Science of Computer Programming 75 (2010) 264–275

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

The Small Project Observatory: Visualizing software ecosystems

Mircea Lungu [a,*], Michele Lanza [a], Tudor Gîrba [b], Romain Robbes [a]

[a] REVEAL @ Faculty of Informatics, University of Lugano, Switzerland
[b] Software Composition Group, University of Bern, Switzerland

ARTICLE INFO

Article history:
Received 1 December 2008
Received in revised form 17 August 2009
Accepted 3 September 2009
Available online 11 September 2009

Keywords:
Software evolution
Software visualization
Software ecosystems
Reverse engineering
Maintenance

ABSTRACT

Software evolution research has focused mostly on analyzing the evolution of single software systems. However, it is rarely the case that a project exists as standalone, independent of others. Rather, projects exist in parallel within larger contexts in companies, research groups or even the open-source communities. We call these contexts software ecosystems. In this paper, we present the Small Project Observatory, a prototype tool which aims to support the analysis of software ecosystems through interactive visualization and exploration. We present a case study of exploring an ecosystem using our tool, we describe the architecture of the tool, and we distill lessons learned during the tool-building experience.

1. Introduction

Software visualization tools span a wide range of abstraction levels. One of the first visualization tools was Eick's SeeSoft [1] which summarizes changes in the software at the level of lines of code. Increasing the level of abstraction, other tools present UML diagrams or other representations that work at the class level [2,3]. Others focus on visualizing higher-level abstractions such as modules and the interdependencies between them [4,5]. At the next higher abstraction level, Holt visualized architectural differences between two versions of a system [6].

An underlying assumption of these approaches is that a project represents one single, complex entity that can be analyzed in isolation. However, we argue that this is often not the case. Instead, projects exist in larger contexts such as companies, research groups or open-source communities and corresponding versioning repositories exist in parallel. We call these context ecosystems.

In this article we present the Small Project Observatory (SPO), a tool that embodies our approach to analyzing software ecosystems. Our first goal when building SPO was to provide an interactive interface for the visualization and exploration of software ecosystems. A few other research projects analyze such groups of projects and treat them as whole, but none of them provides an interactive tool to support the analysis. One such project is FlossMole which provides a database compilation of open-source projects from Source-Forge and several other repositories [7]. Weiss analyzed Source-Forge from a statistical point of view [8].

Groups of software projects have been also analyzed in various instances by the members of the Libresoft research group from Spain whose main interest is analyzing open-source software. In this context they looked at various collections of systems to study their properties and their evolution. In one instance, they analyzed the Debian Linux distribution and estimated the cost of implementing it from scratch [9]. The analysis of the Debian distribution was not focused on the source code but remained at a higher level. In another article, they proposed a quantitative methodology for analyzing how

* Corresponding address: REVEAL @ Faculty of Informatics, University of Lugano, Via Giuseppe Buffi 13, 6900 Lugano, Switzerland.
E-mail addresses: mircea.lungu@usi.ch (M. Lungu), michele.lanza@usi.ch (M. Lanza), girba@iam.unibe.ch (T. Gîrba), romain.robbes@usi.ch (R. Robbes).

0167-6423/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2009.09.004

the developer turnover affects open-source software projects [10]. In this work they took a few representative open-source projects and studied the information in the versioning system repositories without looking at entire groups of projects that belong together. A case where they indeed took entire collections of projects was when they studied the behavior of the developers from a social networking analysis point of view [11]. In this work they looked at the social networks that are built around the Gnome and Apache projects. All of the previous work does however consider group of projects as simple collections of projects, not ecosystems.

The Small Project Observatory is implemented as an online application. There are several other examples of using the Web for software engineering. Holt et al. have developed the Software Bookshelf, which is a web-based paradigm for the presentation and navigation of information representing large software systems [12]. Mancoridis et al. presented REportal which was aimed to be an online reverse engineering portal [13]. Recently D'Ambros et al. proposed an online framework for analysis and visualization in a software maintenance context. Their tool, Churrasco emphasizes flexibility by allowing easy model extension as well as collaboration [14]. Although they support loading multiple models at the same time in Churrasco, they do not consider an ecosystem as a first-class entity in their analysis.

The remainder of this article is organized as follows. In Section 2 we introduce the concept of a software ecosystem. In Section 3 we present the Small Project Observatory, its user interface, and an example of analysis performed using it. In Section 4 we talk about the architecture of the tool. In Section 5 we discuss about validation, interaction, developing for the web and other lessons learned during our tool-building experience. We conclude in Section 6 with a discussion and we outline the directions of future work.

2. No project is an Island

Software systems are seldom developed in isolation [15,16]. 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.

In this context we define a software ecosystem as follows:

A software ecosystem is a collection of software projects which are developed and evolve together in the same environment.

The environment is usually a large company, an open-source community, or a research group. It is even possible for multiple organizations to collaborate and develop software in a common ecosystem.[1]

One particular type of organization which owns ecosystems are telecommunications companies. They are supported by highly integrated IT systems developed decades ago and extended over time. Telecommunications companies usually develop software products for variations of the phone hardware and in these cases they usually have to manage the various versions which are slightly different by introducing product line families. A product family [17,18] is a set of software intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way. A product family is indeed a special type of ecosystem, but the ecosystem concept is much broader.

The large amounts of code that is developed in an ecosystem guarantees that it is very 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 interdependencies 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. In this context we define a super-repository as follows:

A super-repository represents a collection of version control repositories of the projects of an ecosystem.

Beside the information about the evolution of the projects in an ecosystem, super-repositories also contain information about developers. Depending on the organization, developers might work on multiple projects simultaneously, move from one project to another and collaborate on multiple projects at the same time. All this information can be recovered from the information in the super-repository.

Some repositories are dedicated to a particular language such as RubyForge[2] and SqueakSource,[3] while others are language agnostic such as SourceForge[4] and GoogleCode.[5] Although most of the discourse can be generalized to any of these repository types, this article is derived from our experience of analyzing three open-source and one industrial super-repositories, each containing the history of several dozens to hundreds of applications versioned in Store (Smalltalk Open Repository Environment).

---

1 For example, the Gnome family of systems is developed by an open-source community to which various companies contribute, too.

2 http://rubyforge.org.

3 http://squeaksource.org.

4 http://sourceforge.org.

5 http://code.google.com.

Table 1
The analyzed ecosystems.

Ecosystem	Projects	Contributors	Classes	Since
Cincom	288	147	19.830	2000
SCG (Bern)	210	76	10.600	2002
REVEAL (Lugano)	55	11	2.088	2005
Soops	249	20	11.413	2002

Store is a versioning control system for projects written in VisualWorks Smalltalk [19]. A Store repository contains multiple projects. In fact it is common for an organization to keep all its projects in a single Store repository. We have chosen to support the analysis of Store for two reasons: (1) our own projects were versioned with it and we wanted to start the analysis with ourselves and, (2) it is a very powerful version control system. Two of the most important characteristics of Store that we exploited are the detailed change information, and the prerequisites dependencies. Store captures detailed change information not at text level, but at the level of reified programming concepts such as methods and classes. Furthermore, the prerequisite mechanism allows the developer to specify for each project the other projects it depends on (*i.e.*, its prerequisites).

Why reverse engineer software ecosystems?

Reverse Engineering a software ecosystem means recovering high-level, abstracted views of the ecosystem from the low-level facts that exist in the super-repository associated with the ecosystem. Such facts can be relationships between projects, relationships between developers, or other evolutionary and structural views. Examples of questions that need to be answered in this context are: "What are the old projects that everybody depends on?", "How stable is a dependency between two or more projects?", "What are the projects that have been modified recently?", "What are the projects that gain importance?", "What are the projects which currently take the most effort?".

We identified at least three main categories of stakeholders that should be interested in the information about the evolution of the code in a super-repository, namely project managers, developers, and quality assessors.

Project Managers are interested in how teams work and how to allocate resources to the projects. Moreover, since in general successful projects need to continuously change [20,21], a project manager needs to be up to date with how projects change and what their current status is.

Developers need to know whom to ask questions about the source code [22]. They care about both the details of a particular project and the inter-project dependencies. In the open-source ecosystems developers are looking for projects they can contribute to. Since not all of them have equal chances of success, it is useful to gain insights about the evolution, activity and the people involved regarding a particular project.

The Quality Assessment Team is interested in the continuous supervision of the evolution of the projects in the ecosystem. The team could define alerts bound to certain events in the evolution of the ecosystem. For example, when a developer is adding a dependency between two projects, the dependency should be added automatically to the next code review. Another type of analysis which can benefit from the continuous monitoring of the system is searching for code patterns that appear multiple times inside a project or across projects. The detection of such code patterns can be a first step towards reusing them.

Case studies

We have analyzed software ecosystems in various contexts: public open-source communities, research groups and software companies. In Table 1 we provide a brief quantitative overview of these ecosystems.

The oldest and largest of them is the Open Smalltalk Repository hosted by Cincom Systems. The next two are maintained at the Universities of Bern and Lugano, in Switzerland. The last one is a repository maintained by Soops BV, a Dutch software development company. The data provided in Table 1 needs to be considered with care as the numbers are the result of a simple project counting. Super-repositories accumulate junk over time, as certain projects fail, some die, short-time experiments are performed, etc. This is inherent to the nature of super-repositories, and only supports the claim that they need to be understood in more depth.

All the examples and views presented in the remainder of this paper are based on analyzing the SCG ecosystem.

3. The Small Project Observatory

The Small Project Observatory (SPO)[6] is a highly interactive web application. In this section we describe some of the interaction modes and present an example exploration session.

6 SPO is available at http://spo.inf.unisi.ch/.

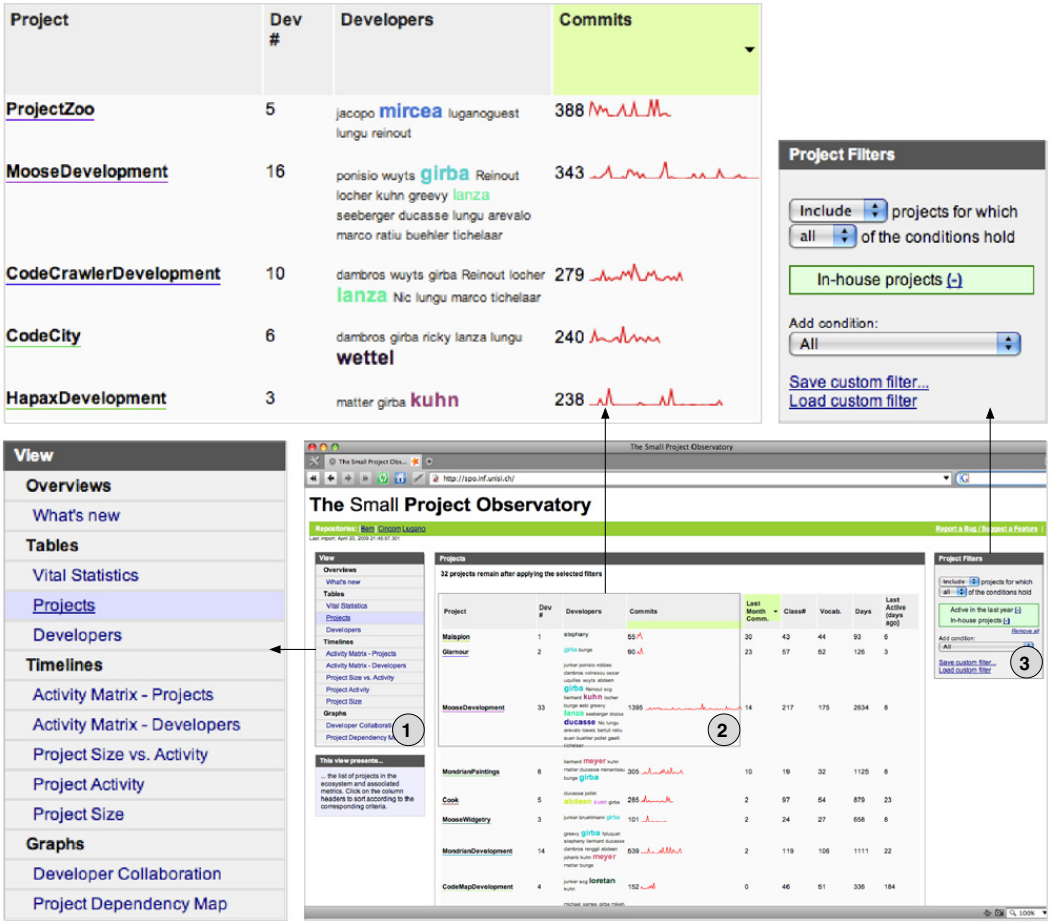

Fig. 1. The user interface of The Small Project Observatory.

Fig. 1 presents a screenshot of SPO running inside the Opera web browser during one of our ecosystem case studies. The central view (a fragment of which is 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 in the sense that the user can select and filter the available projects, sort the displayed projects, obtain contextual menus for the projects or navigate between various perspectives. This is true also for other graphical views - the elements can be individually selected and manipulated.

SPO provides multiple views on a repository and it enables the user to choose the ones which are appropriate for the type of analysis he needs. The Views panel (labeled 1) presents the list of all the available views, some of which we present later in this section.

Given the sheer amount of information residing in a super-repository, filters need to be applied to the super-repository data. The bottom-left panel lists the active filters (labeled 3). In the case of Fig. 1 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.

3.1. Exploring the SCG ecosystem with SPO

To illustrate how SPO is used in practice we present an example of an exploration session. For the purpose of this study we look at the software ecosystem that is hosted by the Software Composition Group from the University of Bern, Switzerland (hereafter referred to as SCG). Most of the projects that belong to the ecosystem are typically research projects developed by PhD and master students. Some other projects are just libraries replicated into this super-repository from elsewhere.

We start our exploration with an overview table of metrics (Fig. 1) that characterize both the structure and the evolution of all projects. Sorting the elements of the table according to the various metrics on the columns reveals the outliers and allows us to get an idea of the size of the ecosystem we are dealing with. In this case, there is a total of 210 projects. Some of these projects are discontinued and some are still active. The oldest project that is still active is MooseDevelopment. This project represents the core of the Moose analysis platform [23]; it has been worked on by more than 32 developers over 6 years. The largest project in terms of classes is Jun which has 896 classes.

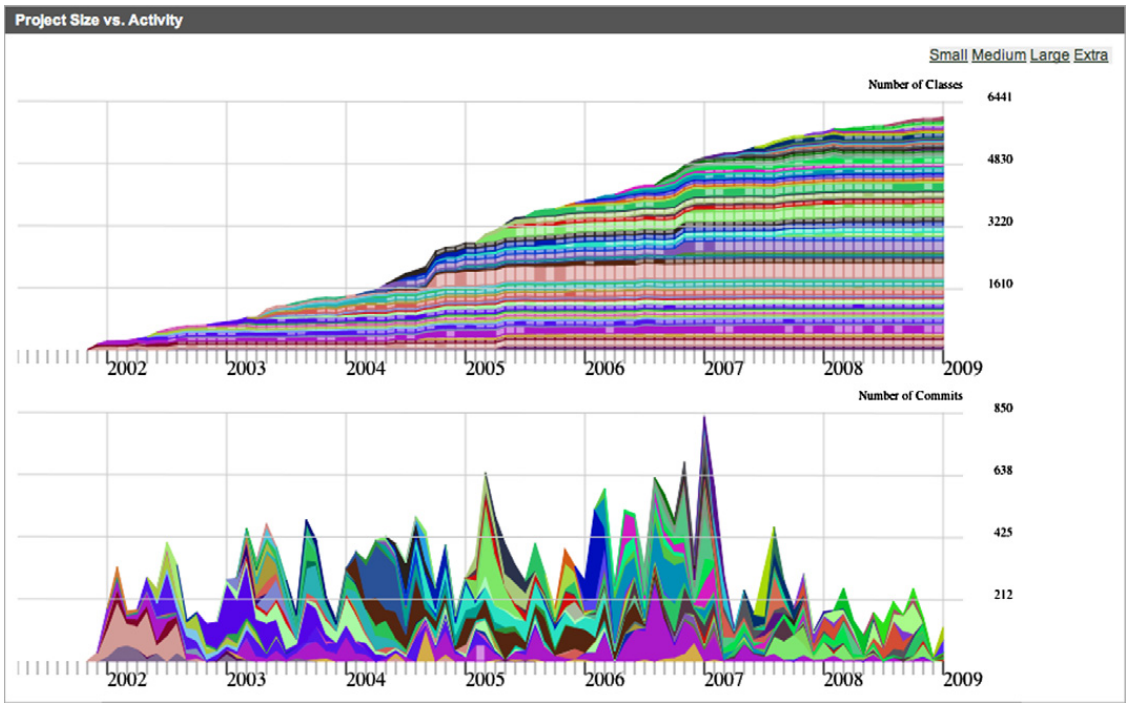

Fig. 2. The evolution of size and activity for the projects in the SCG ecosystem.

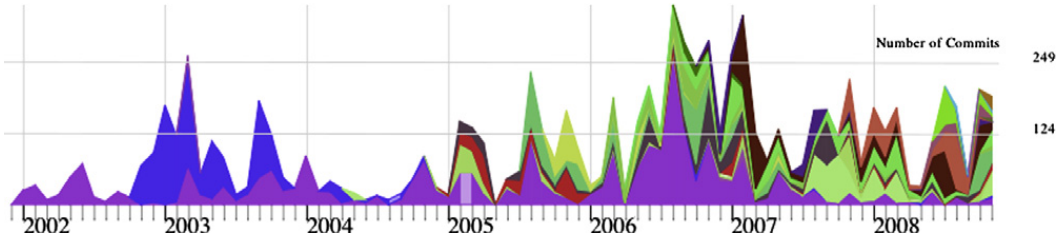

Fig. 3. Evolution of the activity for the projects which are currently active in the SCG ecosystem.

Project timelines

The table-based view is a useful tool for summarizing the status of the repository at the time of analysis but it does not provide evolutionary information about the ecosystem. To address this, SPO provides a set of timeline views that present the evolution over time of chosen metrics. Since the timelines present a high-level overview of the evolution, they are primarily useful for the project manager.

Timelines can be displayed individually or in parallel coordinates. The evolution of each metric is presented as a stacked graph where the overall shape presents the evolution of the metric at ecosystem level but the individual contributions of each project are also visible. Each project has a specific color computed as a hash function applied to the name of the project.

Fig. 2 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 monotonously 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 with 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.

The rightmost end of the activity evolution chart reveals that only a subset of the projects in the ecosystem is active. To learn about these projects we filter out those not active in the last month. The evolution of the remaining 22 projects is presented in Fig. 3. We see that two of the projects that are still active are also the oldest: MooseDevelopment and Smallwiki.

Project dependency map

The timeline views do not present the relationships between the projects in the ecosystem. To address this, SPO provides the Project Dependency Map, a view which presents static dependency relationships between the projects in the ecosystem.

Fig. 4 shows the Project Dependency Map of the recent projects in the SCG ecosystem. The intensity of the gray is proportional to the age of the project. We can see that the projects which are currently active in the repository are related. Many of them depend on the dark (old) project at the bottom. To find details about that project we select the element in

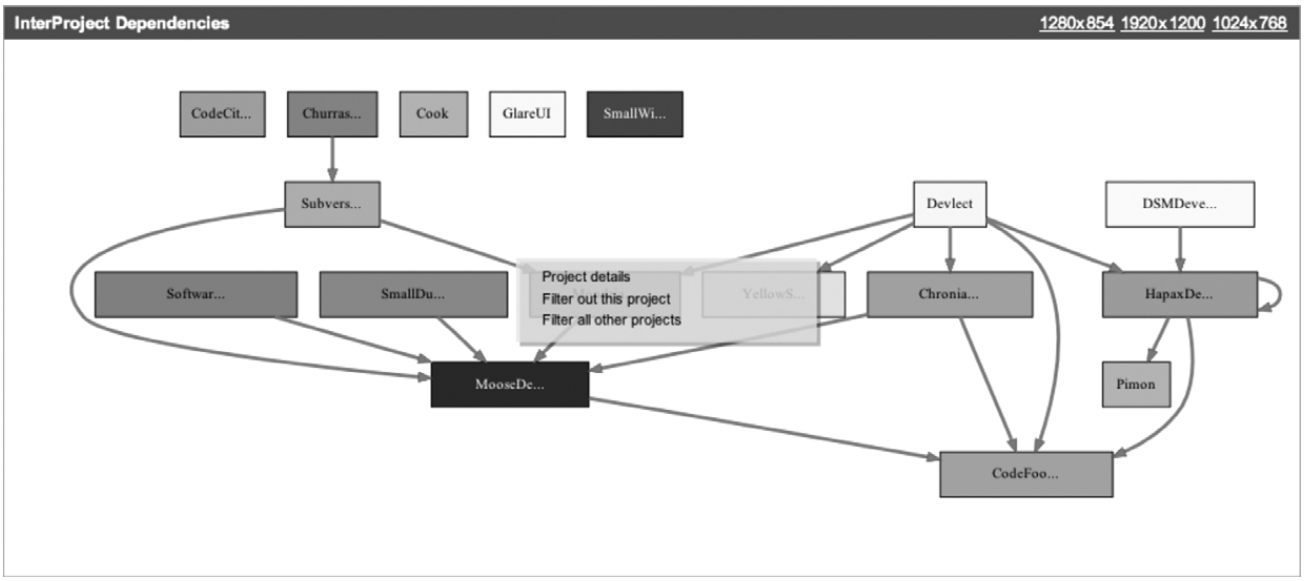

Fig. 4. The static dependencies between the projects in the SCG ecosystem (the darker a node, the older the corresponding project).

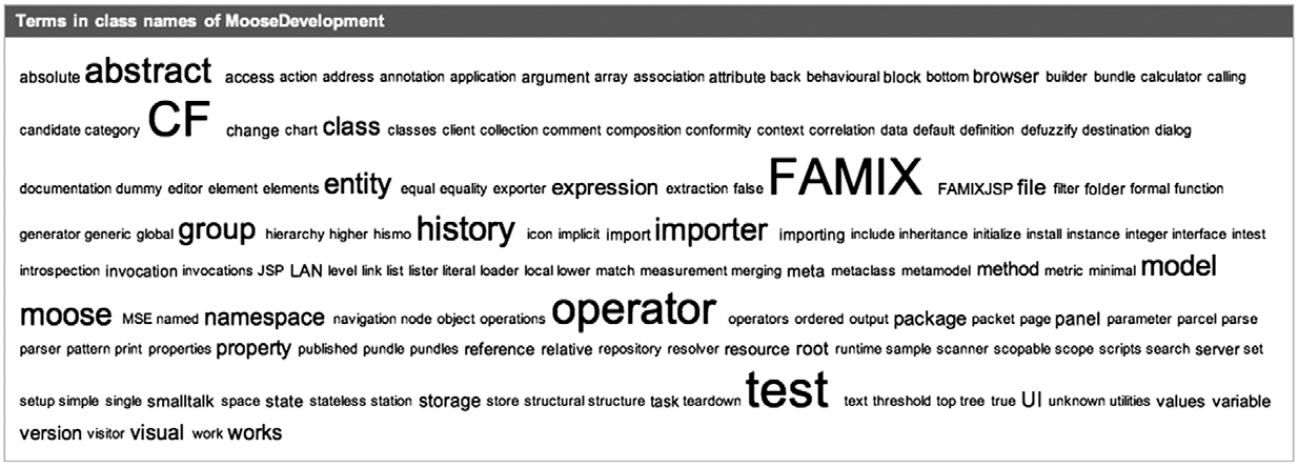

Fig. 5. The result of a lexical analysis of the vocabulary used in Moose.

the graph, open a pop-up menu and select the option Project Details. We are led to a page which presents details about the project, in this case MooseDevelopment.

Visualizing the inter-project relationships highlights the critical projects in an organization because it pinpoints the projects that all the others depend on. As a result, it is a tool for the project manager. However, visualizing the dependencies between projects is also useful for a developer who is trying to understand the way his code fits into the big picture or a developer who is new to the ecosystem.

Project vocabulary map

There are multiple views that present details for a project. We briefly present one such view, the vocabulary map. The Vocabulary Map presents the summary of the terms used in the source code of the project. The code is analyzed, the identifiers split in component words, these words are stemmed and then the final statistics on the frequency of occurrence of the words is presented as a tag cloud. The vocabulary map is a tool for the developer who wants to obtain a general overview of the domain language of a project.

Fig. 5 presents a vocabulary map of a given project. We can see that some of the most important terms used in the project are *FAMIX*, *model*, and *entity*. Indeed, MooseDevelopment contains an implementation of the FAMIX meta-model [23] which describes object-oriented software systems.

Developer collaboration map

Besides the source code of the projects, we also want to learn about the activity of developers. From the palette of views that are available for developers in SPO we choose the Collaboration Map.

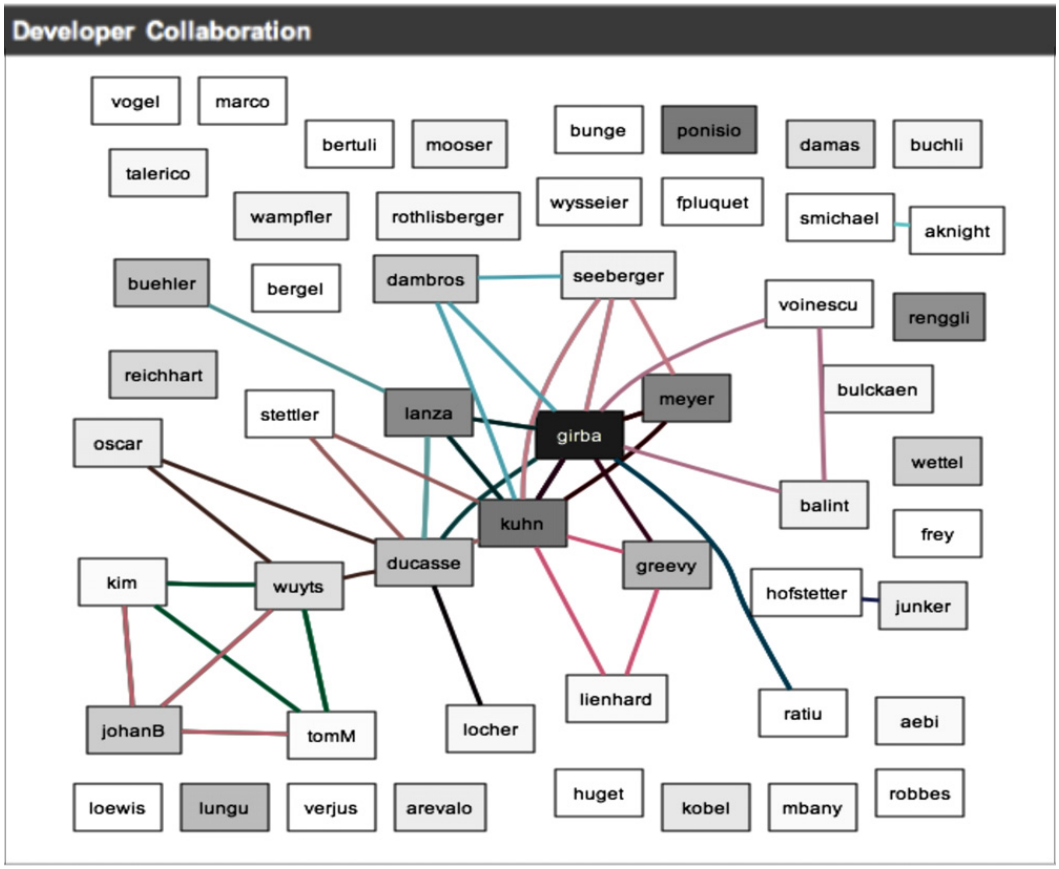

Fig. 6. The collaboration between the developers in the Bern ecosystem.

The Collaboration Map is mainly a tool for managers, and it shows how developers collaborate with each other within an ecosystem, across project boundaries. We say that two developers collaborate on a certain project if they both make modifications to the project for at least a certain number of times. Based on this information we construct a collaboration graph where the nodes are developers and the edges between them represent projects on which they collaborated.

To identify clusters of collaborating developers, we represent the collaboration graph using a force-based layout [24]. Thus, developers which collaborate will be positioned closer together. As an arc between two nodes represents the project on which the two nodes collaborate, the arc has the color of the respective project.

Fig. 6 shows that the SCG ecosystem is a moderately coupled community in which about half of the contributors collaborate on multiple projects while the rest of the contributors work individually. From this view, the exploration can continue by selecting a project edge and navigating to the details of the project, or selecting a developer node and navigating to the details of the developer.

4. The architecture of SPO

At the core of the SPO platform stands an ecosystem meta-model which is independent of the actual super-repository that stores the versioning information of the projects. Fig. 7 presents the diagram of the ecosystem meta-model used in SPO.

The ecosystem contains super-repositories which in turn contain projects. Every project has a history which is formed by an ordered list of Project Versions. Between the project versions there can be dependency relationships.

The relationships between two projects can be diverse. In some cases, a project may require that the source code of another be present at compile time. In other cases, a project may require that a given library is present in the memory at runtime. In the case of web-services, the dependencies between projects can be discovered only when run. When the only available information about the projects is their source code, the type of dependencies that can be recovered the easiest are the static dependencies between projects.

Every project version is modelled as having a developer that committed it and a set of associated changes. The changes are the lowest level of detail to which the ecosystem meta-model can go regarding the modelling of an individual version of a project, without going into detailed and expensive static analysis. A change can be represented by different types of information, depending on the capacities of the extractor and the amount of resources one is willing to use. The three possible change types are: File Change, Diff Change and AST Change. The File Changes are used for keeping track of files that are changed without going into the details of the change - such as binary files that belong to the project. The Diff Changes keep track of the text that is added or deleted from a file. AST Changes keep track of the changes that happen to the

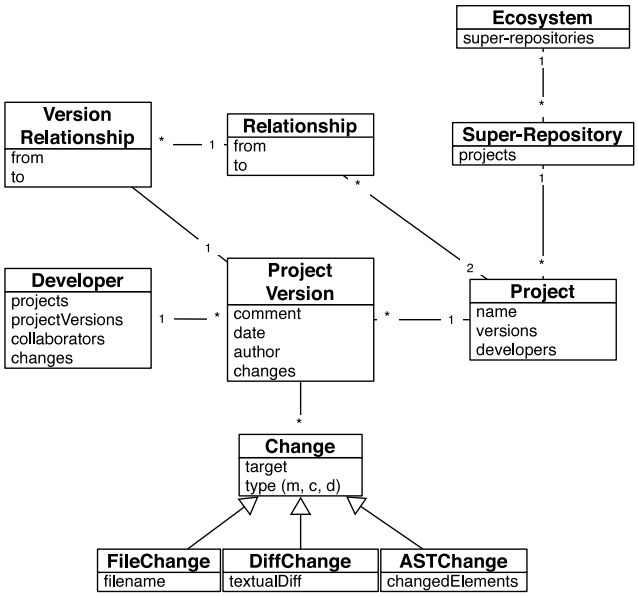

Fig. 7. The ecosystem meta-model used in SPO.

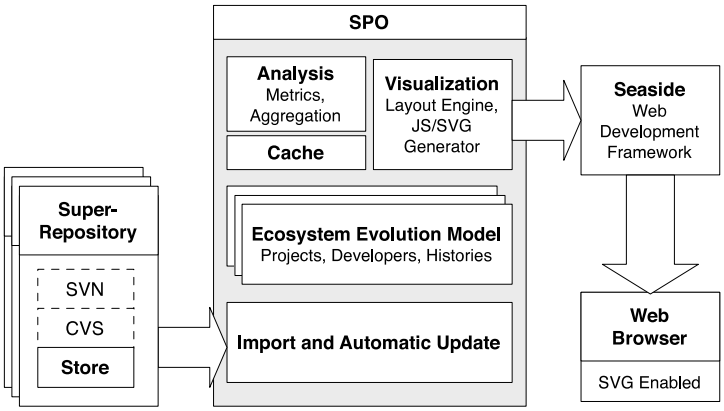

Fig. 8. The architecture of SPO.

actual programming language constructs. An ecosystem model can use any of the three types of changes, depending on the capabilities of the module that imports the data from the super-repository and builds the model.

4.1. Conceptual architecture

Fig. 8 presents a conceptual diagram of the architecture of SPO. The main components are the import module, the ecosystem models, the analysis and cache modules, and the visualization engine. We briefly present each one of the modules.

The import module is responsible for interfacing with the super-repository and pulling data from it. Currently the only type of super-repository that we support is the one for Store . We chose Store because it keeps track of a rich palette of information about the projects it versions. In the current implementation, importing the data about a few hundred projects can be done in a matter of minutes.

At a given time, SPO can contain several models of ecosystems, each of them conforming to the ecosystem meta-model that we presented at the beginning of this section.

The analysis module is computing metrics and any other information that can be derived by analyzing the information in the ecosystem model. One such type of information is the developer collaboration graph.

Due to the highly interactive and exploratory nature of the tool, SPO generates dynamically all the web pages and all the visualizations they contain. The Cache module caches all the information that is needed in order to speed-up the view generation process. Even so, the first time a certain type of analysis is invoked the user experiences a delay.

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 and the SVG generator. The JavaScript interaction code is generated dynamically for every view. Some of the graph-based views (*e.g.,* project dependency) use the hierarchical layout algorithm provided by dot [25], which arranges the nodes from top to bottom based on edges and then attempts to avoid edge crossings and reduce edge length.

Table 2
Questions testing the understanding of the SCG ecosystem.

On Developers
1. Who is more important for the ecosystem – developer X or developer Y? Why?
2. Which three developers would you nominate for distinctions for their overall activity in the ecosystem? Why?
3. How important is the contribution of Wettel and Lanza and Lungu in the SCG ecosystem?
On Projects
4. Which project is more important for the ecosystem – X or Y? Why?
5. Which is the most critical project in the ecosystem? Why?
6. If you were to decide which seven of the active projects to continue funding, which ones would you choose? Why?

Seaside is a web application framework which emphasizes a component-based approach to web application development. We use Seaside because it offers a unique way to have multiple control flows on a page, one for each component [26].

5. Tool-building experience

This section describes our approaches to validating the usefulness of SPO for understanding software ecosystems as well as some of the issues that we encountered during the development of SPO, in particular related to it being a web-based application.

5.1. Validation

We validated SPO in several ways. First we used it ourselves during the analysis of multiple open-source case studies. We listed the case studies in Section 1. We found that the tool enabled us to reason about the project structure and the developer collaboration in the studied ecosystems. Section 3 presented examples of visualizations that were generated by SPO during the aforementioned case studies.

The second way of validating the usefulness of SPO for ecosystem understanding was to use the tool in an industrial context. In our search for an industrial partner interested in analyzing its own project ecosystem we approached Soops BV, an Amsterdam-based software company specialized in Smalltalk development. Soops has experience with creating software for a wide variety of medium sized and major sized organizations such as ministry departments, financial institutions and power exchanges. We asked them if they would allow us to analyze the history of the projects in their ecosystem. They politely denied due to privacy reasons. Instead, they offered to install the tool themselves, perform analysis on their own and then report back on their experience.

The experiment with Soops was the first time we handed over one of our tools to be tested without our presence. Although we did not have control over the experiment we were satisfied to see that the developers were interested in using the tool and reporting on its usage. However, as soon as they tried to apply it, they discovered that the way they were defining their projects was different than the one recommended by Store. They were using an ingenious way of defining projects that although was using the Store facilities, was circumventing the traditional Store conventions in order to obtain increased control and customizability. In order to adapt to their peculiar approach, we had to modify our Store importer to take into account their convention. While we modified the importer, the meta-model needed no modifications. The lesson learned is that we need to be ready to adapt the tools to the peculiarities of the case studies. We presented several visual perspectives on the Soops ecosystem together with the report that we received from them elsewhere [16].

A third way of validating SPO was, given that it currently supports the analysis and visualization of Store ecosystems, to present it to the Smalltalk community. The community was enthusiastic and awarded SPO the first prize in the annual Innovation Awards competition organized at the 15th International Smalltalk Conference.[7]

As a final way of validating SPO we conducted an experiment 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. Then, we gave the students one hour of time to answer questions about an ecosystem case study as well as about the usability of SPO itself.

We asked the students to perform several tasks that were testing their understanding of the projects and developers inside an ecosystem. Table 2 presents six of the questions that we asked. All of the students answered the first five questions, however, several of them reported that they had not enough time to finish answering the last question. The questions did not have unique answers, but in most of the cases the students agreed on the solutions. In general they had to support their answers with arguments. For example, in the case of question 4 all students considered that a project on which many other projects were depending on, was more important than another project that was isolated.

After answering the questions, the students had to rate on a Likert scale [27] their own understanding of the various aspects of the ecosystem. Table 3 presents a summary of the answers.

[7] 15th International Smalltalk Conference - http://www.esug.org.

Table 3
Evaluating the understanding of the ecosystem in the case study (SA=strongly agree, A=agree, N=Neutral, D=disagree, SD=strongly disagree).

Assertion	SA (%)	A (%)	N (%)	D (%)	SD (%)
I succeeded in forming a general idea of the activity in the ecosystem	14	72	14		
I succeeded in developing an understanding of the overall relations between the developers inside the ecosystem	14	72	14		
I succeeded in forming an understanding of the relations between the projects in the ecosystem		72	28		

Table 4
Evaluating the usability of SPO (SA=strongly agree, A=agree, D=disagree, SD=strongly disagree).

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	

5.2. Usability

At the end of the experiment with the students, we asked them to fill a survey on the usability of the tool. Table 4 shows that in general the users were happy with the usability of the tool.

We also asked the students a series of open-ended questions which included what were the most and least useful features of the tool as well as the missing features. The general agreement was that the inter-project dependencies and the developer collaboration views were the most useful. The main complaint was the slowness of the tool and the lack of scalability when presenting large graphs. We used the feedback to improve the Project Observatory.

Interaction. One thing that we observed was that some of the interaction modes which to us seemed intuitive, were not such for the users. For example, when clicking on the name of a developer in any of the views, the users expected to see a new view with details about the author, but the action that was carried out was to keep the same view and add a filter to remove all the projects that did not belong to that author. Students also mentioned that the filtering capacities were very important and the current filtering offer of SPO needs to be improved.

Flexibility. Each time we presented the tool to a colleague, sooner or later, we got the question: *but could you represent that differently?* or *could you visualize that other type of information too?*. This is an instance of a more general type of problem: our users are smart. In Section 2 we present the potential users of our tool: For such users, since most of them are technically savvy, we need to provide more flexibility in view building and customization. Currently, the only way in which one can implement a new view is by writing a new view class. One possible approach is to let the views be declaratively defined, such as in our work on the Mondrian visualization engine [28].

5.3. Developing for the web

One of the reasons for implementing SPO as an online tool was to experiment with the possibilities that web applications offer when it comes to visualization and interaction. In our opinion, the benefits of availability and ease of update compensate for the limitations of doing visualization in the browser. We present here a few of the issues related to providing interactive visualization with SVG and JavaScript.

Scalable Vector Graphics (SVG) is an XML specification and file format for describing two-dimensional vector graphics, both static and animated. SVG offers anti-aliased rendering, pattern and gradient fills, sophisticated filter-effects, clipping to arbitrary paths, text and animations.

SVG is a W3C specification and most of the recent versions of the major browsers of today support it. However, not all the browsers have the same speed in rendering it and this makes the user experience hard to predict. To illustrate this, we wrote a simple JavaScript script which calculates the rendering speed of various browsers. We run the script in OS X on a PowerBook G4 running at 1.5 GHz with 1 G of RAM. The differences between the browsers are very large. For example, in a second, Opera 9.50 renders 595 polygons while Safari only renders 77 elements.

This simple benchmark shows two of the greatest limitations of SVG: the amount of visual elements that one can count on rendering is limited (at least currently) and the user experience is hard to predict as the timings will be different for different users using different system configurations. Moreover, we did encounter problems with the same pop-up menu being rendered differently in two different browsers. These limitations are probably part of reasons for which, at the time of writing this article, the trend for interactive visualization applications is biased towards using Adobe's Flash platform.

The interaction part (mouse-over effects, selection, pop-up menus) were implemented using JavaScript. In terms of expressively and flexibility, JavaScript is a very powerful programming language. However, we found that there was a dearth of good integrated development environments and debugging tools. Still, the Firebug plugin for Firefox proved very useful with debugging.

6. Conclusions and future work

In this paper we have presented the Small Project Observatory, an online visualization tool aimed at the visualization and analysis of super-repositories. We believe that super-repository visualization and analysis is a promising research direction and we plan to continue working on it. Some of the directions in which we would like to focus our efforts are improving the flexibility of the tool as well as providing finer-grained information and supporting other types of super-repositories besides Store.

Extensibility. SPO was initially developed for analyzing Store repositories. However, we realized that with several generalizations, our ecosystem meta-model can become independent of the super-repository that keeps track of the project versions. In order to examine and validate this independence we are currently working on a new importer that extracts information from ecosystems that are hosted in SVN repositories.

Integration with other tools. A platform which is complementary to SPO is the online visualization framework called Churrasco [14]. D'Ambros et al. visualize information from CVS, SVN and Bugzilla repositories. The focus of the Churrasco platform is on supporting collaboration between the users during the reverse engineering of individual systems. In the future we plan to investigate whether it is possible to link the two platforms in such a way that when a user needs to analyze bug information about an individual system he could be directed to Churrasco.

Analyzing fine-grained information. The visual perspectives that we presented show information only about the elements that are relevant at the abstraction level of an entire super-repository: projects, developers, inter-project relationships. However, there are cases in which the application should support navigating to a lower level of abstraction such as an inter-module dependency perspective inside a project or the individual method calls that are responsible for an inter-project dependency. Our current super-repository model does not support such fine-grained information. At the present time we are working towards extending it to allow the navigation down to the code level.

One final direction is to find and study other real-world ecosystems and to apply new types of analyses to the ecosystems that we have already studied.

Acknowledgements

The authors would like to thank the anonymous reviewers of the Experimental Software Toolkits journal for providing valuable feedback on the writing of this paper.

Appendix. Supplementary data

Supplementary data associated with this article can be found, in the online version, at doi:10.1016/j.scico.2009.09.004.

References

[1] S. Eick, J. Steffen Jr., Seesoft — a tool for visualizing line oriented software statistics, IEEE Transactions on Software Engineering 18 (11) (1992) 957–968.
[2] A. Marcus, L. Feng, J.I. Maletic, 3d representations for software visualization, in: SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization, ACM, New York, NY, USA, 2003, pp. 27–36.
[3] M.-A.D. Storey, H.A. Müller, Manipulating and documenting software structures using SHriMP Views, in: ICSM'95: Proceedings of the 12th International Conference on Software Maintenance, IEEE Computer Society Press, 1995, pp. 275–284.
[4] H. Muller, K. Klashinsky, Rigi: A system for programming-in-the-large, in: ICSE'88: Proceedings of the 10th International Conference on Software Engineering, 1988, pp. 80–86.
[5] M. Lungu, A. Kuhn, T. Gîrba, M. Lanza, Interactive exploration of semantic clusters, in: VISSOFT'05: Proceedings of the 3rd International Workshop on Visualizing Software for Understanding and Analysis, 2005, pp. 95–100.
[6] R. Holt, J.Y. Pak, Gase: visualizing software evolution-in-the-large, in: WCRE '96: Proceedings of the 3rd Working Conference on Reverse Engineering, WCRE '96, IEEE Computer Society, Washington, DC, USA, 1996, p. 163.
[7] M. Conklin, J. Howison, K. Crowston, Collaboration using ossmole: A repository of floss data and analyses, SIGSOFT Software Engineering Notes 30 (4) (2005) 1–5.
[8] D.A. Weiss, A large crawl and quantitative analysis of open source projects hosted on sourceforge, in: Research Report ra-001/05, Institute of Computing Science, Pozna University of Technology, Poland, 2005. At http://www.cs.put.poznan.pl/dweiss/xml/publications/index.xml, 2005.
[9] J.J. Amor, G. Robles, J.M. González-Barahona, I. Herraiz, Measuring libre software using debian 3.1 (sarge) as a case study: Preliminary results, Upgrade Magazine.
[10] G. Robles, J. Gonzalez-Barahona, Contributor turnover in libre software projects, in: IFIP International Federation for Information Processing, vol. 203, Springer, Boston, 2006, pp. 273–286.
[11] L. Lopez-Fernandez, G. Robles, J.M. Gonzalez-Barahona, I. Herraiz, Applying social network analysis to community-driven libre software projects, International Journal of Information Technology and Web Engineering 1 (3) (2006) 27–48.
[12] P.J. Finnigan, R.C. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H.A. Müller, J. Mylopoulos, S.G. Perelgut, M. Stanley, K. Wong, The software bookshelf, IBM Systems Journal 36 (4) (1997) 564–593.
[13] S. Mancoridis, T. Souder, Y.-F. Chen, E. Gansner, J. Korn, Reportal: A web-based portal site for reverse engineering, in: WCRE'01: Proceedings of the 8th Working Conference on Reverse Engineering, 2001, pp. 221–230.
[14] M. D'Ambros, M. Lanza, A flexible framework to support collaborative software evolution analysis, in: CSMR'08: Proceedings of the 12th European Conference on Software Maintenance and Reengineering, IEEE Computer Society, 2008, pp. 3–12.
[15] M. Lungu, T. Gîrba, A small observatory for super-repositories, in: IWPSE'07: Proceedings of the 9th International Workshop on the Principles of Software Evolution, 2007, pp. 106–109.

[16] M. Lungu, M. Lanza, T. Girba, R. Heeck, Reverse engineering super-repositories, in: WCRE '07: Proceedings of the 14th Working Conference on Reverse Engineering, IEEE Computer Society, Washington, DC, USA, 2007, pp. 120–129.
[17] M. Jazayeri, A. Ran, F. van der Linden, Software Architecture for Product Families: Principles and Practice, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.
[18] P. Clements, L. Northrop, Software Product Lines : Practices and Patterns, Addison-Wesley Professional, 2001.
[19] Cincom, Team Development with VisualWorks. Cincom Technical Whitepaper, 2000.
[20] M. Lehman, Programs, life cycles, and laws of software evolution, Proceedings of the IEEE 68 (9) (Sept. 1980) 1060–1076 (Special Issue on Software Engineering).
[21] M. Lehman, D. Perry, J. Ramil, W. Turski, P. Wernick, Metrics and Laws of Software Evolution — The Nineties View, in: METRICS '97: Proceedings of the 4th International Symposium on Software Metrics, IEEE Computer Society, Washington, DC, USA, 1997.
[22] D. Cubranic, G. Murphy, J. Singer, K. Booth, Hipikat: A project memory for software development, IEEE Transactions on Software Engineering 31 (6) (2005) 446–465.
[23] O. Nierstrasz, S. Ducasse, T. Gîrba, The story of moose: An agile reengineering environment, SIGSOFT Software Engineering Notes 30 (5) (2005) 1–10.
[24] T.M.J. Fruchterman, E.M. Reingold, Graph drawing by force-directed placement, Software Practice and Experience 2 (1991) 1129–1164.
[25] E.R. Gansner, S.C. North, An open graph visualization system and its applications to software engineering, Software Practice and Experience 30 (11) (2000) 1203–1233.
[26] S. Ducasse, A. Lienhard, L. Renggli, Seaside: A flexible environment for building dynamic web applications, IEEE Software 24 (5) (2007) 56–63.
[27] R. Likert, A technique for the measurement of attitudes, Archives of Psychology 22 (140) (1932) 1–55.
[28] M. Meyer, T. Gîrba, M. Lungu, Mondrian: An agile information visualization framework, in: SoftVis '06: Proceedings of the 2006 ACM Symposium on Software Visualization, ACM, New York, NY, USA, 2006, pp. 135–144.