Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

On Porting Software Visualization Tools to the Web

Marco D'Ambros[1], **Michele Lanza**[1], **Mircea Lungu**[2], **Romain Robbes**[3]

[1] REVEAL @ Faculty of Informatics, University of Lugano, Switzerland
e-mail: {marco.dambros, michele.lanza}@usi.ch
[2] Software Composition Group (SCG), University of Bern, Switzerland
e-mail: lungu@iam.unibe.ch
[3] PLEIAD Lab, Computer Science Department (DCC), University of Chile, Chile
e-mail: rrobbes@dcc.uchile.cl

Received: date / Revised version: date

Abstract. Software systems are hard to understand due to the complexity and the sheer size of the data to be analyzed. Software visualization tools are a great help as they can sum up large quantities of data in dense, meaningful pictures. Traditionally such tools come in the form of desktop applications. Modern web frameworks are about to change this status quo, as building software visualization tools as web applications can help in making them available to a larger audience in a collaborative setting. Such a migration comes with a number of promises, perils and technical implications that must be considered before starting any migration process.

In this paper we share our experiences in porting two such tools to the web and provide guidelines about the porting. In particular, we discuss promises and perils that go hand in hand with such an endeavour and present a number of technological alternatives that are available to implement web based visualizations.

Key words: Software Visualization, Software Analysis Tools, Web Applications

1 Introduction

Developing tools is an important part of software engineering research as they provide a proof-of-concept for an approach. Further, the tool itself can be considered a research contribution. However, tools remain often at the stage of prototypes, not maintained anymore after the corresponding article is published. Little effort is spent in making tools long-lived and used in an industrial context, with a number of notable exceptions such as the Moose reverse engineering framework [13], visualization tools such as Rigi [1, 44], and recommender systems like Mylyn [28].

The vast majority of tools do not survive after research has been published and concluded. One of the reasons is that, unlike in the industry, there is little incentive to keep tools running as most of the times there are few users. In his keynote address at the 31st International Conference on Software Engineering, Carlo Ghezzi stated that a survey of all the papers that appeared in ACM Transactions on Software Engineering and Methodology between 2000 and 2008 showed that 60% of them dealt directly or indirectly with tools. Of those only 20% were actually installable, let alone functional.

In the past years, we have developed a number of software visualizations tools, such as CodeCrawler [31], Softwarenaut [36], BugCrawler [8], Evolution Radar [10], Bug's Life [11], CodeCity [60], Churrasco [7], The Small Project Observatory [38], and Spyware [48]. Many of these tools are available, but some effort from accidental users to make them work is required, decreasing their adoption and impact. A solution is to exploit the web and the available modern technologies. We see the web as an opportunity to improve the accessibility and adoption of research prototypes, since the cost for people to "give it a try" is minimal.

Developing web-based software visualization tools is not easy, and comes with a number of promises to embrace and perils to avoid. In this paper we discuss our experience in building two web-based software visualization tools and distill a number of considerations that need to be made if one wants to port such tools to the web. We present available technologies to develop web based visualizations, discussing their benefits and limitations. The goal is to provide guidance to researchers who want to move their (visualization) tools to the web, or want to create new web-based tools from scratch.

Contributions. The main contributions of this paper are:

- The identification, via our empirical experience building two large-scale, web-based software visualization tools, of 8 promises and 7 perils to be aware of when designing and implementing web-based visualization tools.
- The evaluation of a subset of the perils as they hold in practice, based on two usability studies of our web-based visualization tools.

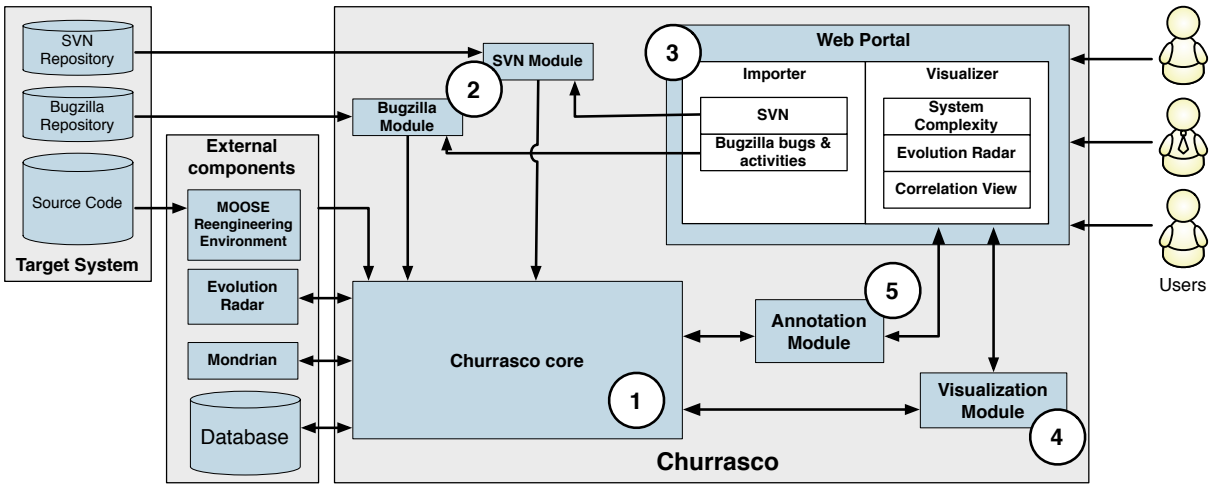


Fig. 1. The architecture of Churrasco.

– An overall discussion of the promises and perils, followed by a detailed survey at the current and incoming technologies in web software development, in order to provide guidance over one of the most important perils, the peril of rapidly changing technologies.

Structure of the paper. In Section 2 we introduce two web-based software visualization tools that we have developed: Churrasco and the Small Project Observatory, and distill a number of promises and perils for porting such tools to the web in Section 3. We report on two small-scale experiments involving our web based tools, highlighting the impact of some of the promises and perils in practice Section 4. We summarize the lessons learned in developing our tools in Section 5. In Section 6 we present the technologies one can use to implement a web-based visual application. We then look at related work on software visualization tools in and out of the web (Section 7), and conclude in Section 8.

2 Churrasco and SPO

In the last years we have developed two web-based software visualization tools: *Churrasco* and the *Small Project Observatory*), available respectively at http://churrasco.inf.usi.ch and http://spo.inf.usi.ch.

2.1 Churrasco

Churrasco [7] is a web platform for collaborative software analysis with the following characteristics:

– It provides a web interface to create models of software systems and of their evolution, and to store them in a database for subsequent analysis.
– It provides a set of visual analyses and supports collaboration by allowing several users to annotate the shared 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.

2.1.1 Architecture

Figure 1 depicts Churrasco's architecture, consisting of[1]:

1. *The core* connects the various modules of Churrasco and external components. It includes the internal representation of a software system's evolution and manages the connection with the database to write models imported from the web interface and to read models to be visualized in the web portal.
2. *The Bugzilla and SVN modules* retrieve and process the data from SVN and Bugzilla repositories.
3. *The Web portal* represents the front-end of the framework (developed using the Seaside framework [12]) accessible through a web browser. It allows users to create the models and to analyze them by means of different web-based visualizations.
4. *The Visualization module* supports software evolution analysis by creating and exporting interactive Scalable Vector Graphics (SVG) visualizations. The visualizations are created by two external tools: Mondrian [42] and the Evolution Radar [10] . The visualization module converts these visualization to SVG graphics. To make them interactive within the web portal, Churrasco attaches Asynchronous Javascript And XML (AJAX) callbacks to the figures, allowing server-side code to be executed when the user selects a figure.
5. *The Annotation module* supports collaborative analysis by enriching any entity in the system with annotations. It communicates with the web visualizations to integrate the annotations within the visualizations.

[1] Churrasco itself, without the external components, is made of 259 classes.

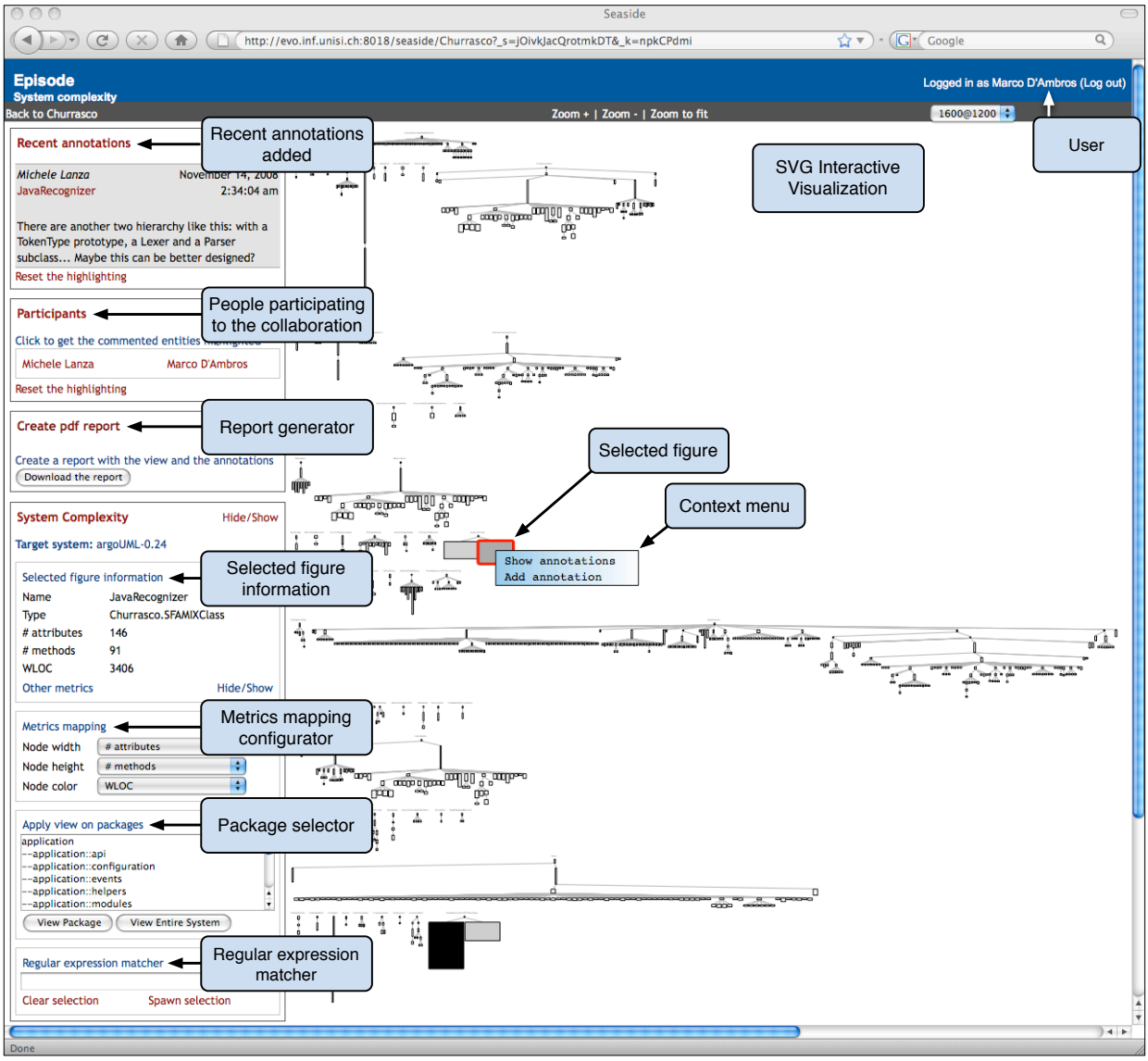


Fig. 2. A screenshot of the Churrasco web portal showing a System Complexity visualization of ArgoUML (http://argouml.tigris.org).

2.1.2 Visualizations

Churrasco offers the following interactive visualizations to support software evolution analysis:

- *The Evolution Radar* [6, 9] 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.
- *The System Complexity* [31] view supports the understanding of object-oriented systems, by enriching a simple two-dimensional depiction of classes and inheritance relationships with software metrics.
- *The Correlation View* shows all classes of a software system 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.

Figure 2 shows an example of a System Complexity visualization [31] 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 to customize the view by changing the metrics mapping.
2. The package selector 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.

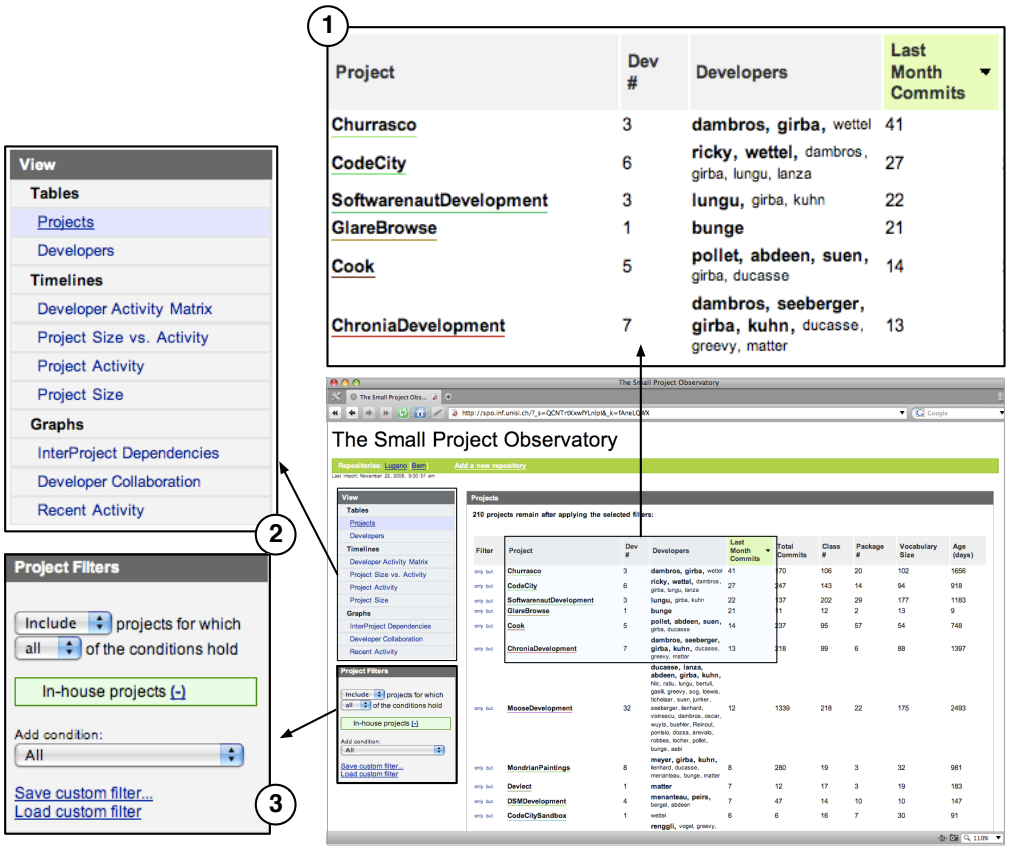


Fig. 3. The user interface of SPO: (1) Detail on the main project overview; (2) the View panel which allows selecting various visual perspectives on the analyzed super-repository; (3) the filter composition panel.

2.1.3 Collaboration Support

A key idea behind Churrasco is collaboration: 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.

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. Since the annotations are stored in a central 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 panels aimed at supporting collaboration:

1. The "Recent annotations" panel displays the most recent annotations added, together with the name of the annotated entity. 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. When one of these names is selected, all figures annotated by the corresponding person are highlighted in the view, to see which part of the system that person is working on.
3. The "Create pdf report" panel generates a pdf document containing the visualization and all the annotations referring to the visualized entities.

2.2 The Small Project Observatory

The Small Project Observatory (SPO from hereafter) is an interactive web application targeted at the visualization and analysis of entire software ecosystems.

Software Ecosystems. Software systems are seldom developed in isolation. On the contrary, many companies, research institutions and open-source communities deal with software projects developed in parallel and depending on one another. Such collections of projects represent assets and analyzing them as a whole can provide useful insights into the structure of the organization and its projects. We define a *software ecosystem* as a collection of software projects which are developed and evolved together in the same environment.

The large amounts of code that is developed in an ecosystem makes it hard, if not impossible for a single person to keep track of the complete picture. Many times, even if there exists documentation to describe the inter-dependencies between the projects and the way the developers and teams are supposed to collaborate, it is out of date or inaccurate. Thus, the only reliable source of information about the ecosystem is the data present in the versioning repositories of the projects. Such a collection of version control repositories for the projects of an ecosystem is called a super-repository.

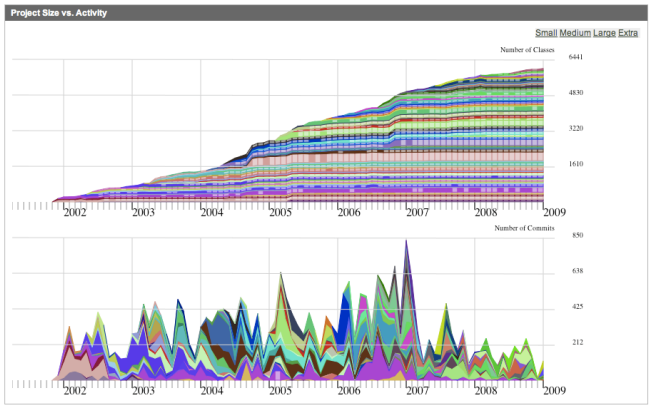


Fig. 4. Size and activity evolution for the projects in the SCG ecosystem.

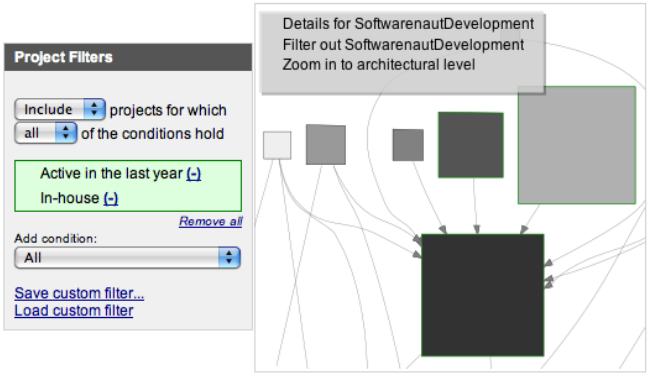


Fig. 5. Two ways of setting project filters in SPO: by composing rules and by interactively eliminating elements from the active viewpoint.

2.2.1 SPO Overview

Figure 3 shows a screen capture of SPO. The figure presents three concepts that are fundamental to the philosophy of SPO:

Multiple Perspectives. SPO provides multiple visual perspectives on a super-repository. The focus of each perspective can be either on the developers or on the projects in the system. Each perspective can present an overview of the entire ecosystem or a detailed view on an individual element (developer or project) which is to be understood in the broader context of the entire ecosystem.

Figure 3 presents the Views panel (labeled 1) which contains a list of all the available perspectives. Once the user has selected one perspective, the central view (labeled 2) displays a specific perspective on a super-repository. In this case it is a table that presents metrics about the projects in the super-repository. The view is interactive: The user can select and filter the available projects, sort the displayed projects, obtain contextual menus for the projects or navigate between various perspectives.

Figure 4 shows a visual perspective of a super-repository hosted by the Software Composition Group from the University of Bern, in Switzerland. The perspective presents two timelines displayed in parallel: the growth of the size (top graph) and the fluctuation of the activity (bottom graph). The size is measured in number of classes while the activity is measured in number of commits. The figure shows that size is monotonically increasing while the activity fluctuates over time with regularities and with a general trend being visible. One of the regularities is the dip in activity towards the end of every year and in the summer. This rhythm corresponds to the holiday periods of students. The general trend shows increase in activity until the peak of January 2007 when there are 700 commits. After that date, the overall activity level seems to have fallen.

Filtering. Given the sheer amount of information residing in a super-repository, filters need to be applied to the super-repository data. The panel labeled (3) in Figure 3 lists the active filters. The only active filter is "In-house projects". The user can choose and combine existing filters. A user can also apply filters through the interactive view, for example by removing a project or focusing on a specific project using the contextual menu (see Figure 5).

Interaction. The visual perspectives are interactive in SPO, meaning that every element of the view can be selected either for navigation, or filtering. The right side of Figure 5 shows a pop-up menu that appears when the user interacts with individual elements in one of the visual perspectives of SPO.

2.2.2 Navigation

Navigation is at the core of every information visualization tool, and this is the case also with SPO. Initially SPO was designed to support navigation between the different perspectives on the system. However, as we were using the tool we realized that one type of navigation it misses is vertical navigation: navigating between views which present information at different levels of abstraction. One example would be, navigating from a view which presents the inter-dependencies between all the systems in an ecosystem to a view which presents the architecture of one of these systems. We already had a tool that was supporting the visualization of software architecture at the individual system level. To support vertical navigation, SPO requests architectural views from Softwarenaut. Softwarenaut, which runs in the background, can export its output to SVG and deliver it to SPO to depict it in the user interface. Figure 6 presents an architectural view loaded in SPO. The two user interface elements highlighted are:

- *The list of available architectural views* presenting all views that are available for the given system. In Figure 6 there are two views available: the one called *main*, and the one called *main with tests*.
- *The list of available queries* that can be used for selection. Currently two types of queries are available:
  1. Queries that detect elements of the system that interact with the ecosystem. For example, all the classes that have methods that are called from the ecosystem, or all the classes that are subclassed in the ecosystem.

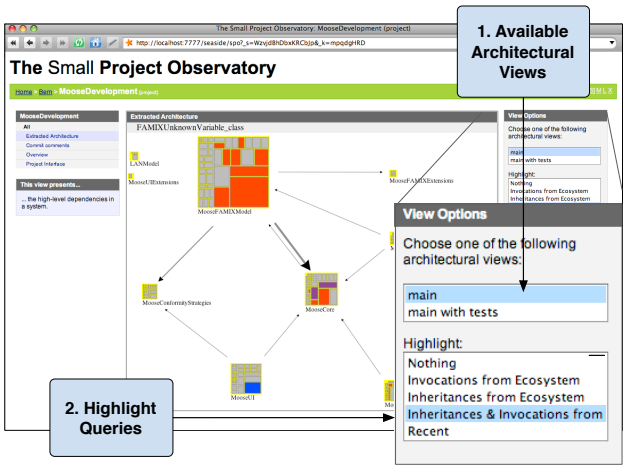


Fig. 6. Visualizing in SPO an architectural view that was generated in Softwarenaut.

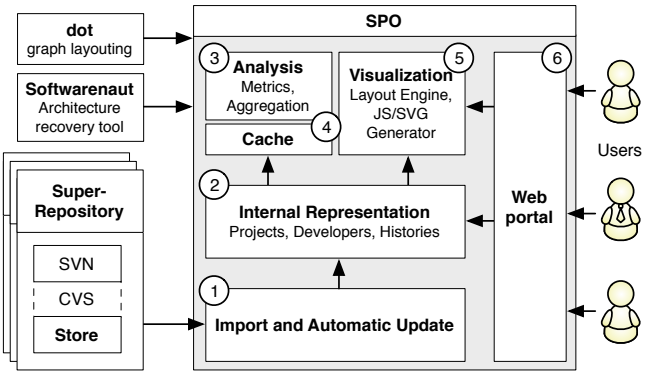


Fig. 7. The architecture of SPO.

2. Queries that detect elements that were active at certain periods in the lifetime of the system. For example, all the classes that were active recently.

2.2.3 Architecture of SPO

Figure 7 presents SPO's architecture, consisting of[2]:

1. *The import module* is responsible for interfacing with the super-repository. Currently SPO supports two types of super-repositories: one based on SVN and another one based on Store, a Smalltalk-specific repository.
2. *The internal representation* is a meta-model [35] for representing super-repositories and ecosystems. SPO supports the analysis of multiple models at the same time.
3. *The analysis module* is responsible with computing metrics, discovering collaborations, analyzing developer and project vocabularies, aggregating dependencies, and all the other types of analysis that are be performed on an ecosystem model.
4. *The cache module*. Due to the highly interactive and exploratory nature of the tool, SPO generates dynamically all the web pages and all the visualizations they contain. This module caches across sessions all the information that is needed in order to speed-up the view generation.
5. *The visualization module* takes as input information from the internal representation, analysis and cache modules and generates views from it. The module contains the layout engine, which delegates the layouting to the Dot external tool[3], and the SVG generator. The generator produces the SVG graphics and the associated Javascript interaction.
6. *The web portal* is the user interface of SPO. Like Churrasco, it is built on top of the Seaside framework, a web application framework which emphasizes a component based approach to web application development.

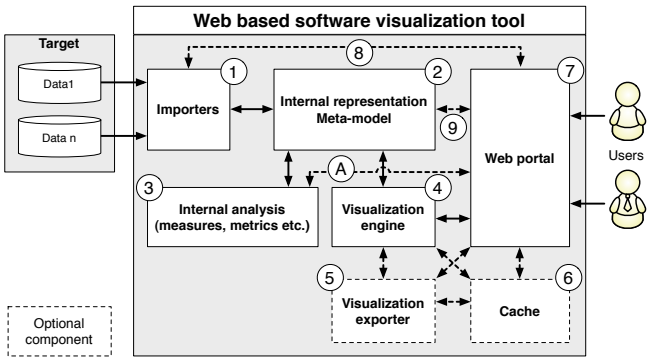


Fig. 8. General architecture of a web-based software visualization tool.

2.3 *Beyond Churrasco and SPO*

Figure 1 and Figure 7 show the architecture of Churrasco and SPO. We abstracted a general architecture for web-based software visualization tools displayed in Figure 8.

Dashed elements are optional components. Software visualization tools provide views on one or more aspects (e.g., source code, bug report, mail archive, etc.) of a software. Therefore, they have an importer module (1) which retrieves the data and stores it according to an internal representation (2). The data is then optionally processed to compute metrics (3) about the considered aspects. The data is finally visualized by means of a visualization engine (4): In case the engine does not produce a web suitable visualization, an exporter (5) is used to create the web visualization. To improve the performances one can use a cache component (6) which avoids recomputing the visualizations. The software visualization tool has a web portal which displays the visualizations (7), imports the data (8), accesses the models (9), and computes the metrics (A).

[2] SPO itself, without dot and Softwarenaut, is composed of 110 classes.

[3] See http://www.graphviz.org

Availability and Privacy	
Promise 1	Porting software visualization tools (SVT) to the web makes them more available than desktop applications
Peril 1	Sensitive information about software systems should not be available for not authorized people
Collaboration and Performance	
Promise 2	Porting SVT to the web eases making them collaborative
Peril 2	Web-based software visualization applications (WBSVA) have to serve large amounts of data to several users, which can be a performance bottleneck impacting all users
Promise 3	WBSVA ease the creation of an incrementally enriched knowledge about a software
Error Handling	
Peril 3	WBSVA are single points of failure
Peril 4	Debugging and testing web applications is hard
Promise 4	WBSVA provide feedback about errors
Promise 5	WBSVA make it possible to gather usage data
Development	
Peril 5	WBSVA have to tackle cross browser issues
Peril 6	Developing interactive web applications is hard
Peril 7	Web technologies are changing fast
Promise 6	WBSVA can use external tools to perform a number of tasks, exposing only the results as services
Promise 7	Updating an WBSVA is easy since it is only done once for all the users
Measurements	
Promise 8	One can selectively deploy changes to a group of users and measure their effect

Table 1. Summary of promises and perils.

3 Promises and Perils

In this section, we recall our experience building Churrasco and SPO and extract various aspects in the form of promises and perils, summarized in Table 1.

Promise 1 - Availability: Porting software visualization tools to the web makes them more available than desktop applications.

Many research prototypes have problems with respect to their availability. Often such prototypes are hard to install because of compatibility issues, missing libraries, missing documentation, etc. Among the various reasons behind the availability problem, one is that researchers do not have the manpower required to create and update documentation, maintain the software, keep the web site (when existing) up-to-date, etc. Moreover, academic research is mostly publication-driven, and not tool-driven, i.e., there is little direct benefit that comes with maintaining tools.

Tracking the evolution of systems and components requires further effort, as compatibility issues occur over time when new versions of components the tool depends on are released. Having the application running on a Web server means that the environment can be frozen, so that supporting the latest version of a component is not a priority.

Indeed, porting research prototypes to the web increases the availability of such tools and avoids installation problems. In the case of both Churrasco and SPO all that needs to be given to users is the url.

Peril 1 - Privacy: Sensitive information about software systems should not be available for unauthorized people.

Having a tool available on the web implies that anybody can access it. Web-based software visualization tools might have access to sensitive information about a software system, which should be accessible only by authorized people. For this reason, such tools should provide an authorization mechanism that is not required for desktop applications.

In Churrasco we tackled this problem by letting only registered users access the visualizations, and by giving different users different privileges. SPO does not implement authentication yet. As a result, when we approached an industrial partner for a case study on the ecosystem of the company, the partner declined to import their data in the online version of SPO. They installed a local version of SPO on their intranet and performed the analysis themselves.

Promise 2 - Collaboration: Porting software visualization tools to the web eases the process of making them collaborative.

Sharing the data among users naturally leads to collaboration. Virtually all software is nowadays built in a collaborative fashion. This ranges from the usage of software configuration management systems (SCM) supporting distributed development, now widely used in practice [18], awareness tools to prevent upcoming conflicts [50], to fully integrated solutions such as IBM's Jazz [21].

Just as the software development teams are geographically distributed, consultants and analysts are too. Analysis tools supporting collaboration would allow different experts with a distinct range of skills to collaboratively analyze a software system from various locations and/or time zones.

Churrasco supports collaboration using a central database: Different users access the same web portal, and analyze the

same models of software systems. Users collaborate by annotating the model entities and by looking at other people's annotations. This simple collaboration facility proved useful in the experiment we report on in Section 4. Improving it via the addition of richer communication channels, such as chat or tagging, is easy to achieve in a web application.

Desktop applications can also support collaboration, but we argue that this is harder to implement. In this case, the various instances of the application need a communication channel among themselves directly in a peer-to-peer fashion or using a centralized server. This leads to networking issues due to firewalls. We are not aware of software visualization tools which support collaboration, but a number of visualization tools in other domains support it [2, 17].

Peril 2 - Performance and Scalability: Collaborative, visual web applications have to serve large amounts of data to several users at the same time, which can be a performance bottleneck impacting all users.

Web applications have to serve several users at the same time, and collaborative applications even more so. Depending on the number of users and the type of application, the performance per user might decrease. This is especially true for visualization applications, where for large datasets both the computation time and the size of the data to be sent to the user's browser might be large, increasing the user's waiting time and thus decreasing the usability of the application.

Visualization must scale up as it is most useful to deal with large amounts of data. Since the visualizations are rendered on the client side, bandwidth can become and issue. For example, in Churrasco an SVG graphic visualizing the ArgoUML software system (ca. 1800 classes) is larger than 1 MB, while SPO generated SVG images going up to 2MB. SPO however reduces the bandwidth by compressing the data to be sent, effectively trading CPU usage for increased bandwidth. In that case the 2MB file was reduced to 150KB.

The standard way of rendering a web visualization is that every time something changes in the page, the whole page is refreshed. 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. Refreshing the entire web page for every action introduces latencies which make the web application slow when it comes to rendering large SVG files. One way to avoid this problem is to use semantic zoom and details on demand to keep the rendered image small. Churrasco can focus on a single package of a system, while SPO allows the definition of filters. Another possibility is to minimize the page refreshes by using AJAX updates, which refresh only the changed part of the page, as Churrasco does. However, while the use of AJAX has been simplified, it is still non-trivial. The current standard is to use libraries such as Prototype or jQuery.

Concurrent usage is an issue in the context of collaborative work. With Churrasco and SPO we performed two experiments, with 8 participants each, with mixed results with respect to performance (see Section 4). Due to the small number of participants we refrain from making general statements.

This peril can be tackled by having several instances of the web application running on several servers, with a web server responsible of dispatching the requests and balancing the CPU and bandwidth loads. While this solution is standard fare in web applications, for research prototypes such a hardware infrastructure is often not available. However, when infrastructure is an issue, one can exploit cloud computing services which provide data replication and scalability transparently. Typical examples of cloud computing service providers are Google, Amazon and Salesforce.

Promise 3 - Incremental results: Web-based software visualization tools ease the creation of an incrementally enriched body of knowledge on software systems.

Despite performance and scalability issues, sharing the data paves the way for new possibilities. Results of analyses on software systems produced by tools are often written into files and/or manually crafted reports, and have therefore a limited reusability. To maximize their reuse, analysis results should be incrementally and consistently stored back into the analyzed models. This would allow researchers to develop novel analyses that exploit the results of previous analyses, leading to a cross-fertilization of ideas and results. It can also serve as a basis for a benchmark for analyses targeting the same problem (i.e., by tagging entities that a candidate analysis should detect, we can compare approaches), and ultimately would also allow one to combine techniques targeting different problems.

By using a central database where all the models are stored, and by letting users annotate the entities composing the models, the users can store the results of the analysis on the model itself, thus supporting the incremental storage of results. This is supported in Churrasco, and can be easily implemented in other web-based software visualization applications, in the same fashion.

Peril 3 - Single point of failure: Web-based applications are single points of failure.

Excessive centralization reduces the reliability of the application. Web-based applications run on a server, and usually have a unique instance of the application which all the users access. As a consequence, if the application crashes it will lock out all its users, i.e., the application represents a single point of failure, whereas in desktop applications each user has a private instance of the application, where a crash does not impact the other users. This peril can be tackled, together with performance, by distributing the computation on several servers for redundancy.

Peril 4 - Debugging and testing: Debugging and testing web applications is hard.

A barrier to develop web applications is the lack of support for debugging. Even if there are some applications like Firebug (http://getfirebug.com) providing HTML inspection, Javascript debugging and DOM exploration, the debugging support is not comparable with the one given in mainstream IDE such as Eclipse. Moreover, the testing of a web-based system is hard to perform, due to the lack of consolidated techniques and supporting tools.

Promise 4 - Feedback: Web-based software visualization tools provide feedback about errors and failures.

If debugging a web application is more difficult than a desktop one, being notified of bugs and deploying the fixes is actually easier. Because of the restricted manpower available when developing them, research prototypes are far from being mature and stable applications. Indeed, researchers do not have the resources to invest a significant amount of time testing their application. These problems impact the usage of the tools and therefore their adoption by other researchers or people from industry. One way to be notified about these issues is to instrument the tool so that if it crashes, it collects information about the run-time scenario and then asks the users to send this information back to the developers. This widely adopted approach requires a significant infrastructure and is therefore mostly used in commercial applications.

By having the tool as a web service, the tool is always running on the server, and therefore the tool developer can be notified of all bugs and failures. Bug fixes also do not need to be distributed to individual users, but are available to all users at once.

Promise 5 - Usage report: Web applications make it possible to gather precise usage data.

Similarly to error notifications, gathering usage data is easy. With desktop applications it is possible to track the number of downloads of a tool, and the tool might be instrumented to send back feedback about how it is used. This is however not straightforward to implement. Web-based applications offer the possibility to exploit standard solutions to the usage statistics problem, such as Google analytics. This allows developers to easily gather usage statistics and infer popular features or usability problems, to continuously improve the tool. As with bug fixes, deploying updates is transparent.

Peril 5 - Browser compatibility: Web applications have to tackle cross browser issues.

Web browsers are a rather diverse crowd, and the fact that a web application works with one browser does not guarantee that it works with other browsers. While many compatibility issues can be solved, such as how CSS (Cascading Style Sheets) are interpreted, others cannot. In these cases the users have to focus on a particular web browser to exploit the full functionality of the web application.

Visualization applications have requirements which make this situation more probable: For instance, Churrasco uses AJAX callbacks to update SVG depictions without refreshing the entire web page. The SVG DOM update in AJAX is supported only by Firefox and, as a consequence, Churrasco is only fully functional with Firefox.

SVG is a W3C specification and most of the recent versions of major web browsers support it: Opera and Safari support it without AJAX update and Internet Explorer supports it through a third party plug-in. However, not all the browsers have the same speed in rendering it, which makes the user experience unpredictable. To test this, we wrote a simple Javascript program which calculates the rendering speed of various browsers. We ran the script in OS X on a PowerBook G4 running at 1.5GHz with 1GB of RAM. The differences between the browsers are very large. For example, in one second Opera 9.50 renders 595 polygons while Safari only renders 77. This simple benchmark shows two of the greatest limitations of SVG: The amount of visual elements that one can render is limited (at least currently) and the user experience is hard to predict, as the timings will be different for users with different system configurations. Also, we encountered problems with the same pop-up menu being rendered differently in two browsers.

Other technical choices such as Flash or Javascript (with APIs such as Processing.js or the Javascript InfoVis Toolkit) may alleviate these problems. Javascript in particular has seen a resurgence of interest among web browser builders who now compete over their Javascript performance (see Section 6 for details about these issues).

Finally, it is not unreasonable to require a widespread browser such as Firefox over Internet Explorer if the benefits of the application are promising enough.

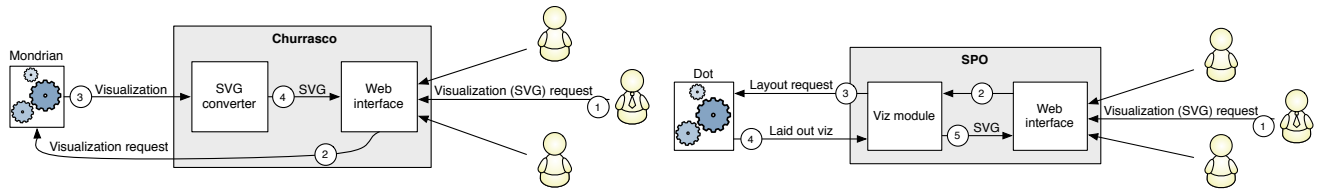
Peril 6 - Interaction: Developing interactive web applications is harder than desktop applications.

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

However, libraries of reusable components are quickly developing, such as Prototype, script.aculo.us and jQuery for Javascript, which should alleviate this problem. We provide a more detailed discussion on this in Section 6.

Peril 7 - Rapid evolution: Web technologies are changing fast.

The dust is far from settled in the web technology arena. As we saw above, several technologies (SVG, Flash, Javascript, etc.) are currently competing. These technologies are rapidly



(a) Churrasco uses the Mondrian framework in the backend to create visualizations and then it converts them as SVG interactive graphics.

(b) SPO uses the Dot external tool to layout its visualization.

Fig. 9. Two examples of using external tools in Churrasco and SPO.

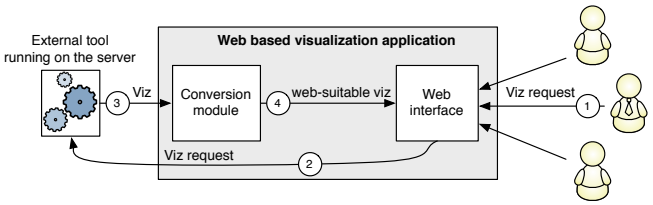


Fig. 10. The general schema for using external tools in web-based visualization applications and hiding them behind the web interface.

evolving: New possibilities are emerging, and the amount of support among browsers varies. This rapid evolution makes it difficult to choose which tools/libraries/technologies to use, and to maintain the web application aligned with the rapidly evolving technologies. Developers must be watchful of new opportunities and potentially capable to switch to newer technologies when needed. We hope that, with time, standard solutions will emerge for highly interactive, graphical web applications.

Promise 6 - Hiding tasks and exposing services: Web-based visualization applications can use external tools to perform tasks, exposing the results as services.

Some aspects of web application development are however easier. Implementing software visualization tools as web applications allows the developer to use external tools in the backend, hiding them from the users. On the contrary, in desktop applications external tools have to be included in the application distribution, and they should run on the client machine (which might also have installation problems like the application itself). In short, the web application developer has total control over the environment the application is executing in.

The use of external tools offers a lot of reuse opportunities, such as layout engines. For example, Churrasco reuses two external tools (Mondrian [42] and the Evolution Radar [10]) to create visualizations, which are then converted to SVG by a dedicated module of Churrasco (see Figure 9(a)). This enables us to freely reuse all the visualizations and layouts provided by Mondrian and the Evolution Radar. SPO is dispatching the layouting of its visualizations to Dot, a Unix command line layout algorithm library (see Figure 9(b)).

SPO also exposes the service of Softwarenaut [36], an architecture recovery tool whose visualizations where adapted to the Web. Moreover, SPO is processing huge amounts of data (entire super repositories) when there are no user connected, i.e., exploiting idle time, caching the results and presenting them on-demand to the users. In this way, SPO is hiding heavy computations and presenting only the results as a lightweight service. Churrasco does the same thing when, given the url of a SVN or Bugzilla repository, it sends an email to the user when the data is imported.

Figure 10 shows how the usage of external tools can be generalized: The web interface gets the request for a visualization and dispatches it to an external tool. The result is then converted in a web-suitable format and sent back through the web interface to the clients' web browsers.

Promise 7 - Updating and maintaining: Updating a web-based visualization application is easy since it is only done once for all the users.

In our experience with developing visualization tools as desktop applications, usually deploying a new version takes weeks or months, since one needs to put up a new release and then inform all the users.

One of the main advantages of having a visualization tool available for the web is the ability to update and maintain the application without distributing and installing software on numerous client computers. The updates can be done only once on the server. This promise is one of the building blocks of promises 4 and 5, as they rely on the instant availability of updates. The associated risk is that defective updates will also propagate instantly to all users; careful testing is needed.

This promise is more general then just for visualization applications, but we feel like this is one of the strong arguments that will bring more software visualization applications to the web in the future.

Promise 8 - Selective deployment and feedback: One can selectively deploy changes to a group of users and measure their effect.

Web applications being easier to update and providing feedback allows one to measure the effects of changes on the users. Assuming an application has a steady amount of users, and gathers usage statistics about how the users are using it, one can measure the effect of changes in the following way:

- The users are divided in two groups, one using the application with the change (such as the introduction of a novel visualization, or changes to an existing one), while the second group uses the application without the change. The possibility of deploying updates transparently, thanks to Promise 7, makes this possible.
- The application gathers usage statistics about both groups of users as they are using the application (using Promise 5 in order to do so). The monitoring can be as fine-grained as needed (i.e., recording individual mouse clicks on web page elements, with their time stamps). If the monitoring already in place is insufficient, it can be deployed as another update as well (and removed later on if it proves to be detrimental to performance, such as if it increases communications between clients and the server beyond what is expected).
- A suitable performance metric can be devised and computed on the collected data, in order to assess the impact of the change introduced. One could for example measure if a novel visualization produces a statistically significant decrease of the time needed to perform a given task by comparing the timestamps of events, or evaluate differences in correctness if one took time to tag beforehand the entities that a given task is supposed to uncover (as mentioned in Promise 3).

This promise is important for visualization techniques, which are usually hard to evaluate without performing a controlled experiment. Such a technique could allow one to deploy enhancements and measure their impact using a lighter and more automated process than a regular controlled experiment would allow.

4 Promises and Perils in Practice

We report on two experiments we performed on small groups of users, in order to test some of the promises and perils we described in a real-life setting. In particular, we test the impact of Peril 2 (performance), and the benefits of Promise 1 (availability), Promise 2 (Collaboration), and Promise 7 (Ease of updates).

4.1 A Collaboration Experiment with Churrasco

We performed a collaboration experiment using Churrasco, with the following goals: (1) evaluate whether Churrasco is a good means to support collaboration in software evolution analysis (Promise 2), (2) test the usability of the tool as an exemplar of a web-based reverse engineering and visualization tool (Promise 1), and (3) test the scalability of the tool with respect to the number of participants (Peril 2).

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 out 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.

4.1.1 Case study and tasks

The task consisted in using two Churrasco visualizations (System Complexity and 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. 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 11 shows a System Complexity of JMol in which the size of nodes maps to the number of attributes (width) and methods (height) and the nodes' color represents the amount of annotations they received (the darker the color, the more the annotations), 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.

4.1.2 Usage of collaborative annotations

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.).

4.1.3 User survey

At the end of the experiment all participants but one filled out a survey about the tool and the collaboration experience. The survey used a Likert scale [33]; its results are shown in Table 2.

Although not a full-fledged user experiment, it provided us with information about our initial goals. The survey shows that the participants found the tool easy to use: This is important in the context of web-based tools, and especially with respect to Promise 1 as the goal is to lower the users' barrier to entry. Moreover, the survey provides us feedback about Promise 2: Participants found collaboration important in reverse engineering and Churrasco as a good means to support collaboration (for the participants the experiment was the first collaborative reverse engineering experience). Informal user

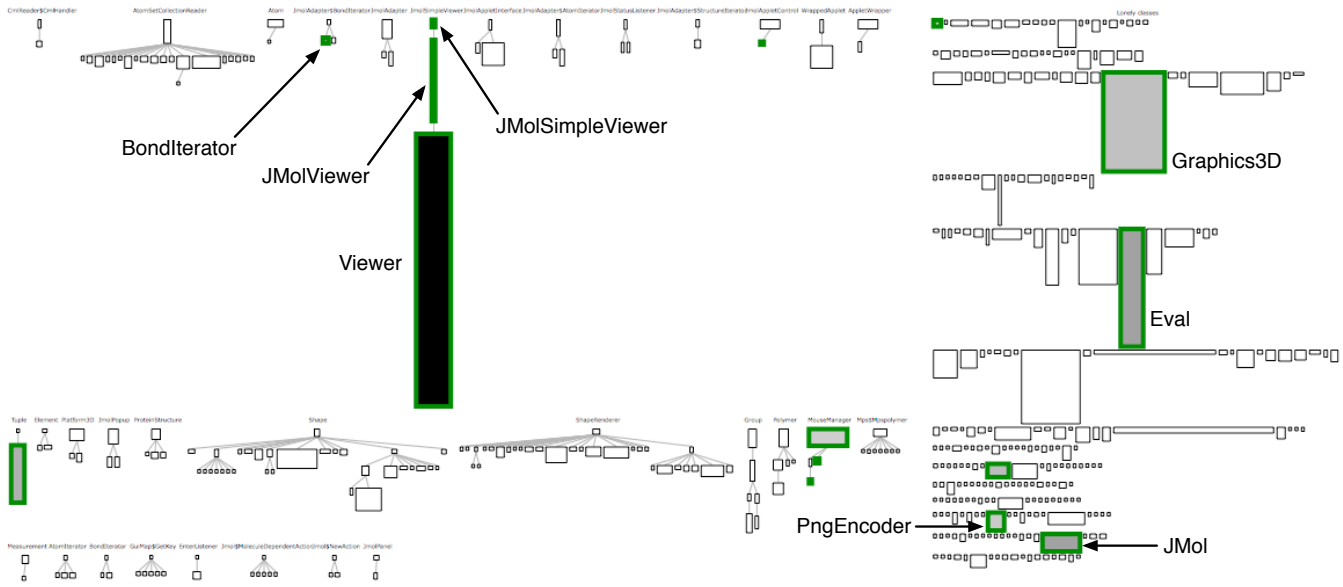


Fig. 11. A System Complexity of JMol. The color denotes the amount of annotations made by the users. The highlighted classes (thick boundaries) are annotated classes.

Assertion	SA (%)	A (%)	N (%)	D (%)	SD (%)
Churrasco is easy to use	33	50	17		
System Complexity view is useful	73	27			
Correlation view is useful	72	14	14		
Churrasco is a good means to collaborate	100				
Collaboration is important in reverse engineering	14	72	14		

Table 2. Evaluating the usability and collaboration support of Churrasco (SA=strongly agree, A=agree, N=Neutral, D=disagree, SD=strongly disagree).

comments from the users stated that they especially liked to be notified of 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. Further, Churrasco scaled well with 8 people accessing the same model on the web portal at the same time, without any performance issue, even if we did not implement any load-balancing scheme: Churrasco was running on a 3 GHz, dual-processor server at the time. This alleviates our concerns about the scalability peril somewhat.

4.2 A Usability Experiment with SPO

To verify the usability and usefulness of SPO, we conducted an experimental study in the context of the Software Evolution course at the University of Lugano. The course is a master level course.

During one of the labs we introduced the students to the concept of a software ecosystem and then presented the Project Observatory. After that, we gave the students one hour of time to analyze an academic ecosystem and report on their understanding as well as the usability of the tool. The ecosystem that we used as case study is the one hosted by the Software Composition Group from the University of Berne, an ecosystem which contains tens of developers and hundreds of projects.

At the end of the analysis the students had to answer several questions that were testing their understanding of the relationships between the developers in the ecosystem as well as the importance and relationships between the projects in the ecosystem (e.g., Which project is more important in for the ecosystem A, or B? Which developer is more critical to the ecosystem?).

During the experiment, we had the chance of testing Promise 7. At one point, soon after the beginning of the experiment, one of the students discovered a bug in the application. We immediately fixed the bug and updated the application on the server, such that all the participants could benefit from the fix.

After answering the questions regarding the ecosystem, the students had to rate on a Likert scale their own understanding of the various aspects of the ecosystem. The majority felt that the analysis session was useful in supporting their understanding of the analyzed ecosystem.

At the end of the experiment, we asked the participants to fill out a survey on the usability of the tool. Table 3 shows that in general the participants were happy with the UI and ease of use of the tool. We report more on the case-study elsewhere [39].

The main complaint was the slowness of the tool and the lack of scalability when presenting large graphs. These problems were not inherent in the web-based nature of the application, but rather they were problems with the back-end implementation which represented a computational bottleneck. In fact the application was not slow in our previous tests, but

Assertion	SA (%)	A (%)	N (%)	D (%)	SD (%)
Application was easy to use	20	70	10		
Application was responsive enough		10	30	40	20
Interaction features were satisfying		30	60	10	

Table 3. Evaluating the usability of SPO (SA=strongly agree, A=agree, N=Neutral, D=disagree, SD=strongly disagree).

that was because we only tried it with a single user at a time before. This was a confirmation of Peril 2 - the use of the tool by multiple users at the same time resulted in a performance degradation that we did not see before.

When asked about the interaction capabilities of the tool 30% of the students were satisfied, 60% were neutral and 10% were not satisfied. This means that we have to work more on the interaction aspects of SPO. Students also mentioned that the filtering capacities were very important and the current filtering that SPO offers needs to be improved. However, none of the observations were really specific to the fact that the tool was run in the browser. In fact, the high expectations that the students had from the tool were probably the result of being used to highly interactive web-based applications.

5 Discussion

We argue that in developing a web-based software visualization tool the benefits of the promises are greater than the mostly technical issues and challenges of the perils. In particular, we argue that the most important promises are:

- *Availability*. In the Introduction we observed that 80% of tools presented in TOSEM in the last 8 years are not even installable. The web can improve this situation.
- *Reuse*. We showed that with web applications it is possible to hide tasks and provide services. Porting or creating a web visualization requires a smaller implementation effort, as not only libraries but even entire external tools can be reused.
- *Collaboration*. Collaboration is getting more and more attention both in forward and reverse engineering. We believe that this trend will continue and collaboration will play a key role in these domains in the following years. We discussed how and why, with web applications, supporting collaboration is easier with respect to desktop applications. Our experiment with Churrasco showed that users used the collaborative annotions when presented with the option to do so.
- *Selective Deployment*. Once an application gathers a steady stream of users, selective deployment of enhancements allows one to measure their effect in a convenient fashion. The ease of access of a web application allows one to easily recruit potential users to evaluate the enhancement on as well.

To increase their survival chances, every software visualization tool, in the long run, should have a web front-end. This does not require a huge implementation effort because many existing tools can be just reused, and it will increase the accessibility of the application and its adoption.

The perils of developing web applications should be however taken into account. The peril of performance in particular is one we were confronted with when we performed our experiments on Churrasco and SPO: Not all the users found the applications responsive enough for their tastes. However, no measure was taken to ensure performance at the time. Standard techniques such as load-balancing can alleviate this problem. Finally, the peril of rapid evolution is also a concern: In such a rapidly evolving domain, it is especially important to evaluate which technology fits best the developer needs when it comes to porting or creating a web visualization. Nowadays the choice is among a number of technologies that we discuss in the next section.

6 Technologies

In this section, we list the array of technologies available presently to implement software visualization applications, with a focus on the ones allowing rich presentations with graphical and interactive elements. The technologies we consider are Javascript (using Canvas and/or SVG), Flash, Silverlight, and Java applets. We summarize all the libraries and frameworks that we mention in this section in Table 4.

6.1 Javascript and DHTML

Javascript is the standard scripting language of web pages. It is a powerful language which combines functional and prototypical paradigms. Historically, the support for Javascript was variable among browsers, with some browsers providing the same functionality differently. With time the browser implementations of the language became better and more performant and the popularity of the language increased. With the standardization of the DOM by the W3C the way was paved for building interactive web applications by dynamically modifying the content of a page; This combination of Javascript and DOM manipulation is called Dynamic HTML (DHTML).

Once DHTML started to get traction, frameworks and libraries that mask the quirks and differences of individual browsers have emerged, offering a unified front to the programmer. Two of the most widespread libraries are Prototype and jQuery, which simplify the operations needed to manipulate the contents of a web page, and do so while abstracting the behavior differences of browsers. Several frameworks also exist to ease the building of applications featuring a graphical user interface, such as Dojo, script.aculo.us, Sprout Core, Mootools, the Yahoo UI Library, or the Google Web Toolkit. All these frameworks provide both traditional GUI widgets and advanced graphics, charting, and interaction widgets.

Library/Framework	Available at	Goal	License
Prototype	http://www.prototypejs.org	Simplify Javascript programming and DOM manipulation	MIT
Dojo	http://dojotoolkit.org	Provide basic language extensions, and a rich set of widgets	BSD
jQuery	http://www.jquery.com	Ease DOM traversing, event handling, animating, Ajax interactions	GPL, MIT
script.aculo.us	http://script.aculo.us	Improve user interface	MIT
Sprout Core	http://www.sproutcore.com	Move the app logic to the client (the server deliveries only the data)	MIT
Mootools	http://mootools.net	Simplify and improve Javascript programming	MIT
Yahoo UI Library	http://developer.yahoo.com/yui/	Build scalable, fast, robust and interactive web applications	BSD
Google Web Toolkit	http://code.google.com/webtoolkit/	Create JavaScript front-end applications in Java	Apache 2.0
Processing.js	http://processingjs.org	Program visualizations, animations, and interactions in Javascript	MIT
Cake	http://code.google.com/p/cakejs/	Support scene graph visualizations in Javascript	MIT
Raphael	http://raphaeljs.com	Simplify working with vector graphics in Javascript	MIT
InfoVis Toolkit	http://thejit.org	Create interactive data visualizations for the web	BSD
Flare	http://flare.prefuse.org	Create interactive visualizations in Flash	BSD
Google Data Explorer	http://www.google.com/publicdata/home	Explore datasets with interactive Flash-based visualizations	-

Table 4. Libraries and frameworks available to improve the web experience and to support web-based visualization.

With Javascript one can dynamically modify a page based on interaction events triggered by the user, allowing for the production of interactive graphics on a web page. At the moment, there are two main supporting technologies that allow the insertion of graphics in a page. The first is SVG (Scalable Vector Graphics), a declarative XML-based language for vector graphics specification. The second is the Canvas element introduced by Apple in their WebKit component and part of the forthcoming HTML 5 standard.

SVG has a tree structure just as the HTML DOM, and this allows current browsers to make SVG elements become part of the DOM. This means that approaches that generate and manipulate HTML can be easily adapted to integrate with SVG as well. One can attach event handlers to SVG elements, and use Javascript to add or alter the structure of the SVG graphic. SVG also supports animations.

The canvas tag allows one to define a zone on the web page where one can draw programmatically through Javascript. Several visualization libraries have been built on top of the HTML canvas to abstract commonly used functionalities, such as Processing.js, Cake, Raphael and the InfoVis Toolkit. All these libraries allow one to build event handlers on top of graphical elements as well.

These technologies are based on standards, yet the support for those is not complete. For example, as of February 2010 in a sample of web accesses retrieved by Stat Owl[4], 67% were performed by browsers not supporting SVG. Internet Explorer's support for SVG and the canvas element is weak. There exist workarounds, but they are not fully satisfactory yet. Version 9 of Internet Explorer should address these issues, but it is far from being released at this moment of writing. On the subject of performance, Javascript and especially SVG are slower than Flash, Java applets and Silverlight, although the situation is changing as browsers are competing on Javascript performance nowadays. According to the JS Benchmark[5] Chrome 4.0 is the browser with the best Javascript performance, followed by Safari 4.0 (1.1 times slower), Opera 10.50 (1.4 times slower), Firefox 3.6 (2.6 times slower), Konqueror 4.3 (5.2 times slower) and IE 8.0 (5.6 times slower).

6.2 Java applets, Flash and Silverlight

Java applets are Java applications that can run in a web browser through a Java Virtual Machine. They were designed to provide interactive features to web applications that could not be provided by HTML alone. Applets were introduced in the first version of the Java language in 1995. Although applets were supported by the majority of web browsers, and had the advantage of being cross-platform, they did not become mainstream. Another Java technology that makes applications easier to deploy and install is Java Web Start. It allows applications to be downloaded in the browser, and to be run in an independent sandbox. However, as applications deployed with Java Web Start do not run in a web browser, they do not benefit from the novel advantages offered by web technologies.

Adobe Flash and Microsoft Silverlight are multimedia platforms that integrate graphics, animations, multimedia and interactivity into a single runtime environment. While Flash is an well-established technology, introduced in 1996, Silverlight is relatively new, as its first version was released in April 2007.

The Flash, Silverlight and Java technologies require the installation of plugins to launch the applications that they are written in, as they are not natively supported by browsers. Of the three, Flash has the most significant market-share: According to Stat Owl, 96% of the browsers have Flash support, while for Java and SilverLight the percentages are respectively 81% and 39%[6]. Two popular Flash-based visualization frameworks are Flare and Google Public Data Explorer. The latter is an application which provides four types of interactive visualizations to "explore" datasets: Line chart, bar chart, maps and bubble chart. Flare is an ActionScript (the language to write Flash application) library for creating visualizations: from simple charts to complex interactive graphics.

[4] http://www.statowl.com

[5] http://jsbenchmark.celtickane.com

[6] In a sample population of web accesses retrieved from September 2009 to February 2010.

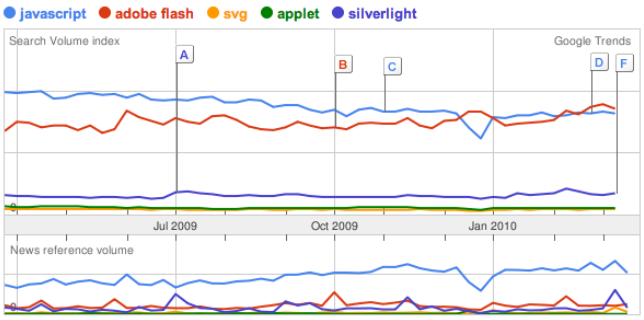


Fig. 12. Technological trends over the last year (as of March 2010), in terms of Google searches and mentions in news articles of Javascript, Adobe Flash, Java applets, SVG and Silverlight.

One downside of these technologies is their weak integration with the browser. A Flash application is usually seen as a "black box", which does not communicate with the rest of the web page. This implies that a web visualization tool would probably need to be implemented either entirely in Flash, or suffer from the limitations of the communication between the components of the application. Using Javascript on the other hand allows one to access all the elements of the web page at once. Churrasco and SPO use SVG graphics, that, in response to user interactions, alter also the HTML content of the page. Were the visualizations to be implemented in Flash, most of the HTML content would have to be rewritten in Flash as well, in order to be updated in response to interactions.

6.3 The Bottom Line

Figure 12 shows the result of Google Trends on how much the technologies we present in this section are discussed over time. We can consider it as a predictor of how these technologies are supported among developers. In the figure, we clearly see that the two main contenders are Flash and Javascript. This is reflected in terms of available libraries: For instance, considerable effort has been invested to make Javascript frameworks able to support cross-browser compatibility, while other solutions, such as Java applets and SVG graphics, do not have such a support.

Deciding between a Javascript solution and a Flash-based solution depends on several factors. For example, in terms of current and future compatibility with browsers, at the moment Flash enjoys a wider compatibility. However, this might not continue, since, on the one hand, several mobile devices do not support Flash and, on the other hand, the support for the HTML 5 standard is growing. Other two factors against Flash are its proprietary technology and the fact that its content does not cooperate well with the host HTML. However, Flash still offers better performances and multimedia capabilities (although a visualization application may not need to perform advanced tasks, such as playing back video).

7 Related Work

7.1 Software Visualization

The goal of software visualization is to support the understanding of large amounts of data, when the question one wants to answer about the data cannot be expressed as queries. Software visualization approaches vary with respect to two dimensions. The first dimension is the type of visualized data, for which visualizations can be classified as: Static (using the system's structure), dynamic (using it's runtime behavior), or evolutionary (using it's history). The second dimension is the level of abstraction on the data. Different levels exist for each visualization type of the first dimension. Based on their abstraction level, we distinguish three main classes of software visualization approaches: Code-level, design-level and architectural-level.

Code-Level Visualization Line-based software visualization has been addressed in a number of approaches. The first tool which uses a direct code line to pixel line visual mapping to represent files in a software system is SeeSoft, proposed by Eick et al. in 1992 [16]. On top of this mapping, SeeSoft superimposes other types of information such as which developer worked on a given line of code or which code fragments correspond to a given modification request. Later, Ball and Eick focused on the visualization of different source code evolution statistics such as code version history, difference between releases, static properties of code, code profiling and execution hot spots, and program slices [3]. Marcus et al. extended the visualization techniques of SeeSoft by exploiting the third dimension in a tool called sv3D [41].

Ducasse et al. worked at a finer granularity level, using a character to pixel representation of methods in object-oriented systems. The authors enriched this mapping with semantic information to provide overviews of the methods in a system [15].

Telea et al. proposed a code level visualization technique called Code Flows, which displays the evolution of source code over several versions [53]. The visualization, based on a code matching technique which detects correspondences in consecutive ASTs, is useful to both follow unchanged code and detect important events such as code drift, splits, merges, insertions and deletions.

Augur [20] is a code level visualization tool which combines, within one visual frame, information about both software artifacts and the activities of a software project at a given moment (extracted from SCM logs). Another tool working at the code level is CVSscan [58].

Design-Level Visualization The next level of abstraction, after code, is the design level where visualizations focus on self contained pieces of code, such as classes in object oriented systems. UML diagrams are the industry standard for representing object-oriented design. Researchers investigated techniques to enrich and extend standard UML diagrams. Termeer et al. developed the MetricView tool which augments

UML class diagrams with visual representation of class metrics extracted from the source code [54].

Researchers also investigated different visualization techniques to represent source code at the design level. Lanza introduced the *polymetric views* [31], a lightweight software visualization technique which renders software entities and software relationships enriched with software metrics. Polymetric views can be enriched with dynamic or semantical information. Orla et al. exploited a 3D visualization to add execution trace information to polymetric views in a tool called TraceCrawler [24]. The tool is a 3D extension of CodeCrawler [32], the tool where Lanza originally implemeted polymetric views. Ducasse et al. enriched polymetric views with information extracted from control flow analysis in a visualization called *class blueprint* [14].

Cornelissen et al. proposed a trace visualization method [5] based on a massive sequence and circular bundle view [25], implemented in a tool called ExtraVis. ExtraVis shows the systems structural decomposition (e.g., in terms of package structures) and renders traces on top of it as bundled splines, enabling the user to interactively explore and analyze program execution traces.

Another direction of research is the use of metaphors to represent software. Wettel et al. argue that a city is an appropriate metaphor for the visual representation of software systems [59] and implement it in their CodeCity tool [60], where buildings represent classes and districts represent packages. Kuhn et al. used a cartography metaphor to represent software systems [30]. In their Software Cartographer tool the authors use a consistent layout for software maps in which the position of a software artifact reflects its vocabulary, and distance corresponds to similarity of vocabulary.

A number of evolutionary visualizations were proposed at the design level, rendering information extracted from SCM logs. Taylor and Munro used visualization together with animation to study the evolution of a CVS repository [52]. Rysselberghe and Demeyer used a simple visualization of CVS data [56] to recognize relevant changes in the software system. Wu et al. used the spectograph metaphor to visualize how changes occur in software systems [61]. The Ownership Map [23], introduced by Gîrba et al., visualizes code ownership of files over time, based on information extracted from CVS logs. The Evolution Radar visualizes co-change information extracted from SCM logs, integrating different levels of abstraction, to support the analysis of the coupling at the module level and the understanding of the causes at the file level [10].

Architectural-Level Visualization The highest level of abstraction is the architecture level, consisting of system's modules and relationships among them. In 1988 Müller et al. introduced *Rigi* [43], the first architectural visualization tool. Rigi is a programmable reverse engineering environment which provides interactive visualizations of hierarchical typed graphs and a Tcl interpreter for manipulating the graph data. Other architecture visualization tools were built on top of it [27, 46] and it inspired other architectural visualization projects. Two of them were Shrimp [51] and its Eclipse-based continuation Creole [34]. These tools display architectural diagrams using nested graphs where graph nodes embed source code fragments.

Lungu et al. introduced Softwarenaut [36], an architectural visualization and exploration platform on top of which they experimented with automatic exploration mechanisms [37]. Knodel et al. proposed a tool called SAVE [29] which uses UML-like figures to represent architectural components. Jazayeri et al. used a three-dimensional visual representation at the architectural level for analyzing a software system's release history [26]. Gall et al. used a graph based representation to visualize historical relationships among system's modules extracted from the release history of a system [22]. The authors applied the visualization to analyze historical relationships among modules of a large telecommunications system and showed that it supported the understanding of the system architecture. Pinzger et al. proposed a visualization technique based on Kiviat diagrams [47]. The visualization provides integrated views on source code metrics in different releases of a software system together with coupling information computed from CVS log files.

Summing Up We surveyed software visualization approaches and tools in the literature: They vary with respect to the data they visualize and the abstraction level they address. However, none of the mentioned approaches is web based, while both Churrasco and SPO are. Concerning the classification of our tools as software visualization approaches, Churrasco is an evolutionary approach at the design level, while SPO is an evolutionary approach at the architecture level.

7.2 Web Based Software Visualization

There is a wide range of visualization tools that work on the web. One of the most well-known is ManyEyes, a web site where users may upload data, create interactive visualizations, and carry on discussions [57]. The goal of the site is to support collaboration around visualizations at a large scale by fostering a social style of data analysis. Recently Google introduced the Data Explorer, another application targeted at general data visualization.

To our knowledge, besides Churrasco and SPO, the only web-based software visualization tools are Tesseract [49] and the Java applet version of Shrimp[7]. Tesseract is a Flash-based tool which provides interactive visualizations of relationships between files, developers, bugs and e-mails. The main difference between our tools and Tesseract is that Churrasco and SPO address the problem of understanding a system's (or eco-system's) evolution, while the goal of Tesseract is to support the analysis of the socio-technical relations between code, developers, and issues. Shrimp running as a Java applet is identical to the desktop version, functionality wise, but slower in terms of performance. While Shrimp supports the

[7] Available at http://www.thechiselgroup.com/shrimp

exploration of software architecture, our tools focus on software evolution analysis.

Apart from Tesseract and Shrimp, the most related work are software visualization tools which produce outputs readable by web browsers, and web-based software analysis tools without visualizations. Beyer and Hassan proposed Evolution Storyboards [4], a technique that offers dynamic views. The storyboards, rendered as SVG files, depict the history of a project using a sequence of panels, each representing a particular time period. These visualizations are partially interactive: They only show the names of the entities in the figures. In contrast the views offered in Churrasco and SPO are fully interactive, providing context menus for the figures and navigation capabilities. The Evolution Storyboard is not a web application, but a tool producing SVG files readable by browsers.

Nentwich et al. introduced BOX, a portable, distributed and interoperable approach to browse UML models [45]. BOX translates a UML model in XMI to 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. As the Evolution Storyboard, BOX is not a web application but a tool which can produce output readable by some web browsers.

Mancoridis et al. presented REportal, a web-based portal site for the reverse engineering of software systems [40]. 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. For doing that the authors exploited promise 6 - Hiding tasks and exposing services. REportal supports software analysis through browsing and querying, but does not offer interactive visualizations.

Finnigan et al. developed the Software Bookshelf, a web-based paradigm for the presentation and navigation of information representing large software systems [19].

While we are aware of one research project that aims at developing a web-based IDE [55], we believe that this trend will continue and this kind of efforts will be duplicated by other researchers in the future.

8 Conclusion

Building software visualization tools for the web is a daunting task that we experienced first-hand when we implemented two web-based tools, Churrasco and SPO. We documented our experiences in the form of promises and perils of such a transition, and evaluated some of these promises and perils in practice by means of two usabilities studies of the tools we implemented.

The transition to the web has a variety of technological consequences making some tasks harder (e.g., debugging, scaling), but some other easier (e.g., error reporting, maintenance). The web is a moving target: Technologies and standards are rapidly changing, and one must regularly assess the technological choices made in the light of changing support across browsers. We did such an assessment, as of March 2010, and found that the leading contenders are Javascript and Flash. Flash is currently more performant, but tends to not cooperate well with the rest of the web page, hence limiting its usefulness if the visualizations and the rest of the page need to communicate.

If completed, a transition to the web is rewarding: A web-based tool has a greater visibility and potential impact, as people can work with it without needing to install it. A web platform also makes collaboration a more probable possibility, as the costs to implement it are lower than in standalone applications. As our experiment showed, once given the possibility, people will effortlessly use the collaborative facilities.

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

References

1. Rigi–an environment for software reverse engineering, exploration, visualization, and redocumentation. *Science of Computer Programming*, 75(4):247 – 263, 2010. Experimental Software and Toolkits (EST 3): A special issue of the Workshop on Academic Software Development Tools and Techniques (WASDeTT 2008).
2. C. Bajaj and S. Cutchin. Web based collaborative visualization of distributed and parallel simulation. In *Proceedings of the IEEE symposium on Parallel visualization and graphics (PVGS 1999)*, pages 47–54. IEEE Computer Society, 1999.
3. Timothy Ball and Stephen Eick. Software visualization in the large. *IEEE Computer*, 29(4):33–43, 1996.
4. Dirk Beyer and Ahmed E. Hassan. Animated visualization of software history using evolution storyboards. In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE 2006)*, pages 199–210. IEEE CS Press, 2006.
5. Bas Cornelissen, Andy Zaidman, Danny Holten, Leon Moonen, Arie van Deursen, and Jarke J. van Wijk. Execution trace analysis through massive sequence and circular bundle views. *J. Syst. Softw.*, 81(12):2252–2268, 2008.
6. Marco D'Ambros and Michele Lanza. Reverse engineering with logical coupling. In *Proceedings of WCRE 2006 (13th Working Conference on Reverse Engineering)*, pages 189–198. IEEE CS Press, 2006.
7. Marco D'Ambros and Michele Lanza. A flexible framework to support collaborative software evolution analysis. In *Proceedings of CSMR 2008 (12th IEEE European Conference on Software Maintenance and Reengineering)*, pages 3–12. IEEE CS Press, 2008.
8. Marco D'Ambros and Michele Lanza. Visual software evolution reconstruction. *Journal of Software Maintenance and Evolution: Research and Practice (JSME)*, 21(3):217–232, May 2009.
9. Marco D'Ambros, Michele Lanza, and Mircea Lungu. The evolution radar: Visualizing integrated logical coupling information. In *Proceedings of MSR 2006 (3rd International Workshop on Mining Software Repositories)*, pages 26–32, 2006.

10. Marco D'Ambros, Michele Lanza, and Mircea Lungu. Visualizing co-change information with the evolution radar. *Transactions on Software Engineering (TSE)*, 35(5):720 – 735, 2009.
11. Marco D'Ambros, Michele Lanza, and Martin Pinzger. "a bug's life" — visualizing a bug database. In *Proceedings of VISSOFT 2007 (4th IEEE International Workshop on Visualizing Software For Understanding and Analysis)*, pages 113–120. IEEE CS Press, 2007.
12. S. Ducasse, D. Pollet, M. Suen, H. Abdeen, and I. Alloui. Package surface blueprints: Visually supporting the understanding of package relationships. In *Proceedings IEEE International Conference on Software Maintainance (ICSM 2007)*, pages 94–103, Los Alamitos CA, October 2007. IEEE CS Press.
13. Stéphane Ducasse, Tudor Gîrba, and Oscar Nierstrasz. Moose: an agile reengineering environment. In *Proceedings of ESEC/FSE 2005*, pages 99–102, 2005.
14. Stéphane Ducasse and Michele Lanza. The class blueprint: Visually supporting the understanding of classes. *Transactions on Software Engineering (TSE)*, 31(1):75–90, January 2005.
15. Stéphane Ducasse, Michele Lanza, and Romain Robbes. Multi-level method understanding using microprints. In *Proceedings of VISSOFT 2005 (3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis)*, pages 33–38, 2005.
16. Stephen G. Eick, Joseph L. Steffen, and Sumner Eric E., Jr. SeeSoft—a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, November 1992.
17. Klaus Engel and Thomas Ertl. Texture-based volume visualization for multiple users on the world wide web. In Michael Gervaut, Dieter Schmalstieg, and Axel Hildebrand, editors, *Proceedings of the Eurographics Workshop in Vienna, Austria*, pages 115–124, 1999.
18. Jacky Estublier, David Leblang, André van der Hoek, Reidar Conradi, Geoffrey Clemm, Walter Tichy, and Darcy Wiborg-Weber. Impact of software engineering research on the practice of software configuration management. *ACM Transactions on Software Engineering and Methodology*, 14(4):383–430, October 2005.
19. P. Finnigan, R. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. Mueller, J. Mylopoulos, S. Perelgut, M. Stanley, and K. Wong. The software bookshelf. *IBM Systems Journal*, 36(4):564–593, November 1997.
20. Jon Froehlich and Paul Dourish. Unifying artifacts and activities in a visual tool for distributed software development teams. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 387–396, Washington, DC, USA, 2004. IEEE Computer Society.
21. Randall Frost. Jazz and the eclipse way of collaboration. *IEEE Software*, 24(6):114–117, 2007.
22. Harald Gall, Karin Hajek, and Mehdi Jazayeri. Detection of logical coupling based on product release history. In *Proceedings International Conference on Software Maintenance (ICSM '98)*, pages 190–198, Los Alamitos CA, 1998. IEEE Computer Society Press.
23. Tudor Gîrba, Adrian Kuhn, Mauricio Seeberger, and Stéphane Ducasse. How developers drive software evolution. In *Proceedings of International Workshop on Principles of Software Evolution (IWPSE 2005)*, pages 113–122. IEEE Computer Society Press, 2005.
24. Orla Greevy, Michele Lanza, and Christoph Wysseier. Visualizing live software systems in 3d. In *SoftVis '06: Proceedings of the 2006 ACM symposium on Software visualization*, pages 47–56, New York, NY, USA, 2006. ACM.
25. Danny Holten. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):741–748, 2006.
26. Mehdi Jazayeri, Harald Gall, and Claudio Riva. Visualizing Software Release Histories: The Use of Color and Third Dimension. In *Proceedings of ICSM '99 (International Conference on Software Maintenance)*, pages 99–108. IEEE Computer Society Press, 1999.
27. R. Kazman and S. J. Carrière. View extraction and view fusion in architectural understanding. In *Proceedings of the 5th International Conference on Software Reuse (ICSR 1998)*, page 290, Washington, DC, USA, 1998. IEEE Computer Society.
28. Mik Kersten and Gail C. Murphy. Using task context to improve programmer productivity. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT 2006/FSE-14)*, pages 1–11, New York, NY, USA, 2006. ACM.
29. Jens Knodel, Dirk Muthig, Matthias Naab, and Mikael Lindvall. Static evaluation of software architectures. In *CSMR'06*, pages 279–294, Los Alamitos, CA, USA, 2006. IEEE Computer Society.
30. Adrian Kuhn, Peter Loretan, and Oscar Nierstrasz. Consistent layout for thematic software maps. In *WCRE '08: Proceedings of the 2008 15th Working Conference on Reverse Engineering*, pages 209–218, Washington, DC, USA, 2008. IEEE Computer Society.
31. Michele Lanza and Stéphane Ducasse. Polymetric views — a lightweight visual approach to reverse engineering. *Transactions on Software Engineering (TSE)*, 29(9):782–795, September 2003.
32. Michele Lanza, Stéphane Ducasse, Harald Gall, and Martin Pinzger. Codecrawler — an information visualization tool for program comprehension. In *Proceedings of ICSE 2005 (27th IEEE International Conference on Software Engineering)*, pages 672–673. ACM Press, 2005.
33. R. Likert. A technique for the measurement of attitudes. *Archives of Psychology*, 22(140):1–55, 1932.
34. Rob Lintern, Jeff Michaud, Margaret-Anne Storey, and Xiaomin Wu. Plugging-in visualization: experiences integrating a visualization tool with eclipse. In *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization*, pages 47–ff, New York, NY, USA, 2003. ACM.
35. Mircea Lungu. *Reverse Engineering Software Ecosystems*. PhD thesis, University of Lugano, Switzerland, October 2009.
36. Mircea Lungu and Michele Lanza. Softwarenaut: Exploring hierarchical system decompositions. In *Proceedings of CSMR 2006 (10th IEEE European Conference on Software Maintenance and Reengineering)*, pages 349–350. IEEE CS Press, 2006.
37. Mircea Lungu, Michele Lanza, and Tudor Gîrba. Package patterns for visual architecture recovery. In *Proceedings of CSMR 2006 (10th IEEE European Conference on Software Maintenance and Reengineering)*, pages 183–192. IEEE CS Press, 2006.
38. Mircea Lungu, Michele Lanza, Tudor Gîrba, and Reinout Heeck. Reverse engineering super-repositories. In *Proceedings of WCRE 2007 (14th IEEE Working Conference on Reverse Engineering)*, pages 120–129. IEEE CS Press, 2007.
39. Mircea Lungu, Michele Lanza, Tudor Girba, and Romain Robbes. The small project observatory: Visualizing software

ecosystems. *Journal of Science of Computer Programming (SCP)*, 75(4):264–275, April 2010.
40. Spiros Mancoridis, Timothy S. Souder, Yih-Farn Chen, Emden R. Gansner, and Jeffrey L. Korn. Reportal: A web-based portal site for reverse engineering. In *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE 2001)*, page 221. IEEE Computer Society, 2001.
41. Andrian Marcus, Louis Feng, and Jonathan I. Maletic. 3d representations for software visualization. In *Proceedings of the ACM Symposium on Software Visualization*, pages 27–ff. IEEE, 2003.
42. Michael Meyer, Tudor Gîrba, and Mircea Lungu. Mondrian: An agile visualization framework. In *Proceedings of Softvis 2006 (3rd International ACM Symposium on Software Visualization)*, pages 135–144. ACM Press, 2006.
43. H. A. Müller and K. Klashinsky. Rigi-a system for programming-in-the-large. In *ICSE '88: Proceedings of the 10th international conference on Software engineering*, pages 80–86, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.
44. Hausi A. Müller. *Rigi — A Model for Software System Construction, Integration, and Evaluation based on Module Interface Specifications*. PhD thesis, Rice University, 1986.
45. Christian Nentwich, Wolfgang Emmerich, Anthony Finkelstein, and Andrea Zisman. BOX: Browsing objects in XML. *Software Practice and Experience*, 30(15):1661–1676, 2000.
46. Liam O'Brien and Christoph Stoermer. Architecture reconstruction case study. Technical report, CMU/SEI-2001-TR-026, 2001.
47. Martin Pinzger, Harald Gall, Michael Fischer, and Michele Lanza. Visualizing multiple evolution metrics. In *Proceedings of SoftVis 2005 (2nd ACM Symposium on Software Visualization)*, pages 67–75, St. Louis, Missouri, USA, May 2005.
48. Romain Robbes and Michele Lanza. Spyware: A change-aware development toolset. In *Proceedings of ICSE 2008 (30th ACM/IEEE International Conference in Software Engineering)*, pages 847–850. ACM Press, 2008.
49. Anita Sarma, Larry Maccherone, Patrick Wagstrom, and James Herbsleb. Tesseract: Interactive visual exploration of socio-technical relationships in software development. *Software Engineering, International Conference on*, 0:23–33, 2009.
50. Anita Sarma, Zahra Noroozi, and André van der Hoek. Palantír: Raising awareness among configuration management workspaces. In *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003)*, pages 444–454, 2003.
51. Margaret-Anne D. Storey and Hausi A. Müller. Manipulating and documenting software structures using SHriMP Views. In *Proceedings of ICSM '95 (International Conference on Software Maintenance)*, pages 275–284. IEEE Computer Society Press, 1995.
52. Christopher Taylor and Malcolm Munro. Revision towers. In *Proceedings 1st International Workshop on Visualizing Software for Understanding and Analysis*, pages 43–50, Los Alamitos CA, 2002. IEEE Computer Society.
53. Alexandru Telea and David Auber. Code flows: Visualizing structural evolution of source code. In *Proc. 10th Eurographics/IEEE Symposium on Data Visualization (EuroVis 2008)*, volume 27, pages 831–838. Eurographics, 2008.
54. M. Termeer, C. F. J. Lange, A. Telea, and M. R. V. Chaudron. Visual exploration of combined architectural and metric information. In *VISSOFT '05: Proceedings of the 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*, page 11, Washington, DC, USA, 2005. IEEE Computer Society.
55. Arie van Deursen, Ali Mesbah, Bas Cornelissen, Andy Zaidman, Martin Pinzger, and Anja Guzzi. Adinda: A knowledgeable, browser-based ide. In *Companion Proceedings of the 32nd International Conference on Software Engineering (ICSE NIER)*. ACM, 2010.
56. Filip Van Rysselberghe and Serge Demeyer. Studying software evolution information by visualizing the change history. In *Proceedings 20th IEEE International Conference on Software Maintenance (ICSM '04)*, pages 328–337, Los Alamitos CA, September 2004. IEEE Computer Society Press.
57. Fernanda B. Viegas, Martin Wattenberg, Frank van Ham, Jesse Kriss, and Matt McKeon. Manyeyes: a site for visualization at internet scale. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1121–1128, 2007.
58. Lucian Voinea, Alex Telea, and Jarke J. van Wijk. CVSscan: visualization of code evolution. In *Proceedings of 2005 ACM Symposium on Software Visualization (Softviz 2005)*, pages 47–56, St. Louis, Missouri, USA, May 2005.
59. Richard Wettel and Michele Lanza. Program comprehension through software habitability. In *Proceedings of ICPC 2007 (15th IEEE International Conference on Program Comprehension)*, pages 231–240. IEEE CS Press, 2007.
60. Richard Wettel and Michele Lanza. Codecity: 3d visualization of large-scale software. In *ICSE Companion '08: Companion of the 30th ACM/IEEE International Conference on Software Engineering*, pages 921–922. ACM, 2008.
61. Jingwei Wu, Richard Holt, and Ahmed Hassan. Exploring software evolution using spectrographs. In *Proceedings of 11th Working Conference on Reverse Engineering (WCRE 2004)*, pages 80–89, Los Alamitos CA, November 2004. IEEE Computer Society Press.