

Softwareonaut: Explorative Software Architecture Recovery

Mircea Lungu

Software Composition Group - University of Bern, Switzerland

Michele Lanza

REVEAL @ Faculty of Informatics - University of Lugano, Switzerland

Abstract

When the initial architecture of a system has eroded the only solution is architecture recovery. However, when the system under analysis is large, the user must use dedicated tools to support the recovery process. In this article we present Softwareonaut – a tool which supports architecture recovery through interactive exploration and visualization. The tool provides overview and detailed perspectives which support the architecture recovery process as well as powerful navigation primitives and filtering mechanisms that allow managing the complexity of large software systems. The recovered architectural views can be shared between users through a Global Architectural View Repository allowing collaborative architecture recovery.

Keywords: Architecture Recovery, Visualization, Reverse Engineering

1. Introduction

No software system is an island. Instead a 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 a system is critical for its maintenance and evolution [2, 3].

As a system evolves, the architecture erodes [4] and an architectural mismatch appears between the *as-defined* and *as-is* architecture [5]. 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 [4]. The main reason for 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 [6].

When the drift and erosion have brought the system architecture too far from the initial state, the solution is to recover the architecture of the system from the source code. Jazayeri defined architecture recovery as “*the techniques and processes used to uncover a system’s 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 can be automated (e.g., fact extraction, view generation), no tool works completely without human intervention. In some cases the user has to group related artifacts together based on their similarity of purpose [12]. In other cases the user has to compare the architecture as-extracted with the architecture as-predicted [11] or the user has to decide which exploration paths to follow [13].

We present Softwareonaut [14, 15] - the architecture recovery tool that we have developed. Softwareonaut 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. Section 2 is an overview of Softwareonaut. The following two sections discuss two important phases in the workflow of any architecture recovery tool: information aggregation (Section 3) and interactive exploration (Section 4). In Section 5 we discuss architectural considerations and in Section 6 we discuss the tool-building experience. In Section 7 we present related work. In Section 8 we conclude and outline future directions.

2. Softwarenaut in a Nutshell

Softwarenaut is an architecture recovery tool based on interactive exploration, with the goal of discovering architecturally relevant views [15].

The main interaction approach in Softwarenaut is to let the user navigate from an automatically aggregated high-level view on the system downwards, also known as *top-down exploration*.

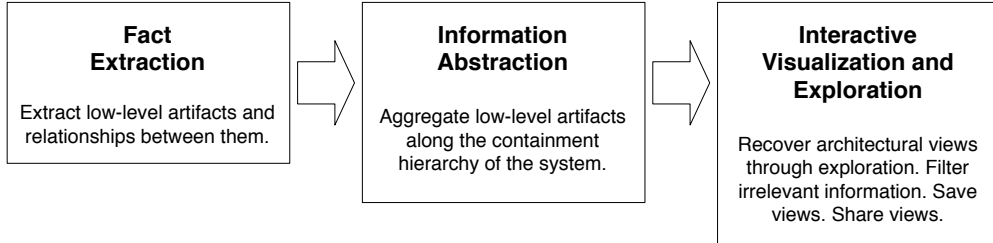


Figure 1: Softwarenaut supports an extract-abstract-view workflow model

Like other architecture recovery tools [3], Softwarenaut conforms to a classical extract-abstract-view architecture. Figure 1 presents the three main steps of this architecture. In each of the three steps the tool offers state of the art features:

- *Fact Extraction.* The tool can analyze any object-oriented system through a system of pluggable fact extractors which export information to a well-known meta-model.
- *Information Aggregation.* The tool takes into account the hierarchical decomposition of a system as a basis for aggregating artifacts and relationships. In the case of a missing hierarchical decomposition the tool can automatically generate one.
- *Interactive Visualization and Exploration.* The tool allows for an overview, zoom and filter, and details on demand approach. The tool provides a powerful set of filtering mechanisms. The recovered architectural views can be shared between users, and reused by other tools.

The article continues with detailing the second and third step, the first one being discussed later in the section on architectural concerns regarding the tool.

3. Information Abstraction

Aggregating horizontal relationships between software artifacts along the containment relationships is the fundamental technique that enables abstraction in Softwareaut.

Figure 2 exemplifies the aggregation of low-level relationships between methods and classes along the containment hierarchy of a Java system.

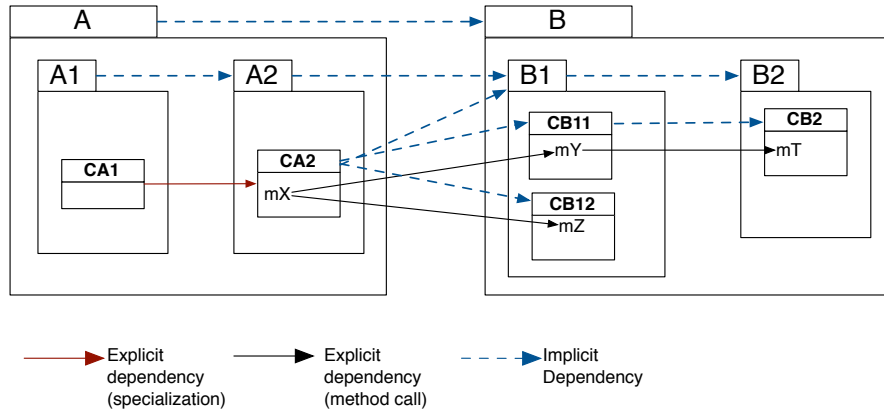


Figure 2: The aggregation of explicit dependencies into implicit ones along the containment relationships between methods, classes, and packages in a Java system

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.

A data structure that allows for this is the Higraph introduced by Harel [16]. In our case, the DPMHigraph¹ 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 3 presents two types of abstract entities and two types of abstract relations:

¹DPM stands for Detail Project Model. For the importance of the concept see [32]

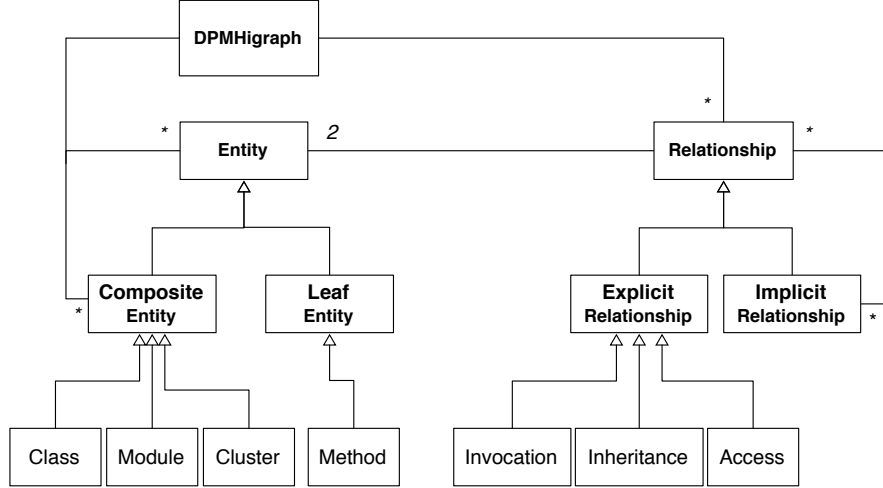


Figure 3: The DPMHigraph model is the core representation of a system in Softwareaut

- *Leaf Entities.* They are the basic object-oriented programming building blocks used for the structuring of software. Depending on the analyzed relationship, 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. Softwareaut 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.

4. Interactive Exploration

The UI of Softwarenaut contains three linked complementary visual perspectives that present information about a system during the exploration. Figure 4 presents Softwarenaut visualizing an architectural view of itself. The linked complementary perspectives support the “overview first, zoom and filter, and details on demand” principle of information visualization [17].

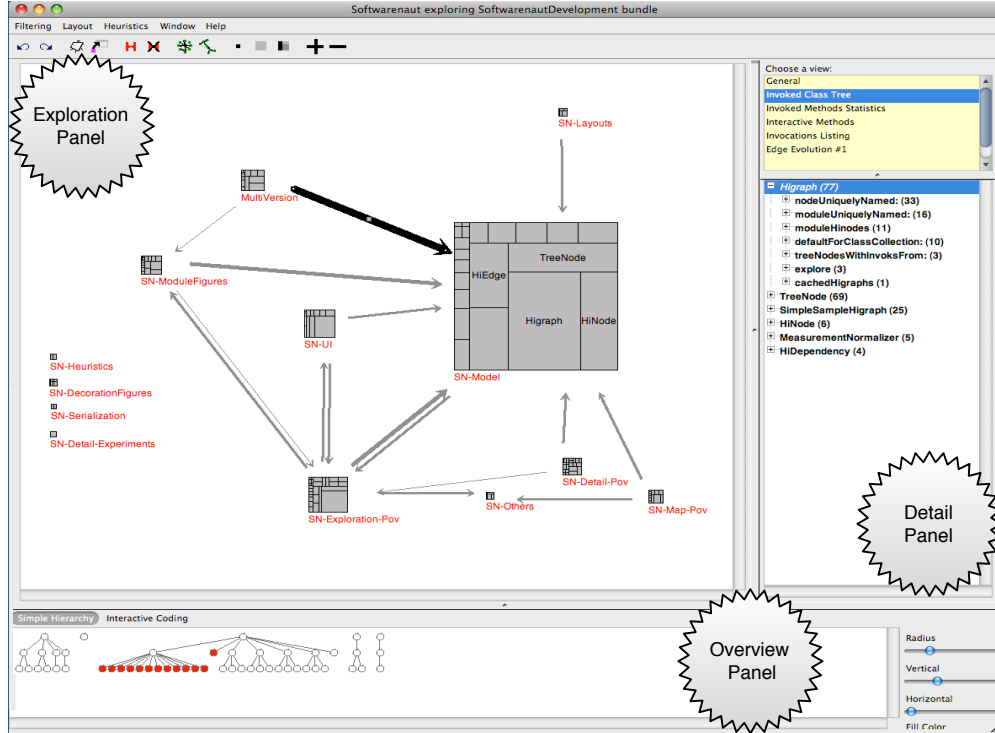


Figure 4: Linked views present complementary perspectives on a software system

The three complementary views that the tool supports:

1. The *Exploration* panel is Softwarenaut’s main view. 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 here we use Lanza’s polymetric approach [18, 19].

2. The *Detail* panel presents details for an entity selected in the Exploration panel. The goal is to supplement details of the element selected in the exploration panel. The detail panel implements the “details-on-demand” part of Shneiderman’s visualization mantra [17].
3. The *Overview* panel presents the entire hierarchy of a system and highlights the modules that are visible in the exploration panel. The Overview panel presents a horizontal slice through a system [20], offering an orientation aid which is critical for successful navigation [21].

In this section we detail the interactive techniques that support the exploration process in Softwareonaut: navigation (4.1), details-on-demand (4.2), rule-based filtering (4.3) and first-class views and collaboration (4.4).

4.1. Navigation

The dominant exploration mechanism of Softwareonaut 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 [22]. At any 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 high-level nodes. As the user explores the system he transforms the working set by performing exploration operations on it and thus changes the contents of the view in the exploration view.

The exploration operations supported by Softwareonaut are:

- *Expand*. The expand operation applied to a node of the working set 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 with their parent module. We define the operation 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 .

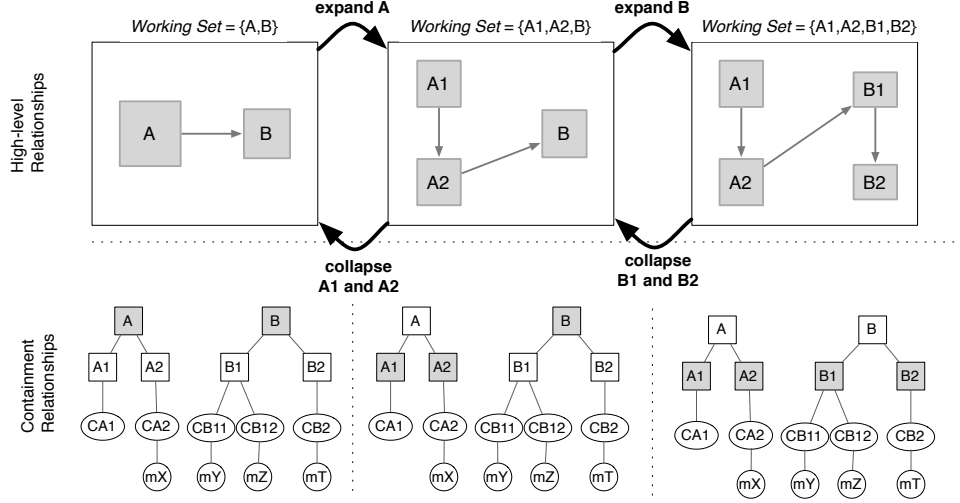


Figure 5: The expand and collapse complementary operations that allow vertical navigation in the Higraph

- *Filter*. The filter operation applied to a node removes that node from the working set. We define the operation 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 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. He 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 large graphs. 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 [23].

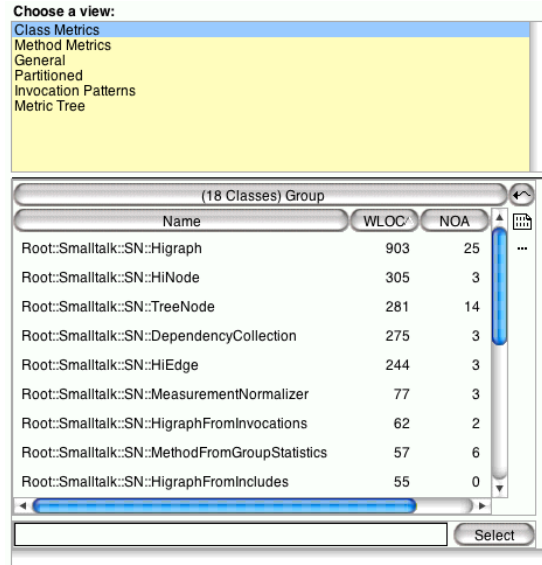
4.2. Details-on-Demand

The elements of every architectural view in Softwarenaut 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 Softwarenaut 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.

4.2.1. 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 a customizable set of metrics for them.



(18 Classes) Group		
Name	WLOC	NOA
Root::Smalltalk::SN::Higraph	903	25
Root::Smalltalk::SN::HiNode	305	3
Root::Smalltalk::SN::TreeNode	281	14
Root::Smalltalk::SN::DependencyCollection	275	3
Root::Smalltalk::SN::HiEdge	244	3
Root::Smalltalk::SN::MeasurementNormalizer	77	3
Root::Smalltalk::SN::HigraphFromInvocations	62	2
Root::Smalltalk::SN::MethodFromGroupStatistics	57	6
Root::Smalltalk::SN::HigraphFromIncludes	55	0

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.

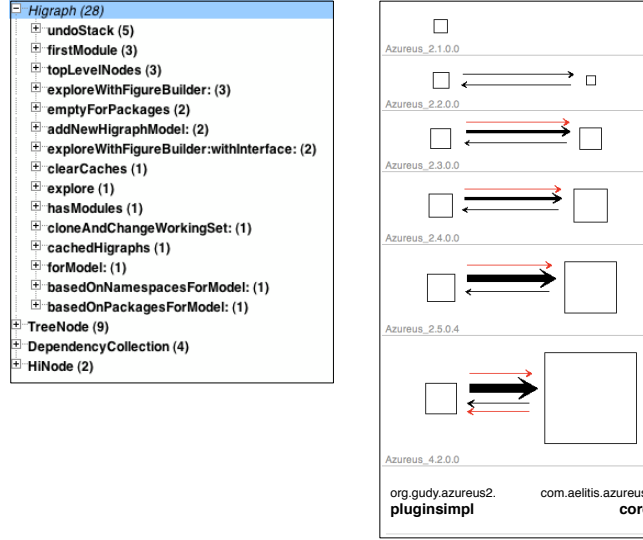


Figure 7: Relationship details: Invoked Artifacts (left) and Evolution Filmstrip (right)

4.2.2. 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, Softwrenaut provides a broad set of detail views for relationships which cover both structural and evolutionary aspects of the relationships [24, 14]. Figure 7 presents two such interactive detail views:

The *Invoked Artefacts* view lists all the artefacts invoked by the selected high-level dependency. The tree has on its first level the invoked classes, on the second level invoked methods, and on the third level call sites. The user can navigate to any of the presented code artifacts from this view.

The *Evolution Filmstrip* view presents the evolution of the given relationship in all the versions of the system which are available for analysis [14]. The two modules and the associated relationship are represented in every analyzed version. Figure 7 shows a relationship which changed dramatically during the six versions of the system under analysis.

4.3. 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 [25]. Softwarenaut implements several categories of advanced *rule-based filters* for nodes and edges. Softwarenaut supports two types of basic filters:

1. *Low-level filters* act on the higraph itself. They remove from the higraph the low-level elements that match a given condition, e.g., all the invocation relationships that go to polymorphic classes.
2. *High-level filters* act on the high-level elements and relationships between them in the working set, e.g., 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:

1. *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.
2. *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 advanced filters for relationships:

- *Evolutionary filters* are defined based on the historical evolution of an inter-module relationship in the system². For example showing only the *relationships that existed in all the versions of a system* or hiding all the *unstable relationships* [14].
- *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.

²Evolution-based filters require models of multiple versions of a system loaded

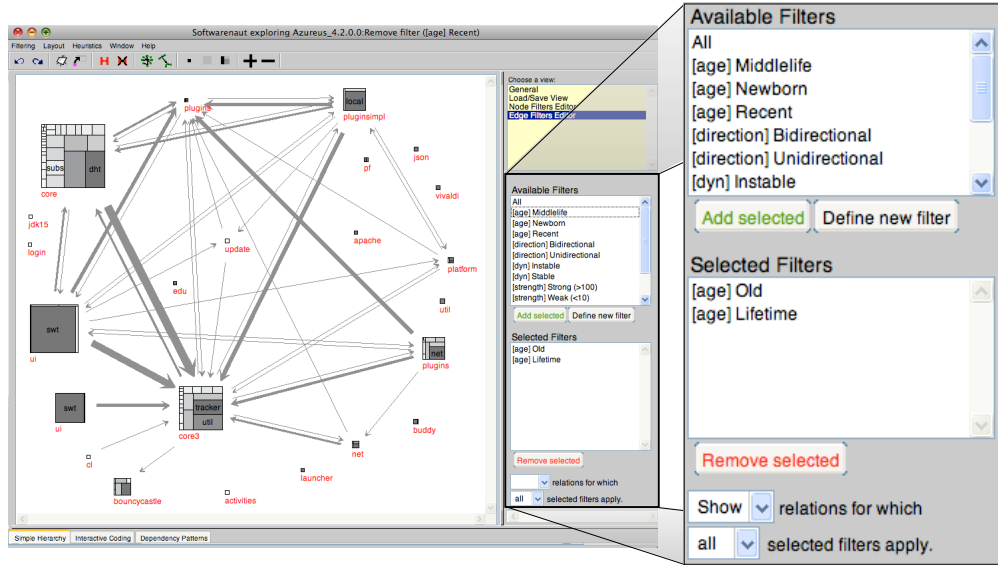


Figure 8: The UI for applying relationship filters in Softwarenaut

Figure 8 presents the relation filtering panel as implemented in Softwarenaut. 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.

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 focuses first on those relationships that are most relevant for the chosen goal. Two different goals are architecture recovery and architecture quality assessment:

- *Architecture Recovery.* 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.

- *Architectural Quality Assessment.* 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.

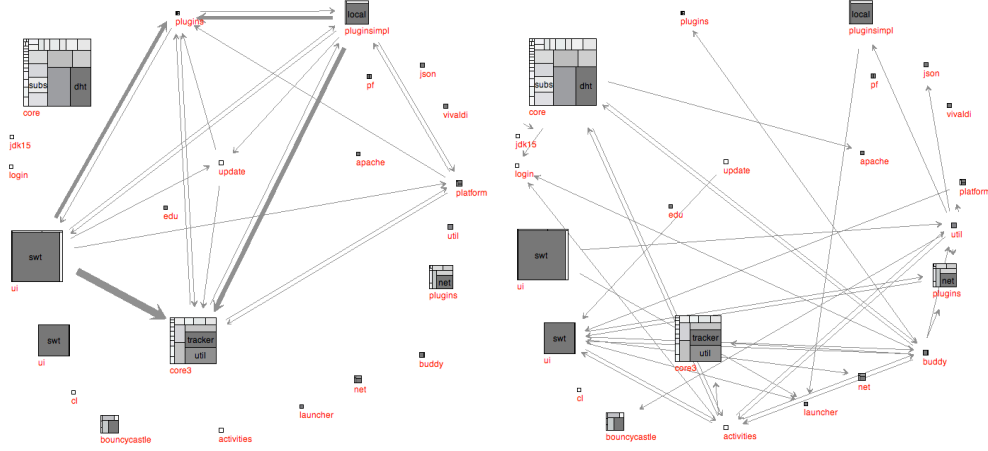


Figure 9: The left part of the figure shows the Lifetime Relationships while the right part of the figure presents the Recent Relationships in Azureus/Vuze

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.

4.4. First-Class Views and Collaboration

The designers of a system use multiple diagrams for the specification of the architecture of a system. Also the reverse engineers need to recover multiple views when trying to understand the architecture of the system. To avoid information 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.

Views are first-class entities in Softwarent: 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. Moreover, views can be shared with other reverse architects.

View persistence requires the following information: The name and version of the system under analysis, the current working set with the positions of all the nodes in it, the active node and edge filters (both explicit and rule based), the creator of the view, and a name and 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 ³.

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 generated with Softwarent. The GVR is implemented as a relational database in PostgreSQL which can be publicly accessed by instances of Softwarent or other tools.

The architectural views as saved in the GVR can serve as the basis for monitoring the future architectural evolution of the system. After the publication of a new version of the system, Softwarent can automatically detect the views affected by the new changes, and present a *visual diff* on top of the

³This requires nevertheless a way of uniquely identifying the entities in the view. In our case, for each of the entities in the working set we save the fully scoped name

view retrieved from the GVR between the state of the system at the moment the view was created and the latest commit.

Figure 10 presents several considerations regarding the interaction with the architectural views in Sofwarenaut:

- 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.
- The top part of the inset presents the locally stored views which the user has in the image. For each of them the user can load it, delete it, or push it to the GVR. In the inset, the buttons which have black text represent operations that can be applied on the local views.
- 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 Softwareonaut.

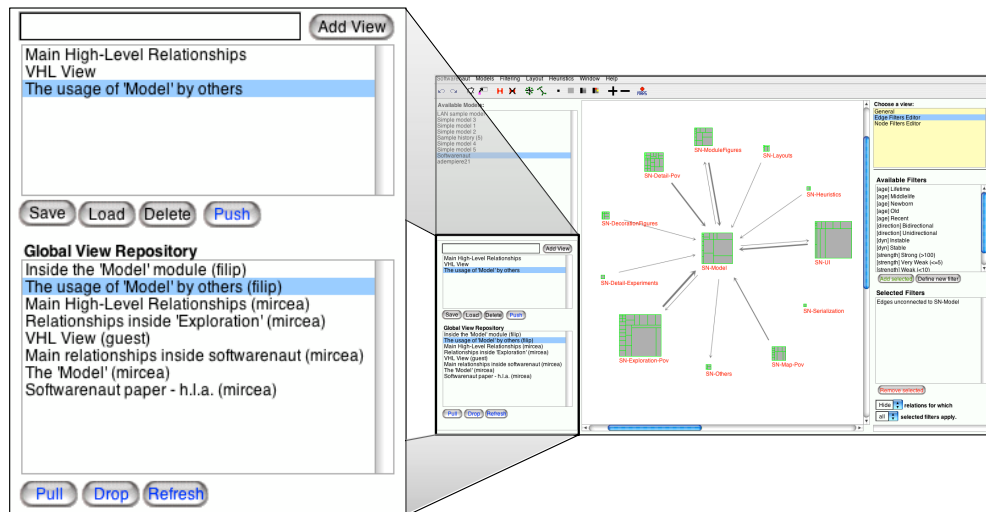


Figure 10: Views are first-class entities in Softwareonaut. They can be saved, deleted locally, but also published and retrieved from the Global Architectural View Repository.

5. Architectural Considerations Regarding Softwarentaut

5.1. Fact Extraction and Modelling

At the lowest level, Softwarentaut models a system using the Core of the FAMIX meta-model [26], a language independent meta-model that describes the static structure of object-oriented software systems. Given that the FAMIX meta-model is language independent, Softwarentaut 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 [27] or inFusion⁴.

FAMIX 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 [26].

A class of relationships which have a special importance in Softwarentaut 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 an 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 [28]. We presented elsewhere an experiment with clustering the classes in a system based on natural language similarity[29].

5.2. Integration with the Moose Analysis Platform

We built Softwarentaut on top of the Moose Analysis Platform [30]. The main feature that the tool uses from Moose is the FAMIX Core meta-model for representing object-oriented systems. Softwarentaut makes uses 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 Softwarentaut lays a HiNode and behind it there is a FAMIX entity one can spawn

⁴See <http://www.intooitus.com/inFusion>, verified Jan 25 2011.

other Moose analyses by selecting any of the elements of a SoftwareNaut architectural view. In the same time, any tool in the Moose platform can spawn a SoftwareNaut architectural analysis on any group of entities which have containment relationships and dependencies between themselves.

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

5.4. History Operations

SoftwareNaut keeps a history of the exploration and filtering actions to support undo and redo operations. This is a requirement for information exploration tools [17] that few other architecture recovery tools implement.

6. Tool-Building Considerations

Softwareonaut served as the research prototype for many of the research ideas we had during the PhD of the first author. As a result, the tool was the basis for a number of research papers [24, 33, 15, 14]. Many of these papers could have not been written if it were not for the Softwareonaut which was a testbed for our ideas.

We have obviously used Softwareonaut to to analyze Softwareonaut itself, and this has determined several re-architecting sessions. However, all throughout the development of the tool we wanted to have a tool that was usable. This allowed us to collaborate with others. We have integrated the tool with others [33, 34, 30] and we provided a framework in which masters projects to be developed [35]. We have also used the tool several times to perform reverse engineering in consulting projects.

In the next section we present a brief discussion on evaluating the tool with students.

6.1. 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 ⁵ they have never seen before with the help of Softwareonaut and to produce an architectural report. 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. To guide their analysis we asked them to perform a series of tasks which are presented in Figure 12. After the students submitted the reports, we analyzed them ourselves and found them to be of various detail and quality. We list here several of the observations that we made after analyzing the reports that the students submitted:

- At question 1 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 architecture being “*is in the eye of the beholder*” just as quality [8].

⁵In this case it was ArgoUML

	Questions
1	Discover one or more architectural views on the system which present modules and their interactions
2	Is there a subset of the modules that you consider to be at the core of the system?
3	Is there a core module in the system? Why? How does it interact with the others?
4	Choose one inter-module dependency in the system and analyze it. What is the reason for its existence?
5	Choose one other module in the system. Analyze its interface.
6	Are there cases in which two modules depend on one another that you would have not expected from the conceptual architecture?
7	Overall what do you think about the structure of the system? Is it well modularized?
8	You want to add support for generating code in a new language. Which module do you change? Which others are impacted? How much time do you need?

Figure 12: The eight tasks the students had to solve during two hours of the software design and evolution lab

- The opinions were divided for question 2 and 3. Each group was ready to argue its position.
- Questions 4, 5, and 6 were not solved by everybody. We think the reason was the lack of sufficient training with the tool as well as some of the limitations that the users reported and which we present later.
- All the groups that answered thought the system was well structured, fact with which we agree. However, one group blamed information overload for not answering.

We also asked the students to fill a post-analysis questionnaire.

By analyzing the post-experiment questionnaire we observed that most of the students found the tool easy to use and most of them considered that the reports they generated were reliable. They estimated that they got a better understanding of the system by using Softwareonaut. The most important feedback for us was related to the features of the tool that they students found useful, or the ones that they wanted to have but did not have:

- 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 (in decreasing order of the frequency of request) were: undo and redo facilities, user-defined filters, arbitrary and rule-based grouping of elements, view persistence.

Since then we have implemented most of the features that the students requested. One of these features is the view persistence and sharing as a means of supporting collaboration. In the future we plan to perform usability experiments that will evaluate the usability and usefulness of collaboration in architecture recovery and its implementation in the GVR by running some more controlled experiments.

6.2. Depending on other research prototypes

Depending on other research prototypes and platforms has been a benefit because we had the opportunity of using cutting edge technology, and building on the shoulders of giants, and in the same time, it made our life harder since the tools that we depended on kept moving “under our feet” and at times they were not maintained anymore. We believe however that the benefits outweighed the difficulties.

6.3. Installation and Documentation

Softwareonaut 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/softwareonaut/>. 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.

7. 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 [3]. In this section we take several of the core aspects of Softwrenaut and we discuss how they are similar and how they differ from other state of the art tools.

7.1. Exploration and Navigation

The first architectural visualization prototype was Rigi, a programmable reverse engineering environment which emphasizes visualization and interaction [36]. Rigi can visualize the data as hierarchical typed graphs and provides a Tcl interpreter for manipulating the graph data. The reconstruction process is based on a bottom-up process of grouping the software elements into clusters by manually selecting the nodes and collapsing them. The approach does not scale when analyzing very large systems because the number of low-level artifacts is too large. This is why in Softwrenaut we automatically aggregating the low-level relations, and then letting the user navigate from the highest abstraction level downwards is the main interaction approach in Softwrenaut.

One of the projects that was inspired by Rigi was the SHriMP tool [13] and its Eclipse-based continuation [37]. SHriMP and Creole display architectural diagrams using nested graphs. Their user interface embeds source code inside the graph nodes and integrates a hypertext metaphor for following low-level dependencies with animated panning, zooming, and fisheye-view actions for viewing high-level structures. The difference between our tool and Creole is that Softwrenaut can also perform evolutionary analysis.

Pinzger proposed the ArchView approach [38] which provides visualizations that present the evolution of the modules in a system. His evolution analysis takes into account the annotations from the versioning system repository. However, there is no support for first-class views in ArchView and the dependencies between the modules are only based on logical coupling.

7.2. 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 [39]; 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. [40] 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.

7.3. First-Class Views and Collaboration

The work of Storey et al. on Shrimp also allows for saving and restoring views [41]. 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 DAmbros et al. [42]. 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 [43]. 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.

8. Conclusions and Future Work

In this article we presented SoftwareNaut, our tool for architecture recovery. The tool allows the recovery of architectural views of a software system through interactive exploration. The tool supports the “overview first, zoom and filter, and details on demand” principle of information visualization. It provides powerful filtering mechanisms and the capacity of saving and sharing architectural views. The tool was the test-bed for a variety of research projects and is still serving us in our research and consulting practice.

There are two main research directions in which we would like to bring SoftwareNaut beyond architecture recovery:

Reengineering. One of the tools built on top of SoftwareNaut is MARS: an automated architecture refactoring recommender tool [35]. The tool starts from a given SoftwareNaut 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. In the future we plan to explore more the space of reengineering operations that can be performed on an architectural view.

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 SoftwareNaut 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, SoftwareNaut can automatically detect the views affected by the new changes, and visualize these changes on them.

Acknowledgements. We would like to thank Fabrizio Perin and Oscar Nierstrasz for feedback on earlier drafts of 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. 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, pp. 137–148. Best Paper Award.
- [4] D. E. Perry, A. L. Wolf, Foundations for the study of software architecture, *SIGSOFT Softw. Eng. Notes* 17 (1992) 40–52.
- [5] D. Garlan, R. Allen, J. Ockerbloom, Architectural mismatch: Why reuse is still so hard, *IEEE Softw.* 26 (2009) 66–69.
- [6] 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.
- [7] M. Jazayeri, On architectural stability and evolution, in: *Reliable Software Technologies-Ada-Europe 2002.*, Springer, 2002, pp. 304–315.
- [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, 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.
- [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] 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.
- [17] 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.
- [18] M. Lanza, S. Ducasse, Polymetric views - a lightweight visual approach to reverse engineering, *Software Engineering, IEEE Transactions on* 29 (2003) 782–795.
- [19] M. Lanza, R. Marinescu, *Object-Oriented Metrics in Practice*, Springer-Verlag, 2006.
- [20] K. Wong, *The reverse engineering notebook*, Ph.D. thesis, University of Victoria, Victoria, B.C., Canada, Canada, 2000.
- [21] 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.
- [22] 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.
- [23] B. S. Mitchell, S. Mancoridis, On the automatic modularization of software systems using the bunch tool, *IEEE Trans. Softw. Eng.* 32 (2006) 193–208.
- [24] M. Lungu, M. Lanza, Softwareonaut: cutting edge visualization, in: *SoftVis '06: Proceedings of the 2006 ACM symposium on Software visualization*, ACM, New York, NY, USA, 2006, pp. 179–180.
- [25] 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.
- [26] S. Tichelaar, *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*, Ph.D. thesis, University of Bern, 2001.

- [27] P. F. Mihancea, G. Ganea, 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.
- [28] R. Koschke, Atomic architectural component recovery for program understanding and evolution, Software Maintenance, 2002. Proceedings. International Conference on (2002) 478–481.
- [29] 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.
- [30] O. Nierstrasz, S. Ducasse, T. Gîrba, The story of moose: an agile reengineering environment, SIGSOFT Softw. Eng. Notes 30 (2005) 1–10.
- [31] M. Lungu, M. Lanza, T. Gîrba, R. Robbes, The small project observatory: Visualizing software ecosystems, EST special issue of the Science of Computer Programming (2009).
- [32] M. Lungu, Reverse Engineering Software Ecosystems, Ph.D. thesis, University of Lugano, 2009.
- [33] 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.
- [34] M. Lungu, M. Lanza, T. Gîrba, R. Robbes, The small project observatory: Visualizing software ecosystems, Science of Computer Programming, Elsevier 75 (2010) 264–275.
- [35] A. Boeckmann, MARS - Modular Architecture Recommendation System, Bachelor’s thesis, University of Lugano, 2010.
- [36] 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.
- [37] R. Lintern, J. Michaud, M.-A. Storey, X. Wu, Plugging-in visualization: experiences integrating a visualization tool with eclipse, in: SoftVis ’03: Proceedings of the 2003 ACM symposium on Software visualization, ACM, New York, NY, USA, 2003, pp. 47–ff.
- [38] M. Pinzger, ArchView - Analyzing Evolutionary Aspects of Complex Software Systems, Ph.D. thesis, Vienna University of Technology, 2005.

- [39] 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.
- [40] A. Hindle, Z. M. Jiang, W. Koneilat, M. W. Godfrey, R. C. Holt, Yarn: Animating software evolution, Visualizing Software for Understanding and Analysis, International Workshop on V (2007) 129–136.
- [41] D. Rayside, M. Litoiu, M.-A. Storey, C. Best, R. Lintern, Visualizing flow diagrams in websphere studio using shrimp views, Information Systems Frontiers 5 (2003) 161–174. 10.1023/A:1022649506310.
- [42] 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.
- [43] 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.