Distributed and Collaborative Software Evolution Analysis with Churrasco

Marco D'Ambros[a], Michele Lanza[a]

[a] *REVEAL @ Faculty of Informatics - University of Lugano, Switzerland*

Abstract

Analyzing the evolution of large and long-lived software systems is a complex problem that requires extensive tool support due to the amount and complexity of the data that needs to be processed. In this paper we present *Churrasco*, a tool to support *collaborative software evolution analysis* through a web interface. After describing the tool and its architecture, we provide a usage scenario of Churrasco on a large open source software system and we present two collaboration experiments performed with respectively 8 and 4 participants.

Key words: Software Evolution Analysis, Collaboration, Visualization

1. Introduction

Software evolution analysis is concerned with the causes and the effects of software change. There is a large number of approaches, which all use different types of information about the history and the (evolving) structure of a system. The overall goal is on the one hand to perform retrospective analysis, useful for a number of maintenance activities, and on the other hand to predict the future evolution of a system. Such analyses are intrinsically complex, because modeling the evolution of complex systems implies

1. the retrieval of data from software repositories, managed by software configuration management systems such as CVS or SVN,
2. the parsing of the obtained raw data to extract relevant facts and to minimize the noise that such large data sets exhibit, and

Email addresses: marco.dambros@lu.unisi.ch (Marco D'Ambros), michele.lanza@unisi.ch (Michele Lanza)

3. the population of models that are then the basis for any analysis. Tools supporting software evolution analysis should hide these tasks from the users, to let them focus on the actual analysis.

Moreover, such tools should provide means to break down information complexity, typical for large and long-lived software systems. We argue that any software evolution analysis tool should possess the following characteristics:

Flexible Meta-model. Several, and largely similar, approaches have been proposed to create and populate a model of an evolving software system, considering a variety of information sources, such as the histories of software artifacts (as recorded by a versioning system), the problem reports stored by systems such as Bugzilla [1], e-mail archives, user documentation [2], *etc.* Even if such models are appropriate for modeling the evolution, they are "hard-coded" in the sense that their creators took deliberate design choices in accordance with their research goals. We postulate that *software evolution tools should be flexible with respect to the underlying meta-model*: If the meta-model is changed or extended because some new type of information is at hand or because some new analysis is required, the tool should adapt itself to the new meta-model.

Accessibility. Researchers have developed a plethora of evolution analysis tools and environments. One commonality among many prototypes is their limited usability, *i.e.,* often only the developers themselves know how to use them, thus hindering the development and/or cross-fertilization of novel analysis techniques. There are some notable exceptions, such as Moose [4], which have been used by a large number of researchers over the years. Researchers also investigated ways to exchange information about software systems [5, 6], approaches which however are seldom followed up because of lack of time or manpower. We argue that *software evolution tools should be easily accessible*: They should be usable from any machine running any operating system, without any strings attached.

Incremental Storage of Results. Results of analyses and findings on software systems produced by tools are often written into files and/or manually crafted reports, and are therefore of limited use. We claim

that *analysis results should be incrementally and consistently stored back into the analyzed models*: This allows researchers to develop novel analyses that exploit from the results of a previous analysis (cross-fertilization of ideas/results). It can also serve as a basis for a benchmark for analyses targeting the same problem, and ultimately would also allow one to to combine techniques targeting different problems.

Support for Collaboration. The need of collaboration in software development is getting more and more attention. Tools which support collaboration, such as Jazz for Eclipse [7], were only recently introduced, but hint at a larger current trend. Just as the software development teams are geographically distributed, consultants and analysts are too. Specialists in different domains of expertise should be allowed to collaborate without the need of being phisically present together. Because of these reasons, we argue that software evolution analysis should be a collaborative activity. As a consequence, *software evolution analysis tools should support collaboration*, by allowing different users, with different expertises, from different locations, to collaboratively analyze a system.

We present *Churrasco* [8], a tool for collaborative software analysis, which is available at `http://churrasco.inf.unisi.ch`. Churrasco has the following characteristics:

- It hides all data retrieval and processing tasks from the users, to let them focus on the actual analysis, and provides an easily accessible interface over a web browser to model the data sources to be analyzed.
- It copes with modeling and populating problems by providing a flexible and extensible object-relational persistency mechanism. Any data meta-model can be dynamically changed and extended, and all the data is stored in a central database.
- It provides a set of collaborative visual analyses and supports collaborative analysis by allowing users to annotate the analyzed data.
- It stores the findings into a central database to create an incrementally enriched body of knowledge about a system, which can be exploited by subsequent users.

Structure of the paper. In Section 2 we describe the Churrasco framework, its architecture, and its main components. We then provide an example of a collaborative session and describe two collaboration experiments performed with Churrasco (Section 3). We discuss our approach in Section 4 and examine tool building issues in Section 5. We survey related work in Section 6, and conclude in Section 7 with a summary of our contributions and directions of future work.

2. Churrasco

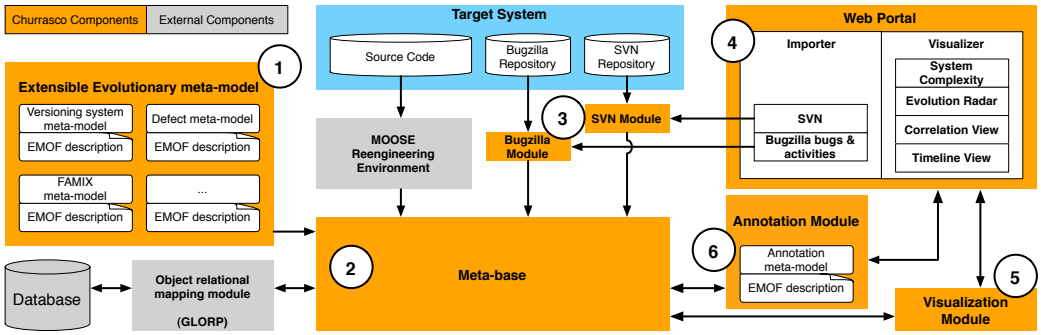


Figure 1: The architecture of Churrasco.

Figure 1 depicts Churrasco's architecture, consisting of:

1. *The Extensible Evolutionary meta-model* describes the internal representation of software systems' evolution, which can be extended using the facilities provided by the Meta-base module.
2. *The Meta-base* supports flexible and dynamic object-relational persistency. It uses the external component GLORP [9] (Generic Lightweight Object-Relational Persistence), providing object-relational persistency, to read from/write to the database. The meta-base also uses the Moose reengineering environment [4] to create a representation of the source code (C++, Java or Smalltalk) based on the FAMIX language independent meta-model [10].
3. *The Bugzilla and SVN modules* retrieve and process the data from SVN and Bugzilla repositories.
4. *The Web portal* represents the front-end of the framework accessible through a web browser.

5. *The Visualization module* supports software evolution analysis by creating and exporting interactive Scalable Vector Graphics (SVG) visualizations.
6. *The Annotation module* supports collaborative analysis by enriching any entity in the system with annotations. It communicates with the web visualizations to depict the annotations within the visualizations.

2.1. The Meta-base

Churrasco's *Meta-base* [11] provides flexibility and persistency to any meta-model, in particular to our evolution meta-model. It takes as input a meta-model described in EMOF and outputs a descriptor, which defines the mapping between the object instances of the meta-model, *i.e.,* the model, and tables in the database. EMOF (Essential Meta Object Facilities) is a subset of MOF[1], a meta-meta-model used to describe meta-models. The Meta-base ensures persistency with the object-relational module GLORP. By generating descriptors of the mapping between the database and the meta-model, the Meta-base can be adapted dynamically and automatically to any meta-model. This allows Churrasco users to modify and extend dynamically any meta-model. For more details, we refer the interested reader to [11].

2.2. The SVN and Bugzilla modules

These modules retrieve and process data from, respectively, Subversion and Bugzilla repositories. They take as input the URL of the repositories and then populate the models using the Meta-base. They are initially launched from the web importer (discussed later) to create the models, and then they automatically update all the models in the database every night, with the new information (new commits or bug reports).

The SVN module populates the versioning system model, by checking out (or updating) the project with the given repository, creating and parsing SVN log files. The checked out system is then used to create the FAMIX model of the system with the external component Moose.

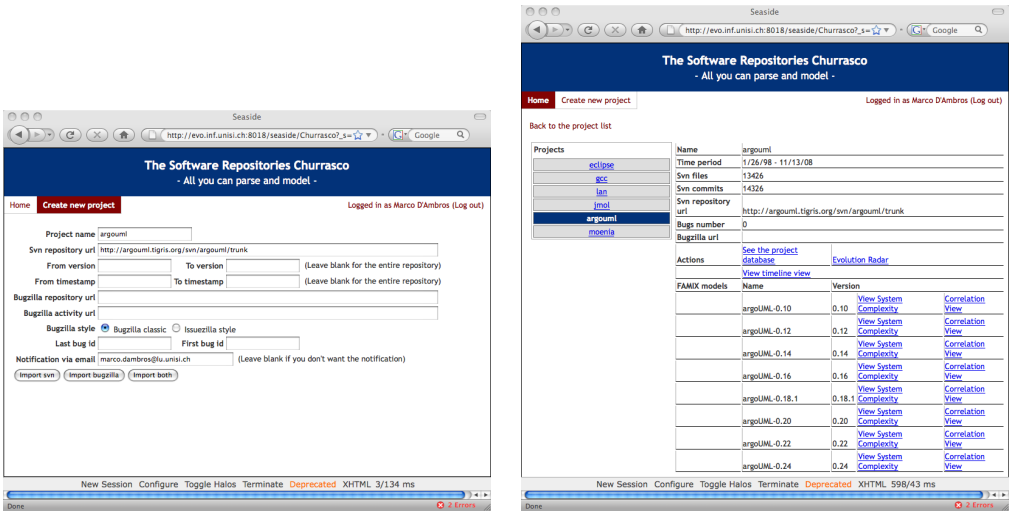
The Bugzilla module retrieves and parses all the bug reports (in XML format) from the given repository. Subsequently it populates the corresponding part of the defect model. It then retrieves all bug activities from the given

[1]MOF and EMOF are standards defined by the OMG (Object Management Group) for Model Driven Engineering. For more details consult the specications at: `http://www.omg.org/docs/html/06-01-01/Output/06-01-01.htm`

repository. Since Bugzilla does not provide this information in XML format, Churrasco parses HTML pages and populates the corresponding part of the model. Finally, it links software artifacts with bug reports. To do this it combines the technique proposed by Fischer *et al.* [1] (matching bug report IDs and keywords in the commit comments) with a timestamp-based approach.

2.3. The Web Portal

The web portal is the front-end of Churrasco, developed using the Seaside framework [12]. It allows users both to create the models, and to analyze them by means of different web-based visualizations. To create new models and access the visualizations the user has to log in the web portal.



(a) The importer page.

(b) The projects page.

Figure 2: The Churrasco Web Portal.

Figure 2(a) shows the importer web page of Churrasco, ready to import the ArgoUML software project. All that is needed to create the model is the URL of the SVN repository and the URLs of the bugzilla repository (one for bug reports, one for bug activities). Since, depending on the size of the software system to be imported, this can take a long time, the user can also indicate an e-mail address to be notified when the importing is finished.

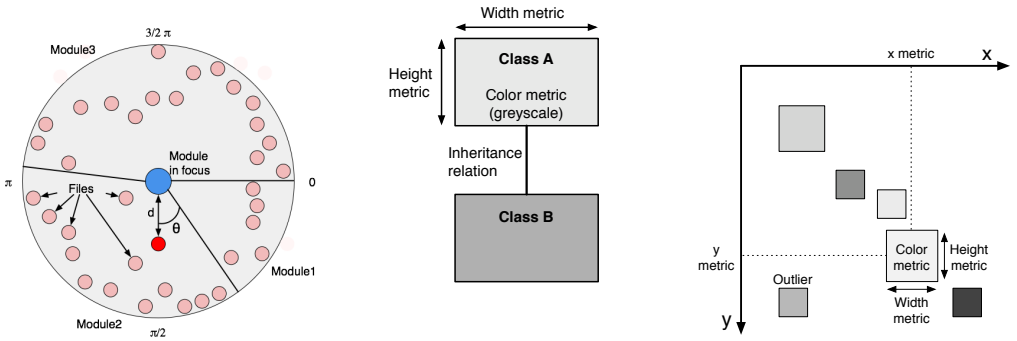
Figure 2(b) shows the projects web page of Churrasco, which contains a list of projects available in the database and, for a selected project, information such as the number of files and commits, the time period (time between

the first and last commit), the number of bugs, a collection of FAMIX models corresponding to different versions of the system *etc.* Finally, the page also provides a set of *actions* to the user, *i.e.,* links to the web visualizations provided by Churrasco.

2.4. The Visualization Module

This module offers the following set of interactive visualizations that support software evolution analysis:

1. *The Evolution Radar* [13, 14] supports software evolution analysis by depicting change coupling information. Change coupling is the implicit dependency between two or more software artifacts that have been observed to frequently change together during the evolution of a system. There are several ways of computing the change coupling measure, where the simplest one consists in counting the number of transactions in which the considered software artifacts were changed together. Discussing other techniques to compute change coupling goes beyond the scope of this paper, and we refer the interested reader to [14].
   Figure 3(a) illustrates the principle of the Evolution Radar. It shows the dependencies between a module, represented as a circle and placed in the center of a pie chart, and all the other modules in the system represented as sectors. In each sector, all files belonging to the corresponding module are represented as colored circles and positioned according to the change coupling they have with the module in the center (the higher the coupling the closer to the center). With respect to the angle, we sort the files alphabetically (considering the entire directory path) and uniformly distribute them in their containing module.
2. *The System Complexity* [15] view supports the understanding of object-oriented systems, by enriching a simple two-dimensional depiction of classes and inheritance relationships with software metrics (see Figure 3(b)). By default, the size of the nodes is proportional to the number of attributes (width) and methods (height), while the color represents the number of lines of code. This mapping can be changed from the web interface, by assigning any software metric from a rich catalog to the width, height and color of the nodes. The goal of the view is to provide clues on the complexity and structure of a software system.



(a) Evolution Radar. (b) System Complexity. (c) Correlation View.

Figure 3: Evolution Radar, System Complexity and Correlation View principles.

3. *The Correlation View* shows all the classes of a software system in a two-dimensional space, using a scatterplot layout and mapping up to five software metrics on them: On the vertical and horizontal position, on the size and on the color (see Figure 3(c)). The default mapping is the following: The nodes' coordinates represent the number of attributes (x) and methods (y), the color represents the number of lines of code, while the size of the nodes is fixed. As for the System Complexity view, the mapping can be changed at any time using the web interface. The Correlation view is useful to understand the correlation between different metrics in a software system and to detect outliers, *i.e.,* entities having metric values completely different with respect to the majority of entities in the system.
   In the experiments discussed later in the paper, we provide examples of Evolution Radar and System Complexity visualizations, but not of Correlation view. We give an example of this view here, depicted in Figure 4. Nodes represent classes of the ArgoUML software system[2], where the x position is proportional to the number of lines of code, the y is proportional to the number of post release bugs and the color maps the number of methods. Such a choice of metrics mapping can be useful to understand whether larger classes (higher number of lines of

[2] `http://argouml.tigris.org`

Figure 4: A Correlation view applied to the ArgoUML software system. Nodes represent classes, nodes' position represents number of lines of code (x) and number of post release bugs (y), and nodes' color maps the number of methods.

code) generate more bugs. This correlation does not hold in the case of ArgoUML (see Figure 4). Moreover, we spot some outliers in the view: The one marked as "A", which has an outstanding number of bugs, and the ones marked as "B", with an outstanding number of lines of code.

The visualizations are created using the Mondrian framework [16] (residing in Moose) and the Episode framework [17] (residing in Churrasco's visualization module). To make the visualizations interactive within the web portal, Episode attaches Ajax callbacks to the figures.

Figure 5 shows an example of a System Complexity visualization rendered in the Churrasco web portal. The main panel is the view where all the figures are rendered as SVG graphics. The figures are interactive: Clicking on one of them will highlight the figure (red boundary), generate a context menu and show the figure details (the name, type and metrics values) in the figure information panel on the left. Under the information panel Churrasco provides three other panels useful to configure and interact with the visualization:

1. The metrics mapping configurator which allows the user to customize the view by changing the metrics mapping.
2. The package selector which allows the user to select, and then visualize, multiple packages or the entire system.
3. The regular expression matcher with which the user can select entities in the visualization according to a regular expression.

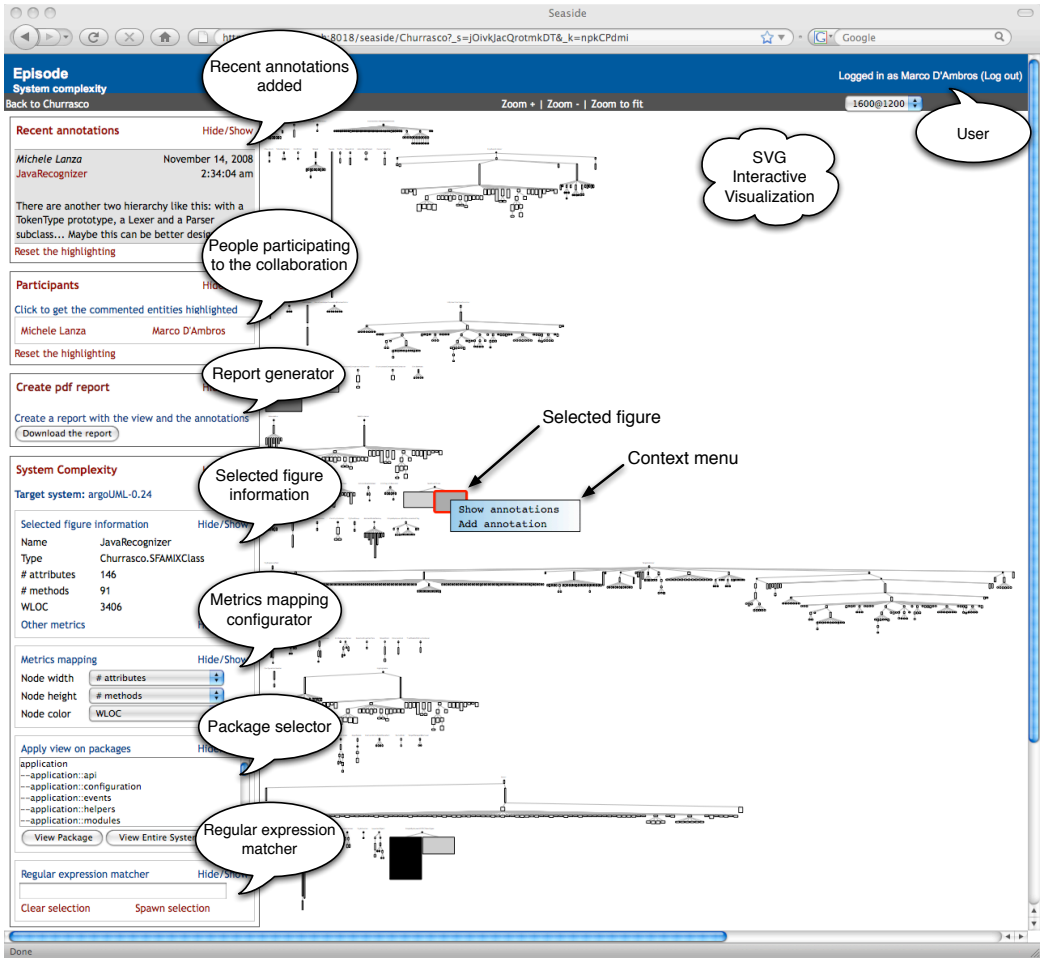


Figure 5: A screenshot of the Churrasco web portal showing a System Complexity visualization of ArgoUML.

2.5. The Annotation Module

The idea behind Churrasco's annotation module is that each model entity can be enriched with annotations to (1) store findings and results incrementally into the model and to (2) let different users collaborate in the analysis of a system in parallel. Annotations can be attached to *any* visualized model entity, and each entity can have several annotations. An annotation is composed of the author who wrote it, the creation timestamp and the text. When the user clicks on the menu action "Show annotations" an additional panel is

rendered at the top left corner of the web page (above the recent annotation panel). The panel shows all the annotations for the selected entity and allows the user to delete (only) his/her annotations. Clicking on the "Add annotation" menu item will result in displaying another panel (again in the top left corner) that allows the user to write and add new annotations to the selected entity. Since the annotations are stored in a centralized database, any new annotation is immediately visible to all the people using Churrasco, thus allowing different users to collaborate in the analysis. Churrasco features three other panels aimed at supporting collaboration:

1. The "Recent annotations" panel displays the most recent annotations added, together with the name of the annotated entity, and by clicking on it the user can highlight the corresponding figure in the visualization.
2. The "Participants" panel lists all the people who annotated the visualizations, *i.e.,* people collaborating in the analysis. When one of these names is clicked, all the figures annotated by the corresponding person are highlighted in the view, to see which part of the system that person is working on.

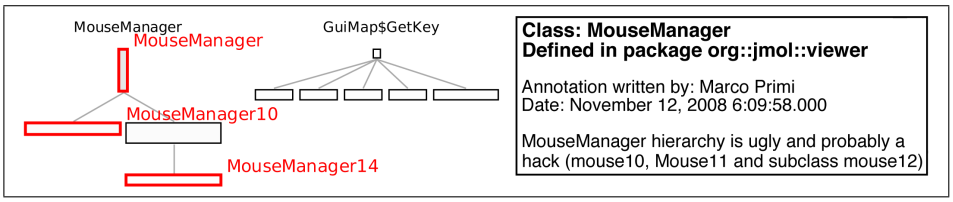


Figure 6: An excerpt of a pdf report generated by Churrasco. The entities with one or more annotations are highlighted in red, and the corresponding annotations are provided.

3. The "Create pdf report" panel generates a pdf document containing the visualization and all the annotations referring to the visualized entities. Figure 6 shows a modified excerpt[3] of such a report: In the visualization part the entities with at least one annotation are highlighted in red, and the corresponding annotations are listed together with the author and date information.

[3]We modified the excerpt of the report to make it fit in the page.

3. Churrasco in Action

We show Churrasco's use through one simple example scenario, presented next, and two collaboration experiments with respectively 8 and 4 participants.

3.1. Analyzing ArgoUML

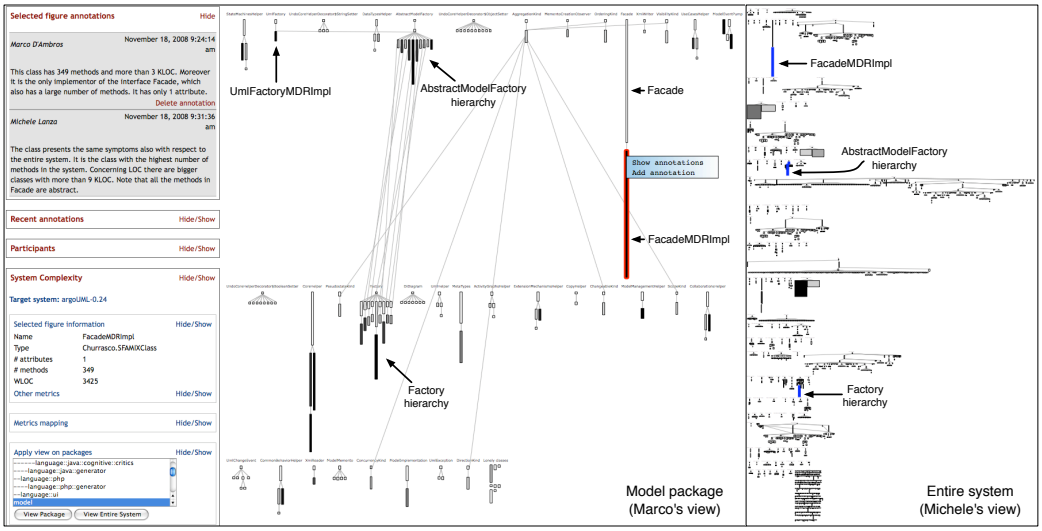


Figure 7: The web portal of Churrasco visualizing the system complexity of the *Model* package of ArgoUML on the left and the entire ArgoUML system on the right.

We use the following simple scenario to exemplify Churrasco's usage: The authors of this article, working on different machines in different locations, study the evolution of ArgoUML, a UML modeling tool composed of ca. 1800 Java classes, developed over the course of ca. 7 years. The users first create the evolutionary model by indicating the URL of the ArgoUML SVN repository in the importer page of Churrasco (bug information is not needed in this example scenario). Once the model is created and stored in the centralized database, they start the analysis with a system complexity view of the system. Each user renders the visualization in his web browser, and attaches annotations to interesting figures in the visualizations. The annotations are immediately visible to the other user on the left side of the browser window (in the annotation panels).

While Michele is analyzing the entire system, Marco focuses on the *Model* package, which contains several classes characterized by large number of methods and many lines of code. The entities annotated by Marco in the fine-grained view are then visible to Michele in the coarse-grained system complexity. Marco has the advantage of a more focused view, while Michele sees the entire context. Figure 7 shows Marco's view on the left, while Michele's one is depicted on the right. Marco selected the *FacadeMDRImpl* class (highlighted in red in Marco's view), and is reading Michele's comments about that class (highlighted in blue in Michele's view). These are two examples of collaboration:

1. Marco, focusing on the *Model* namespace, annotates that the class *FacadeMDRImpl* shows symptoms of bad design: It has 350 methods, 3400 lines of code, only 3 attributes, and it is the only implementor of the *Facade* interface. Michele adds a second annotation that Marco's observation holds also with respect to the entire system, and that *FacadeMDRImpl* is the class with the highest number of methods in the entire system.
2. Marco sees that several classes in the *Factory* hierarchy implement the *Factory* interface and also inherit from classes belonging to the *AbstractModelFactory* hierarchy. This is not visible in Michele's view (where *Factory* and *AbstractModelFactory* are highlighed in blue), who discovers that fact by highlighting the entities annotated by Marco and then reading the annotations.

Both now want to find out whether these design problems have always been present in the system. They analyze the system history in terms of its change coupling using the Evolution Radar. This visualization is time-dependent, *i.e.,* different radar views are used to represent different time intervals. Figure 8 shows on the left an evolution radar visualization corresponding to the time interval Oct 2004 – Oct 2005, and on the right the radar corresponding to Oct 2005 – Oct 2006. They both represent the dependencies of the *Diagram* module (displayed as a cyan circle in the center) with all the other modules of ArgoUML, by rendering individual classes. Marco is looking at the time interval 2004/05 (left part of Figure 8). He selects the class *UMLFactoryMDRImpl* (marked in red), belonging to the *Model* module, because it is the closest to the center (highest coupling with the *Diagram* module in the center) and because it is large (the size maps the number of changes in

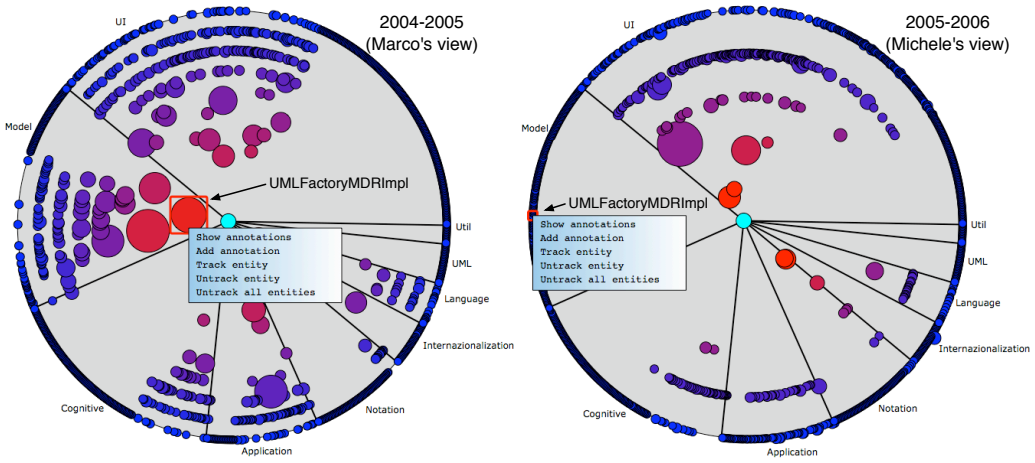


Figure 8: Evolution Radars of ArgoUML.

the corresponding time interval). Marco attaches to the class the annotation that it is potentially harmful, given the high coupling with a different module (*Diagram*), with respect to the one the class belongs to (*Model*). In the meantime Michele is looking at the time interval 2005/06 (right part of Figure 8). He highlights the classes annotated by Marco and sees the *UMLFactoryMDRImpl* class. In Michele's radar the class is not coupled at all with the *Diagram* module, *i.e.,* it is at the boundary of the view (marked in red). Therefore, Michele adds an annotation to the class saying that it is probably not harmful, since the coupling decreased over time. After reading this comment, Marco goes back to the system complexity view, to see the structural properties of the class in the system. The *UMLFactoryMDRImpl* class (marked in the left part of Figure 7) has 22 methods, 9 attributes and 600 lines of code. It implements the interfaces *AbstractUmlModelFactoryMDR* and *UMLFactory*. After seeing the class in the system complexity, Marco adds another annotation saying that the class is not harmful after all.

This information can then be used by other users in the future. Suppose that Romain wants to join the analysis with Marco and Michele, or to start from their results. He can first see on which entities the previous users worked, by highlighting them, and then reading the corresponding annotations to get the previously acquired knowledge about the system.

This simple scenario shows how (1) the knowledge about a system, gained in software evolution analysis activities, can be incrementally built, (2) dif-

ferent users from different locations can collaborate, and (3) different visualization techniques can be combined to improve the analysis.

3.2. First Collaboration Experiment

The previous example showed that Churrasco supports collaborative analysis. However, the example is hardly a collaborative experiment, because (1) there were only two participants (2) who were the developers of the tool, (3) possessing prior knowledge about the analyzed software system. Therefore, we performed a collaboration experiment, in a more realistic setting, with the following goals: (1) evaluate whether Churrasco is a good means to support collaboration in software evolution analysis, (2) test the usability of the tool, and (3) test the scalability of the tool with respect to the number of participants.

We performed the experiment in the context of a university course on software design and evolution. The experiment lasted 3 hours: During the first 30 minutes we explained the concept of the tool and how to use it, in the following two hours (with a 15 minutes break in the middle) the students performed the actual experiment and in the last 15 minutes they filled in a questionnaire about the experiment and the tool. The participants were: 5 master students, 2 doctoral students working in the software evolution domain and 1 professor. The Master students were lectured on reverse engineering topics before the experiment.

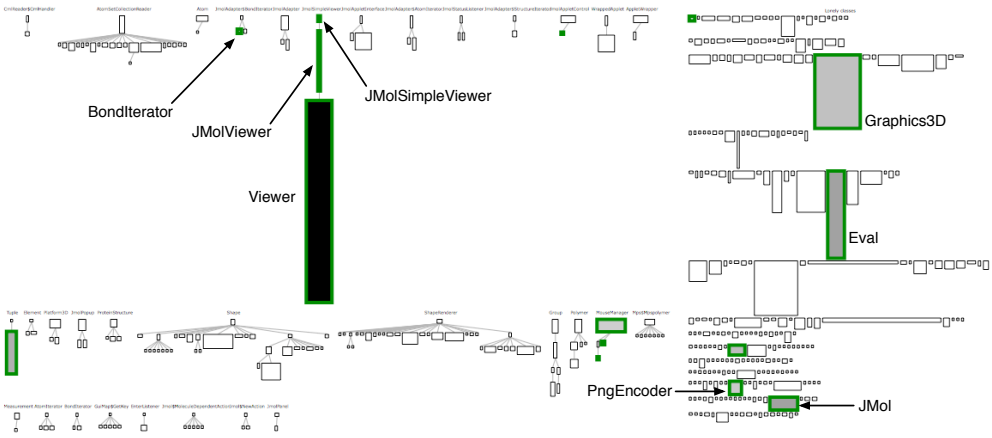


Figure 9: A System Complexity of JMol. The color denotes the amount of annotations made by the users. The highlighted classes (green boundaries) are annotated classes.

The task consisted in using the System Complexity and the Correlation View and looking at the source code to (1) discover classes on which one would focus reengineering efforts (explaining why), and to (2) discover classes with a big change impact and explain why. The target system chosen for the experiment was JMol, a 3D viewer for chemical structures, consisting of ca. 900 Java classes. Among the participants only one possessed some knowledge about the system.

Figure 9 shows a System Complexity of JMol in which nodes' size maps number of attributes (width) and methods (height) and nodes' color represents the amount of annotations they received, *i.e.,* number of annotations weighted with their length. We see that the most annotated class is *Viewer*, the one with the highest number of methods (465). However, we can also see that not only the big classes (with respect to methods and/or attributes) were commented, but also very small classes.

Class	**NOA**	**NOM**	**Annotation**
JmolSimpleViewer	0	8	"This is a strange hierarchy. There is only one subclass per superclass (all with many method and few attributes)."
JMolViewer	0	135	"Strange: 134 abstract methods, only 1 concrete, only 1 subclass."
Viewer	54	465	"This class seems to be the "thing" in the system, at least in terms of functionality", "Strong dependency with Eval.", "High fan out (25) and many LOC (>1k).", "High number of access to foreign data."
Eval	34	198	"This class should probably be broken down.", "Very strong dependency with Viewer."
JMol	60	25	"This class has the largest fan out (78). Probably part of the core of the system.", "High coupling, low cohesion", "13 protected methods and no child!"
PngEncoder	23	26	"17 protected attributes, completely useless since there's no child!"
BondIterator	5	5	"There are ca. 6 classes with Iterator logic. The implementation is strange. I would expect them to be in some hierarchy."
Graphics3D	101	166	"This can be probably broken down. It's an implementation of a 3d engine."

Table 1: Subset of the annotations made on Jmol. NOA stands for number of attributes and NOM for number of methods.

In Table 1 we list a subset of the annotations made by the users during the experiment. In the assigned time the participants annotated 15 different classes for a total of 31 annotations, distributed among the different participants, *i.e.,* everybody actively participates in the collaboration. The average number of annotations per author was 3.87, with a minimum of 2 and a maximum of 13.

The annotations were also used to discuss about certain properties of the analyzed classes. In most of the cases the discussion consisted in combining different pieces of knowledge about the class (local properties as number of methods with properties of the hierarchy with dependency *etc.*).

At the end of the experiment all participants but one filled in a survey about the tool and the collaboration experience. The survey results are shown in Table 2. In the cases where the sum of the answers is not 7, a participant did not indicate an answer.

Statement	Strongly disagree	Disagree	Neither	Agree	Strongly Agree
Churrasco is easy to use			1	3	2
System Complexity view is useful				2	5
Correlation view is useful			1	1	5
Churrasco is a good means to collaborate					7
Collaboration is important in reverse engineering			1	5	1

Table 2: Subset of the results from the questionnaire, using a Likert Scale.

Although not a full-fledged experiment, it provided us with information about our initial goals: The survey shows that the participants found the tool easy to use, collaboration important in reverse engineering and Churrasco as a good means to support collaboration (for all the participants the experiment was the first reverse engineering collaboration experience). Another result is that they found the provided visualizations useful to achieve the given tasks. Churrasco scaled well with 8 people accessing the same model on the web portal at the same time, without any performance issue.

A final comment given by the users during an informal conversation after the experiment, is that they had fun in the collaborative session: They especially liked to wait for annotations from other people on the entity they already commented, or to see what was going on in the system and which classes were annotated, to also personally look at them.

3.3. Second Collaboration Experiment

The purpose of the first experiment was to perform a preliminary evaluation of Churrasco's usability and whether users would find Churrasco a good means to collaborate. In our second experiment we wanted to simulate a more structured form of collaboration, where users do not have all equal roles.

We performed the experiment in the context of a university course on software engineering, with a set-up very similar to the one of the previous experiment. This time the participants were 4 bachelor students with little knowledge about reverse engineering. The experiment took place during the last week of a project in which all students had developed a web application in Smalltalk during 6 weeks. During the last week of the project the students could not add new features to the system, but they could only restructure / refactor it to improve its design and code quality.

The task that the students had in the experiment was to identify which parts of the system should be refactored, using the System complexity and Correlation views in Churrasco. The students had different roles in the collaboration: One acted as a leader, responsible to analyze the system, by selecting classes which he thought were candidates for refactoring, while the other students would check in detail whether the classes in question needed to be refactored or not.

With the annotations, the leader could also ask questions that the followers then answered. Typical questions were: "What is the responsibility of this class?", "Can we remove this class?", "These hierarchies seems to be duplicated, can we merge them?","Why this class is in this hierarchy? Shouldn't it be a subclass of that class?" *etc.*

The target software system was composed of 166 classes, 983 methods for a total of ca. 5,000 lines of Smalltalk code.

Class ElementModel
What is the difference between Element Model and Element? Are both hierarchies replicated?
One is the model that manages the functionalities of the element, the other one manages the displaying of the element (it is a proxy pattern).
One is for the layout behavior while the other is for the widget behavior.
Class WBLBorderLayoutModel
This layout seems to have more behavior than the others, even though it has the same number of attributes. Maybe it is doing too much and it should be a composite layout?
It has a lot of complex operations which being detached can raise the complexity much more. As you say it has functionalities that can be put in more than 1 class.
The layout is complex. Dividing it into several classes will require too much time and effort.

Table 3: Subset of the annotations made on the Smalltalk web application.

Table 3 shows a subset of the annotations made by the users during the experiment, the ones written for a couple of classes. These two groups of annotations exemplify how the collaborative session was performed: The leader was asking questions about the design of classes and hierarchies, and

the other students were answering these questions.

During the experiment the participants annotated 11 different classes for a total of 27 annotations, 9 written by the leader and 18 by the other students. Since the participants knew the system, they were faster in writing annotations with respect to the participants of the first collaboration experiment.

Statement	Strongly disagree	Disagree	Neither	Agree	Strongly Agree
Churrasco is a good means to collaborate				1	3
Collaboration helped me in understanding the system				3	1
The proposed methodology (leaders) helps in structuring the collaborative effort				3	1
Reading other users' annotations eases the given tasks				2	2
	1-2	3-4	5-6	7-8	9-10
Quantify (1-10) the added value of the collaborative support provided by the tool				2	2

Table 4: Subset of the results from the questionnaire, using a Likert Scale.

As in the previous experiment, at the end of the collaborative session the participants filled in a survey about the tool and the collaboration experience. A subset of the results are shown in Table 4. The survey shows that the participants found that collaboration helped them in understanding the software system and the used methodology with the leader useful to structure the collaborative effort. Moreover, the use of annotations eased the task of selecting potential candidates for refactoring. Another result is that the students found that the collaborative support provided by Churrasco has an added value.

4. Discussion

The main benefits of Churrasco consist in its accessibility and flexibility. All the features of the framework can be accessed through a web browser: (1) The importers to create and populate evolutionary models of software systems, (2) the system complexity and correlation views, to support the understanding of the structure of the system and (3) the evolution radar view to study the evolution of the system modules in terms of change coupling. The visualizations are interactive, and they allow the user to inspect the entities represented by the figures, to apply new visualizations on-the-fly from the context menus and to navigate back and forth among different

views. The framework can be extended with respect to the meta-model and with respect to the visualizations. Using the facilities provided by the meta-base, the underlying evolutionary meta-model of Churrasco can be enriched with new types of information.

5. Tool Building Issues

Developing a web-based tool that supports scalable and interactive visualizations raises issues related to interacting, updating, and debugging.

Interacting. Supporting interaction through a web browser is still a non-trivial task, and even supposedly simple features, such as context menus, must be implemented from scratch. In our Churrasco tool we have implemented the context menus as SVG composite figures, with callbacks attached, which are rendered on top of the SVG visualization. Moreover, it is hard to guarantee a responsive user interface, since every web application introduces a latency due to the transport of information.

Updating. The standard way of rendering a web visualization is that every time something changes in the page, the whole page is refreshed to show the updated version. In the context menu example, whenever the user clicks on a figure the page changes because a new figure appears, and therefore the page needs to be refreshed to show the menu. This introduces latencies which make the web application unusable when it comes to rendering very large SVG files. For this reason, we implemented many actions that do not require a complete re-rendering of a page using Ajax requests. Examples of such actions are: Rendering of context menus, highlighting figures, displaying figure information, displaying and adding annotations.

Debugging. A barrier to develop web applications is the lacking support for debugging. Even if there are some applications like Firebug providing HTML inspection, Javascript debugging and DOM exploration, the debugging support is not comparable with the one given in mainstream integrated development environments such as Eclipse.

All in all, while building Churrasco we learned that creating a web application that supports interactive visualizations implies a number of technological challenges. With the current *status quo* of web development frameworks it should make any intention of porting existing applications to the web forego a careful evaluation of whether the result is worth the effort. On the other hand, web applications introduce a number of novel ways to interact with systems that will open up new research directions.

6. Related Work

A number of approaches support web-based software evolution analysis and visualizations.

Beyer and Hassan proposed Evolution Storyboards [18], a visualization technique that offers dynamic views. The storyboards, rendered as SVG files (visible in a web browser), depict the history of a project using a sequence of panels, each representing a particular time period in the life of a software project. These visualizations are not, or only partially, interactive, *i.e.,* they only show the names of the entities represented by the SVG or VRML figures. In contrast the views offered in the Churrasco web portal are fully interactive, providing context menus for the figures and navigation capabilities.

Lungu *et al.* presented an web-based approach to visualize entire software repositories [19]. Their technique, validated on Smalltalk repositories, focuses on understanding the structure of the organization behind the repositories, by studying the interaction among the developers. They also provide views to see the evolution of the repositories over time. Both the approaches are fully interactive and web-based, but while Lungu's approach focuses on the entire repository evolution with coarse-grained views, Churrasco targets single projects with fine-grained visualizations.

In [20] Mancoridis *et al.* presented REportal, a web-based portal site for the reverse engineering of software systems. REportal allows users to upload their code (Java or C++) and then to browse, analyze and query it. These services are implemented by reverse engineering tools developed by the authors over the years. REportal supports software analysis through browsing and querying, whereas Churrasco supports the analysis by means of interactive visualizations.

In [21] Nentwich *et al.* introduced BOX, a portable, distributed and interoperable approach to browse UML models. BOX translates a UML model that is represented in XMI into VML (Vector Markup Language), which can be directly displayed in a web browser. BOX enables software engineers to access and review UML models without the need to purchase licenses of tools that produced the models. While BOX is focused on design documents, such as UML diagrams, in Churrasco we focus on the history and structure of software systems.

A major difference between all the mentioned approaches and Churrasco is that these techniques support single user software evolution analysis, while Churrasco supports collaborative analysis.

7. Conclusions

We have presented Churrasco, a tool which supports collaborative software evolution analysis and visualization. The main features of Churrasco are:

- *Flexible meta-model support.* The meta-model used in Churrasco to describe the evolution of a software system can be dynamically changed and/or extended, by means of the meta-base component.
- *Accessibility.* The tool is fully web-based, *i.e.,* the entire analysis of a software system, from the initial model creation to the final study, can be performed from a web browser, without having to install or configure any tool.
- *Modeling of results.* Churrasco relies on a centralized database and supports annotations. Thus, the knowledge of the system, gained during the analysis, can be incrementally stored on the model of the system itself.
- *Collaboration.* We have shown, through a couple of collaboration experiments with respectively 8 and 4 participants, how Churrasco supports collaborative software evolution analysis.

7.1. Future Work

Our future work targets two main directions: Extending the tool and performing more experiments. Concerning the tool, we plan to extend the Churrasco meta-model by including information extracted from mail archives, and we plan to create the corresponding importer which retrieves such data in batch mode. Moreover, we want to enrich the set of views offered by Churrasco with visualizations of bug information. For examples of such bug visualizations, we refer the reader to [22].

With respect to the experiments, we plan to perform both a quantitative experiment on Churrasco's usefulness and a qualitative one (by means of interviews) on its usability and usefulness.

Acknowledgments. We gratefully acknowledge the financial support of the Swiss National Science foundation for the project "DiCoSA" (SNF Project No. 118063).

References

[1] M. Fischer, M. Pinzger, H. Gall, Populating a release history database from version control and bug tracking systems, in: Proceedings of the International Conference on Software Maintenance (ICSM 2003), IEEE CS Press, 2003, pp. 23–32.

[2] D. Cubranic, G. Murphy, Hipikat: Recommending pertinent software development artifacts, in: Proceedings of the 25th International Conference on Software Engineering (ICSE 2003), ACM Press, 2003, pp. 408–418.

[3] S. Ducasse, T. Gîrba, O. Nierstrasz, Moose: an agile reengineering environment, in: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE 2005), 2005, pp. 99–102.

[4] S. Kim, T. Zimmermann, M. Kim, A. Hassan, A. Mockus, T. Gîrba, M. Pinzger, J. Whitehead, A. Zeller, TA-RE: An exchange language for mining software repositories, in: Proceedings of the 3rd International Workshop on Mining Software Repositories (MSR 2006), ACM, 2006, pp. 22–25.

[5] S. Tichelaar, S. Ducasse, S. Demeyer, FAMIX: Exchange experiences with CDIF and XMI, in: Proceedings of the ICSE 2000 Workshop on Standard Exchange Format (WoSEF 2000), 2000.

[6] R. Frost, Jazz and the eclipse way of collaboration, IEEE Software 24 (6) (2007) 114–117.

[7] M. D'Ambros, M. Lanza, A flexible framework to support collaborative software evolution analysis, in: Proceedings of the 12th IEEE European Conference on Software Maintenance and Reengineering (CSMR 2008), IEEE CS Press, 2008, pp. 3–12.

[8] A. Knight, Glorp: generic lightweight object-relational persistence, in: Proceeding of OOPSLA 2000 (Addendum), ACM Press, 2000, pp. 173–174.

[9] S. Demeyer, S. Tichelaar, S. Ducasse, FAMIX 2.1 — The FAMOOS Information Exchange Model, Tech. rep., University of Bern (2001).

[10] M. D'Ambros, M. Lanza, M. Pinzger, The metabase: Generating object persistency using meta descriptions, in: Proceedings of the 1st Workshop on FAMIX and Moose in Reengineering (FAMOOSR 2007), 2007.

[11] S. Ducasse, A. Lienhard, L. Renggli, Seaside: A flexible environment for building dynamic web applications, IEEE Software 24 (5) (2007) 56–63.

[12] M. D'Ambros, M. Lanza, M. Lungu, The evolution radar: Visualizing integrated logical coupling information, in: Proceedings of the 3rd International Workshop on Mining Software Repositories (MSR 2006), ACM, 2006, pp. 26–32.

[13] M. D'Ambros, M. Lanza, Reverse engineering with logical coupling, in: Proceedings of the 13th Working Conference on Reverse Engineering (WCRE 2006), IEEE CS Press, 2006, pp. 189–198.

[14] M. Lanza, S. Ducasse, Polymetric views — a lightweight visual approach to reverse engineering, Transactions on Software Engineering (TSE) 29 (9) (2003) 782–795.

[15] M. Meyer, T. Gîrba, M. Lungu, Mondrian: An agile visualization framework, in: ACM Symposium on Software Visualization (SoftVis 2006), ACM Press, 2006, pp. 135–144.

[16] M. Primi, The episode framework - exporting visualization tools to the web, Bachelor's thesis, University of Lugano (Jun. 2007).

[17] D. Beyer, A. E. Hassan, Animated visualization of software history using evolution storyboards, in: Proceedings of the 13th Working Conference on Reverse Engineering (WCRE 2006), IEEE CS Press, 2006, pp. 199–210.

[18] M. Lungu, M. Lanza, T. Gîrba, R. Heeck, Reverse engineering super-repositories, in: Proceedings of the 14th IEEE Working Conference on Reverse Engineering (WCRE 2007), IEEE CS Press, 2007, pp. 120–129.

[19] S. Mancoridis, T. S. Souder, Y.-F. Chen, E. R. Gansner, J. L. Korn, Reportal: A web-based portal site for reverse engineering, in: Proceedings of the 8th Working Conference on Reverse Engineering (WCRE 2001), IEEE Computer Society, 2001, p. 221.

[20] C. Nentwich, W. Emmerich, A. Finkelstein, A. Zisman, BOX: Browsing objects in XML, Software Practice and Experience 30 (15) (2000) 1661–1676.

[21] M. D'Ambros, M. Lanza, M. Pinzger, "a bug's life" — visualizing a bug database, in: Proceedings of the 4th IEEE International Workshop on Visualizing Software For Understanding and Analysis (VISSOFT 2007), IEEE CS Press, 2007, pp. 113–120.