Overview

Softwareaut is an architecture recovery tool which is based on interactive exploration. The goal of the interactive exploration **process the** discovery of multiple architecturally relevant views [15].

Automatically aggregating the low-level relations, and then letting the user navigate from the highest abstraction level downwards is the main interaction approach in Softwareaut.

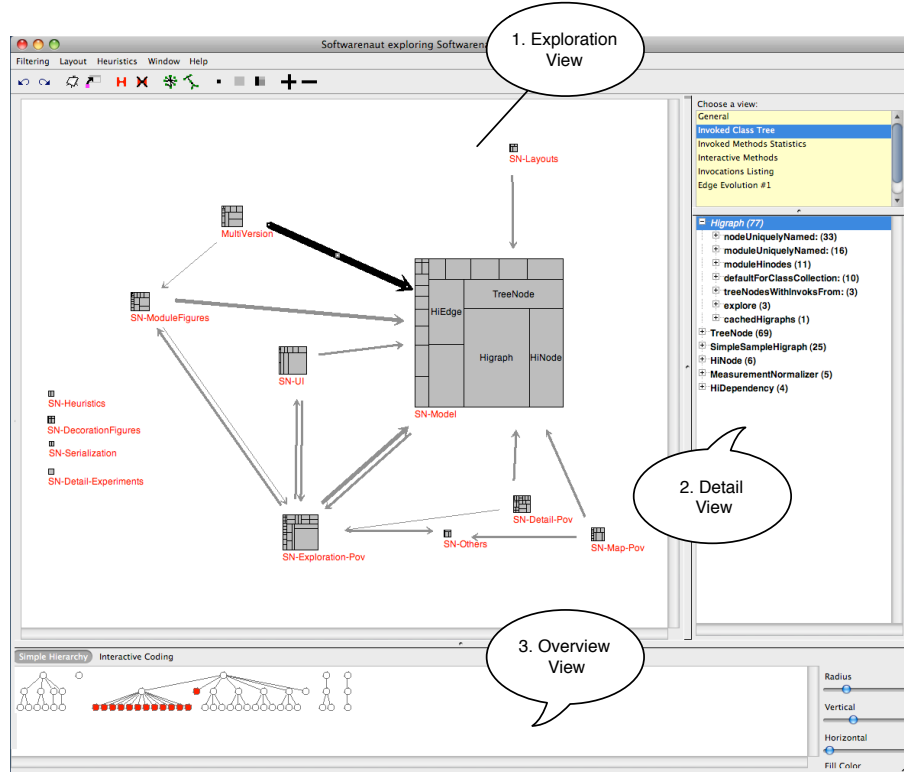


Figure 1: Softwareaut uses linked views to present multiple perspectives on a software system at once

The UI of Softwareaut contains three linked complementary visual perspectives that present information about the system during the exploration. Figure 1 presents Softwareaut visualizing an architectural view of itself. The figure illustrates the three complementary views that the tool supports:

1. The Exploration is the main view in Softwarenavt. It is a graph-based representation of modules and their dependencies. The nodes in the graph represent modules and the edges represent the dependencies between the modules. Each dependency edge is an aggregation of low-level dependencies between the two associated modules. To represent the nodes and edges we use a polymetric view metaphor as introduced by Lanza [16, 17].
2. The Detail view presents details for the entity selected in the Exploration View. The goal of this view is to provide insight into the details of the element selected in the exploration view. This view implements the “details-on-demand” part of the visualization mantra of Shneiderman [18].
3. The Overview view presents the entire hierarchy of the system and highlights on it the modules that are visible in the exploration view. The Overview view presents a horizontal slice through the system [19]. This view is significant because it offers a sense of orientation which is critical for successful navigation [20].

Like other architecture recovery tools [6], Softwarenavt conforms to a classical extract-abstract-view architecture. Figure Figure 2 presents the three main steps of this architecture:

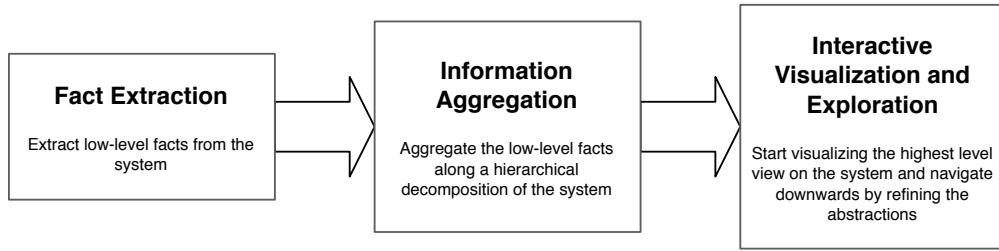


Figure 2: Softwarenavt supports an extract-abstract-view workflow model

Dedicated sections will treat each of the three steps of the workflow but the Interactive Visualization and Exploration part will be treated in the most detail.

### 3. Fact Extraction and Modelling

At the most basic level, SoftwareNaut models a system using the Core of the FAMIX meta-model. The Core of FAMIX is a language independent meta-model that describes the static structure of object-oriented software systems. Since its introduction by Tichelaar [21], FAMIX has evolved through multiple versions, the latest one being FAMIX 3.0.

#### 3.1. Entities and relationships in FAMIX

FAMIX Core represents both artifacts and relationships as first-class entities. The main artifacts are namespaces, packages, classes, attributes, methods, fields. The main relationships between these entities are method invocations, variable accesses, class inheritances, and package include relationships [21].

A class of relationships which have a special importance in SoftwareNaut are the containment relationships which organize a software system in a vertical hierarchy: classes contain methods, modules contain classes, systems contain modules. This containment mechanism is a powerful way of coping with the complexity of large software systems.

At the architectural level, different languages provide different mechanisms for the hierarchical organization of the system. C/C++ developers use the directory structure to organize systems hierarchically; Java developers use the package hierarchy; Smalltalk developers use the bundles hierarchy, etc.

When a hierarchical decomposition is not provided, we can automatically generate one using clustering techniques [22]. We presented elsewhere an experiment with clustering the classes in a system based on natural language similarity [23].

#### 3.2. Language Independence

Given that the FAMIX meta-model is language independent SoftwareNaut can analyze any system written in an object-oriented language. This requires fact extractors that analyze the source code and build the intermediary FAMIX model. For this we use various third-party tools like McC [24] or inFusion <sup>1</sup>.



---

<sup>1</sup>At the time of writing this at <http://www.intooitus.com/inFusion>

#### 4. Information Abstraction

Aggregating horizontal relationships between software artefacts along the containment relationships is the fundamental technique that allows for abstraction in SoftwareNaut.

Figure 3 exemplifies the aggregation of low-level relationships between methods and classes along the containment hierarchy of a Java system. The example contains two explicit low-level dependencies: the method calls between *mc1* and respectively *mc2* and *mc5*. The explicit dependencies propagate as implicit relationships vertically along the containment relationships  $mc1 \rightarrow c1 \rightarrow B \rightarrow A$ ,  $mc5 \rightarrow c5 \rightarrow d \rightarrow c$  and  $mc2 \rightarrow c2 \rightarrow d \rightarrow c$ . The highest level implicit relationship is the one between A and C.

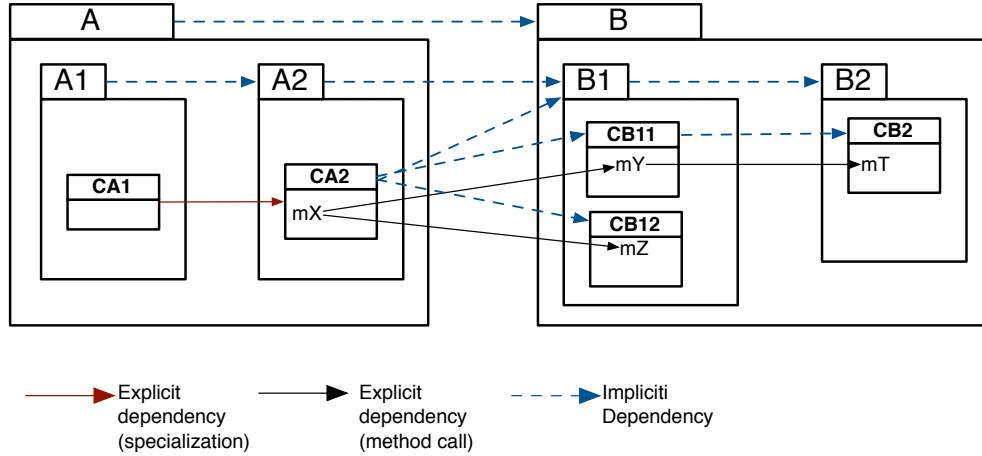


Figure 3: The aggregation of explicit dependencies into implicit ones along the containment relationships between methods, classes, and packages in a Java system



A data structure **that allows for this** is the Higraph introduced by Harel [25]. In our case, the DPMHigraph<sup>2</sup> is a data structure formed by taking the graph of basic artefacts and relationships between them and aggregating them along the containment hierarchy of the system.

The diagram from Figure 4 presents two types of abstract entities and two types of abstract relations:

<sup>2</sup>DPM stands for Detail Project Model

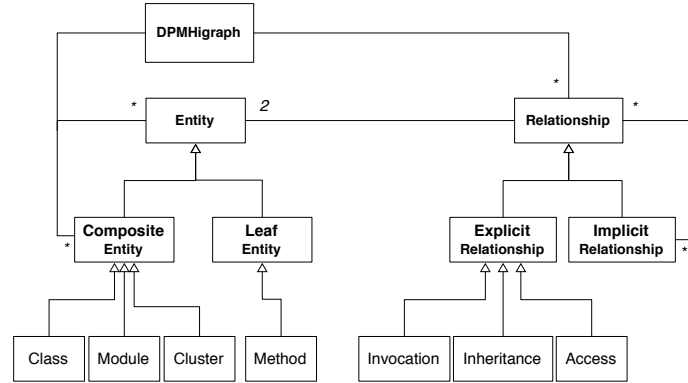


Figure 4: The DPMHigraph model is the core representation of a system in SoftwareNaut

- Leaf Entities. The leaf entities are the basic object-oriented programming building blocks used for the structuring of software. Depending on the analyzed **relation-ship**, the leaf entities can be either methods or classes.
- Composite Entities. The composite entities are containers for other entities. They can have direct mappings to programming language entities, such as classes, packages, namespaces or modules but can also represent abstract composites such as clusters.
- Explicit Relationships. These are the relations between two entities as they exist in the code. SoftwareNaut models method invocations, class inheritance, and field access.
- Implicit Relationships. The model admits relationships between any abstract entities. However, in software systems explicit relationships usually exist only between the leaf entities. Therefore, the relations between the composite entities must be inferred bottom-up from the relations existing between the leafs. The result is that between any two high-level components, we have a relation that represents a collection of all the relations between the leaf components aggregated in them.

## 5. Interactive Exploration

In this section we detail the interactive techniques that support the exploration process in Softwareaut: navigation (5.1), details-on-demand (5.2), rule-based filtering (5.3) and first-class views (5.4).

### 5.1. Navigation

The dominant exploration mechanism of Softwareaut is navigation along the vertical decomposition of the system. One starts with a very high-level abstracted view of a system and continuously **refines** by using exploration operations [26]. At a given moment the set of visible nodes in the exploration view with which the user interacts constitutes the working set (WS). Initially the working set contains very few nodes and they are very high-level. As the user explores the system she transforms the working set by performing exploration operations on it and in this way, she changes the contents of the view in the exploration view.

The exploration operations supported by Softwareaut are:

- Expand. The expand operation applied to a node of the working set

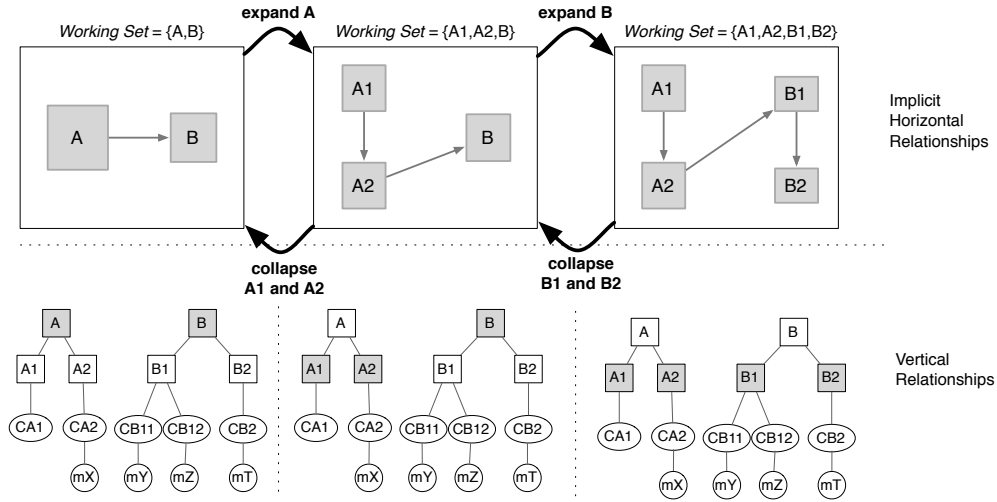


Figure 5: The expand and collapse complementary operations that allow vertical navigation in the Higraph



replaces in the working set the node with nodes that represent its children. We define the operation formally as follows:

$$Expand_{N,HG}(WS) = WS - N + children(N, HG),$$

where  $children(x, h)$  is a function which returns the children of node  $x$  in higraph  $h$ .

- Collapse. The collapse operation applied to a module in the working set, removes the module and all its siblings from the view and replaces them their parent module. We define the operation formally as follows:

$$Collapse_{N,HG}(WS) = WS - N - siblings(N, HG) + parent(N, HG)$$

where  $siblings(x, h)$  is a function which returns the nodes that have the same parent with  $x$  in higraph  $h$ , and  $parent(x, h)$  is a function which returns the parent of node  $x$  in higraph  $h$ .

- Filter. The filter operation applied to a node removes that node from the working set. We define the operation Formally as follows:

$$Filter_{N,HG}(WS) = WS - N$$

- Group. The group operation applied to several modules removes the modules from the view and replaces them with a new virtual module. By grouping several modules together the user reduces the clutter in the view. We define the group operation formally as follows:

$$Group_{N_i,HG} = WS - N_i + NewGroupNode$$

where  $N_i$  is the set of nodes the user wants to group.

As the user refines the view, and climbs down in the hierarchy of the system, he brings more and more elements into the view. She can use the Filter and Group operations on explicit sets of nodes to decrease the number of nodes displayed on screen and therefore cope with the complexity of a graph with a large number of nodes and edges. One type of modules that benefit the user when filtered out are the *omnipresent modules* which contribute little to the understanding of the architecture of the system, and heavily clutter the view [27].

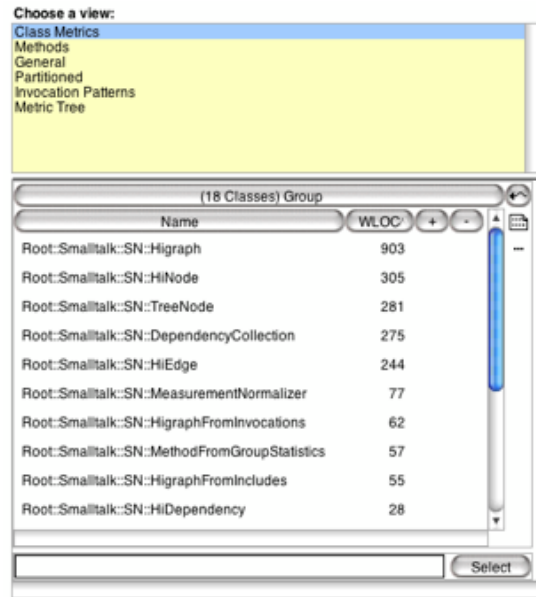
### 5.2. Details-on-Demand

The elements of every architectural view in Softwareaut are nodes that represent modules and edges that represent relationships between them. For the user to be confident that he *understands* such a view he needs to understand the role of every node and the meaning of every edge. Metaphorically, if the view were a phrase, the nodes would be the nouns and the edges would be the verbs. Only when one has understood all the nouns and the verbs he has really understood the story of the view.

The Detail panel of Softwareaut is one of the ways in which one can understand the individual nodes and edges in an architectural view; it presents different views depending on the selected element in the exploration view.

#### 5.2.1. Detail Views for Modules

The detail views for modules present various aspects of a selected module. Figure 6 presents the *Class Metrics* detail view which lists all the classes contained in a module together with metrics for them. The set of displayed metrics is customizable.



(18 Classes) Group	
Name	WLOC
Root:Smalltalk:SN:Higraph	903
Root:Smalltalk:SN:HiNode	305
Root:Smalltalk:SN:TreeNode	281
Root:Smalltalk:SN:DependencyCollection	275
Root:Smalltalk:SN:HiEdge	244
Root:Smalltalk:SN:MeasurementNormalizer	77
Root:Smalltalk:SN:HigraphFromInvocations	62
Root:Smalltalk:SN:MethodFromGroupStatistics	57
Root:Smalltalk:SN:HigraphFromIncludes	55
Root:Smalltalk:SN:HiDependency	28

Figure 6: The Class Metrics view shows the list of classes in a module. The classes are sorted based on their size and the user can navigate to the source of individual classes.

### 5.2.2. Detail Views for Relationships

The detail views for relationships present information about the selected relationships. Since understanding the relationships is critical for understanding the view Softwarenaut provides a broad set of detail views for relationships which cover both structural and evolutionary aspects of the relationships [28, 29].

Figure 7 presents two such examples: to the left is the Invoked Artifacts view and to the right is the Interactive Methods view.

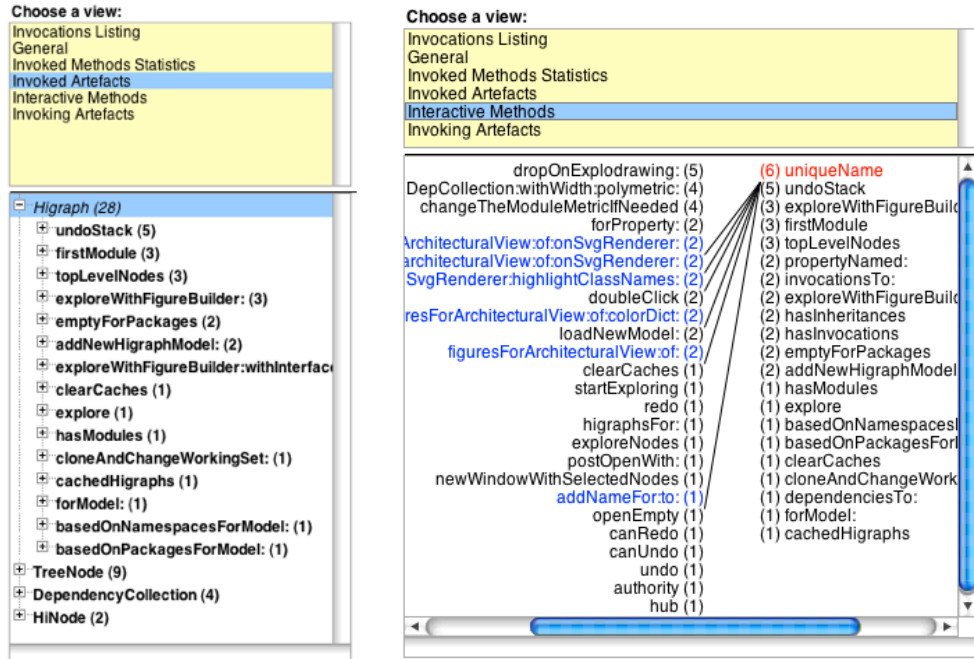


Figure 7: Two example detail views for a relationship: (left) the Invoked Artifacts detail view; (right) the Interactive Methods detail view

The Invoked Artefacts view lists all the artefacts invoked by the selected high-level dependency. The tree has on its first levels the names of the invoked classes. The children of every class node are the invoked methods. The method nodes can be further expanded and the call sites will be revealed.

The Interactive Methods detail view presents the call graph of the method calls abstracted in the selected high-level dependency.

### 5.3. Rule-Based Filtering

In the previous section we have introduced the Filter operation which works on explicit sets of nodes. Most software architecture recovery tools offer such a filter operation [30]. Softwarenaut implements several categories of more advanced *rule-based filters* for nodes and edges. At the most basic level, Softwarenaut supports two types of filters: low-level filters and high-level filters:

- Low-level filters act on the higraph itself. They remove from the higraph the low-level elements that match a given condition. For example, all the invocation relationships that go to polymorphic classes.
- High-level filters act on the high-level elements and relationships between them in the working set. For example, hiding all the high-level dependencies that abstract few low-level dependencies.

During exploration, the user interacts mostly with the High-level filters. There are two types of filters that apply to both artifacts and relationships:

- Metric-based filters for entities and relationships are defined with respect to the metrics computed for artifacts. For example filtering out *the weak dependencies* or *the small modules* in a view.
- Type-based filters for entities and relationships are defined with respect to the type of the artifacts. For example showing *only inheritance relationships* or hiding *all the classes* from a view.

Softwarenaut also implements several advanced filters only for relationships:

- Evolutionary filters are defined based on the historical evolution of an inter-module relationship in the system<sup>3</sup>. For example showing only the *relationships that existed in all the versions of a system* or hiding all the *unstable relationships* [29].
- Directional filters are defined based on the direction of the relationship between two modules. For example, filtering out the unidirectional relationships from an architectural view is useful for highlighting the modules that have mutual dependencies between themselves.

---

<sup>3</sup>Evolution-based filters require models of multiple versions of a system loaded

Figure 8 presents the relation filtering panel as implemented in Software-naut. Several filters can be combined to obtain more powerful ones with either the “and” or the “or” operator. The elements that match the filter can be either “shown” or “hidden”. The user can define new filters by writing simple scripts in Smalltalk. The system is fully reflective, and as soon as a new filter is defined, it will immediately appear in the list of available filters.

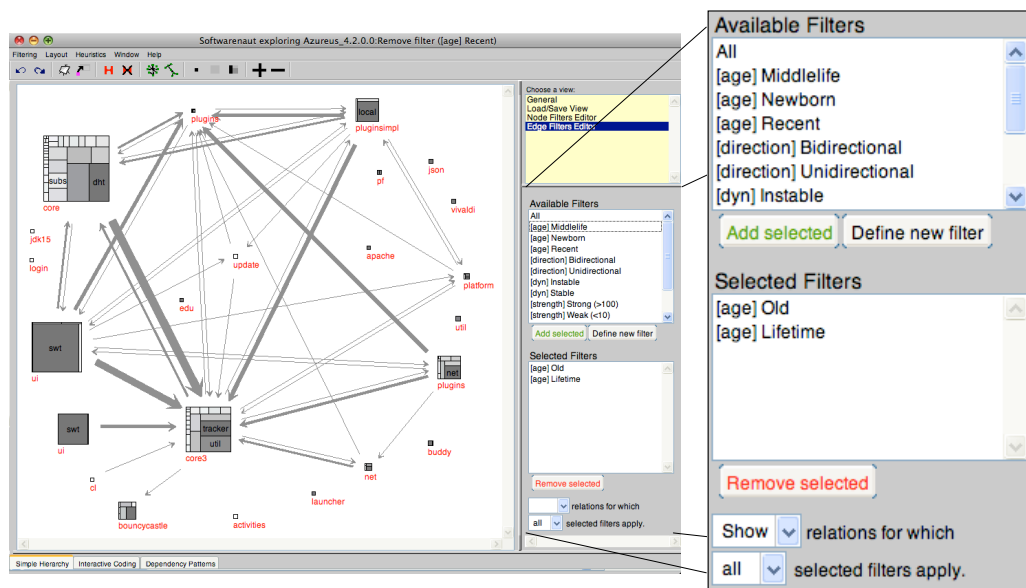


Figure 8: The UI for applying relationship filters in Softwrenaut

Rule-based relationships are a powerful way of reducing information in the view. The principle is simple: not all the relationships are equally relevant for the task at hand. When analyzing a system with a specific goal, the analysis will focus first on those relationships that are most relevant for the chosen goal. Two different goals are architecture recovery and architecture quality assessment:

- When recovering the architecture of a system, the lifetime relationships are more relevant. They represent the architectural backbone of the system and their stability over time insures that it is worth analyzing them first.
- When assessing the quality of an architecture, the recent relationships

are of higher interest. Since they were recently introduced, they are more likely to be contrary to the original intended architecture. They might be the result of architectural decay or of changes to the system performed by new developers that are unaware of the architecture. Continuously monitoring these relationships can be a good quality assurance policy.

Figure 9 presents two views on the same modules that are also present in Figure 8 but with age-based filters activated.

- The left side presents only the Lifetime Relationships - 21 relationships that existed between the displayed modules in all the versions of the system.
- The right side presents only the Recent Relationships - 36 relationships introduced in the system in the latest version.

Both the numbers are very low in comparison with the total number of relationships that are present in the last version of the system; both function as powerful filters.

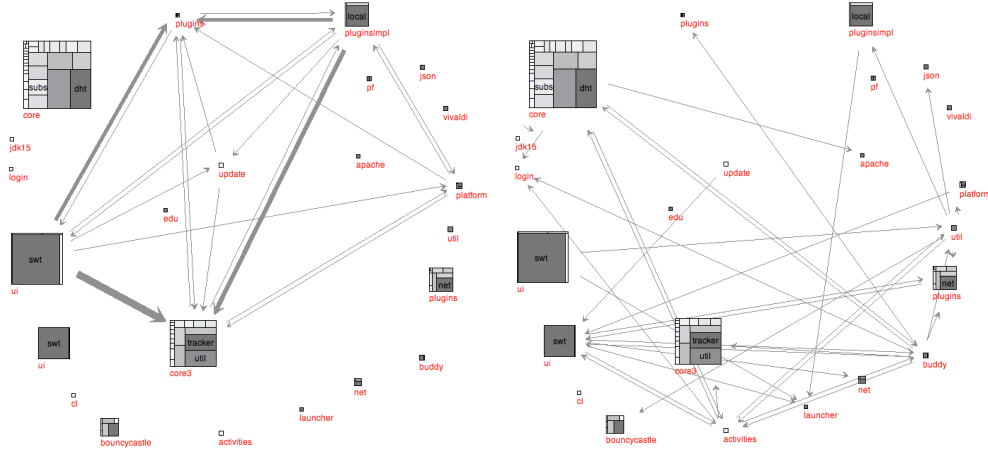


Figure 9: The left part of the figure shows the Lifetime Relationships while the right part of the figure presents the Recent Relationships in Azureus/Vuze

#### 5.4. First-Class Views

The designers of the system use multiple diagrams for the specification of the architecture of a system. Also the reverse engineers will need to recover multiple views when trying to understand the architecture of the system. If it was possible to present all the aspects of the architecture of a system in a single view such a view would overload the user with too much information.

To avoid this cognitive overload, architecture recovery tools usually support some kind of semantic zooming [13]. We address this problem by directing the analysis towards the recovery of multiple architectural views. The technical detail that allows this is view persistence: in SoftwareNaut one can save and restore any view during the exploration. This this allows for a *divide et impera* approach to architecture recovery and representation: each time a view risks becoming too complex the user saves it and the further exploration focuses only on a sub-part of the view.

View persistence requires six chunks of information:

1. The name of the system under analysis
2. The version of the system
3. The current working set with the positions of all the nodes in it
4. The active node and edge filters, both explicit and rule based
5. The creator of the view
6. A name and a description of the view

The model of the system is not saved together with the view. We assume that model construction is deterministic and the name and version of the system will suffice for model reconstruction at a later time.

Figure 10 highlights the way the user can interact with the local views. He has a set of local views that he can save, load, delete locally.

## 6. Collaboration

Once published, a given version of a system never changes. It makes therefore sense to publish all the analyses regarding that version, such that when other users analyze the same version or system they can benefit from the previous work. In the context of architectural view recovery this means discovering architectural views that others have already defined. To support sharing architectural information we have created a *Global Architectural View Repository* (GVR) – a public repository that indexes architectural views<sup>4</sup>.

As soon as a user starts the analysis of a given system Softwareonaut checks the available view repositories and retrieves the list of views that are already defined for that system and/or version.

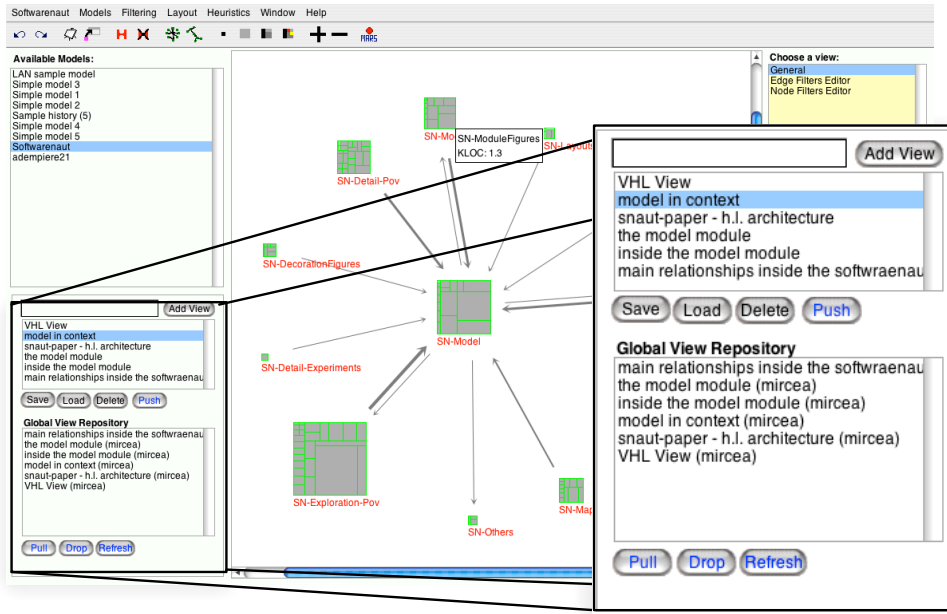


Figure 10: Views are first-class entities in Softwareonaut. They can be saved, deleted locally, but also published and retrieved from public or private repositories

Figure 10 presents the view UI in Softwareonaut. The top part of the inset

<sup>4</sup> Organizations can create their own private view repositories if privacy is a concern



presents the locally stored views which the user can loaded, delete, or push to the GVR. The bottom part presents the views which exist for the given system in the GVR. From the global view repository he can pull views in the local repository, or if he is the creator of such a view he can delete it from the global repository too. The view sharing mechanism is the latest addition to Softwarenaut and we are still waiting to test it with users.

### 6.1. Softwareonaut Synergies

One of the tools that benefits from the Global View Repository is the Small Project Observatory (SPO), an ecosystem analysis tool that we have introduced elsewhere [31]. SPO works at an abstraction level above the architectural level of individual systems: the *ecosystem abstraction level* [32].

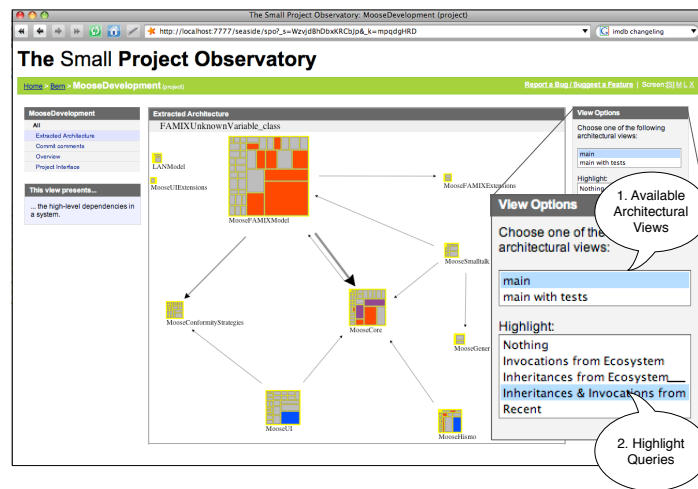


Figure 11: SPO imports architectural views saved in Softwarenaut

SPO needs to support navigation between the two abstraction levels to support the understanding of the ecosystem abstraction level. When navigating from the ecosystem abstraction level down to the architectural level SPO must present architectural views of the individual systems. When they are available, the architectural views of the individual systems are obtained from the Global View Repository. SPO can therefore reuse architectural information generated with Softwarentaut.



## 7. Discussion

### 7.1. Evaluating the Usability and Usefulness of the Tool

We have tested the Softwareonaut tool with students in the Software Evolution master course at the University of Lugano. In a qualitative experiment we asked the students to analyze a large software system they have never seen before with the help of Softwareonaut and to produce an architectural report. We also asked them to fill a post-analysis questionnaire. Our goal in organizing the experiment was to test Softwareonaut and get feedback on both the usability of the tool and its usefulness for architecture recovery in the context of large software systems.

To make the assignment more engaging for the students we grouped them in teams of two. After the students submitted the reports, we analyzed them ourselves and found them to be of various detail and quality. We observed that every group came with a different perspective and with a set of different views on the system. We considered this to be an argument for supporting multiple views in architecture recovery. From our experiments we could conclude that “*architecture is in the eye of the beholder*” just as quality [8].

By analyzing the post-experiment questionnaire we made several observations:

- Most of the students found the tool easy to use
- Most of the students considered that the reports they generated were reliable and they got a better understanding of the system by using Softwareonaut
- The features that the students found most useful in their analysis were the details-on-demand views, and the predefined filters for both nodes and edges.
- The features that the students thought were missing were: undo/redo facilities, parametrizable filters, arbitrary and rule-based grouping of elements<sup>5</sup>

In the future we plan to empirically validate the usefulness of collaboration in architecture recovery and its implementation in the GVR.

---

<sup>5</sup>Since then we have implemented most of the features that the students requested.

### *7.2. Integration with the Moose Analysis Platform*

We built Softwareaut on top of the Moose Analysis Platform [33]. The main feature that the tool uses from Moose is the FAMIX Core meta-model for representing object-oriented systems. The reliance of Softwareaut on the FAMIX Core meta-model allows it to be independent of the programming language and analyze systems written in any language as long as there is a fact-extractor for that particular language.

Softwareaut makes use of several of the views defined in Moose for the detail panel (e.g. the views that present entity metrics). Also, since behind any visual element of Softwareaut lays a HiNode and behind it there is a FAMIX entity one can spawn other Moose analyses by selecting any of the elements of a Softwareaut architectural view.

### *7.3. Beyond Architecture Recovery(I): Reengineering*

One of the tools built on top of Softwareaut is MARS: an automated architecture refactoring recommender tool [34]. The tool starts from a given Softwareaut view and checks whether move operations applied on classes can improve the architecture of the system by increasing coupling and decreasing cohesion. Preliminary results show a good recall but a low precision of the automated refactoring recommendations generated by MARS. We plan to further study this phenomenon.

### *7.4. Beyond Architecture Recovery (II): Monitoring Software Evolution*

One of our future projects is exploring ways in which the recovered views can function as a live documentation of an evolving system. The Softwareaut views are not simple pictures but instead they code relationships between the modules of the system. A view recovered for a given version of the system can function as a reference point for presenting the future evolution of the system. After the publication of a new version of the system, Softwareaut can automatically detect the views affected by the new changes, and visualize these changes on them.

### *7.5. Installation and Documentation*

Softwareaut is written in the Smalltalk programming language and is released under the open source MIT Licence. The tool is available online at <http://www.inf.usi.ch/phd/lungu/softwareaut/>. On the homepage of the tool there are a set of screencasts that present its various features together with other documentation, installation instructions, and instructions on how to obtain the source code.

## 8. Related Work

There is an extended tradition of architecture recovery tools in software engineering research. Pollet et al. have presented a comprehensive overview of the work in architecture recovery in their survey article [6]. In this section we take several of the core aspects of SoftwareNaut and we discuss how they are similar and how they differ from other state of the art tools.

### 8.1. Exploration and Navigation.

Automatically aggregating the low-level relations, and then letting the user navigate from the highest abstraction level downwards is the main interaction approach in SoftwareNaut. This is the exact opposite of the approach that Müller proposed with Rigi [35]. In their case, the user starts from the lowest-level facts and aggregates them as he climbs up in the abstraction hierarchy. Their approach does not scale when analyzing very large systems because the number of low-level artifacts is too large. Storey took the same top-down navigation approach in her work on SHriMP [13].

### 8.2. Filtering.

As far as we are aware we are the only ones to propose filters based on evolutionary aspects of the relationships in the context of software exploration. However, there are two related works. The first is Wierda et al. who recover the architectural decomposition of a system through clustering; they observe that if they use for clustering only those dependencies that were in the system in both the first and the last versions, the decompositions are more precise [36]; this observation supports our approach of using the lifetime relationships as more architecturally relevant than the other relationships.

The second related work is of Abram Hindle et al. [37] who introduce the YARN visualization prototype which animates the evolution of dependencies between the modules of a system. The main difference between our approach and theirs is that we work on a snapshot-based model and they work on a commit-based model. Their commit-based model is advantageous since they benefit from more detailed information about the system; the disadvantage is that the animation of all the commits is time consuming. They do not support a query mechanism for visualizing only special types of relations.

### 8.3. *First-Class Views and Collaboration.*

The work of Storey et al. on Shrimp also allows for saving and restoring views [38]. The views are saved inside a “Filmstrip” which is persistent. Through the intermediation of the filmstrips the users can restore exploration sessions or even share certain views. This type of information allows people that know about each other to share information by emailing the files. The advantage of the Global View Repository is that it allows for discovering information that other users have discovered and about which the analysis is not aware.

Relevant for the collaborative reverse engineering part of our work is the Churrasco work of D'Ambros et al. [39]. Churrasco supports software evolution modeling, visualization and analysis through a web interface. Through an example scenario they show that Churrasco allows for collaborative software evolution analysis, and they attribute this to the availability on the web of the tool. The collaboration happens by annotating the various elements in the views of Churrasco. However, their tool presents a set of predefined views and they are not that much architectural views but rather design-level views.

One project developed with collaboration support as the main goal is the Jazz IDE of IBM [40]. The goal of the Jazz “collaborative development environment” is to enhance and enrich collaboration in small, informal software development teams. Jazz has several features to support awareness of team member activities in addition to screen sharing such as local history of chats that are anchored in the code providing therefore context to the discussions. The main difference between their work and ours is the goal of the project.

## 9. Conclusions and Future Work

In this article we have presented Softwrenaut, our tool which is state of the art in architecture recovery. The tool allows the recovery of architectural views of a software system and supports collaboration and integration with other tools by publicly sharing architectural views.

**Acknowledgements.** We would like to thank Fabrizio Perin for feedback on this paper.

## References

- [1] M. Lehman, Programs, life cycles, and laws of software evolution, *Proceedings of the IEEE* 68 (Sept. 1980) 1060–1076.
- [2] S. Ducasse, D. Pollet, Software architecture reconstruction: A process-oriented taxonomy, *IEEE Transactions on Software Engineering* 35 (2009) 573–591.
- [3] D. E. Perry, A. L. Wolf, Foundations for the study of software architecture, *SIGSOFT Softw. Eng. Notes* 17 (1992) 40–52.
- [4] D. Garlan, R. Allen, J. Ockerbloom, Architectural mismatch: Why reuse is still so hard, *IEEE Softw.* 26 (2009) 66–69.
- [5] C. Riva, Reverse architecting: An industrial experience report, in: *WCRE '00: Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*, IEEE Computer Society, Washington, DC, USA, 2000, p. 42.
- [6] D. Pollet, S. Ducasse, L. Poyet, I. Alloui, S. Cîmpan, H. Verjus, Towards a process-oriented software architecture reconstruction taxonomy, in: R. Krikhaar, C. Verhoef, G. Di Lucca (Eds.), *Proceedings of 11th European Conference on Software Maintenance and Reengineering (CSMR'07)*, IEEE Computer Society, 2007. Best Paper Award.
- [7] M. Jazayeri, On architectural stability and evolution, in: *Reliable Software Technologies-Ada-Europe 2002.*, Springer, 2002.
- [8] L. Bass, P. Clements, R. Kazman, *Software Architecture in Practice*, Addison-Wesley Professional, 1997.
- [9] P. Kruchten, The 4+1 view model of architecture, *IEEE Softw.* 12 (1995) 42–50.
- [10] C. Hofmeister, R. Nord, D. Soni, *Applied Software Architecture*, Addison-Wesley, 2000.
- [11] G. Murphy, D. Notkin, K. Sullivan, Software reflexion models: Bridging the gap between source and high-level models, in: *Proceedings of SIGSOFT '95, Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ACM Press, 1995, pp. 18–28.
- [12] H. Muller, K. Klashinsky, Rigi: a system for programming-in-the-large, *Software Engineering*, 1988., *Proceedings of the 10th International Conference on* (1988) 80–86.
- [13] M.-A. D. Storey, H. A. Müller, Manipulating and documenting software structures using SHriMP Views, in: *Proceedings of ICSM '95 (International Conference on Software Maintenance)*, IEEE Computer Society Press, 1995, pp. 275–284.

- [14] M. Lungu, M. Lanza, Softwrenaut: Exploring hierarchical system decompositions, in: Proceedings of the 10th European Conference on Software Maintenance and Reengineering (CSMR'06), pp. 349–350.
- [15] M. Lungu, M. Lanza, T. Gîrba, Package patterns for visual architecture recovery, in: Proceedings of CSMR 2006 (10th European Conference on Software Maintenance and Reengineering), IEEE Computer Society Press, Los Alamitos CA, 2006, pp. 185–196.
- [16] M. Lanza, S. Ducasse, Polymetric views - a lightweight visual approach to reverse engineering, *Software Engineering, IEEE Transactions on* 29 (2003) 782–795.
- [17] M. Lanza, R. Marinescu, *Object-Oriented Metrics in Practice*, Springer-Verlag, 2006.
- [18] B. Shneiderman, The eyes have it: A task by data type taxonomy for information visualizations, in: *IEEE Visual Languages*, College Park, Maryland 20742, U.S.A., pp. 336–343.
- [19] K. Wong, *The reverse engineering notebook*, Ph.D. thesis, University of Victoria, Victoria, B.C., Canada, Canada, 2000.
- [20] M.-A. D. Storey, D. Čubranić, D. M. German, On the use of visualization to support awareness of human activities in software development: a survey and a framework, in: *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization*, ACM, New York, NY, USA, 2005, pp. 193–202.
- [21] S. Tichelaar, *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*, Ph.D. thesis, University of Bern, 2001.
- [22] R. Koschke, Atomic architectural component recovery for program understanding and evolution, *Software Maintenance, 2002. Proceedings. International Conference on* (2002) 478–481.
- [23] M. Lungu, A. Kuhn, T. Gîrba, M. Lanza, Interactive exploration of semantic clusters, in: *3rd International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2005)*, pp. 95–100.
- [24] P. F. Mihancea, G. Ganeva, I. Verebi, C. Marinescu, R. Marinescu, Mcc and mc#: Unified c++ and c# design facts extractors tools, *Symbolic and Numeric Algorithms for Scientific Computing, International Symposium on* 0 (2007) 101–104.
- [25] D. Harel, On visual formalisms, in: J. Glasgow, N. H. Narayanan, B. Chandrasekaran (Eds.), *Diagrammatic Reasoning*, The MIT Press, Cambridge, Massachusetts, 1995, pp. 235–271.
- [26] G. G. Robertson, J. D. Mackinlay, S. K. Card, Cone trees: animated 3d visualizations of hierarchical information, in: *CHI '91: Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM Press, New York, NY, USA, 1991, pp. 189–194.

- [27] B. S. Mitchell, S. Mancoridis, On the automatic modularization of software systems using the bunch tool, *IEEE Trans. Softw. Eng.* 32 (2006) 193–208.
- [28] M. Lungu, M. Lanza, Softwrenaut: cutting edge visualization, in: *SoftVis '06: Proceedings of the 2006 ACM symposium on Software visualization*, ACM, New York, NY, USA, 2006, pp. 179–180.
- [29] M. Lungu, M. Lanza, Exploring inter-module relationships in evolving software systems, in: *Proceedings of CSMR 2007 (11th European Conference on Software Maintenance and Reengineering)*, IEEE Computer Society Press, Los Alamitos CA, 2007, pp. 91–100.
- [30] I. Aracic, T. Schaeffer, M. Mezini, K. Osterman, A Survey on Interactive Grouping and Filtering in Graph-based Software Visualizations, Technical Report, Technische Universität Darmstadt, 2007.
- [31] M. Lungu, T. Gîrba, M. Lanza, The small project observatory: Visualizing software ecosystems, *EST special issue of the Science of Computer Programming*. DOI:10.1016/j.scico.2009.09.004 (2009).
- [32] M. Lungu, Reverse Engineering Software Ecosystems, Ph.D. thesis, University of Lugano, 2009.
- [33] O. Nierstrasz, S. Ducasse, T. Girba, The story of moose: an agile reengineering environment, *SIGSOFT Softw. Eng. Notes* 30 (2005) 1–10.
- [34] A. Boeckmann, MARS - Modular Architecture Recommendation System, Bachelor's thesis, University of Lugano, 2010.
- [35] H. A. Müller, S. R. Tilley, M. A. Orgun, B. D. Corrie, N. H. Madhavji, A reverse engineering environment based on spatial and visual software interconnection models, in: *SDE 5: Proceedings of the fifth ACM SIGSOFT symposium on Software development environments*, ACM, New York, NY, USA, 1992, pp. 88–98.
- [36] A. Wierda, E. Dortmans, L. Lou Somers, Using version information in architectural clustering - a case study, in: *CSMR '06: Proceedings of the Conference on Software Maintenance and Reengineering*, IEEE Computer Society, Washington, DC, USA, 2006, pp. 214–228.
- [37] A. Hindle, Z. M. Jiang, W. Kuleilat, M. W. Godfrey, R. C. Holt, Yarn: Animating software evolution, *Visualizing Software for Understanding and Analysis, International Workshop on V* 0 (2007) 129–136.
- [38] D. Rayside, M. Litoiu, M.-A. Storey, C. Best, R. Lintern, Visualizing flow diagrams in websphe studio using shrimp views, *Information Systems Frontiers* 5 (2003) 161–174. 10.1023/A:1022649506310.



- [39] M. D'Ambros, M. Lanza, A flexible framework to support collaborative software evolution analysis, in: Proceedings of CSMR 2008 (12th European Conference on Software Maintenance and Reengineering), IEEE Computer Society, 2008, pp. 3–12.
- [40] S. Hupfer, L.-T. Cheng, S. Ross, J. Patterson, Introducing collaboration into an application development environment, in: Proceedings of the 2004 ACM conference on Computer supported cooperative work, CSCW '04, ACM, New York, NY, USA, 2004, pp. 21–24.