Developer-centric Analysis of SVN Ecosystems

Master's Thesis submitted to the
Faculty of Informatics of the University of Lugano
in partial fulfillment of the requirements for the degree of
Master of Science in Informatics

presented by
Jacopo Malnati

under the supervision of
Prof. Michele Lanza and Eng. Mircea Lungu

June 2009

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Jacopo Malnati
Lugano, 5 June 2009

Abstract

A project that does not evolve is a dead product.

Software is meant to be used by people in the real world. The context in which software operates is subject to change, making its specification, design and implementation rapidly obsolete. Politic, economic or marketing interests cannot be completely (or at all) anticipated and they lead to *software aging*. *Software Evolution Analysis* is a set of models and technologies whose aim is to understand the evolution, in time, of a given project. Monitoring the evolution of software systems and analyzing their history is extremely useful in order to understand how serious a project is, what the dependencies are, who to ask for documentation or explanations, how teams work and, of course, how projects evolve.

The goal of our project was to analyze the evolution of sets of projects, characterizing a single *ecosystem*. Projects rarely evolve in isolation, but rather belong to a larger set of products that coexist and live, usually, within the same organization or group of developers. Each developer contributes to the ecosystem in his own way, according to his skills and roles. Our analysis is focused on developers, inferring the evolution of a project from the actions of its contributors. We analyze SVN repositories and extract from their databases information about each commit, inferring what each developer contributed to the system. Comparing the versions of the committed files allows us to create the vocabulary of developers, describing their *added value*, what were they working on, and for how long.

We developed a library that extracts information about developers from SVN repositories. This information is then imported into a web application that depicts, through different perspectives, the evolution of developers in terms of what they were contributing to the system. Using our tools we can inspect the vocabulary of developers, projects or even ecosystems through time and analyze the evolution of projects in terms of files, changes or commits. Refining this

huge amount of data and depicting it through *tag clouds* and *graphs* in a web application, it is possible to infer the profile of each developer, the way he is contributing to the system and which are his areas of interests and competence. We can perform searches to find specific developers responsible for particular topics or technologies, identifying who to ask for explanations or who could be assigned to fix a specific bug. Comparing the evolution of different entities allows us to spot patterns, dependencies and collaborations that would be difficult to detect otherwise.

Acknowledgements

Much of the work presented here comes from the collaboration with PhD students of the REVEAL research group of the Faculty of Informatics of the University of Lugano: Romain Robbes, Marco D'Ambros, Lile Hattori, Mircea Lungu, Richard Wettel and, of course, Michele Lanza. I am grateful to all of you guys.

In particular, many thanks go to Mircea Lungu for being my thesis advisor for both my bachelor and master thesis. Thanks for the endless hours spent coding and reasoning together.

Michele Lanza was my first contact with research, *real research*. Thanks for believing in me, for giving me countless opportunities and being so patient. Thanks, really. You introduced me to the beauty of software visualization, but first of all you were with me five years ago when I wrote my first, clumsy, line of code.

Thanks to all the people that influenced my work, without even knowing me, in particular Tudor Gîrba, Stéphane Ducasse, Radu Marinescu and Adrian Kuhn. You are doing a great job.

Nothing would have been possible were it not for my parents. Thanks for supporting me when I needed that.

Last, but not least, thanks to my wife Giulia, for being so patient and believing in me. The last five years were really hard even for you, I know. I love you. If I had the strength to keep fighting, it was for you.

Contents

Chapter 1

Introduction

Software evolution analysis is a relatively new research area, therefore its literature lacks exhaustive and complete models or data mining technologies. Different research groups developed different methodologies at different levels of abstraction and granularity. Initially, meaningful data was extracted from versioning systems databases, while lately we are witnessing the creation of tools aimed at capturing each single change. Our approach exploits databases of SVN repositories. The *pro et contra* of this technique will be described in depth in Chapter 2, along with a more detailed description of the state of the art of several techniques.

Modeling the evolution of an ecosystem through time implies handling a huge amount of raw data that must be refined and stored according to the required level of abstraction and granularity. In a detailed model like ours, where changes are first class entities, time and space efficiency must be taken into consideration. Chapter 3 describes how we extract information from SVN repositories and how other applications can use this data. Our work is not composed of a single, monolithic component, but rather is a set of libraries and tools that, together, provide the wanted functionalities to the user. The data extractor operates as a library whose task is to gather all the required information given an ecosystem of SVN repositories. The content of its model can then be inspected, exported or graphically represented by other applications. Chapter 3 will describe how to use our model within any application, while Chapter 4 will describe how we can export and then import it in the *Small Projects Observatory* [LLG07], a web application extended by us to exploit and visualize all the data collected by the data extractor. We chose tag clouds and graphs as principal means of navigation and visualization to depicts the underlying data.

Chapter 5 provides several examples and case studies in which our approach

helps to characterize single developers, projects and ecosystems. We analyze the evolution and look for patterns and clusters emerging from the visualizations. In particular, we model and analyze the activity and vocabulary of developers to describe them and their history. In order to have a broad range of subjects, we decided to model 67 projects belonging to the GNOME ecosystem and 70 projects belonging to the repository used by the REVEAL group at the Faculty of Informatics of Lugano. Our analysis covers very different kind of projects and ecosystems giving us the possibility to compare different styles and properties.

Chapter 6 summarizes the main contributions of our work as well as some weaknesses that should be taken into consideration while exploiting our model or using our application. Although our approach provides tangible evidences of its usefulness, it is far from being exhaustive and there are several issues that still need be addressed. The model extraction could be computed in parallel, instead of on a single machine and faster visualizations and more options could be added.

Chapter 2

Related work

Software is an abstract, intangible, product.

Like each human artifact, it starts with an idea and slowly takes shape. There is an important difference with respect to many other artifacts. An idea is behind the blueprint of a bridge and these blueprints communicate concepts that, eventually, will become iron, wires and cement. These blueprints are abstractions of a set of physical, concrete goods. When we reason about software, we are reasoning about abstractions describing intangible concepts, not physical artifacts. Software is about concepts and they way they interact and depend on each other.

As human beings, we are used to reason in terms of concrete elements. Think about how difficult it is to communicate or describe a feeling or an opinion. To understand abstract concepts we need a concrete description of them or a metaphor involving understandable or common situations and elements.

Using visualizations and metaphors is an efficient way to fill the gap between intangible and concrete objects.

2.1 Software Visualization

Software Visualization is a research field that exploits graphics in two and three dimensions to represent the abstract concepts encoded in software systems. Entities, relationships and their evolution is depicted to provide a concrete interpretation of their abstractions. Many stand-alone tools or plug-ins have been developed so far, exploiting different metaphors and *metrics* for the visualization of software systems (object oriented in particular). A metric is a function that maps a property of an object to a numerical value. This numerical value

will be then used to characterize visual properties in the visualizations. A typical example could be the *number of methods* metric applied to a class; the given result could then be mapped to a geometric property of a graphical object (the width of a rectangle, for example).

There are multiple dimensions along which visualizations can be classified: *in primis*, the scope. It is possible to visualize a single component, the whole system or a system of systems: an ecosystem. Then, there could be several goals: understanding a system (reverse engineering), quality assurance (hunting defects) or understanding the evolution of a software artifact. Finally, there are several aspect of a system that could be visualized: its static structure, its dynamic behavior (at run-time) or its evolution through time.

CodeCity [WL07], for example, is an integrated environment for software analysis, in which software systems are visualized as interactive, navigable 3D cities. The classes are represented as buildings in the city, while the packages are depicted as the districts in which the buildings reside. The visible properties of the city artifacts depict a set of chosen software metrics. This is a stand alone application whose aim is to provide the user the possibility to visualize single components (up to the method level) as far as entire projects, as shown in Figure 2.1. This tool could be useful both in forward and reverse engineering and it contains feature to automatically detect design flaws, according to a given set of design metrics. Systems (and their components) can be analyzed through time, comparing several releases of the depicted projects (Figure 2.2).

2.1.1 Software Evolution Visualization

Some researchers focused their attention on the visualization of evolution; in particular, visualizing entities and their changes.

CodeCity is an example of software visualization tool that copes, to some degree, with evolutionary analysis. One of its limitation is that evolution is represented as changes between releases. This is a very coarse level of granularity if the goal is to understand what is the contribution of the single developer. All the changes performed in months (potentially) are flattened and analyzed in one shot. Moreover, the context of the analysis is at the project level and it is difficult to compare the evolution of different products in the same time span.

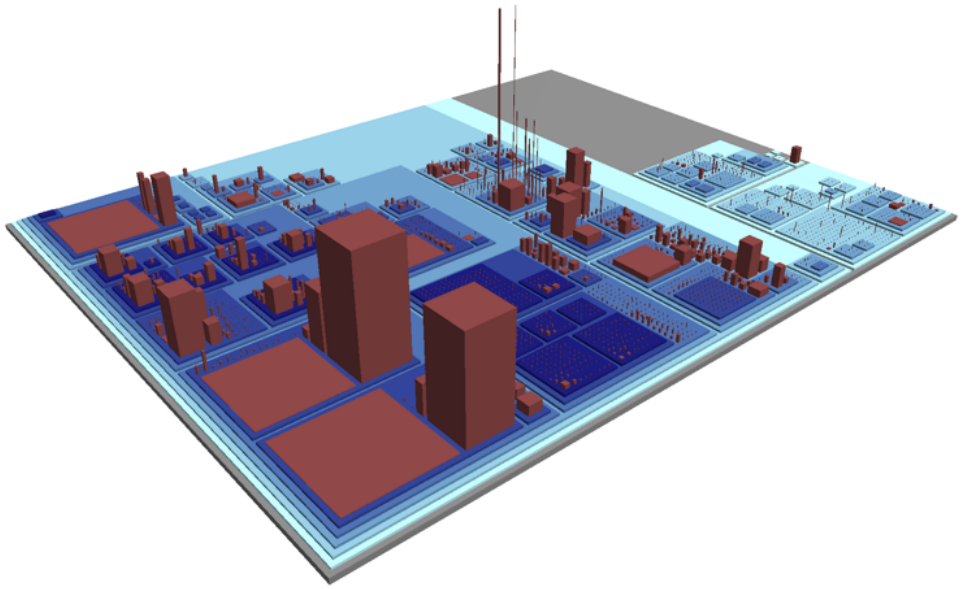

Figure 2.1: CodeCity visualizing ArgoUML, release 0.24. The buildings represent classes, the height is mapped to the number of methods and the width to the number of attributes in that class. Districts depict packages and their color is a measure of their position in the package hierarchy.

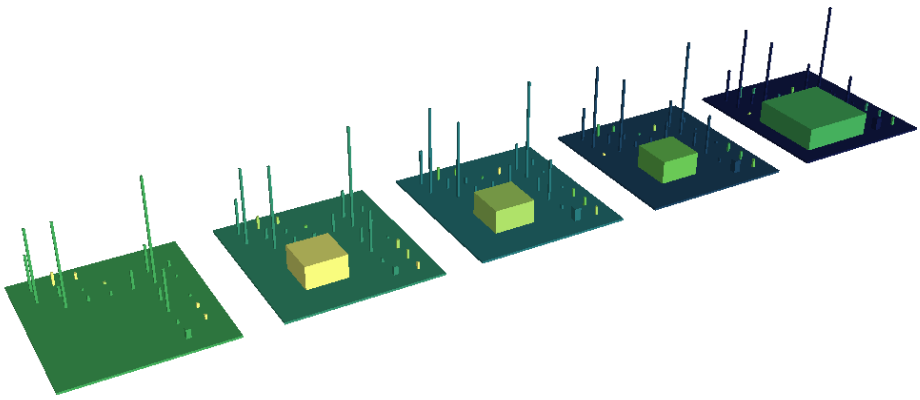

Figure 2.2: CodeCity visualizing the evolution (left to right) of the content of a package through 5 releases of a project. The size of the entities is consistent with the metrics described in Figure 2.1. An age map that associates darker colors to older entities is applied to each entity. Entities that have been modified in the current release are bright yellow.

2.1.2 Mining Software Repositories

To gather data about developer's activity, many researchers are mining information from software repositories. They rely on data provided by versioning systems, which keep track of changes by storing the text differences of a particular file. Versioning systems (CVS, SVN, STORE, etc..) are meant to be used as synchronization aid and backup for software systems. They are mainly based on text (with the notable exception of STORE), most of them do not contain any semantic information about their content. They perceive systems as a set of files and provide facilities to automatically assign versions to them. Beside versioned files, they usually store a registry containing information about each commit (the act of versioning a set of files). Commits are made by developers, have a timestamp, a comment and information about what has been changed in the system. This is exactly the data we need for our analysis. The granularity of this information is at the commit level, each time a developer decides to commit a set of files, we will be able to understand what he did with respect to the previous version. Each commit represents a project version.

Chronia [SKGD06], for example, is a tool that supports the exploration the CVS repository of a software system, featuring a visualization that shows who owned which files at which time in a system's evolution. Figure 2.3 shows the *ownership map* of a project through time. On the X-axis is time and on the Y-axis are files. The circles are commits and the colors show authors. While Chronia

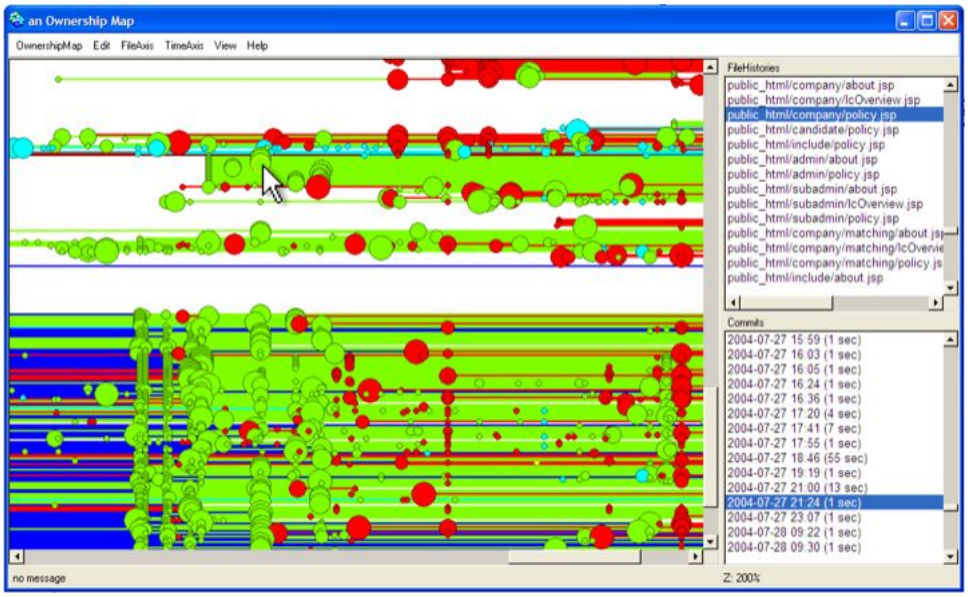

Figure 2.3: An ownership map in Chronia

is a 2D static visualization of the evolution of a project, *code_swarm* [1] exploits animations to describe the evolution of a system (Figure 2.4). Both developers and files are represented as moving elements. When a developer commits a file, it lights up and flies towards that developer. Files are colored according to their purpose, such as whether they are source code or a document. If files or developers have not been active for a while, they will fade away. A histogram at the bottom keeps a reminder of what has come before.

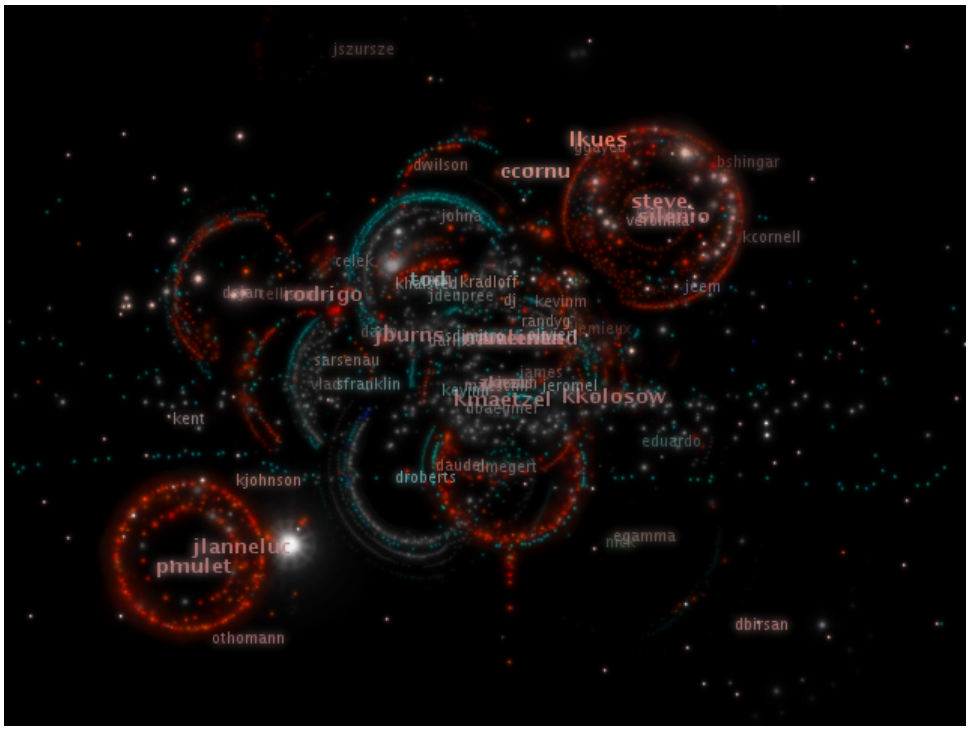

Figure 2.4: A frame of the animation produced by code_swarm while describing the evolution of Eclipse

Both Chronia and code_swarm describe the evolution of a system, according to different facets. The first provides, with a single image, a complete representation of the history of the system, with emphasis on the ownership of files. Files are the main entities and developers are used as metric to represent colors. The second one describes the evolution through an animation that must be watched from the beginning to the end to have a complete overview. Developers are the main entities and files, in time, are used to characterize their behavior.

[1] `http://vis.cs.ucdavis.edu/~ogawa/codeswarm/`

The *Evolution Radar* [DLL06] is a tool to analyze the evolution of software systems from the logical coupling perspectives. The visualization technique integrates both file-level and module-level logical coupling information. The tool facilitates an in-depth analysis of the logical couplings at all granularity levels, and leads to a precise characterization of the system modules in terms of their logical coupling dependencies. The Evolution radar, Figure 2.5, is a visualization technique which renders dependencies between groups of entities. The module in focus is visualized as a circle and placed in the center of a pie chart. All the other system modules are represented as sectors. The size of the sector is proportional to the number of files contained in the corresponding module. Within each sector files are represented as colored circles and positioned using polar coordinates where the radius is inversely proportional to the logical coupling the file has with the module in focus.

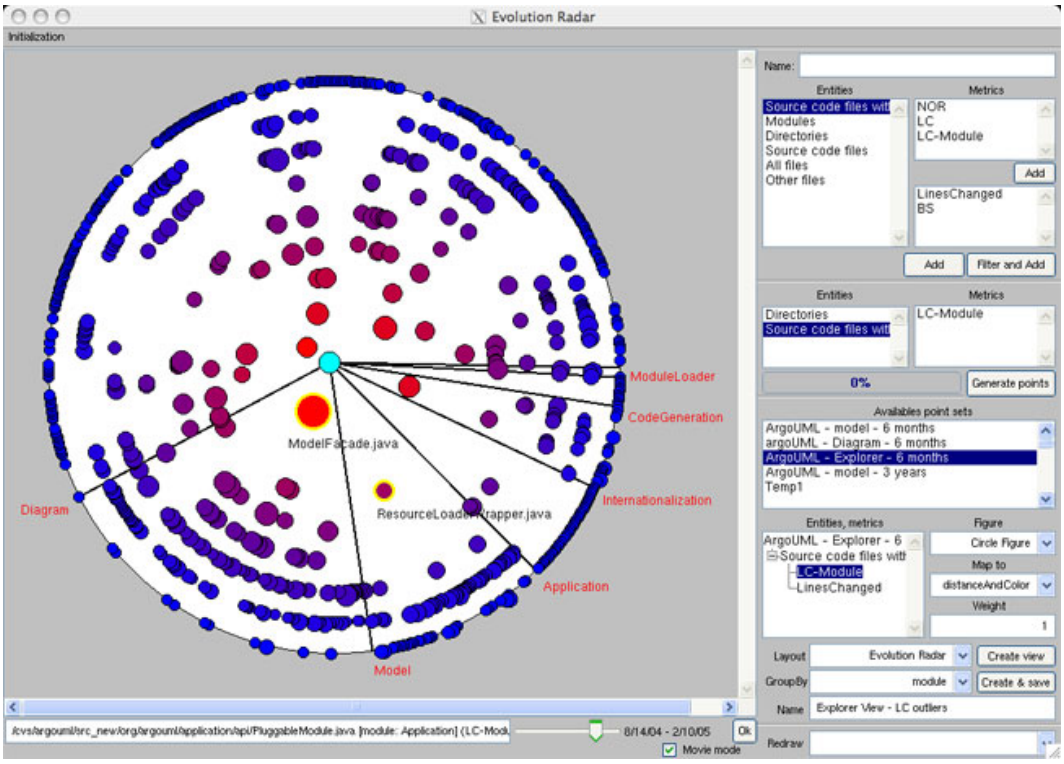

Figure 2.5: The *Evolution Radar*.

The value of the coupling between two files is equal to the number of transactions which include both files, throughout their evolution. This logical coupling is relative to a period of time, it can be computed either considering the entire history of files or with respect to a given time window. The main contribution of

this tool is to characterize the evolution of the coupling between entities.

Ohloh is a website that provides a web services suite and online community platform that aims to map the landscape of open source software development. One of the main goals of this web portal is to provide more visibility into software development. Ohloh analyzes the project source code and determines the language of each line of code, excluding comments and blanks, moreover it searches the source code for individual license declarations that can differ from the project's official license. Some of the features of this web application are aimed at visualizing metrics computed on top of projects, developers and their evolution.

The results of these analyses are statistics and data about the languages used in each given project and the evolution of their lines of code. This data is then visualized through pie-chart and graphs, as far as textual descriptions, as shown in Figure 2.6, a portion of the details of project *Firefox*.

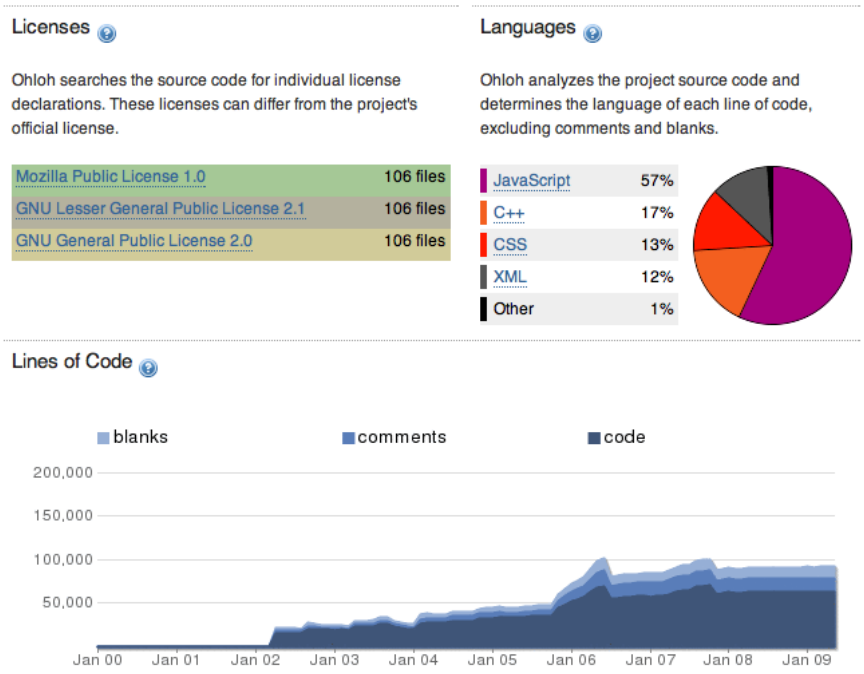

Figure 2.6: Portion of the details of project *Firefox* in Ohloh.

Similar visualizations are available for developers, depicting their commits through time and dividing their contributions according to the extensions of files. This web application extracts data out of software repositories and process

it to compute several software metrics. These metrics are then combined, refined and visualized to characterize a developer or project. Most of them, anyway, do not take into consideration time or evolution, and are flattened to provide a general overview, other than an historical description.

2.1.3 Change Based Systems

According to the type of analysis, modeling the evolution of a system through a set of versions (commits) could not be enough. Version control tools do not store all the information generated by developers and the intermediate versions of the system are flattened into a single one, the one that gets committed. Using *change operations* retrieved directly from the programming environment as the developer is creating them ([RL08][HL09]) opens new ways for both developers and researchers to explore and evolve systems. Changes are detected and handled as soon as the developer modifies the content of a file, this is the case of SpyWare and Syde. This approach enriches the information available to analysis tools, and could also increase awareness, communication and synchronization, given that changes are captured on the fly. As limitation, this technique can not be applied to projects that evolved in the past. For this kind of projects the required detail of information is not available and a more common technique, such as mining their repositories, could be applied.

2.2 Small Project Observatory

The *Small Project Observatory* (SPO) is a web application developed by Mircea Lungu. It is an easily accessible platform which supports the analysis of software ecosystems. The platform can be valuable for reverse engineering both the projects and the structure of the organization as reflected in the interactions and collaborations between developers.

We extended this web application to import and visualize data extracted through our library, as described in the next Chapter. The following description is about the tool before our intervention. A detailed description of the changes and enhancements performed to the application, will be presented in Chapter 4.

SPO is different with respect to the previous examples. Its main contribution is that it gives the possibility to analyze the evolution of a set of projects rather than a single one, featuring visualizations able to show or suggest patterns that would be extremely difficult to observe otherwise. It is possible to infer which

developers worked on which project at which time, to what extent and collaborating with whom.

This web application, before our intervention, focused on the analysis of Smalltalk repositories. SPO, at that time, was already particularly suited for *project managers*, *developers* and *researchers* for different reasons.

- Project managers could understand how teams work, how projects evolve and who has worked on a similar project already.
- Developers, in particular newcomers, can understand the dependencies of the projects they will be working on and assign skills to names, becoming aware of who they should ask if they want experts in a given field.
- Researchers can identify case studies and extract high level lessons. SPO is an easily accessible platform which helps in identifying the appropriate case studies.

Figure 2.7 represents the Small Project Observatory analyzing the Bern ecosystem, a set of Smalltalk projects developed at the University of Bern.

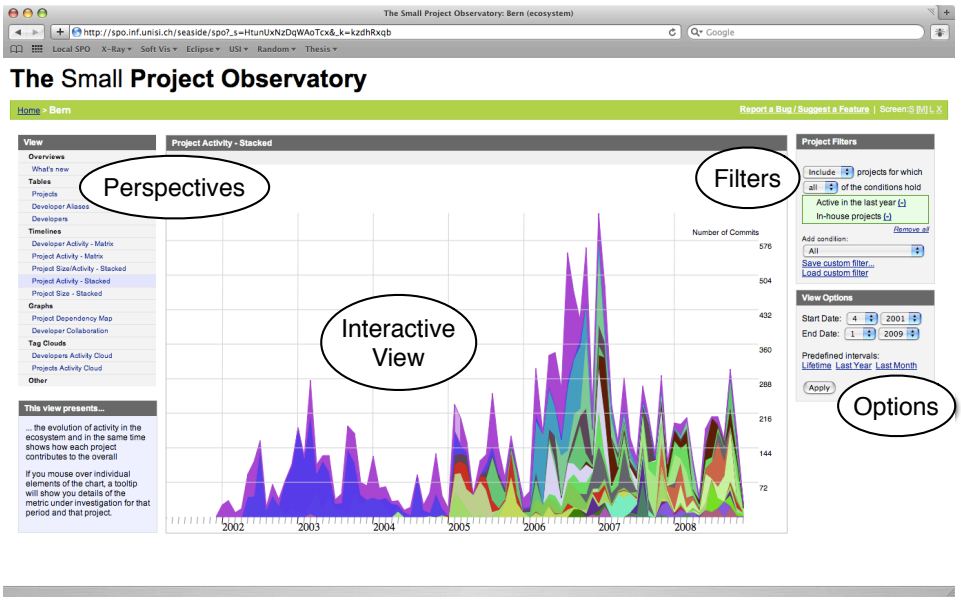

Figure 2.7: SPO depicting the *project activity* perspective of the Bern ecosystem

The central view displays a specific perspective of an ecosystem. Here we see the activity, measured in terms of commits, over a period of more than 6

years. Each colored layer in the view represents a different application. This visualization represents projects in the context of their ecosystem, according to the *number of commits* metric. Interaction is provided by *filters and options* (the panels on the right), as far as *contextual menus* on the perspective that give the possibility to further inspect entities. Views can be configured in terms of the displayed time interval. Several perspectives are available for a single project. Users can choose the ones which are appropriate through the panel on the left, containing a list of the available views.

The Small Project Observatory maintains an up to date model of an ecosystem. Based on this model, a multitude of analyses can be performed. The *size evolution* perspective illustrates the evolution of projects with respect to various metrics. Each project is assigned a specific color and is represented as a surface where the horizontal axis shows time and the height of the surface is given at every point by a certain metric computed at the respective time in the life of the project. Given that at that time SPO handled just Smalltalk, it considered the *number of classes* to be a good estimation for the evolution of the size of the projects. The upper part of Figure 2.8 illustrates the concept on the Bern ecosystem. The time interval is divided into months but can also be divided into days or weeks. The project surfaces are stacked to provide an overview of the size evolution of the ecosystem. The order in which they are stacked is chronological starting with the oldest project at the bottom. This view emphasizes the evolution of size as far as the specific time intervals when each project's size changed. The saturation of the project color is lower in the periods when the size remains constant. In the given example, the project at the top has been discontinued after an initial size increase.

The *activity evolution* perspective (at the bottom of Figure 2.8) complements the previous one. This view depicts the activity over time, rendering the efforts of developers using the *number of commits* metric. This view does not present its data in a cumulative way, given that it wants to emphasize peaks and valleys due to high or low efforts (in terms of commits) in the given time span.

The colors of projects remains constant within the different perspectives.

The two views can be combined in a parallel evolution perspective, the *project size/activity* view (the complete Figure 2.8), that provides a comparison between the two concepts. Between 2002 and the beginning of 2005 there were just 3 active projects. In January 2005 new projects were introduced and the overall size of the ecosystem slowly started to increase. The biggest project,

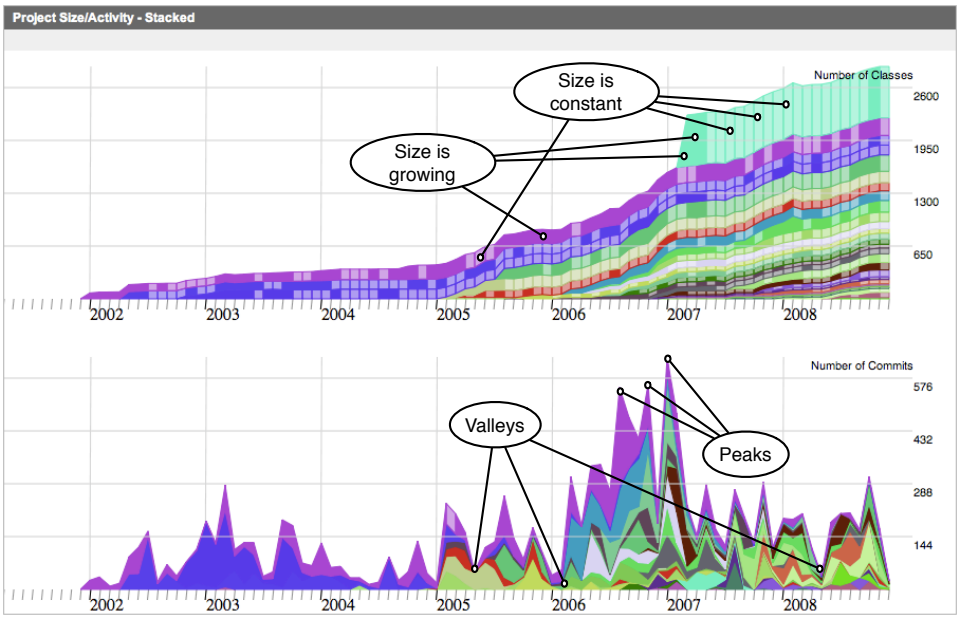

Figure 2.8: SPO depicting the *project size* (at the top) versus the *project activity* (at the bottom) of the Bern ecosystem

in terms of number of classes, started at the beginning of 2007 and is visible as light-green in the *project size* at the top of the perspective. After 3 months, anyway, its development went idle for a long time. In the *project activity* at the bottom of the perspective it is possible to notice peaks and valleys distributed within the whole ecosystem. Most of the valleys are due to summer holidays and peaks reflect the introduction of new projects (it becomes clear comparing this view with the *project size* one) or coincide with important deadlines.

The *developer activity lines* perspective presents a visual summary of the developer activity in the repository. Each contributor to the ecosystem has an associated activity line which sumarizes his activity by marking the periods in time when he was committing changes to the ecosystem. Figure 2.9 represents a portion of the *developer activity lines* of the Bern ecosystem. It is possible to quickly assign categories to contributors simply looking at their effort in time. Some of them contribute for a short period, such as the master students who work on their thesis project (*i.e.* the developer tagged as B). Other developers contribute for long period of time (the ones marked as A and C in the figure), mostly PhD students or Post-docs. Some developers contribute intermittently (C) while others contribute continuously (A and B). It is likely that the first ones are maintaining some projects while the others are actively developing them.

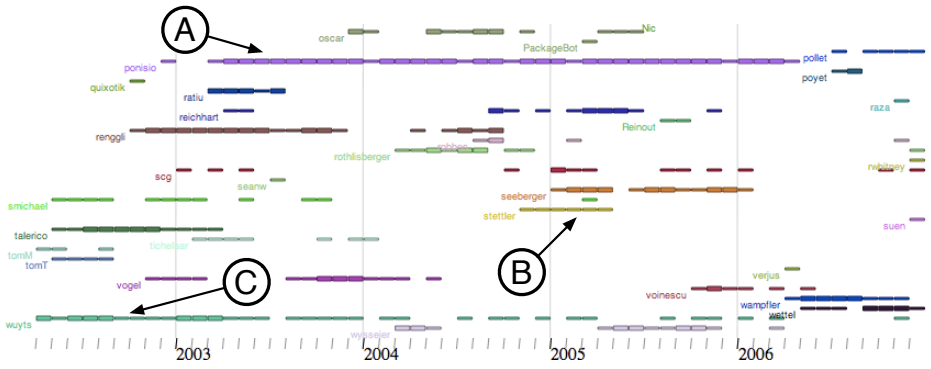

Figure 2.9: A portion of the *developer activity lines* of the Bern ecosystem

The *inter-project dependency* perspective presents the static dependencies between projects of an ecosystem. This visualization emphasizes the critical projects in a company. The projects that are mostly depended upon are at the bottom, various metrics computed for the individual projects can be mapped on the color and dimensions of the project representations.

A similar concept is applied in the *developer collaboration* view that shows how developers collaborate with each other within an ecosystem. Two developers collaborate on a certain project if they both make modification to the project for a certain number of times above a given threshold. Based on this information a *collaboration graph* is constructed, where the nodes are developers and the edges between them represent projects on which they collaborated.

To represent the collaboration graph for an ecosystem, this visualization draws the graph using a force-based layout algorithm which clusters connected nodes together and offers an aesthetically pleasing layout. Thus, developers which collaborate will be positioned closer together. The intensity of a node's color can be proportional to other metrics. Because an arc between two nodes represents the project on which the two nodes collaborate, the arc has the color of the respective project.

The Small Project Observatory addresses two issues that have been ignored by many other tools. First of all, most of them are implemented as stand-alone applications. Usually they are applied on the system that needs to be analyzed, the results retrieved and reasoned on. The majority of tools developed so far poorly address accessibility and usability. These issues are usually why most of their potential users simply gives up even before seeing the results of the pro-

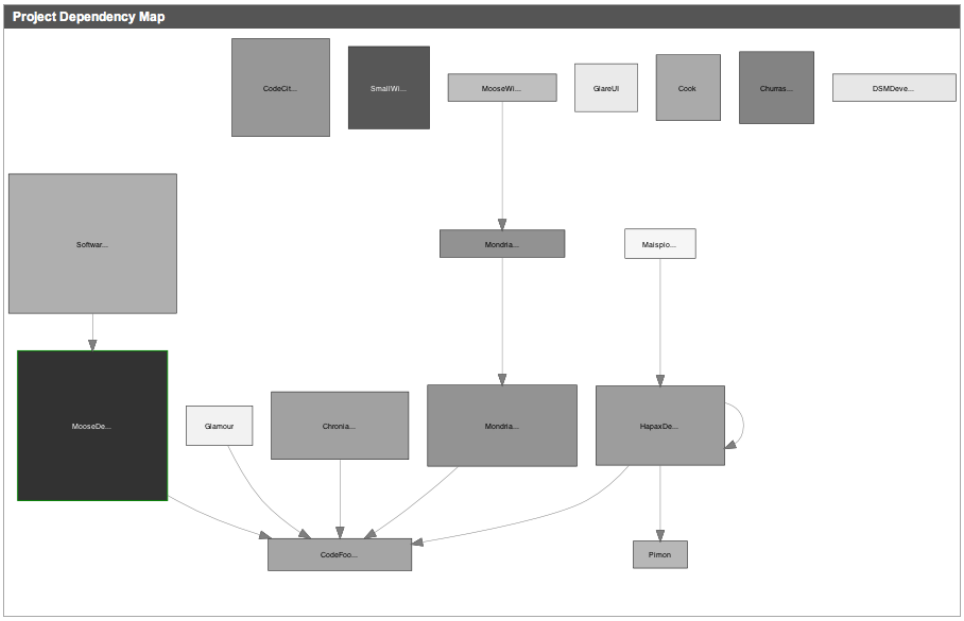

Figure 2.10: *Inter-project dependency* perspective of the projects active in the last 6 months in the Bern ecosystem

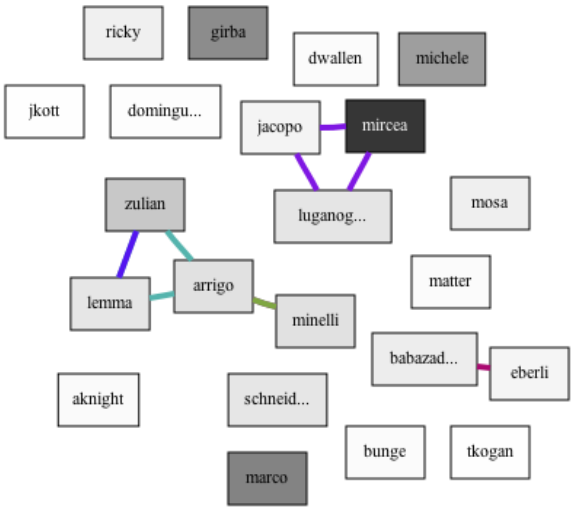

Figure 2.11: *Developer collaboration* view of a subset of the developers in the Lugano ecosystem

posed techniques. SPO is a web application that, once set up, automatically updates its content according to the last versioned data, provides simple but clear visualizations that can be highly customizable. This web application can be used by unskilled users that simply want to analyze the results, without caring about the machinery behind them. Another underestimated or totally ignored issue is that software systems are seldom developed in isolation. One of the main goals of the Small Project Observatory is to provide the analysis of projects and developers in the context of an ecosystem, a living entity that exhibits its own patterns and cross-project information.

Chapter 4 describes in depth the extensions and modifications that we performed to this web application to analyze our data and visualize even more information. Chapter 5 goes through different scenarios explaining how we used an enhanced SPO to analyze data extracted by several ecosystems.

Chapter 3

SVNMole, Modeling SVN ecosystems

In order to acquire information about the evolution of an ecosystem we analyze SVN repositories. SVN stands for *Subversion*, the version control system initiated in 2000 by CollabNet Inc. By SVN ecosystem we mean a set of SVN repositories, each of them versioning a single project. Our model extractor, *SVNMole*, acts as a library that gathers information out of these repositories and stores the refined data in the context of a single ecosystem. In the next sections we describe the structure and design of our model, how we get the needed information and how we remove noise from the data. We then go through some of the implementation decision that we took to maximize the performance of SVNMole and explain how the model could be updated, from time to time, to get the latest data out of an evolving ecosystem.

3.1 Ecosystem Metamodel

This section describes the model that we designed to represent an ecosystem. Given that one of our goals was to integrate the extracted data to the Small Project Observatory web application, the internal representation of an SVN ecosystem in SVNMole is similar to the representation of a STORE ecosystem in SPO. Figure 3.1 is a UML class diagram depicting the main entities that our library creates and handles while modeling an ecosystem. An `Ecosystem` is a set of `Projects`, it is the context in which they evolve. `Projects` are described as a set of `ProjectVersions` (commits) that contributed to the latest status of the project. Each `ProjectVersion` is performed by a `Developer`, its author, at a given date with a given comment. `ProjectVersions` contain a list of changes that have been committed, each of them modeled by a `FileChange`. A `FileChange` has a target, the path of the file involved in this change, a type (added, modified or deleted) and its precious added value: a dictionary of words

and occurrences distilling the real, actual, change performed on that file by that author. Analyzing the commit history of each `Developer`, it is possible to assign collaborators. `Ecosystems`, `Projects` and `Developers` have a given name while the other entities, although being uniquely identified by other properties, do not have a specific name.

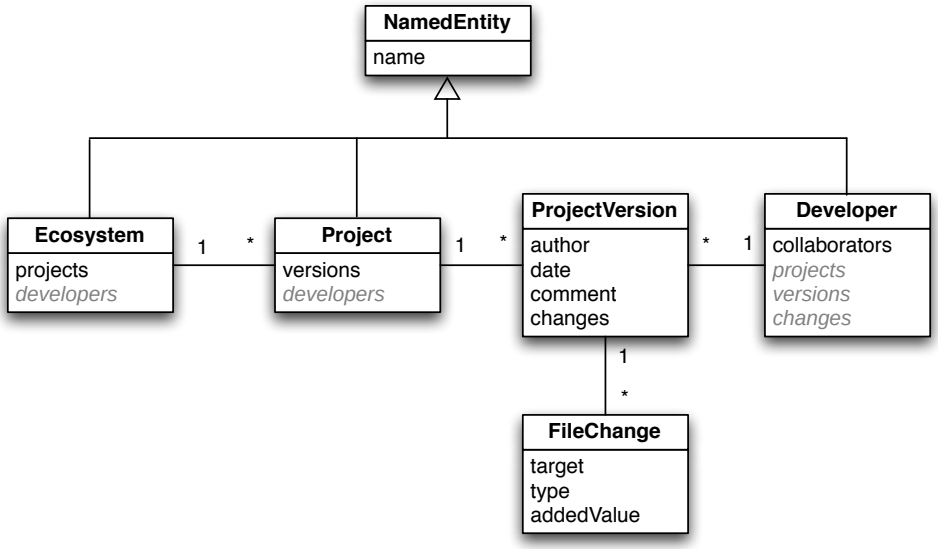

Figure 3.1: UML Class Diagram of the main entities used by SVNMole to model an SVN ecosystem. Gray attributes represent cached information.

The gray attributes in each entity represent cached information. This information could, in principle, be inferred by clients inspecting the entities of the ecosystem but that would be time consuming. Cached information is added to these entities while the model gets incrementally built or updated and it can be used for a quick lookup.

These few entities are the core of our model. Having these concepts as first class entities provides the level of abstraction required by the granularity of our analysis. Modeling each single change is useful to characterize the effort of developers and, as consequence, the evolution of projects and ecosystems.

3.2 Data Collection

SVNMole is written in Java and exploits the SVNKit[1] library developed by TMate-Software to query local or remote databases of SVN repositories. This library gives us the possibility to go through all the changes characterizing a project from its beginning to the latest, available, version. Even if the SVN protocol is text based, SVNKit provides a first level of abstraction on top of it, modeling commits as first class entities. Unfortunately, this is away from being enough to model an ecosystem, therefore we had to create our own set of entities as previously described in Section 3.1.

Going through all the commits of a set of projects, we can incrementally populate our model with projects, developers, project versions (commits) and even changes. Almost all of these entities can be directly inferred from the textual components available after inspecting the commit objects provided by SVNKit.

An important exception is a subset of data that we want to model inside each change. As previously described, a change represents a new version of a file that has been changed and committed by its author. The basic reason why versioning systems are used is that developers want to keep track of each file they change. Each version is different with respect to the previous one. Our goal is to capture the essence of this difference, the added value that there is from change to change, and assign it to the right developer. To capture this information we need to fetch from the SVN database the latest and previous version of a given textual file (we do not focus our attention on the content of binary files). Performing a textual *diff* on the two versions will give us a set of textual additions and deletions that led to the latest one. Given that one of our goals is to understand what is the vocabulary of a developer and what is he working on, we decided to use the textual addition of each file change as our source of raw data.

Let us say that a developer commits a new version of the file `Example.java` consisting in the given source code:

```
public final class Example extends JPanel {
  public static void main(final String[] args) {
    final JFrame frame = new JFrame("Example");
    final PieChartExample chart = new PieChartExample();
    frame.getContentPane().add(chart, BorderLayout.CENTER);
    frame.setSize(640, 480);
    frame.setVisible(true);
```

[1] `svnkit.com`

```
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  }
}
```

SVNmole will then fetch the previous version of the file:

```
public final class Example extends JPanel {
  public static void main(final String[] args) {
    final JFrame frame = new JFrame("Example");
    final PieChartExample chart = new PieChartExample();
  }
}
```

As soon as the two versions are available for inspection, a SVN diff is performed and the result will be something similar to the following textual report:

```
...
+  frame.getContentPane().add(chart, BorderLayout.CENTER);
+  frame.setSize(640, 480);
+  frame.setVisible(true);
+  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
...
```

A "+" sign at the beginning of a line implies that the line has been added in the latest version. This report is parsed and refined to extract the information about the actual changes that have been performed by the developer. Each line added by this file change is divided (tokenized) into its basic words and the added value of that change will be characterized by a set of words and their occurrences within the new content of the analyzed resource.
In case of a file addition, we simply consider the whole content of the file as added value.

For natural language, besides identifying each word as a sequence of characters delimited by white spaces and punctuation, nothing more is required. Contrarily, programming languages impose or suggest coding styles and standards, implying that a sequence of characters delimited by elements belonging to its syntax could not be a single word. `setDefaultCloseOperation`, for example, is a *camel case* string containing 4 words (set, default, close, operation) while `EXIT_ON_CLOSE` contains 3 words (exit, on, close). Camel case strings or words separated by "_" are correctly handled by SVNMole while extracting information from each change involving a textual file. Section 3.4 will describe how this data gets handled and stored efficiently.

Each time a new file is added to the repository, or an old one gets modified and committed, there is a change that we need to handle. Taking into consideration the content of each file that is reported by the SVN log as *added* to the repository is a wrong approach that leads to biases in the resulting data. There are two situations in which the content of a resource should not be taken into consideration. We describe them through two examples.
The following is a portion of an SVN log, describing what has been added (A) or deleted (D) from the repository:

```
revision: 5
author: jacopo
date: Wed Aug 03 21:19:55 NOVST 2009
log message: moving the model description

changed paths:
 D      /trunk/data/model.mse
 A      /trunk/versioned/model.mse
```

The log is a typical textual SVN log containing the number of the revision (5), the author, date and message of the commit. Under `changed paths` it contains the list of all the files that have been changed and committed by the developer. For each one of them our model will create a change object and store inside it, if it is the case, textual additions (they make sense just for added or modified files). If we look closer to the type of action (D and A) and the path (`/trunk/data/model.mse` and `/trunk/versioned/model.mse`) of these two `changed paths` it becomes highly probable that the developer simply *moved* the file `model.mse`. Every time our data extractor detects a *deletion-addition* of a file, it checks whether this is a simple file move without any changes or if there are textual additions to take into account. If it is the case of a simple move, nothing will be set as added value provided by that change. If it is a file move in which the file has been modified, a textual diff will be used other than retrieving the source code of the whole resource. The following algorithm is a high level description of our strategy:

```
1 getAdditionsFromAddedFile(FileA, version){
2  if(isFileMove(FileA)){
3    // if the file has been deleted and re-added
4    diff = textualDiff(FileA, version, version-1)
5  } else {
6    // it is a real addition
7    diff = surceCodeOf(FileA)
8  }
```

```
9  return diff
10 }
```

The textual diff that is performed at line 4 implies to retrieve the source code of two files with different paths, the old and the new one. In our example, SVNMole will compare the source code of `/trunk/data/model.mse` at revision 4 and the content of `/trunk/versioned/model.mse` at revision 5 (the current one). According to whether the developer modified the content of the moved file before committing it, the textual difference computed at line 4 could be empty.

Another situation in which the content of an added file should not be taken into consideration is when that file does not belong to the user, but rather is something that he uses or needs. The typical example are libraries. Some of them come with their source code attached and their license implies the possibility to use them only if their code will be distributed along with the final product. This is the case of SVNKit. If we look at the log of the SVN repository of SVNMole, we can see something like:

```
revision: 6
author: jacopo
date: Wed Aug 04 22:19:55 NOVST 2009
log message: adding SVNkit library

changed paths:
 A      /trunk/libs/svnkit/jar/a.jar
 A      /trunk/libs/svnkit/jar/b.jar
 A      /trunk/libs/svnkit/sources/a.java
 A      /trunk/libs/svnkit/sources/b.java
 A      /trunk/libs/svnkit/...
```

All of these added files are not likely to be changed in the future (being a third party library, developers will probably focus on writing code that exploits it, not on modifying it) and do not belong, strictly, to the project. Accounting their content as added value could result in a bias. This is the reason behind the choice of giving the user of SVNMole the possibility to state a set of *stop-paths* that will not be taken into consideration for specific projects. Changes involving these particular paths will be completely discarded and will not be stored in the model. Unfortunately these paths must be known *a priori* and there is not a reliable way of detecting them automatically.

3.3 Data Cleaning

Theoretically, all the textual files belonging to the analyzed ecosystem should be compared to their previous version and the resulting difference should be then assigned to the developer responsible for the modification of the given file. In a software ecosystem there is a broad variety of textual files, they range from source code (java, python, c, etc.) to documentation (txt, html, etc) and descriptors (databases schema descriptions, build files, etc.). The content of these files is code or natural language. The vocabulary used while coding is smaller than the one used while writing documentation. When we write software we reason in terms of abstractions and concepts that we tend to express in the most readable and simple way. Natural language is richer in terms and more complex in its structure. If we document something about *visualization*, for example, it's likely that we will use the following set of words: *visual, visualization, visualize, visualizing, visualized*, and so on. If we consider each of these words to be a different contribution, we will lose the fact that they all represent the same concept. In order to cope with this problem, SVNMole reduces inflected or derived words to their stem (root form) using the Porter Stemmer algorithm by Martin Porter[2]. The stem need not be identical to the morphological root of the word, it is usually sufficient that related words map to the same stem, even if this stem is not in itself a valid root.

Unfortunately, even if stemming provides the important feature of collapsing different words to the same concept, it introduces two problems. Both of them come from the fact that stemming is a lossy procedure, in which *different* words become a single one. This procedure, as previously stated, not always provides as result a valid word and this affects the understandability and readability of the visualizations built on top of stemmed data. Moreover, even if the root of all the words stemmed to the same one represents all of them, it could be the case that it introduces a bias in the understanding of the context that generated it. Take as example the stemmed word *develop*, suppose that in the whole project the only word that led to this stemmed root was *developer*. It will then be difficult to understand that one of the concepts characterizing the project is the *developer entity* and not the *develop action*.
These issues convinced us to create and maintain a special data structure whose goal is to keep track, for each stemmed root, of the original words (and their occurrences) that generated it. Let's say that the root *develop* has been encountered 100 times in the analysis of a projects and that the root comes from 90 *developer* and 10 *developing* terms. Both of them (*developer* and *developing*) are

[2] `http://tartarus.org/~martin/PorterStemmer/`

correctly assigned to the same root concerning the same concept (*develop*). SVNMole will use this stemmed concept as index for real words, but will present to its clients the word with this root that has been encountered more times. Back to the previous example, our library will state that the word *developer* (and not its root) has been encountered 100 times. This approach solves the issue about readability of incorrect terms (given that roots will no longer be shown to the user, but just be used as indexes in our algorithms) and tends to be more fair with respect to the real concepts expressed by developers.

Not all words are meaningful or describe something. If we want to know on which topics a developer was working on in a given time span, we do not want to take into consideration the syntax of programming languages or common english words. These terms are known as *stop-words* and whenever they are encountered in the analysis of a textual change, SVNMole discards them. This simple procedure removes noise in the data making the overall model more precise and smaller. Our library includes different sets of *stop-words* for the english language and common programming languages syntax. These lists can be integrated by custom ones provided by SVNMole users, simply providing instances of its `util.text.AbstractStopWordContainer` class.

Sometimes we are not interested in the content of all the files in a project, but rather we want to analyze the evolution of the concepts described only in source code, or documentation, or in a subset of them. This is why SVNMole takes as input the set of file extensions that must be inspected, parsed and tokenized. File changes regarding these extensions will contain data about the added value provided by this changes. The remaining file changes will not contain this data. To avoid listing a long and complete set of extension in the case we want to analyze the content of each file, it is possible to set as `true` the `analyzeEveryFileContent` flag of the library. It could also be the case that our analysis is strictly focused on code and should not take into consideration comment or documentation written by developers inside source code files. In this case, the filer is not at the file extension level, but rather at the file content one. Setting the `analyzeSourceComment` flag to `false`, SVNMole will discard comments and documentation for `java, c, c++` and `python` files.

3.4 Performance

SVNMole copes with enormous projects and analyzes years of changes. The procedures to gather the required information are complex and the refined data is huge. Performance optimization has been an important factor that played a cen-

tral role throughout the development of our library.

In order to minimize the latency due to I/O transfer, caching is used to keep track of the latest versions of a subset of files (the ones that are often modified). The initial releases of SVNMole were multithreaded and each thread was responsible for the analysis of a subset of the commits of a project. This was meant to minimize the latency due, in particular, to remote repository analysis. The heaviest operation in terms of memory consumption and CPU cycles is the *diff* between versions. This, depending on the parameters of the analysis and the kind of project, is performed often. Having a multithreaded library on top of CPUs with multiple core does not improve its performances but rather makes them worse. The context switch between threads implies to move back and forth from and to main memory the big data structures needed by the *diff* algorithms. The overhead was so much that we decided to change the architecture of our library to single threaded. Another issue that we had to take into account was the huge amount of strings and numbers inside our model.

Each change has, potentially, added value described in the form of words (`java.lang.String`) and their occurrences (`java.lang.Long`). Applying to these objects the *Flyweight* design pattern[3] we minimized their memory footprint.

Our architecture exploits the *publish/subscribe* message paradigm to asynchronously communicate to clients of our library that a project or an ecosystem has been completely modeled. This is particularly handy for exporters or any application that could start doing something as soon as single projects have been modeled. An implication is that as the client is done with a single project, it can ask SVNMole to remove it from its data structures, reducing its memory footprint.

3.5 Model update

The goal of SVNMole is to analyze the evolution of projects, therefore it was mandatory to provide a way to update the model with respect to the latest changes. The analysis of years of development is a process that takes several hours, according to the project and options, but updating the model implies to analyze just the new commits since the previous update. Whenever a client re-

[3] `http://en.wikipedia.org/wiki/Flyweight_pattern`

quests an analysis, SVNMole will check whether some of the projects that must be analyzed are already in its model; if they are, they are updated to the latest version, otherwise their entire history is analyzed. This is a convenient way to provide an update mechanism that can be used at regular intervals by external applications using our library.

Chapter 4

Extending the Small Project Observatory

The Small Project Observatory, as previously described, was aimed at the analysis of Smalltalk repositories. Part of its structure was bound to object-oriented concepts and to the data that could be extracted from STORE repositories (specific for the Smalltalk language).

After out intervention, this web application is now able to cope with language-agnostic repositories (the versioned data could be of any type) and provides a new set of visualizations. The next sections will explain how we import data from SVN repositories into SPO and how we visualize it with new and enhanced views.

4.1 Importing Models of Ecosystems

SPO is composed of three main subsystems - the smalltalk importer, the in-memory representation and the visualization engine. All three are written in Smalltalk and they run inside of a Smalltalk VM. SVNMole on the other hand is written in Java and takes care of the creation of models of language-agnostic ecosystems. There is no easy way for the Small Project Observatory to directly use the in-memory model created by our library. The following subsections will describe the process of exporting a model from SVNMole and importing it in SPO.

4.1.1 MSE Exporter

We developed the *MSE Exporter* as a client of SVNMole. The MSE Exporter, given the textual description of an ecosystem, uses our library to model it. During the creation of the model, the exporter will by notified of progresses and will incrementally export projects and developers to the *MSE* format[1]. MSE is a file format to store *FM3*[2] compliant metamodels and models. FM3 or Fame meta-metamodel is a model to describe metamodels. The MSE format allows to specify models for import and export with Fame, a polyglot framework for metamodeling at runtime. The following is a portion of the FM3 metamodel description that we use to export and import the model of the ecosystem created by SVNMole. The example describes the *Project* entity that contains three properties: *name, versions* and *developers*, where the last two attributes are lists. The whole FM3 metamodel description is compliant with respect to the description of the model of the ecosystems depicted in Figure 3.1.

```
(FM3.Class (id: 2)
  (name ''Project'')
    (attributes
      (FM3.Property
        (name ''name'')
        (type (ref: String)))
      (FM3.Property
        (name ''versions'')
        (multivalued true)
        (type (ref: 4)))
      (FM3.Property
        (name ''developers'')
        (multivalued true)
        (type (ref: 3)))
      )
    )
```

SVNMole computes the model for the given ecosystem and stores it inside its own data structures. The MSE Exporter client collects these data structures and exports their content according to the FM3 description of the metamodel. The final result is an MSE file that encodes the whole ecosystem and that can be imported in SPO, exploiting the functionalities provided by FAME. Figure 4.1 depicts a simplified UML sequence diagram showing the interactions between

[1] http://scg.unibe.ch/wiki/projects/fame/mse
[2] http://scg.unibe.ch/wiki/projects/fame/fm3

the main actors of this phase.

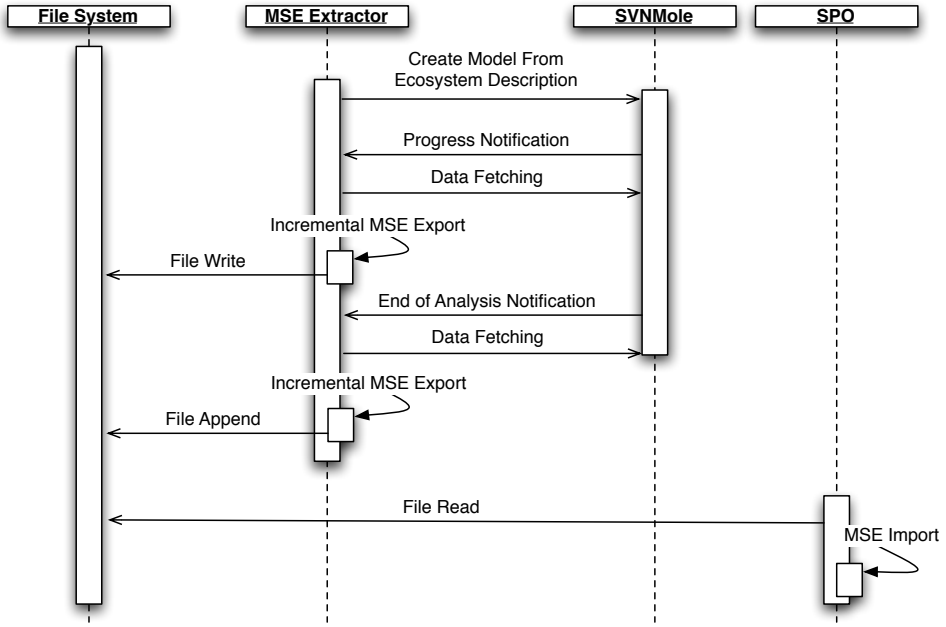

Figure 4.1: UML sequence diagram of the interaction between the main actors involved in the creation, export and import of the model of an ecosystem.

4.1.2 Importing FAMIX Models in SPO

Inside the Small Project Observatory, the concepts of ecosystems, projects and developers were already present, as `Pz.Ecosystem`, `Pz.Project` and `Pz.Developer`. The evolution of the system was described by packages and their versions (modeled as first-class entities as `Pz.PackageHistory`). This was enough and made sense to model the evolution of an object-oriented ecosystem. After our intervention, new entities have been added to SPO to give the possibility to model the history of any content, not just object-oriented systems. We extended the already existing ecosystem, project and developer entities and created classes modeling the content and behavior of project versions (commits) and file changes (textual changes involving a given file). The extensions and new classes can be found in the `ECO-Model` package of the `ProjectZoo` bundle in the Lugano STORE repository.

The content of the `ECO-Model` package reflects the metamodel that gets exported in the MSE format by the MSE Exporter client. These Smalltalk classes

have been annotated in such a way that , exploiting the functionalities of FAME, it is possible to automatically create instances of them filled by data written in an MSE file in compliance with the previously described FM3 description.

The following is an example of annotation in the `Pz.SPOEcosystem` class, the entity we created as extension of the previously existing developer concept.

```
1 developers
2   <MSEProperty: #developers type: #SPODeveloper> <multivalued>
3   ^ developers
```

The previous method is an accessor (`developers`, line 1) that returns the list of developers in the given ecosystem. The `developers` attribute (line 3) of the `Pz.SPOEcosystem` class is populated with data taken from the MSE description according to the annotation at line 2. The annotation refers to concepts described in the FM3 metamodel description file.

Executing the following line in a Smalltalk workspace, the Small Project Observatory will load the content of the given MSE file and instantiate classes according to that. At the end of this procedure, SPO will contain a new ecosystem ready to be analyzed and visualized.

```
1 newlyImported := Pz.SPOEcosystem importFrom: 'gnomeEcosystem.mse'.
2 (Ecosystem caches at: 'Gnome' put: newlyImported)
```

Line 1 assigns the refined content described by the file `gnomeEcosystem.mse` to a `Pz.SPOEcosystem` object that, at line 2, gets added to the list of available ecosystems.

4.2 Visualization Primitives in SPO

In order to analyze the new types of data imported in SPO from SVNMole we needed new viualization mechanisms. To communicate and analyze this new set of information we created new perspectives and view components. Almost all the visualizations that have been added are built on top of two visualization primitives: tag-clouds and stacked-graphs. The next subsections will give an overview of them, while the next Chapter provides examples of their use.

4.2.1 Tag-Clouds

A tag-cloud is a visual depiction of words. The visualized entity is text and the metrics applied to these words are mapped on their color, size and position. A

tag-cloud is particularly useful when we need to visualize text and we want to emphasize a portion of it, according to a given set of rules. In our case, tag-clouds have been originally introduced to visually represent the vocabulary of developers, projects and ecosystems.

The following (Figure 4.2) is an example of tag-cloud in SPO depicting the vocabulary of the developer *jacopo*. The entire vocabulary of this developer has been modeled on top of the added value that he contributed to the system. Each time this developer modified and committed a file, the words constituting the added value of this change were added to its vocabulary.

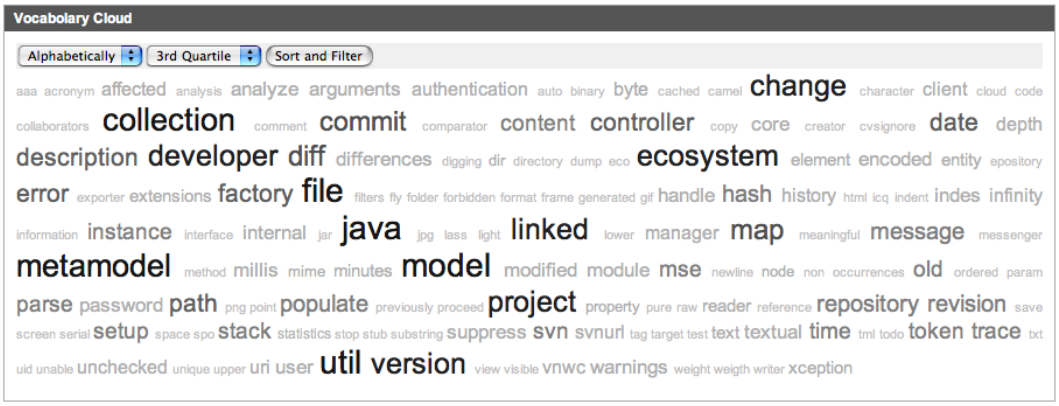

Figure 4.2: The Vocabulary Tag-Cloud of the developer *jacopo*. Just the last quartile of the tags with respect to their occurrences is shown. The order is alphabetic.

The tag-cloud is then created according to the vocabulary of the developer, where each tag represents a word in the developer's vocabulary. The size and color of the tag is proportional to the occurrences of the given word in the vocabulary of the developer. When an author is working on a specific concept, he tends to repeat important keywords that are, much likely, representative of that concept. These repetitions are captured by several added values and, eventually, depicted as bigger tags with respect to those modeling less frequent terms.

The order of the tags in Figure 4.2 is alphabetically, while the same data can be ordered according to the occurrences of its tags (consequently, according to the size of its words) as shown in Figure 4.3. This approach, even if it is less pleasant to the eye, provides a sort of histogram giving an immediate understanding of the most important terms. This way of ordering the tags, moreover,

provides a more compact visualization in terms of space.

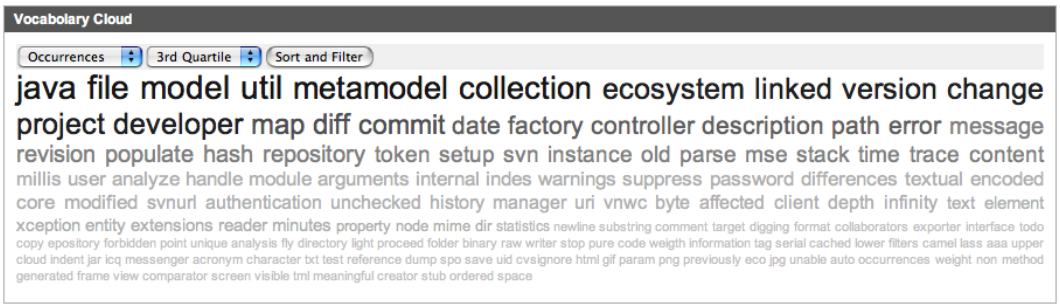

Figure 4.3: The same data of Figure 4.2, but sorted according to the occurrences of the tags in the vocabulary of the developer *jacopo*.

The tag-cloud created by SPO are represented using inline HTML elements. To each tag is associated an hyperlink that points to a specific page containing properties or other kind of information regarding that word. These views will be described in subsection 4.4. The goal of our tag-clouds is twofold: on one hand it is a simple and immediate means of visualization, on the other one is a useful means of navigation.

Moving the mouse over a tag, a small tooltip will appear showing to the user the numerical value of the occurrences of that word in the vocabulary of the developer. This number is used as metric for the size of the tag according to a logarithmic scaling. Given the occurrences of a word, the size of its tag is computed as follows:

```
weight = (log(occurrences)-log(median)) / (log(max)-log(median))
size = minTextSize+round((maxTextSize-minTextSize) * weight)
```

Where `minTextSize` and `maxTextSize` are, respectively, the minimum and maximum size of the text in the tag-cloud. `median` and `max` are computed over the occurrences of the whole set of words in the vocabulary. The logarithmic scale helps preventing the big outliers from shadowing the other terms.

A quick look at the tag-clouds in Figure 4.2 or 4.3 is enough to understand that the concepts exploited by *jacopo* are related to models and metamodels, ecosystems, projects, developers, versions, file and changes. This clearly and quickly describes the added value provided by this user while developing SVN-Mole.

At the top of each tag-cloud, two drop-down menus provide the user the possibility to sort the tags and show all of them, from the second quartile to the last, only the last quartile or the top 100.

Besides being useful to depict vocabularies, we successfully used tag-clouds to quickly communicate to the user information about developer and projects activity, as far as the activity divided by file extensions. Figure 4.4 is an example of that. From this cloud it is immediate that *jacopo* contributed, at the time of that analysis, mainly Java code and Latex (`tex`) documentation.

Figure 4.4: File Extensions Tag-Cloud for all the projects developed by *jacopo*.

An extensive analysis of data on top of tag-clouds will be presented in the next Chapter.

One limitation of tag-clouds is that their results flatten the history that led to that situation, without revealing temporal information about its tags. Even if we apply few evolutionary metrics to the property of tags, the final outcome will not be able to visualize historical patterns or behaviors. If a tag is big, is that so because that word has been important in the past, right now, or always? If we take into consideration this information as metric for tags dimensions, color or position, how can we depict a situation in which the importance of a word was alternating (a so called *pulsar*)? The next subsection describes how we coped with this issue.

4.2.2 Stacked-Graphs

Stacked-graphs, as previously described, were already present in the Small Project Observatory. They provide the notion of time on the horizontal axis and map on the vertical one a specific metric of a given entity at the given time interval. Stacking upon each-other data from all the depicted entities provides an overview of the evolution of each single entity and, at the same time, of the whole context. This graphs are an excellent way of visualizing in a single, static, visualization the evolution of a set of entities. Given that a tag-cloud is not enough to represent even this information, whenever the user feels the necessity

to inspect the evolution of an entity, he can one of the several view components that we created that exploit this kind of graphs. We created them for the evolution of projects in the system, file extensions and vocabularies. These components will be described in the subsection 4.4. To make them more useful, besides using them for different kind of data and evolutions, we added a short legend of the top 15 visualized entities to avoid the user to move the mouse over each stacked entity to see its tooltip (Figure 4.5).

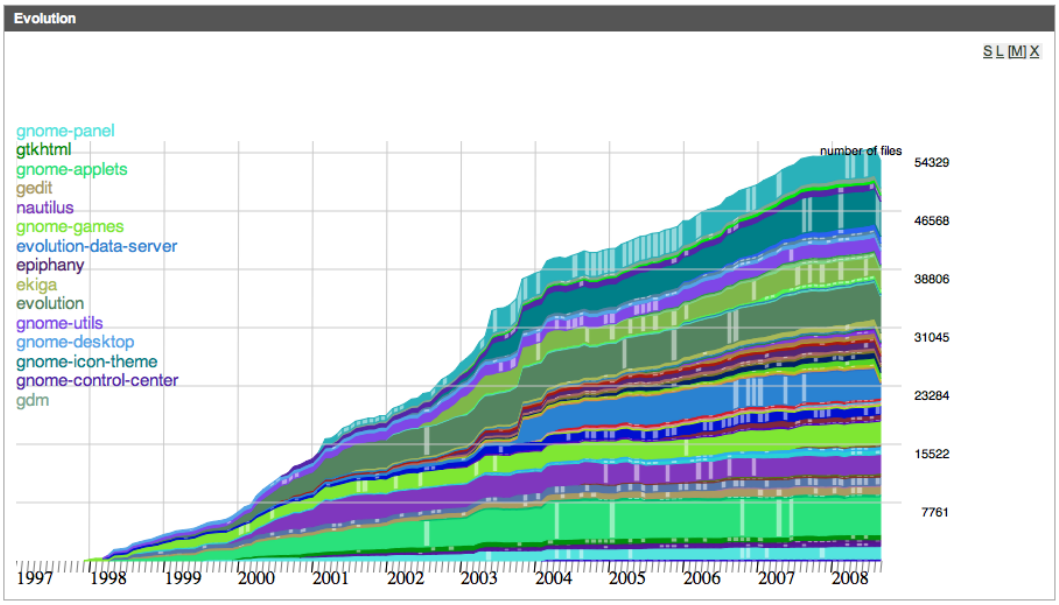

Figure 4.5: A stacked-graph visualizing the growth in terms of number of files for all the projects in the GNOME ecosystem. On the left, a legend with the top 15 projects with respect to the total number of files contributed to the ecosystem.

Right clicking on the visually represented evolution of an entity a context menu will appear and the user could further inspect it.

SPO is now using two kinds of stacked-graphs. Cumulative and non cumulative. Cumulative stacked graphs, like the one in Figure 4.5, accumulates data from the previous time span or at least takes them into consideration before computing the one for the next time span. This approach makes sense when we are handling entities that, potentially, are conserved in time, such as files in a project (if we have 10 files at the end of January and they do not get deleted, we will still have them in February). Other situations, such as activity measured in commits count or the importance of terms measured in their occurrences, are

better depicted through a non cumulative graph, in order to emphasize peaks and valleys in their evolution.

4.3 Additions to the Small Project Observatory

To increase the usefulness of SPO and the amount of information that could be retrieved from an ecosystem, we created new functionalities and visualizations. Their goals and contents are described in the next subsections.

4.3.1 Graphical Visualizations

On top of the visualization primitives previously described, new graphical visualizations have been implemented. The next subsections describe them all.

Scatterplot

A 2D scatterplot uses cartesian coordinates to display values for two variables for a set of data. This data is depicted as a set of points or little squares, each having the value of one variable determining the position on the horizontal axis and the other variable determining the position on the vertical one.

We added a scatterplot perspective to SPO to visualize, for each developer, the amount of contributions to source code versus the contributions to internationalization files. This perspective has been created specifically to visualize this information for the GNOME ecosystem (its analysis will be presented in the next Chapter) but it can be easily modified and applied to any ecosystem for any metric. The goal of our analysis was to understand how the developer's effort was distributed between code and internationalization. The profile of translators is different than the one of coders; this visualization gives us a quick overview of the distribution of efforts and makes it easy to spot outliers.

Figure 4.6 is an example of *Developer Scatterplot* for the GNOME ecosystem. Each dot represents a developer, the horizontal axis represents the amount of contributions in terms of changes to source code (`c, cpp, h, java, python` files) while the vertical axis represents the controbutions involving internationalization files (`po` files).

Outliers and big contributors have their name made explicit next to the dot representing them. Right clicking on each developer it is possible to further inspect its profile or add an alias to him. How, and why, is described in the next

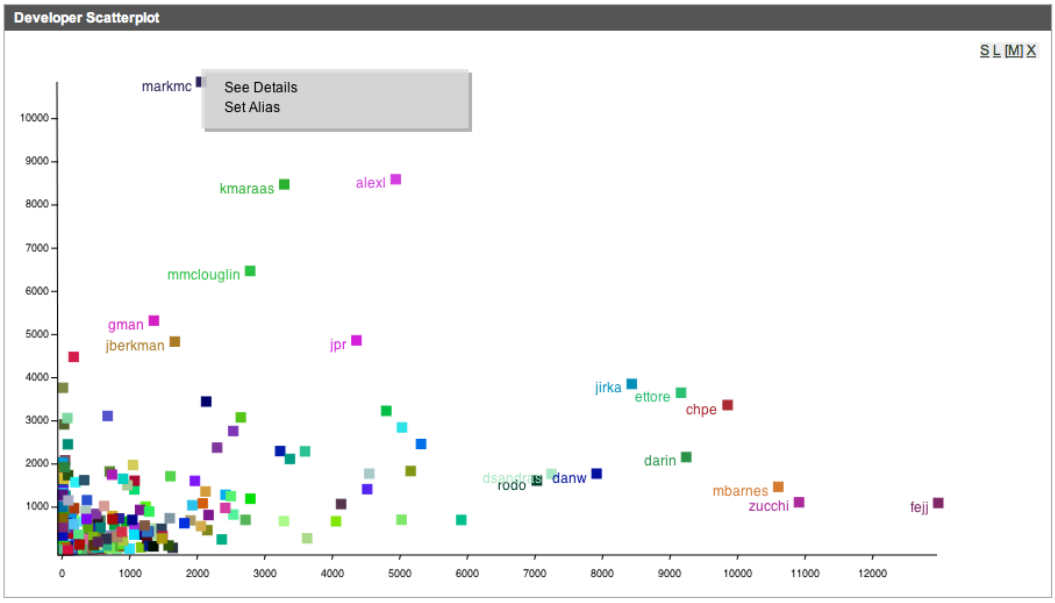

Figure 4.6: Scatterplot in SPO visualizing the amount of contributions for each developer in terms of changes to source code versus internationalization files. Outliers are tagged with their names.

sections.

Figure 4.6 shows that the majority of contributors contributes no more than 3000 file changes, be it source code or internationalization. There are anyway few exceptions that will be described in Chapter 5

Concept Activity Stacked-Graph

The *Concept Activity* view is a developer centric visualization that visualizes, over time, the evolution of the vocabulary of a given developer. The vocabulary of a developer is composed of the terms that he contributed to all of his projects. These terms are extracted from the added value of each file change that is committed to an SVN repository. Monitoring the occurrences of these terms in time, it is possible to depict their evolution. An example of terms evolution is represented in Figure 4.7, the *Concept Activity* of developer *lile* in the Lugano SVN ecosystem.

In this kind of visualization the top 30 terms of the user-selected developer are visualized as *time series*, each with a different color, stacked on each other in a non cumulative stacked-graph. The top 30 terms are not computed flattening

the whole history of a developer and selecting the 30 terms with the most occurrences, because this approach would hide, potentially, terms that have been really important, but just for limited amount of time. Our approach takes into consideration the importance of terms in each component of the *time series*, giving the possibility to visualize also terms that have been highly used for short periods (the so called *meteors*). The stacked-graph is non cumulative to emphasize valleys and peaks in the importance of terms over time. This approach provides a quicker understanding of the set of terms at the base of the development of each period of time. Right clicking on the evolution of a concept, the user can automatically perform a vocabulary search on the whole ecosystem that will bring him to the *Search* view showing a tag-cloud of other developers that have used that concept in their development.

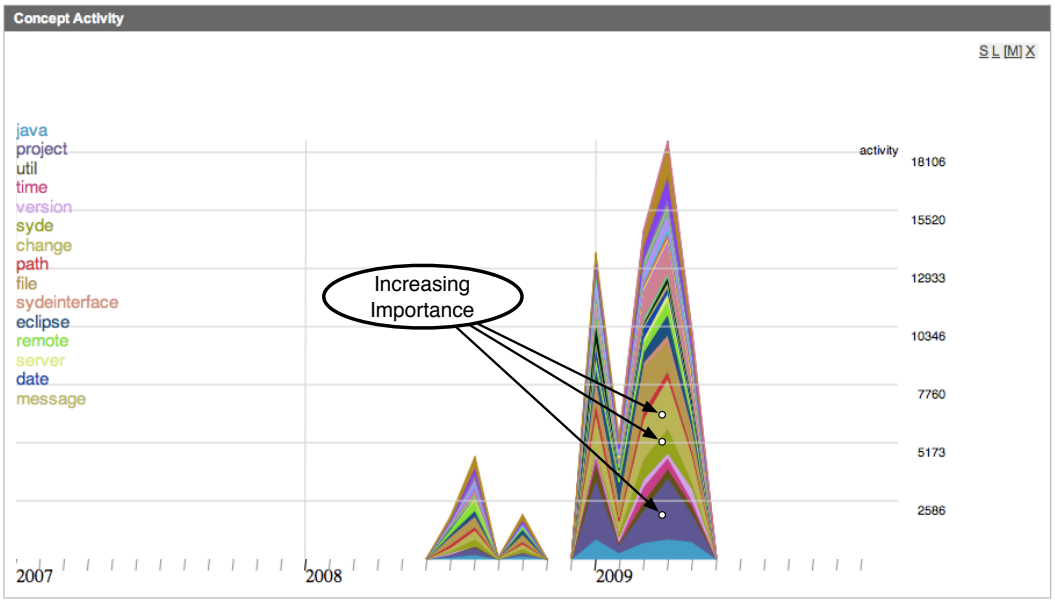

Figure 4.7: *Concept Activity* view of developer *lile* in the Lugano SVN ecosystem.

Figure 4.7 presents the evolution of the frequency of occurence of 30 of the natural language terms that developer *lile* uses the most in his writing. Peaks and valleys characterize their evolution. It is possible to notice the increase of importance of some terms as far as concepts that have been important from the beginning to the end of the modeled history. A deeper analysis of this developer and its concepts will be described in the next Chapter.

Size and Activity by File Stacked-Graph

These perspectives exploit cumulative and non-cumulative stacked-graphs to depicts the evolution of files extensions. The evolution of the size for each file extension is computed with respect to the number of files with the given extension (the sum of the files added to the repository minus the ones deleted). The evolution of the activity for each file extension is computed with respect to the number of changes or additions involving this extension, over time.

The evolution for each file extension can be performed at the developer, project or ecosystem level. The choice of a cumulative graph for the *Size by File* perspective is due to the fact that these visualization involve files, a cumulative entity throughout the life of a project. Analyzing these visualizations and comparing them to the ones about the activity in terms of commits and changes, it is possible to understand patterns and draw conclusions. Right-clicking on the evolution of a project, a context menu allows the user to inspect the profile of that entity. Right-cicking on the evolution of a file extensions, another context menu allows the user to perform an automatic search for developers that modified files with that specific extension. The results will be presented as a tag-cloud in the *Search* view.

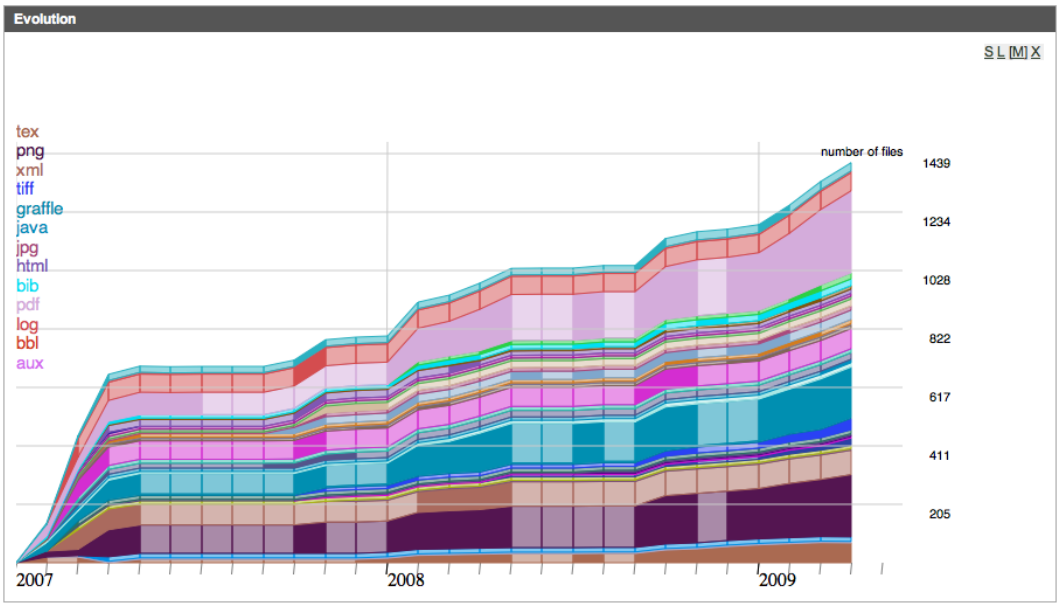

Figure 4.8: Evolution of the file extensions of developer *mircea* during its development lifetime in the LuganoSVN ecosystem.

Figure 4.8 is the evolution of the file extensions for developer *mircea* in the LuganoSVN ecosystem. A similar concept is applied in the *Activity by File* perspective (Figure 4.9) that visualizes the *activity*, in terms of additions and modifications, for each file extension over time. Given that the effort spent on different files is not a cumulative good, this visualization does not use a cumulative stacked-graph.

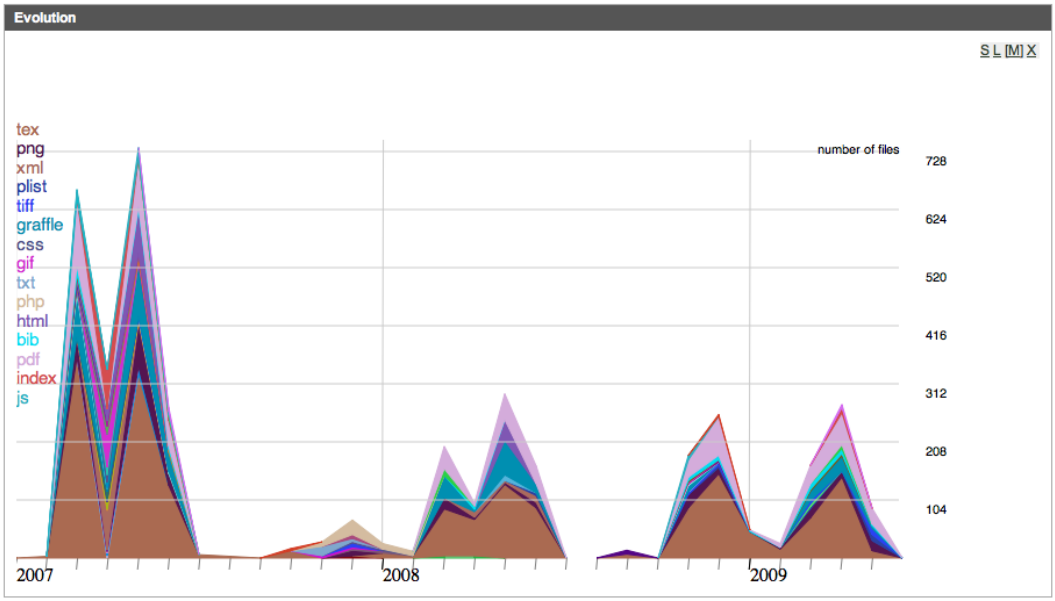

Figure 4.9: *Activity by File* perspective for developer *mircea* in the LuganoSVN ecosystem visualizing the activity on each different file extension.

Projects by Size Stacked-Graph

This perspective depicts the growth of the ecosystem according to the change, in size, of all its projects. By size we mean the number of files contained by each project over time. The previously shown Figure 4.5 describes the growth of the GNOME ecosystem in terms to the size of its projects.

Activity Tag-Cloud

Activity Tag-Clouds exploit tag-clouds to represent the activity of developers or projects. The *Developer Activity Tag-Cloud* view depicts developers as tags and their size is set with respect to the number of changes they performed to the context, be it a project or the entire ecosystem. The *Project Activity Tag-Cloud* visualizes projects as tags and their size is assigned according to the number of

changes that led to their latest version. Clicking on a tag representing a developer or project, their specific information will be displayed in, respectively, a *Developer Details* or *Project Details* view. Figure 4.10 depicts the top 100 contributors to the GNOME ecosystem in terms of file changes. The darker and bigger the name, the more files have been changed or added by that developer. A similar visualization could have been applied just to a given project, other than the whole ecosystem. As each tag-cloud, it can also be sorted according to the size of the tags, other than the default alphabetic order.

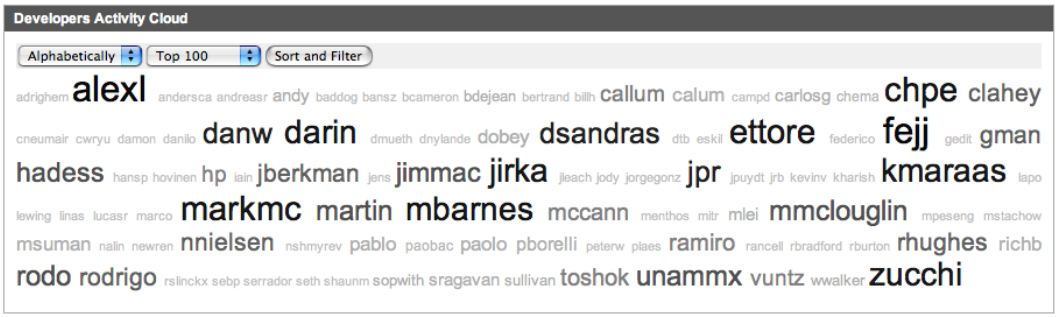

Figure 4.10: *Developer Activity Tag-Cloud* for the gnome ecosystem.

Vocabulary Tag-Cloud

A *Vocabulary Tag-Cloud* has terms as tags and their size and color is assigned on top of the occurrences of that term in the given context. These tag-clouds can visualize the vocabulary of a developer, a project or the vocabulary of the whole ecosystem. Figure 4.11 represents the vocabulary of the LuganoSVN ecosystem, clearly providing evidence of its content in terms of concepts.

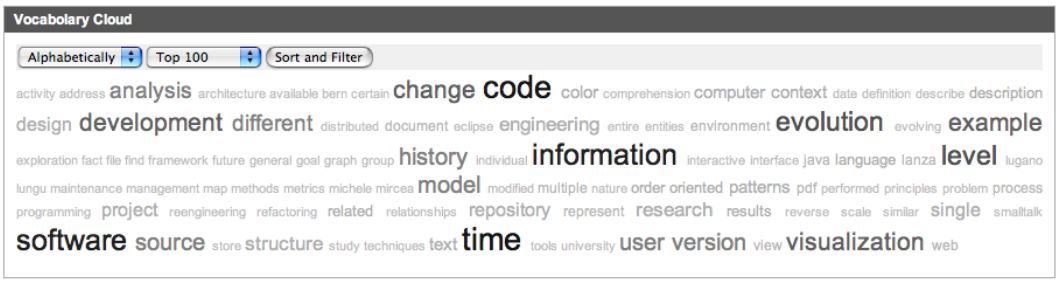

Figure 4.11: *Vocabulary Tag-Cloud* of the LuganoSVN ecosystem.

File Extensions Tag-Cloud

Tag-clouds have also been used to visually represent extensions of files that are modified by developers during their activity. A *File Extensions tag-Cloud* can represent extensions of files that a developer is used to modify and commit, as well as the set of file extensions that, change after change, created a given project or the whole ecosystem.

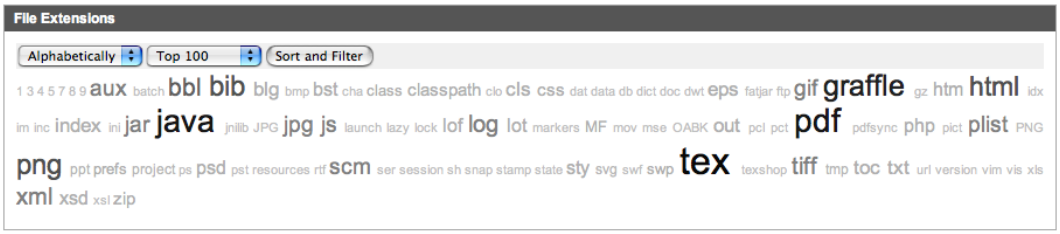

Figure 4.12: *File Extensions Tag-Cloud* for the Lugano SVN ecosystem.

Figure 4.12 shows that, in the LuganoSVN ecosystem, most of the changes involve `text`, `graffle`, `java`, `pdf`, `html` and `png` files. It is worth to emphasize the fact that the size and color of the tags does not represent the absolute number of files with that extension, but rather the number of changes involving files with that extension. This tag-cloud underlines the effort spent by developers on different kind of files.

4.3.2 New Functionalities

Not all the content that we added to SPO is meant to be purely graphical.

Developer Aliases

SVNMole fetches SVN log entries and analyzes them. Each time there is a commit to the repository, its author, comment, timestamp and information about the changes are stored. The author, in particular, is the *user name* of the user who is performing the commit. There are several circumstances, more common that we tend to believe, in which a singe, physical, developer uses more than one user name to commit its changes. Committing changes to an SVN repository is a tedious and mechanical task that is often delegated to scripts, external programs or plug-ins. Once these aids have been set up, the developer does not have to cope with the low level interface of SVN, most of the times he simply pushes a button. Unfortunately, this scenario leads to the fact that user names

contained in these external aids are rarely updated as soon as the situation requires it. Developers tend to care about the results, and the result is that their work gets versioned, it doesn't matter, sometimes, under which name. Moreover, a developer could have several aliases due to different security permissions assigned to each of them. People change their usernames in time and some accounts are used for some automated tasks and they do not represent actual developers, so they can be removed from developer-centric analysis. The reality is that it is unlikely to find an ecosystem without aliases or developers to remove.

To remove biases from the analysis and to capture the whole essence of a developer we created a view in the Small Project Observatory that specifically addresses this issue. The *Developer Aliases* view (Figure 4.13) contains two drop-down menus that allow the user to set an alias to a given developer. A table contains, on the left, the list of developers that are considered to be physical persons and, on the right, their (possibly empty) list of aliases.

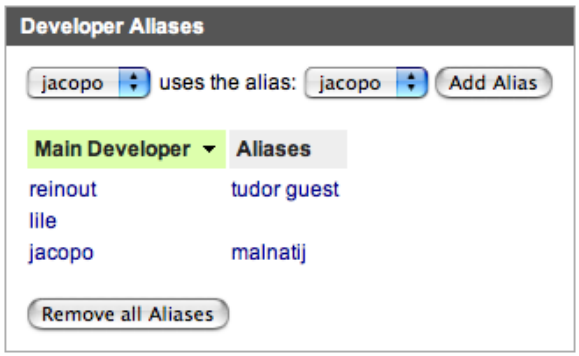

Figure 4.13: The *Developer Aliases* view. Through the menus at the top it is possible to assign aliases to developers. The table at the bottom contains data about the current situation.

Each time an alias is set, the list of developers gets updated according to the user's action. If to a developer A is added developer B as alias, all the aliases of developer B will be added to A. Clicking on the name of an alias, that alias will be removed from the list of aliases of a given developer. Clicking on the name of a developer who is not an alias, will bring the user to that developer's profile.

Assigning aliases to a given developer implies that all the information regarding the aliases are injected into the developer they are assigned to. The system will no longer make difference between the contributions and data of a developer and the one of its aliases.

Search

The data contained in an ecosystem is enormous.
One of the goals of our analysis is to give the possibility to automatically identify experts in a given area, be it a concept, a technology or a programming language.

This possibility is given by the *Search* view. This view allows searching for developers or projects according to concepts (as keywords) or file extensions. A concept related search identifies developers (or projects) that worked with the given concept, expressed as a single word or a set of them. A file extension search retrieves developers that work on certain file extensions. This can provide insights into who is contributing to the ecosystem files with the given extensions, or which projects contain them. If a *Vocabulary Tag-Cloud* or a *File Extensions Tag-Cloud* is useful to characterize a developer and understand his area of interests, the *Search* view is a complementary functionality that, given concepts or a set of extensions, gives the possibility to understand which entities are related with them. We decided to depict the result of the search as a tag-cloud because it immediately provides information about the degree of importance of each result. Mapping the color and size of each tag according to the fitness with respect to the search criteria, it is straightforward to understand which entity should be further inspected (by clicking it, as in each tag-cloud in SPO).

At the top of the *Search* view there is a text field in which the user can enter a set of concepts (*evolution metamodel* in Figure 4.14) or file extensions (*xml* in Figure 4.15). Multiple words need to be separated by spaces. It is then possible to select the kind of search through a drop-down menu: *File Extension* or *Vocabulary*. Finally, the user can select the targets of the search through another drop-down menu: *Projects* or *Developers*. It is therefore possible to search for developers or projects with respect to a given set of concepts or developers and projects according to the extensions of the file they contributed or contain.

Once a user selected the search criteria, a vector containing the search terms is created. For each target entity, be developers or projects, a vocabulary or file extension matrix is created and compared with the vector containing the search terms. The fitness of each entity is computed according to the sum of the occurrences of words or extensions provided by the search criteria. Figure 4.14, for example, depicts the result of a vocabulary search on developers with respect to the concepts *evolution* and *metamodel*. According to the results, developers *lanza, romain, mircea* and *tudor* are likely to be the domain experts in the analyzed ecosystem (LuganoSVN).

Figure 4.14: Vocabulary search on developers

Figure 4.15 is the result of searching for projects containing `xml` files in the GNOME ecosystem. *Evolution, gnome-games, gnome-applets, nautilus* and *gnome-user-docs* are the projects with the biggest amount of `xml` files in the ecosystem. Other six projects (*epiphany, gnome-panel, gdm, gnome-control-center, gnome-utils* and *gedit*) seem to contain a moderate number of them.

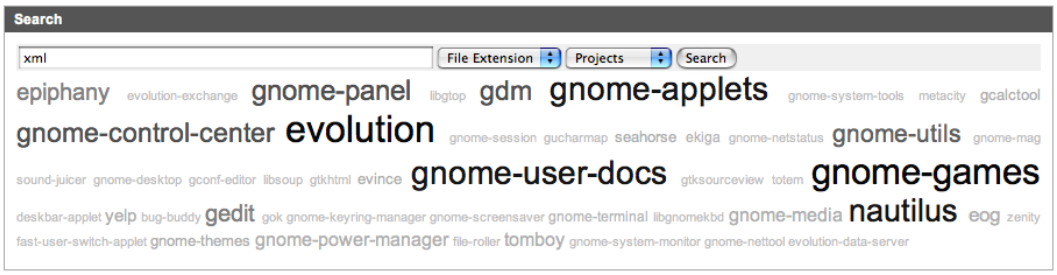

Figure 4.15: File Extension search on projects

4.4 Enhanced Visualizations

All the visualizations that have been described in the previous section can also be used as web application components within other pages. They can be rendered in a single page or be a component, among others, in a bigger page providing a broader set of visual and textual information.

Given that most of these views can be used at the developer and project level, we decided to include them as components in their overviews and details pages. The complete list of projects and developers of the ecosystem can be found, in SPO, in the *Projects* and *Developers* pages. Both of them have been enhanced with a tag-cloud, on the top, depicting the most active entities. The goal of these tag-clouds is twofold: on one hand they quickly provide to the user the most active entities, while on the other they become a useful mean of navigation. Clicking on the interesting entity will bring the user to a page containing

its details.

The same information provided by the tag-cloud is contained in the table listing all the entities and their values for several metrics. The tag-cloud gives information about one of these metrics (number of changes) exploiting the pre-attentive processing of visual information. Users can still inspect the content of the table and mentally process the values of the metric, but at that time they will already have a mental model of the most active entities in terms of effort and activity.

Figure 4.16 is a screenshot of the *Developers* view in which it is clearly visible the tag-cloud on the top and the table containing the list and values for several metrics at the bottom. The *Projects* page has been enhanced in a similar way. The *Project Details* and *Developer Details* views have been enhanced by

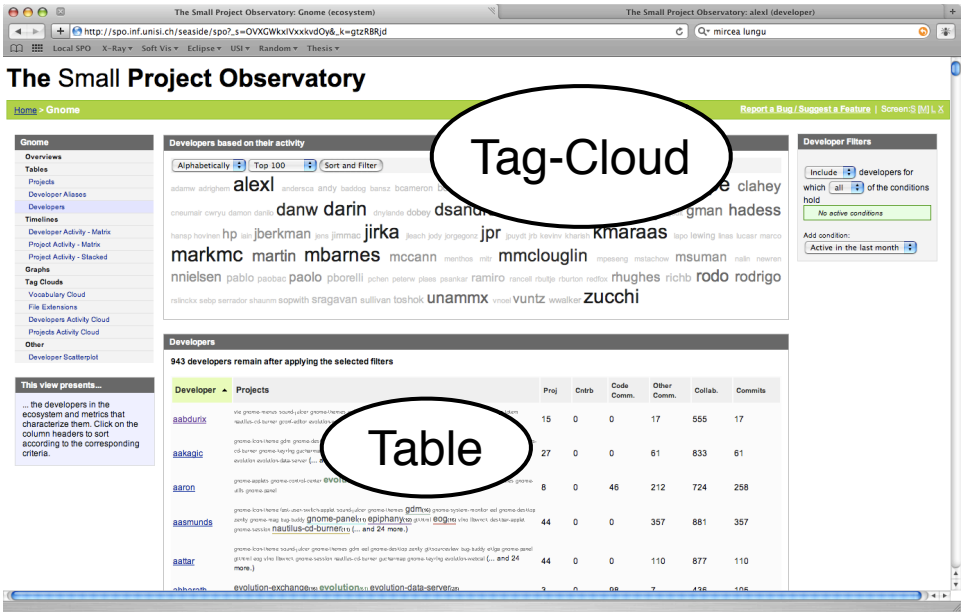

Figure 4.16: A portion of a *Developers* page. At the top, the tag-cloud depicting the activities of developers. At the bottom, the complete list of developers and metrics.

adding a vocabulary and a file extensions tag-clouds components. An activity stacked-graph has been added to the *Developer Details* view, depicting the evolution of the activity (numbers of commits) in each projects to which the developer contributed (Figure 5.36). In the *Project Detail* view, instead, a *Size & Activity Evolution* stacked-graph compares the size of the projects versus the activity of its developers.

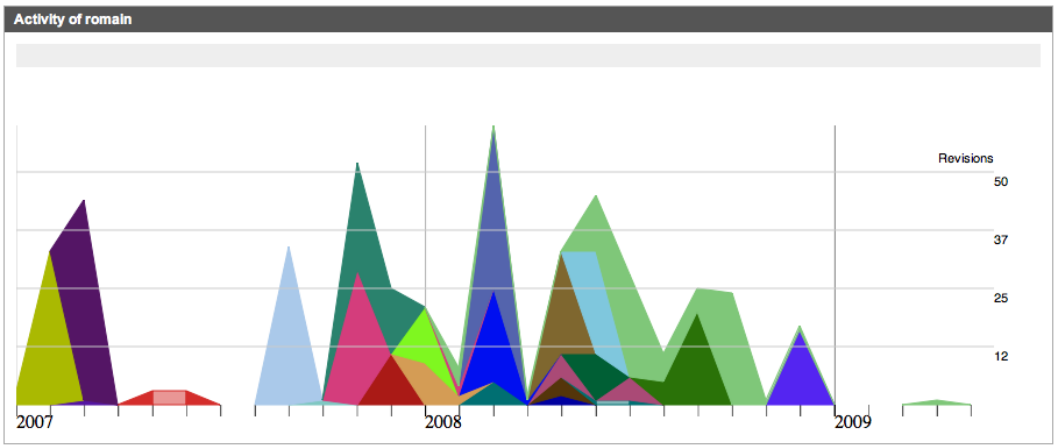

Figure 4.17: Activity stacked-graph of developer *romain* in the LuganoSVN ecosystem. Each color represents a different project.

Chapter 5

Validation

This chapter will describe two case-studies. The first one is the GNOME ecosystem, a set of 67 projects in the GNU Object Model Environment, a user-friendly desktop for Unix operating systems. The second one is the SVN repository (LuganoSVN) of the REVEAL research group headed by prof. Michele Lanza at the *Università della Svizzera Italiana*. The two ecosystems are really different in terms of age and content, the first contains a huge amount of code while the second, mainly research papers. The next sections will go through their analysis, providing examples of how the combination of our applications can be useful in characterizing the content of these ecosystems.

One of the first things to do while analyzing an unknown ecosystem, is to understand the big picture before going in depth. The ecosystem overviews are useful to get a feeling about the size and overall activity of the projects in the ecosystem, as far as the amount of developers and distribution of their efforts. These views highlight interesting or important contributors that can be further inspected. While analyzing the content of a view, it is tempting to draw immediate conclusions. This is the most dangerous interpretation pitfall. Single metrics do not characterize an entity, but rather tell something about a specific aspect. All the perspective should not be considered alone, but in a larger context.

The Small Project Observatory is a versatile application that gives the possibility to depict and analyze ecosystems in many different ways. In the next sections we will use its visualizations and functionalities to characterize developers.

5.1 GNOME

5.1.1 Exporting and Importing the Model

Modeling all the 67 projects and the 943 developers of the GNOME ecosystem took, for our library, about 5 hours on a machine with 6 CPU cores at 1.6Ghz each and 4GB of main memory. Exporting the whole ecosystem to an MSE file of 579MB took, on the same machine, about 3 minutes. The model, then, has been imported on a MacBook Pro with 2 CPU cores at 2.2Ghz and 2GB of main memory in about 30 minutes.

The differences between versions of `c, cpp, h, xml, py, cc, js` and `cs` files was taken into consideration while building the vocabularies of developers, providing this list of extensions to SVNMole.

5.1.2 GNOME Analysis

The first thing that we can do to analyze this set of projects, is to visualize their evolution in terms of activity. This is a good overview of the status of the projects and their importance in the ecosystem. Given that the project activity is composed of developer commits, this view provides a general overview of the distribution of their effort in the whole ecosystem. Figure 5.1 is the *Project Activity* stacked-graph for the whole set of projects. It is straightforward to notice that the ecosystem is composed of a big set of active projects.

Few of them were active since the beginning, while most of them have been introduced after 2002. It is also possible to identify 3 distinct periods: from 1998 to 2000, from 2000 to 2003 and from 2003 to late 2008, the date in which all the SVN repositories of this ecosystems have been archived and given to researchers like us as case-study. The first period is characterized by the development of few projects. there are little peaks and valleys in their activity (in terms of number of revisions) but it is unclear whether they follow a pattern. At the beginning of 2000 two projects became preeminent, in terms of activity: *evolution* and *nautilus*. After 2003 many other projects have been added to the repository and a pattern in their activities starts to emerge. Their evolution exhibits peaks in their activity exactly each 6 months, in January and July. This is a visual expression of the release cycles of GNOME. All of these projects are released each 6 months and these peaks are due to pre-release activity and the consequent post-release calm.

The *Projects Activity* perspective emphasizes peaks and valleys in the effort

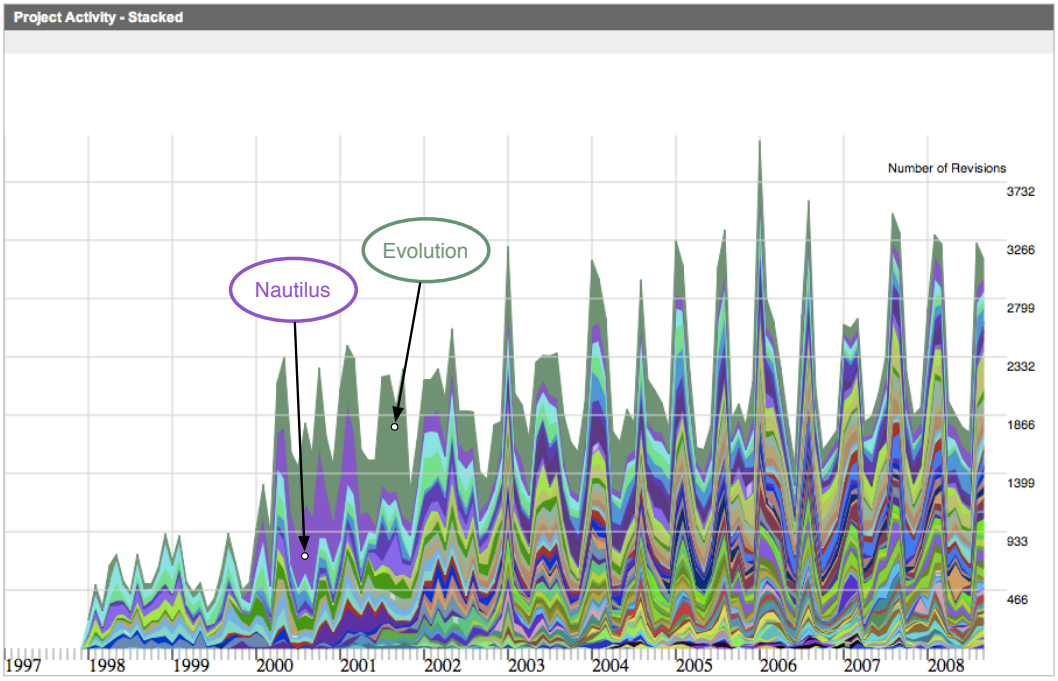

Figure 5.1: *Project Activity* stacked-graph for the GNOME ecosystem.

of developers and the *Projects Activity Tag-Cloud* (Figure 5.2) helps to identify which projects have been the most active so far, without enriching its data with the notion of time. Projects *evolution, nautilus, gnome-panel, evolution-data-server, gnome-applets* and *gnome-games* are the most active ones.

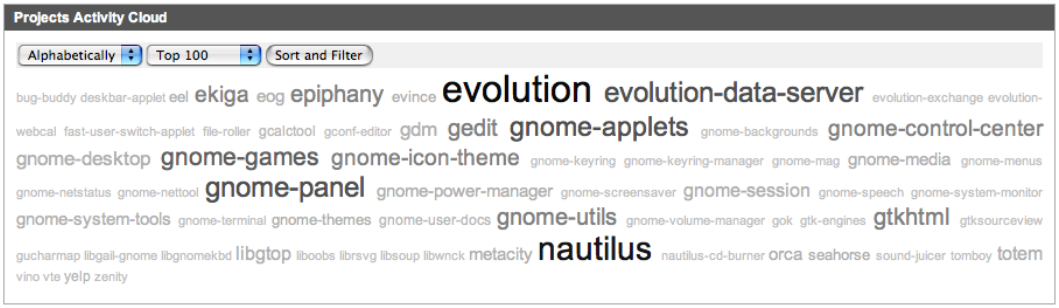

Figure 5.2: *Projects Activity Tag-Cloud* of the GNOME ecosystem.

At this point, it is still unclear if the peaks in the activity of all the projects are just about changes or even file additions. To answer this question, we visualize the evolution of all the projects in terms of their content (files with a given ex-

tension) and in terms of activity on that content (additions and modifications) through the *Size by File* and *Activity by File* perspectives. Figure 5.3 describes the evolution of all the files, divided by file extension, from 1998 to September 2008. This view represents, in each interval of time, the amount of files with the given extensions that have been introduced in the ecosystem. The first thing that we can observe is that there are no peaks. The shape describing the files

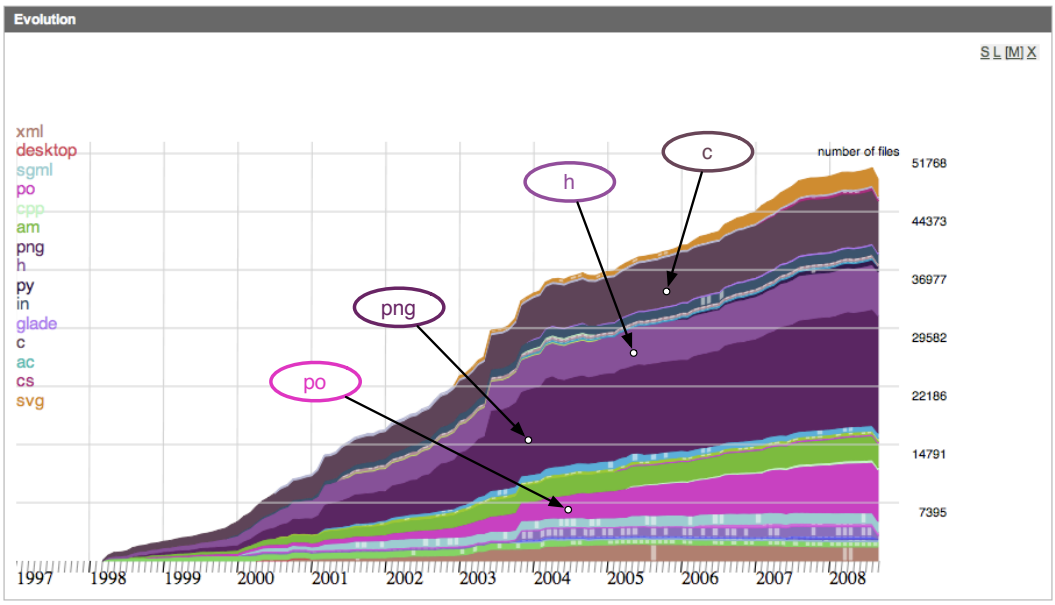

Figure 5.3: *Size by File* perspective of the GNOME ecosystem.

contained in the ecosystem gently grows without noticeable peaks. It is clear, now, that the peaks in the activity of projects is due to modifications and not additions.

This view, moreover, confirms what we already noticed in Figure 5.1: the ecosystem evolution can be divided in 3 separate periods, a first one, with limited activity and content, until 2000, when the activity increased and the content of the ecosystem started to grow. Finally, after 2003, we can see an explosion in the number of files. Most of the files in the ecosystem are `c`, `h`, `png` and `po` files. `c` and `h` files are associated with source code, while `png` files are images. The `po` extension is associated with internationalization files, used by applications to provide the user text and menus in different languages. If we compare the *number* of files with the *activity* on them, we can understand how the focus of developers evolved. In order to do that, we can compare Figure 5.3 with Figure 5.4, the *Activity by File* of the ecosystem. This visualization depicts, over time,

the number of files with a given extension that have been modified or added. The non-cumulative nature of this view emphasizes the focus on the effort of developers, that is subject to change and is not a cumulative good. Analyzing

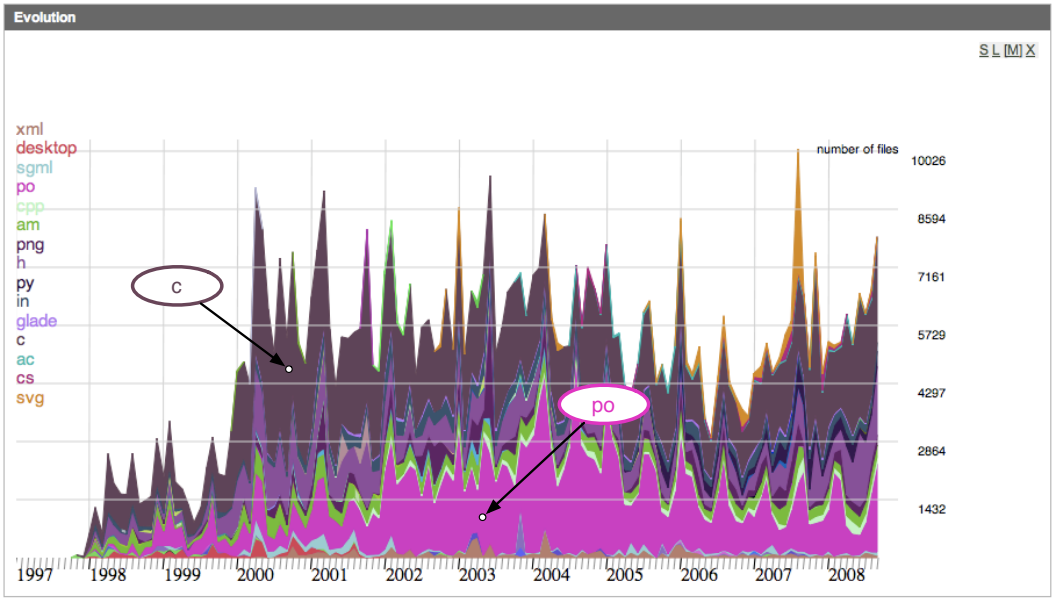

Figure 5.4: *Activity by File* perspective of the GNOME ecosystem

Figure 5.4 we notice that most of the developer efforts are on `c` and `po` files. Moreover, the effort on internationalization files dramatically increased after the last months of 2001. Increasing the quality and keeping up to date texts and menus could be a symptom of the politic of GNOME. `png` files, that according to Figure 5.3 are an important portion of the files in the ecosystem, are almost invisible in the *Activity by File* perspective. This implies that once they are added to the system, they are rarely (or not at all) modified.

The *Activity by File* perspective, at the ecosystem level, does not provide any information about the activity of the single developer. The same thing applies to the *Project Activity* stacked-graph in Figure 5.1. If we want to have an overview of the developers activity we can use a *Developer Activity Tag-Cloud*. This visualization, Figure 5.5, depicts the most active developers as darker and bigger tags in its visualization. A quick analysis of the tag-cloud is enough to understand that the most active developers are, or were, *alexl, chpe, ettore, fejj, kmaraas, markmc, mbarnes* and *zucchi*. This tag-cloud does not take into consideration the notion of time. It is possible that one or several of these developers were only active in the past. Now that we have the log-in names of the most active

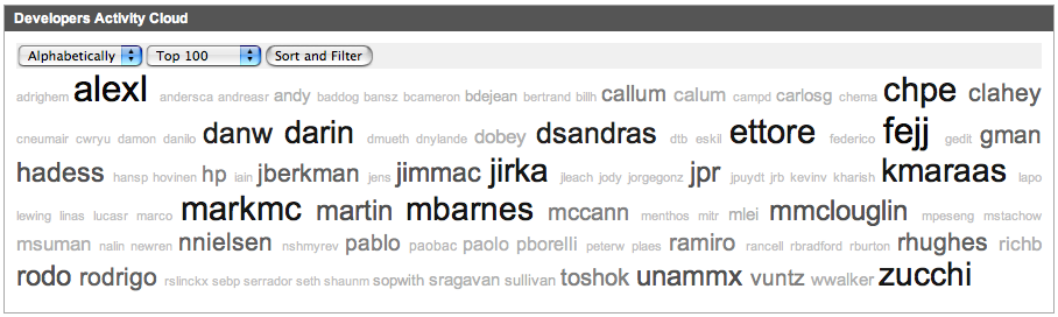

Figure 5.5: *Developer Activity Tag-Cloud* for the GNOME ecosystem.

developers, we can start an internet search to assign them real names, faces, and context. Reading the blog of *markmc*, it is possible to discover his real name: Mark McLoughlin. One of the other log-in names in the ecosystem is *mmclouglin*, it is even depicted as a dark-gray tag in Figure 5.5. Both of these names, *markmc* and *mmclouglin* belong to the same developer. The reason why this developer has 2 aliases is unknown. Analyzing the *Developer Activity Matrix* (Figure *markmc*), we notice that the two aliases have been used in two separate and contiguous periods of time. It is likely that *mmclouglin* has been replaced by *markmc* because the second one is shorter and the first one contains a typo (the surname, McLoughlin, is spelled without an "h"). On top of this information, we can now set *mmclouglin* as alias of *markmc* and proceed with our analysis.

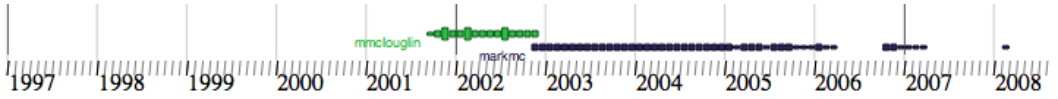

Figure 5.6: *Developer Activity Matrix* for *markmc* and *mmclouglin*.

The *Activity by File* view of the whole ecosystem emphasized a clear distinction between coding and internationalization effort that can be analyzed at the developer level, to understand who is responsible for code and who for po files. The *Effort Distribution for Developers* perspective has been created specifically for this reason. In Figure 5.7 the position of developers on the horizontal axis represents the number of changes performed to source code files, while the position on the vertical one represents the number of changes performed on internationalization files. Analyzing the scatterplot we see that the majority of developer did not contribute more than 3000 file changes, be it code or internationalization. There are also some outliers that are worth to be further inspected. It is not surprising that they are the most active developers depicted in the tag-cloud

of Figure 5.5. Another interesting aspect of this scatterplot is that it describes an ecosystem where the focus is more on source code than internationalization: the majority of developers lay at the bottom of the graph. This is not surprising, the GNOME ecosystem is composed of SVN projects versioning software systems. Some of the outliers have a number of file changes extremely high.

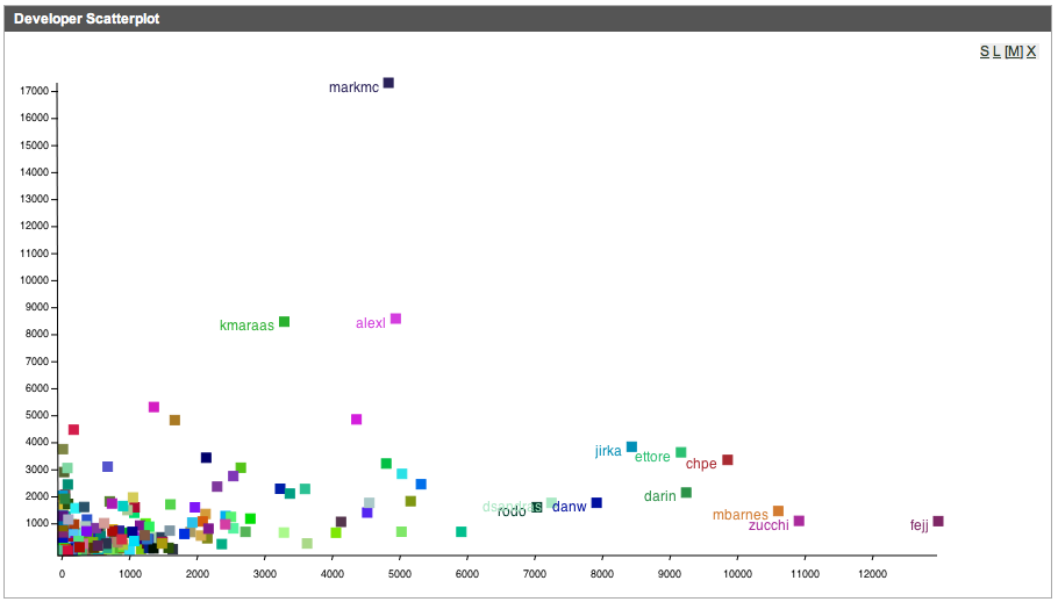

Figure 5.7: *Effort Distribution for Developers* perspective for the GNOME ecosystem. The number of changes of source code files and internationalization files are mapped, respectively, on the horizontal and vertical axis.

Reading documentation and the policy of GNOME, we tend to believe that not all the file changes that are associated with a developer come from that specific developer. It is the case, in fact, that a young contributor submits his changes to an older developer for his blessing. In an open-source community like GNOME, with hundreds of developers and anonymous contributors, sometimes it is necessary for an high rank developer to check the contributions of others. Once their contribution is accepted, it is committed. The contribution now goes under the name of the high rank developer. We do not have explicit documentation about this phenomenon, but we saw several times commit logs belonging to a developer stating that the change was performed by another one. Our models and visualizations do not cope with this issue, yet.

Now that we have a list of important developers, the analysis should continue inspecting them. Given that that our space is limited, we will now inspect a subset of them: *alexl* and *fejj*.

Analysis of developer alexl

From Figure 5.7 we already know that this developer contributed almost the same amount of source code and internationalization files. Its *File Extensions Tag-Cloud* (Figure 5.8) confirms that: `po`, `h` and *c* are extensions on which there is heavy activity.

Figure 5.8: *File Extensions Tag-Cloud* for developer *alexl*.

We can also see that *alexl* contributed `am` and `in` files, usually used on *nix systems to build `c` code. The developer contributed to the system even a discrete quantity of *png* images.

Right now we know that he contributed both source code, images and internationalization files but we do not know when, and to what projects. His *Developer Activity* gives us an overview of his activity, in terms of revisions, for each of the projects he contributed to.

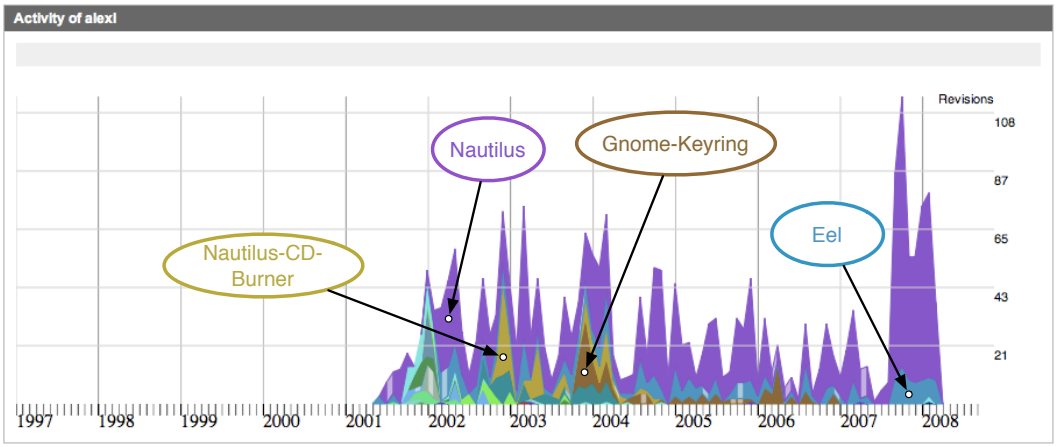

Figure 5.9: *Developer Activity* view for developer *alexl*.

His activity extends form March 2001 to March 2008. During the first year of his activity he took part in several projects, dividing his efforts among them. This distributed effort did not last long. At the end of 2002 he was active just

on projects *nautilus, nautilus-cd-burner* and, to a smaller degree, *eel*. During summer 2003 *alexl* started to work on *gnome-keyring* and kept contributing to that project until late 2006. At any given time in his activity history half of his efforts were always focused on *nautilus*. In particular, between June 2007 and March 2008, there is a considerable burst of activity on this project. There are several peaks in the activity of this developer and most of them coincide with the semestral release dates.

Figure 5.10 depicts his *Files by Type* perspective. The shape described by the evolution of file types is uncommon.

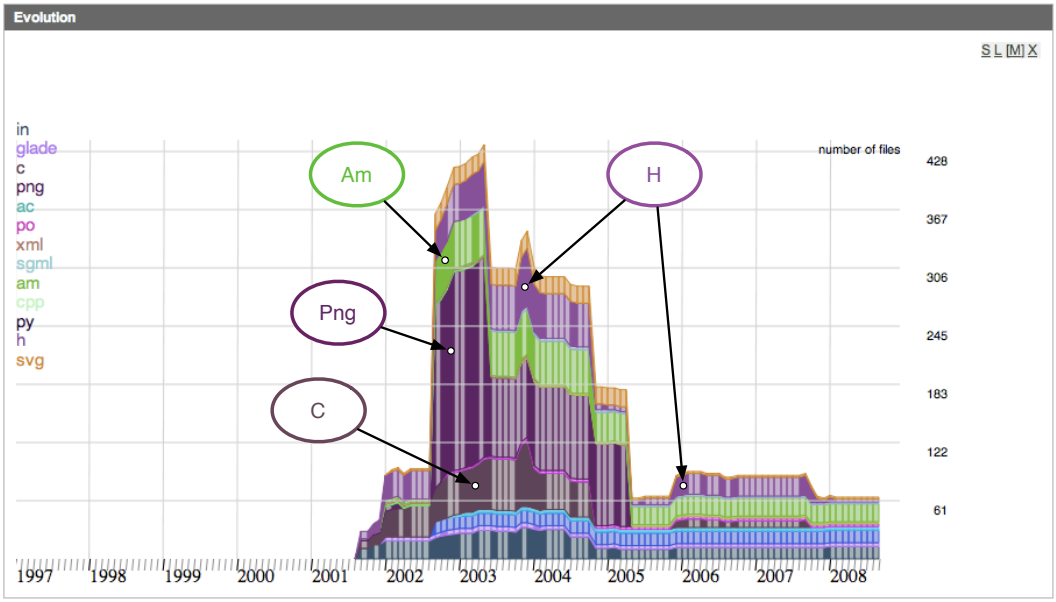

Figure 5.10: *Files by Type* perspective for developer *alexl*.

This visualization tells us that in June 2002 *alexl* contributed a massive number of `png` files as well as `am` files. The quantity of `am` files remained approximatively the same until the SVN repositories we are analyzing were frozen and stored as research data. Te amount of `png` files dramatically decreased in March 2003 and in March 2005 *alexl* removed the remaining ones. A similar thing happened to the evolution of `c` and `h` files. They have been added by *alexl* since the beginning of its activity and then removed almost completely in September 2004 (a small quantity of `h` files have been added between 2006 and late 2007). A possible explanation for this behavior is that his contributions have been reorganized by other developers and *alexl* removed form the repositories his old

files. It is also worth to notice that even if this developer were particularly active on internationalization files, the amount po files is almost invisible in this graph, meaning that *alexl* modified, but not added, internationalization files.

The fact that he removed source code files does not necessarily mean that he stopped working on this file type. This hypothesis is supported by his activity depicted in Figure 5.11.

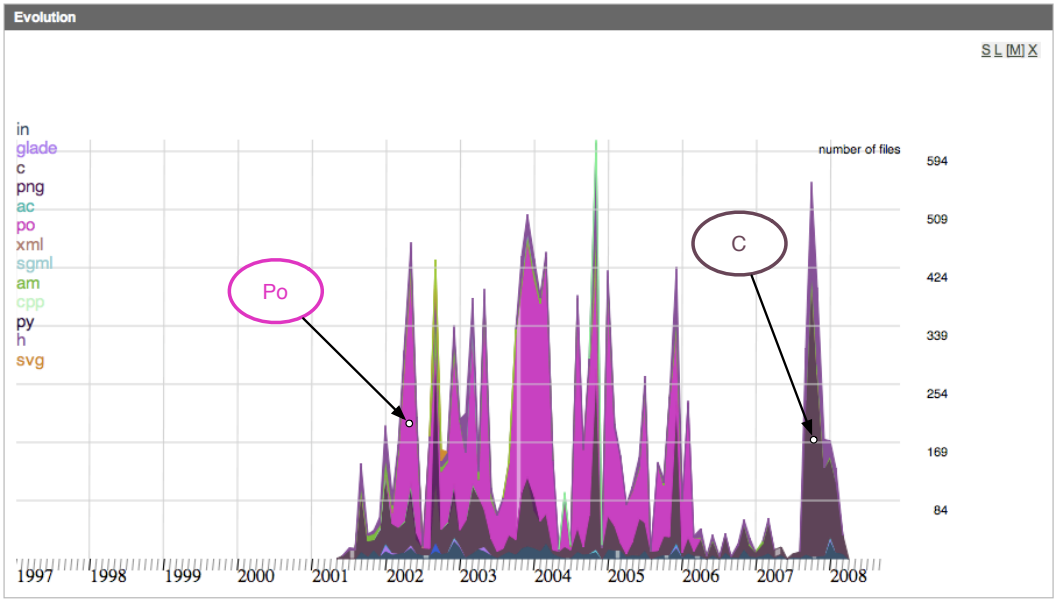

Figure 5.11: *Activity by Type* perspective for developer *alexl*.

His *Activity by Type* perspective shows a clear and preeminent effort on internationalization between 2002 and 2006. *alexl* modified and contributed c code during the entire time span that goes from 2001 to 2008, thus, even after deleting source code files (Figure 5.10). It is also probable, analyzing this perspective, that his burst of activity on the *nautilus* project (Figure 5.9) was focused on c files.

If we want to understand the concepts on which he was working we need to analyze his *Vocabulary Tag-Cloud*. By default, this tag-cloud will show us the top 100 terms that the developer used while working on his projects. Unfortunately, *c* programmers tend to give cryptic names to their entities, especially using abbreviations. This does not help the analysis of the vocabulary of a developer, because it is difficult to understand the real meaning of an abbreviation without

its context. Moreover, the vocabulary expressed over a time span of several years could be so rich and big that, like in this case, the top 100 terms are not enough to characterize the developer. Figure 5.12, in fact, does not give us a complete overview of the concepts associated with this developer.

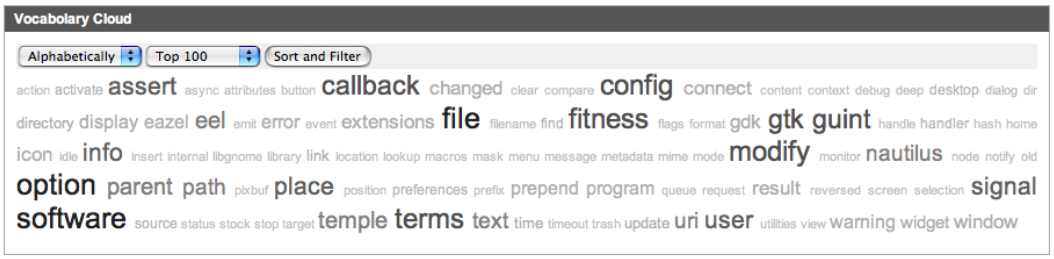

Figure 5.12: *Vocabulary Tag-Cloud* for developer *alexl* showing his top 100 terms.

Using the options at the top of the tag-cloud it is possible to create a bigger cloud containing all the terms, the terms with occurrences bigger than the median or terms with occurrences bigger than the third quartile. Selecting this last option, we obtain a more detailed overview of the vocabulary of *alexl*. We do not show the resulting tag-cloud because it would require too much space. Reading the content of this tag-cloud it becomes clear that the developer focused his efforts on *files, folders, extensions, home, flags, uri, desktop* and *headers*. This is in line with the main concepts of *nautilus*: a file manager application. Many other terms refers to GUI programming: *dialog, panel, button, display, icon*. *alexl* was also handling events and modifications, this is made clear by these terms: *message, modify, signal, event, monitor, notify*. Depicting the evolution of all of these terms is not a good idea. The *Concept Activity* perspective works best with 20 or 30 displayed terms. In a situation like this in which a developer is described by hundreds of them, it is better to rely on a tag-cloud other than an unreadable stacked-graph (due to too many lines and colors).

Analysis of developer fejj

fejj is the biggest contributor in the GNOME ecosystem with respect to source code files. Figure 5.7 clearly shows it. His *Developer Activity* perspective tells us on which projects he focused his attention.

This developer has been active since early 2000 and for about 5 years he contributed just to two projects: *evolution* and *evolution-data-server*. The second project, as the name suggests, is strictly related to the first. The activity

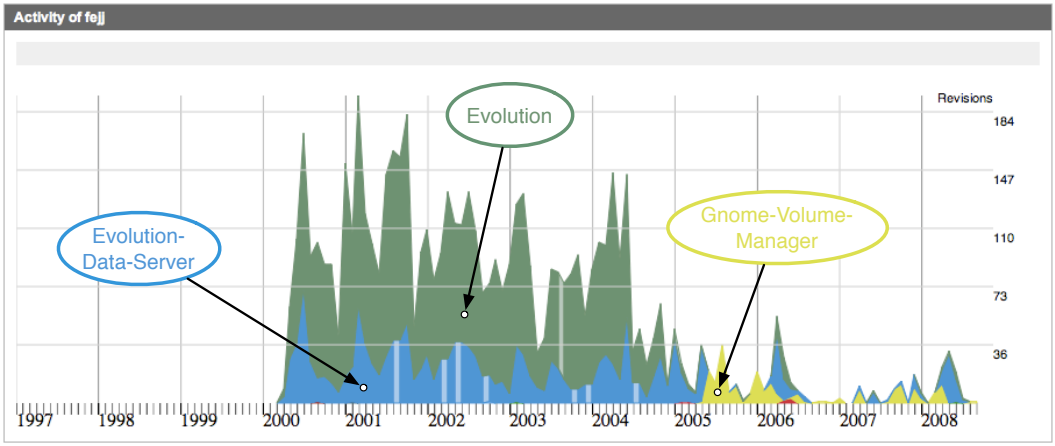

Figure 5.13: *Developer Activity* view for developer *fejj*.

on these projects has been constant and high for the first 4 years and started to decrease in June 2004. Since the end of that year, the amount of revisions on *evolution* and *evolution-data-server* are few or even absent. In April 2005 *fejj* started to contribute to another project: *gnome-volume-manager* but the amount of revisions is small. There are two clear periods characterizing the activity of this developer. The first one goes from 2000 to June 2004; these years look like spent in active development. The second one is since that month; the following years have limited and intermittent activity, a possible evidence of maintenance.

Another hint of the fact that *fejj* has been developing actively between 2000 and the summer of 2004 comes from his *Files by Type* perspective in Figure 5.14. In these years he constantly added `c` and `h` files. Between 2004 and 2005 he stopped adding files and their amount almost did not change in the following 5 years. Comparing this view with the *Activity by Type* we understand that despite *fejj* contributed approximatively the same amount of `c` and `h` files, almost all of his efforts were focused on `c` files. (This makes sense given that, usually, a developer spends much more time on functionalities other than on interface definition). In Figure 5.9 it is possible to see a discrete increase in the activity of *fejj* in the summer of 2008. His *Activity by Type* perspective depicts a surprisingly high number of *c* files that have been modified. This burst in his efforts could be due to a heavy refactoring or deep modification of existing code.

If we analyze his *Vocabulary Tag-Cloud* depicting all the terms in the last quartile, we understand that *fejj* has been actively working on *addresses, attachments, email, editor, send, authentication* and *messages*. He is also frequently using the terms *file, directory, extensions, filename, find, folder, search*. It starts to become

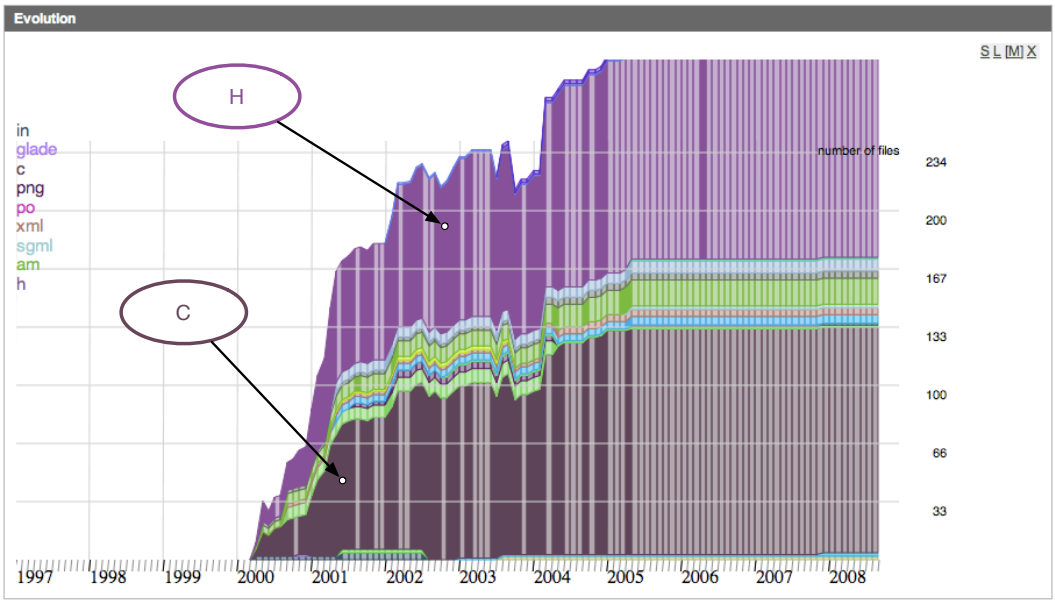

Figure 5.14: *Files by Type* perspective of developer *fejj*.

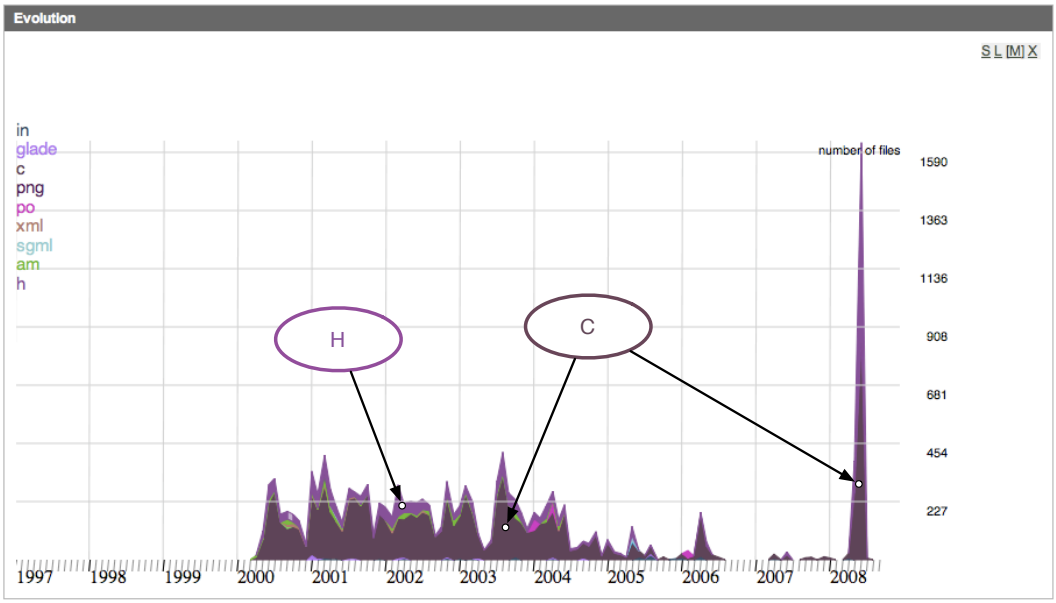

Figure 5.15: *Activity by Type* perspective of developer *fejj*.

clear that he was working on mail clients and file managers. Project *evlution*, in fact, is a mail client, contact manager, address manager and calendar for GNOME. It's interesting to notice that the word *assert* has been used frequently, a possible evidence of testing.

5.2 LuganoSVN

5.2.1 Exporting and Importing the Model

Modeling the 70 projects and 27 contributors of the LuganoSVN ecosystem took approximatively 25 minutes on the same machine were we modeled the GNOME ecosystem. Exporting the whole ecosystem took less than a minute and importing it on the same MacBook Pro where the GNOME ecosystem were imported took approximatively 2 minutes.

The differences between versions of all the textual files was taken into consideration while building the vocabularies of developers.

5.2.2 LuganoSVN Analysis

As we previously did with the GNOME ecosystem, before inspecting the single contributor we should have an overview of the context in which they operated. Figure 5.16 is the *Project Activity* stacked-graph depicting the evolution of all the 70 projects with respect to their number of commits. This visualization does not exhibit peaks following a precise pattern, they rather occur each 2 or 3 months and with different intensity. Given that this ecosystem mainly contains research papers, it is likely that these peaks are due to paper and thesis submissions deadlines. Given that each year a researcher can decide which papers to write, these deadlines will not, most likely, follow precise patterns (or at least this is what happened until now). It is also interesting to notice that the majority of the projects has activity in a short time span, usually not more than 3 months. The projects that are active for more than 3 months are likely to be thesis, curriculum vitae or common documentation, not research papers. *jacopo-devtags*, for example, is the SVN repository of SVNMole, the model extractor developed for this master thesis. *mircea-proposal* is the PhD thesis proposal of Mircea Lungu and *iene-bib* is a project containing bibliography files that are continuously updated.

As we previously said, this ecosystem mainly contains research papers. These papers are usually written in Latex (`tex`) and use `pdf` and `png` images. `bib` and `bbl` files are likely to be used as bibliography databases. Our expectation

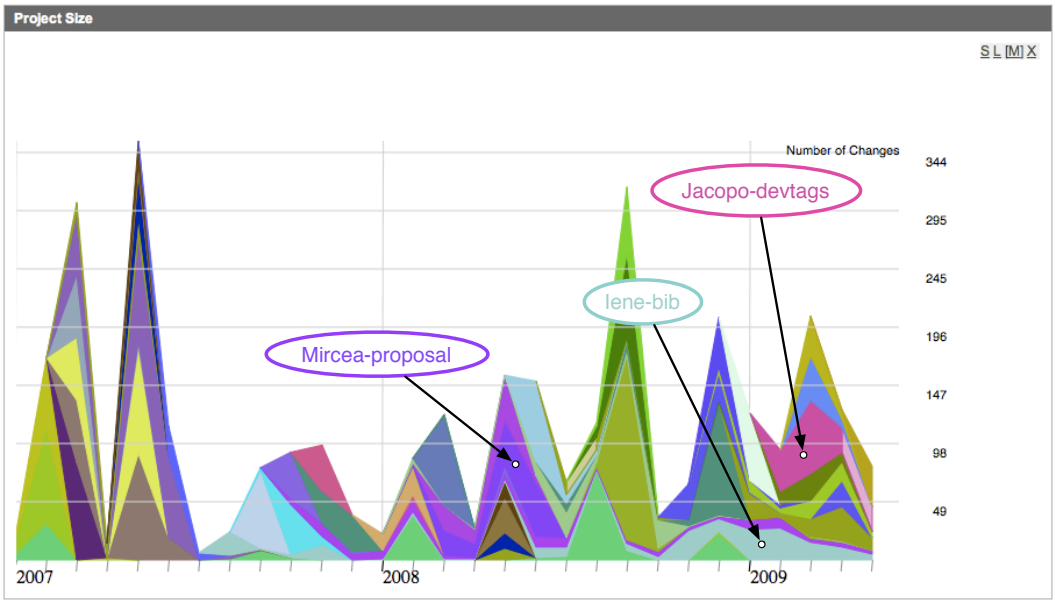

Figure 5.16: *Project Activity* stacked-graph for the LuganoSVN ecosystem.

is confirmed by the *File Extensions Tag-Cloud* of the whole ecosystem, Figure 5.17, that emphasizes the presence of these file extensions and the effort of authors in committing them. `graffle` files seem to be frequently committed too, they are binary files describing vectorial graphs and visualizations created with *OmniGraffle*[1]. `java` is the third file extension with respect to the effort of developers in committing this kind of files to the ecosystem.

Figure 5.17: *File Extensions Tag-Cloud* of the LuganoSVN Ecosystem.

Before using any developer-centric visualization such as the *Effort Distribution for Developers* perspective, we need to update the model according to the aliases in the system. Given that this is an ecosystem of our research group and we know all of its contributors, we have knowledge of the aliases of each author. We know that both *jacopo* and *malnatij* are aliases for Jacopo Malnati and

[1] `http://www.omnigroup.com/applications/OmniGraffle/`

that *mircea, tudor* and *mir* are aliases for Mircea Lungu. Several log-in names are used by scripts or have been used by several persons in different occasions (*guest*, for example) and should not be taken into consideration.

Once we have updated the *Aliases* page, we can continue our analysis. Given that this is a documentation-based ecosystem, we are now interested to see which are the authors that, despite the nature of the ecosystem, are contributing `java` source code. The *Effort Distribution for Developers* immediately answers our question. Developers *jacopo* and *lile* are two heavy contributors of source code. *mircea, romain* and *lanza* are the top three authors in terms of effort on Latex documentation. The dominant position of *lanza* as top contributor for documentation is likely to be due to the fact that prof. Michele Lanza is leading the REVEAL research group. It is highly probable that he supervises most of the research papers created within this group.

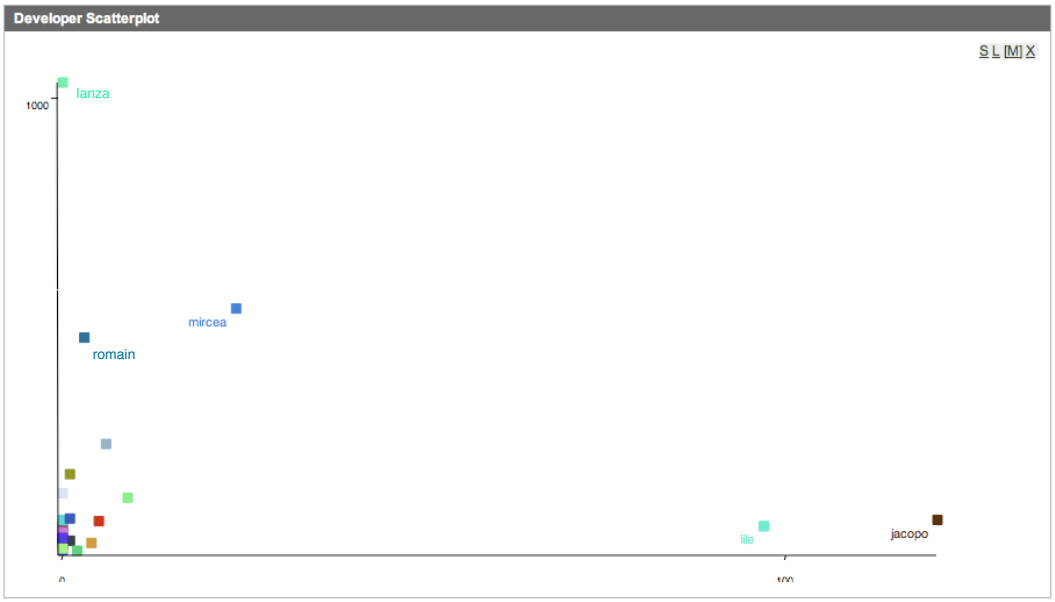

Figure 5.18: *Effort Distribution for Developers* for the LuganoSVN repository. Efforts on source code files is mapped on the horizontal axis; efforts on `tex` files is mapped on the vertical one.

We know that developers *lile, jacopo* and (to a smaller degree) *mircea* contributed `java` code, while other authors focused their efforts on writing `tex` files. What we still do not know, is the amount of files, for each extension, that have been contributed to the ecosystem. Figure 5.19 visualizes the number of files

that were present in the ecosystem each month from 2007 to the current date: June 2009. The *Size by File Type* perspective clearly states that `java` files are few and started to appear at the beginning of 2008, while the number of `tex, pdf, graffle` and `png` files started to increase since the first day of activity.

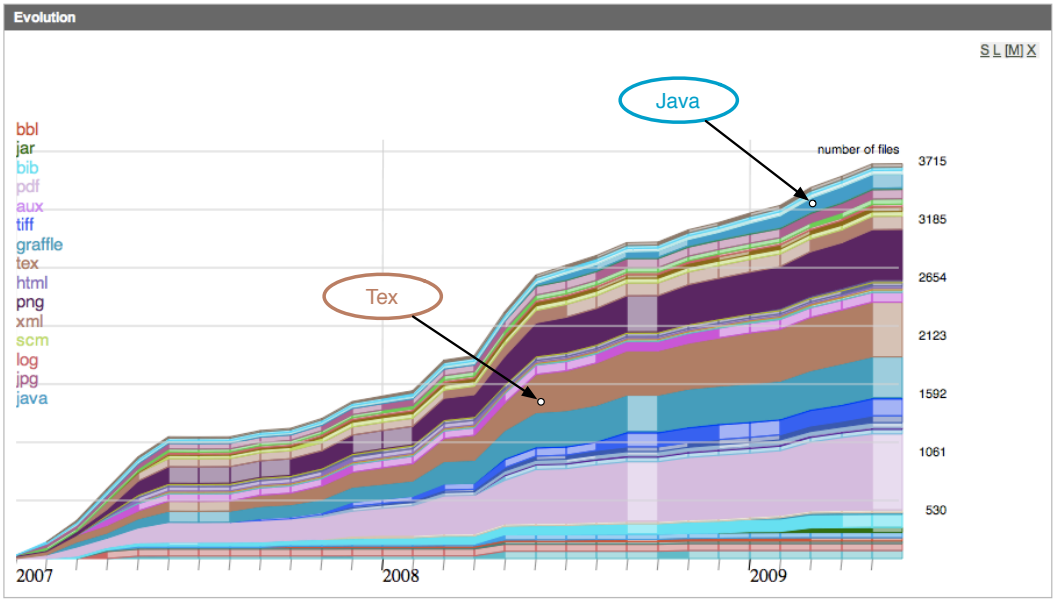

Figure 5.19: *Size by File* perspective for the LuganoSVN ecosystem.

Keeping in mind that the metric applied on the size and color of tags in the tag-cloud of Figure 5.17 is *activity* on files of that extension, we want to check *when* that activity took place. Is the activity on `java` files constant since their introduction in the ecosystem? What is the ratio between the efforts on documentation and source code over time? These questions can be answered at the ecosystem level, analyzing its *Activity by File Type* view (Figure 5.20). This perspective, depicting the activity of developers on files for each given extension, makes clear that despite being introduced in the first months of 2008, effort on `java` files is noticeable only since the last month of that year. Effort on `tex` files is omnipresent in the evolution of the ecosystem, but in the last 6 months the contributions of `java` and `tex` files are approximatively the same.

Now that we have saw the big-picture of the LuganoSVN ecosystem, we can focus our attention on contributors. The *Developers Activity Tag-Cloud* visualizes them according to their activity in the ecosystem. From Figure 5.21 it is clear that the most active authors are *jacopo, lanza, lile, marco, mircea* and *ro-*

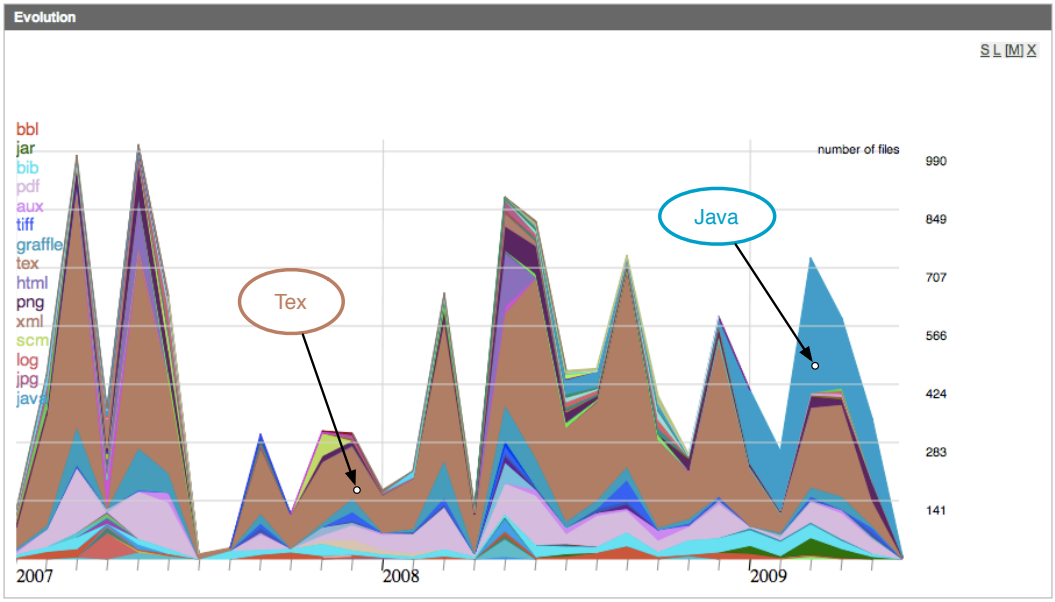

Figure 5.20: *Activity by File* perspective of the LuganoSVN ecosystem.

Figure 5.21: *Developers Activity tag-Cloud* of the LuganoSVN ecosystem

main. From Figure 5.18 we also know if they are contributing source code or documentation, what we still don't know is when they were active. To answer this question we could depict their activity in the *Developer Activity Matrix*, as shown in Figure 5.22. Analyzing the visualization from the bottom to the top, we understand that *lanza* is the only contributor that has been active from the beginning to the present day. *mircea*, depicted above *lanza*, is characterized by three separated periods of activity, each approximatively 6 months long. *romain* has an activity history similar to *lanza* until 2009, when his contributions decreased sensibly. *jacopo* has very limited activity in June and July 2007, a total absence of contributions until february 2009, after which he started again to commit data. Concluding, *lile* has almost uninterrupted activity since she started her PhD in the summer of 2008.

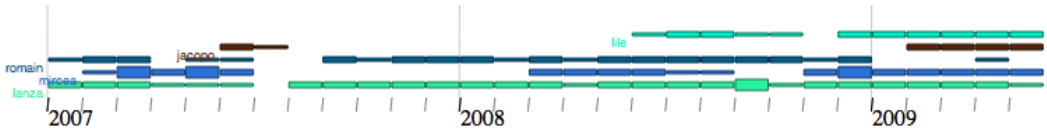

Figure 5.22: *Developer Activity Matrix* for developers *lanza, romain, jacopo, mircea, marco* in the LuganoSVN ecosystem.

Now that we have a list of candidates, we can inspect the evolution of some of them to understand who they are and what they did. For each one of them we will analyze their activity in terms of number of commits per project, we will then depict the number of files, for each file extension, contributed by the developer over time. We will visualize the effort on different file extensions in terms of additions and changes, as well as analyze his vocabulary.

Analysis of developer lile

The *Developer Details* perspective is composed of different components, each of them enriching our understanding of this developer under a different perspective. First of all, we want to understand the amount of contributions of *lile* to the ecosystem. We already know that this developer has an intense activity history (Figure 5.22) after June 2008. The *Developer Activity* component tells us how her efforts are distributed among all the projects she contributed to. Figure 5.23 shows that this developer has contributed to 5 projects during her development life time. The efforts on all of them are limited to two or at most three months, with the noticeable exception of *syde*, whose development is lasting since the beginning of the activities of *lile*. Clicking on this project and inspecting its details, it becomes clear that this is a `java` projects and not a research paper or

thesis. We already knew she was contributing a big amount of source code to the ecosystem, now we even know to which project.

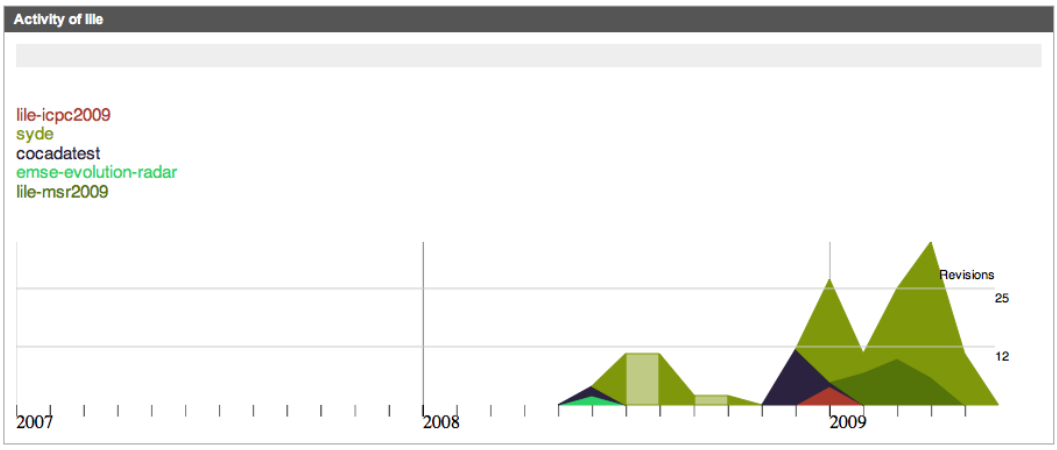

Figure 5.23: *Developer Activity* view for developer *lile*.

From her *File Extensions Tag-Cloud*, Figure 5.24 and the previously analyzed *Effort Distribution for Developers* (Figure 5.18) we know that she focused on `java` files and, to a smaller degree, `tex` and `jar` files. We know that the project *syde* is a `java` project that has been active since the beginning, but we still ignore the distribution of the efforts of *lile* and the amount of files she contributed, over time. At this point we can analyze the *Size by File* perspective that will depict the

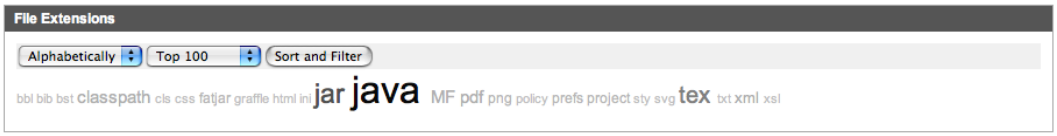

Figure 5.24: *File Extensions Tag-Cloud* for developer *lile*.

amount of files, for each extension, added by *lile* to her projects overt time. Figure 5.25 clearly shows that the majority of files added by this developer are `java` and `jar` files, while `tex` files are really few. If we compare this results with the information provided by the *Activity by File* perspective we can understand the amount of effort spent on each single file extension. Analyzing Figure 5.26 it is possible to divide her activity into two periods. The first one goes from May 2008 to November of that year. During this time span her activity was moderate and focused on `java`. The second period goes from November 2008 to the current date and exhibits a massive effort on `java` and `jar` files. In the second period

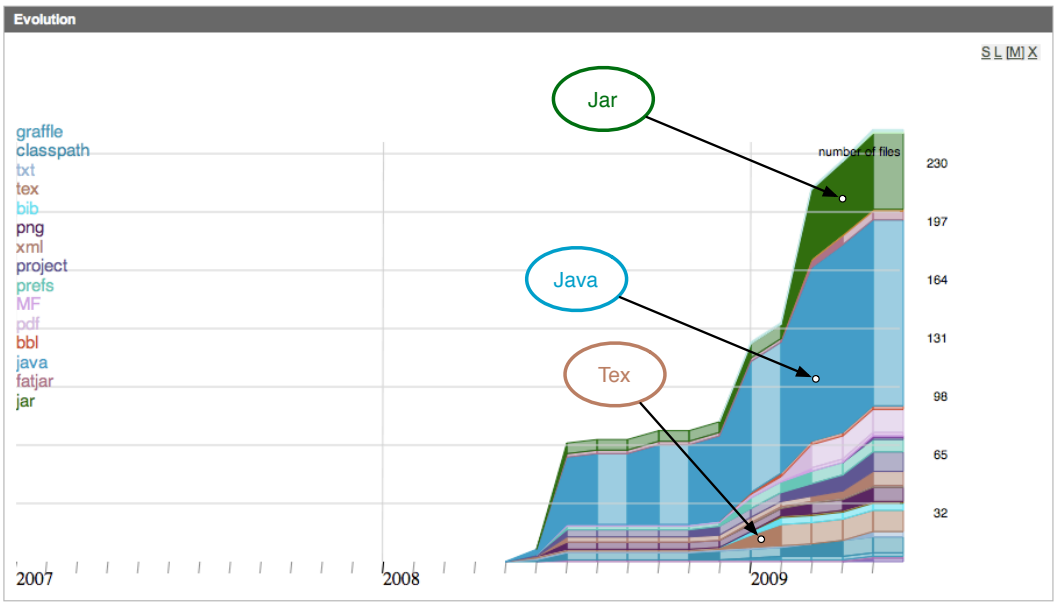

Figure 5.25: *Size by File* perspective for developer *lile*.

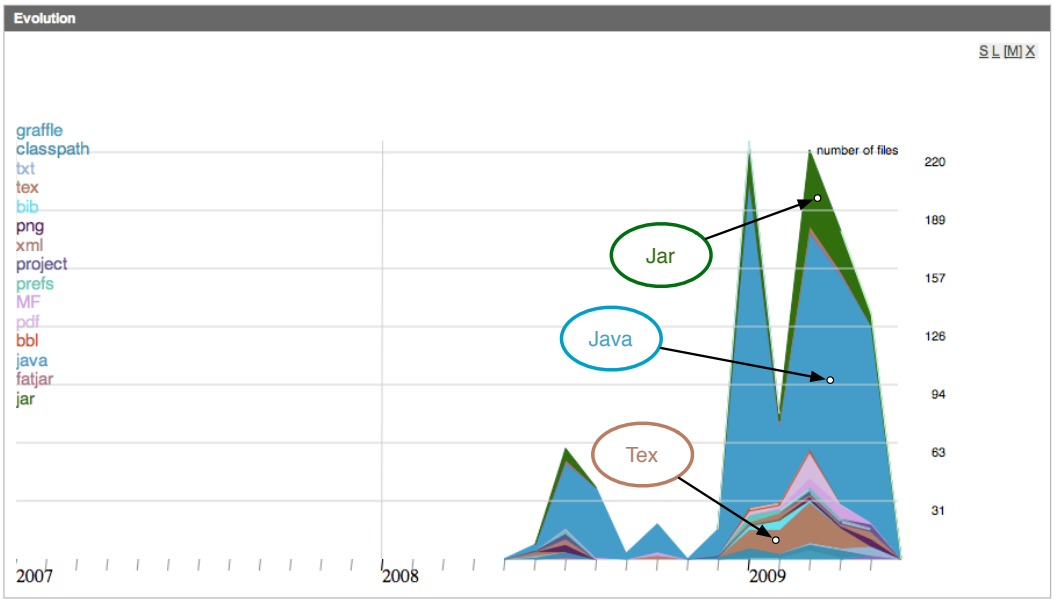

Figure 5.26: *Activity by File* perspective for developer *lile*.

there is also a noticeable, but moderate, activity on `tex` files. Comparing Figure 5.25 with Figure 5.26 it is possible to infer one fact about `jar` files. Usually, they are distributed as libraries and used by client applications. In Figure 5.25 we can see that these files were present since the beginning, being added in the first month of development. Their quantity remained constant for about 8 months until February 2009 when they started to grow, gently, each month. Comparing this information with the activity on `jar` files depicted in Figure 5.26, we notice an increase in the efforts of *lile* spent on this kind of files from February 2008. If these files belonged to libraries, there would not be so much activity on them. It is then likely that they are `jar` files created on top of the `java` code developed by *lile*.

At this point we have information about the activity and efforts in time of this developer, but we still do not know which are her main topics and how her vocabulary evolves over time. The top 100 terms used by *lile* are the ones visualized by the tag-cloud of Figure 5.27.

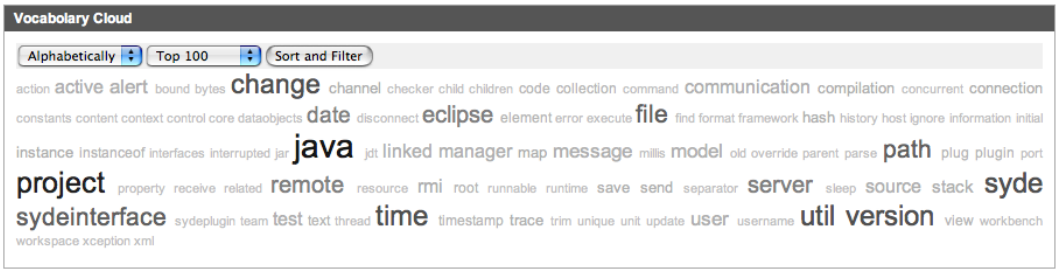

Figure 5.27: *Vocabolary Tag-Cloud* of *lile*.

From the tag-cloud we understand that probably she is working on *java* entities and she is focusing her attention on *changes, time* and *versions*, most likely in the context of *projects*. The presence of terms such as *server* and *remote* are possible evidences of a provider, or consumer, of remote functionalities.

To analyze the evolution of her vocabulary we can use the *Concept Activity* perspective, depicting the occurrences of her terms over time. The two separate periods of development are clearly visible and it is possible to notice how the terms *Java* and *Project* sensibly increased their importance after November 2008. In this period new terms have been introduced and used frequently: *time, version, change, message* and *sydeinterface*. The first three terms exhibit a focus on evolutionary concepts, while the last two could be interpreted as part of the previously hypothesized remote functionalities.

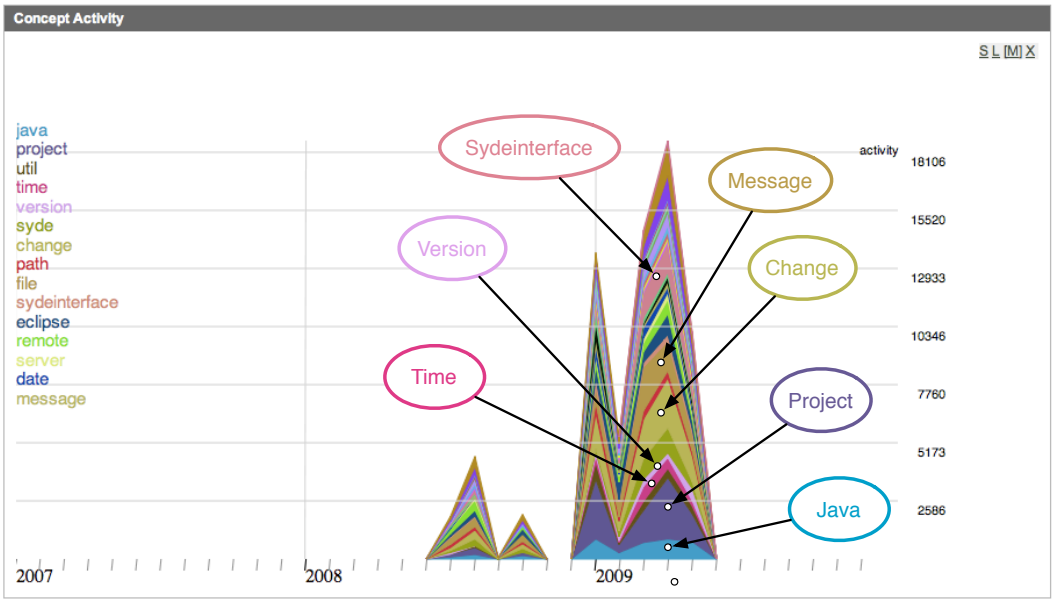

Figure 5.28: *Concept Activity* of *lile*.

Analysis of developer jacopo

Developer *jacopo*, similarly to *lile*, is an heavy contributor of `java` code. His *Developer Tag-Cloud* emphasized this fact and provides evidences about a moderate activity also on `tex`, `png` and `pdf` files. His activity is distributed on 4 projects where just 3 have a reasonable size. This information is depicted by the *Developer Activity* perspective in Figure 5.30. Even for *jacopo* it is possible to

Figure 5.29: *File Extensions Tag-Cloud* for developer *jacopo*.

identify two separate periods in his development. The first one goes from May 2007 to July 2007. In these three months all the efforts were concentrated on a single project: *jacopo-thesis*. The second period goes from january 2009 to the present day. In this last period the effort has been divided between the remaining 3 projects (two of which important: *jacopo-devtags* and *jacopo-master-thesis*). As a side remark, *jacopo-devtags* is the SVN repository of SVNMole and *jacopo-master-thesis* is the master thesis of this developer. From the analysis of Figure 5.30 is is also possible to see that the development of *jacopo-devtags* has been

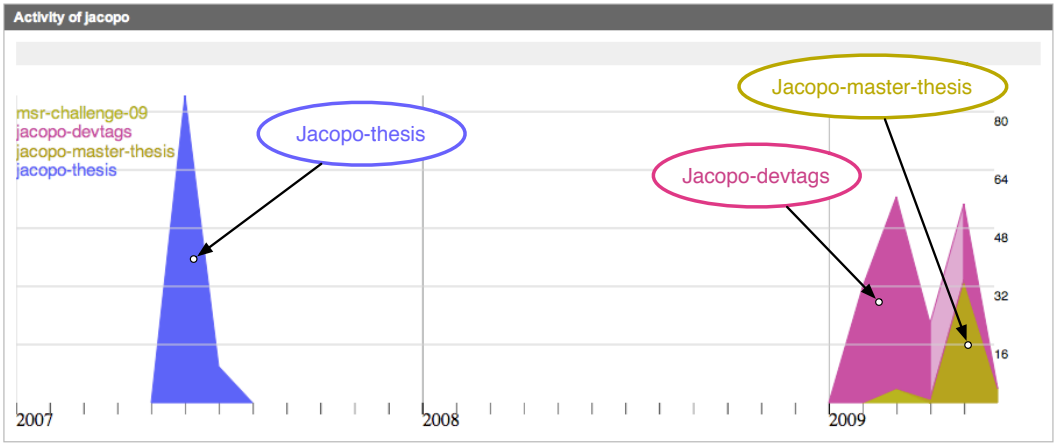

Figure 5.30: *Developer Activity* perspective of developer *jacopo*.

suspended during the first months in which *jacopo* was writing his master thesis. Minor efforts on SVNMole can be noticed in May 2009, as possible evidence of maintenance.

Now that we have an idea of how his effort has evolved through time, we can inspect the *Size by File* perspective to understand when amount of files for each given extension over time. We know that *jacopo* focused his efforts on the project *jacopo-thesis* for three months, but comparing this information with the graphs of Figure 5.31, we notice that he added files to this project only during the first month of development. In the remaining 2 months he just modified and updated existing files.

It is interesting to notice the evolution of the amount of `java` files during the second period of activity of this developer. The number of this files increased during January and March 2008, but decreased in February and April of the same year. This is a clear symptom of refactoring or code cleaning. Comparing the *Size by File* perspective to the *Activity by File* view, we can inspect the situation that led to this symptom.

Figure 5.32 describes a clear lack of activity on `java` files during April 2008. Given that in this visualization the activity is computed with respect to files *addition or modifications*, keeping in mind that the amount of `java` files decreased during this month, it is clear that *jacopo* performed some cleanup, removing useless classes. During February 2008 there is an high activity on `java` files, therefore we can not say with precision if the reason why the number of these files decreased is due to heavy refactoring (deleting one file after moving all its

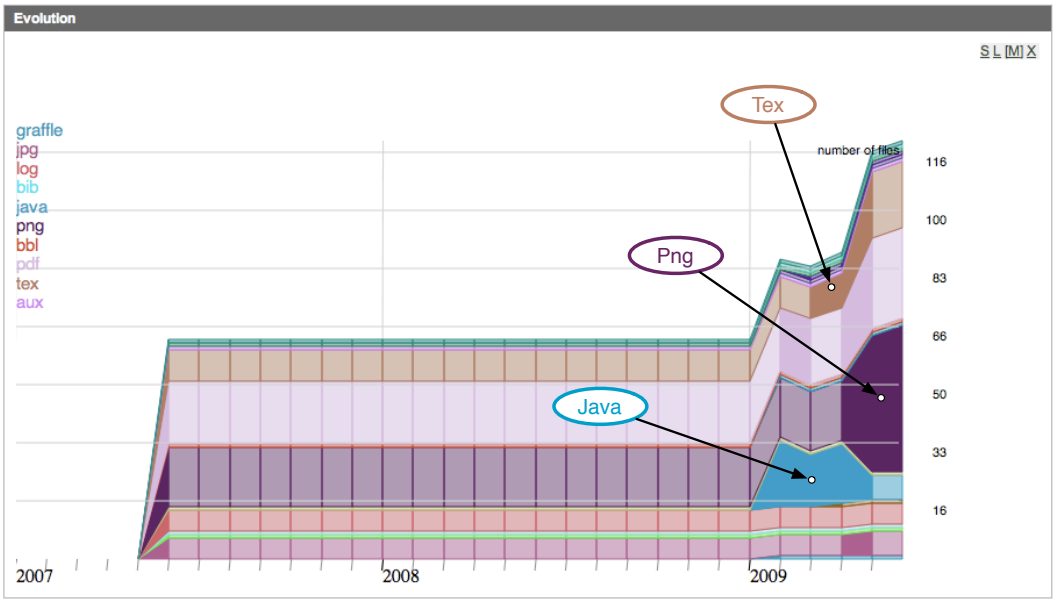

Figure 5.31: *Size by File* perspective for developer *jacopo*.

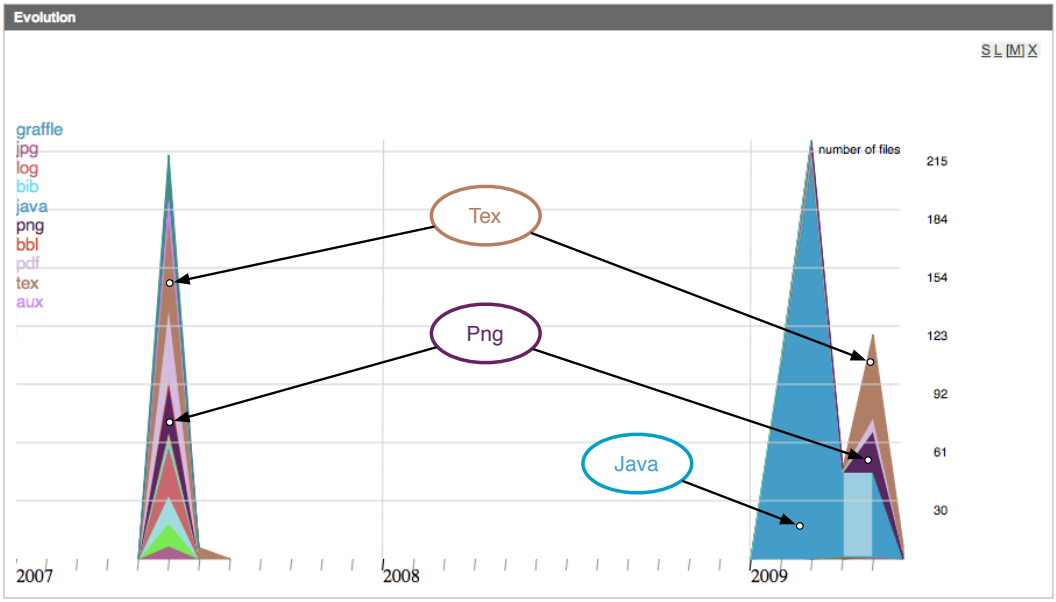

Figure 5.32: *Activity by File* perspective for developer *jacopo*.

content to other files) or simple deletions.

From the *Activity by File* perspective it is also possible to notice that the effort on `png` and `tex` files is approximatively the same, in both the first and second period.

If we want to understand the topics described by the two theses or the concepts modeled by SVNMole we can analyze *jacopo*'s *Vocabolary Tag-Cloud* and then inspect it through time with the *Concept Activity* perspective. Figure 5.33

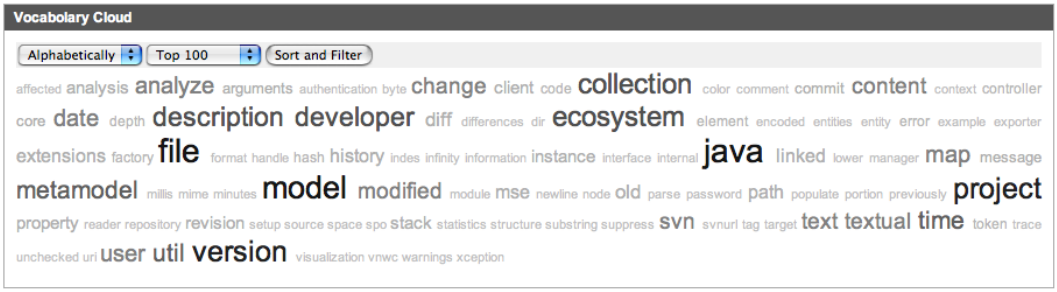

Figure 5.33: *Vocabolary Tag-Cloud* for *jacopo*.

depicts the vocabulary of *jacopo*. From the visualization we can assume that, probably, the developer is *modeling* entities according to a *metamodel*. Some of these entities, could be *Java* files. Important concepts are *project, ecosystem, developer* that are likely to be modeled according to *time, version* and *date*. To enrich this information with the notion of time and vocabulary evolution we can analyze the *Concepts Activity* perspective in Figure 5.34.

The first thing to notice is the big difference in the size of the vocabulary between the first and the second period of development. The first period is characterized by a small vocabulary without any clear outlier in terms of importance. The only exception is *file*, a concept that keeps being important after 2009, providing evidences of the fact that all the approaches described or implemented by *jacopo* are file based or exploit the content of files. After January 2009 the vocabulary of this developer has been enriched by several, important terms. Some of them describes entities, such as *developer, ecosystem* and *project*. Others concepts are related to history and software evolution: *time* and *version*.

Analysis of contributor romain

We already know that *romain* has almost continuous activity between January 2007 and January 2009. We also know that he is among the top contributors

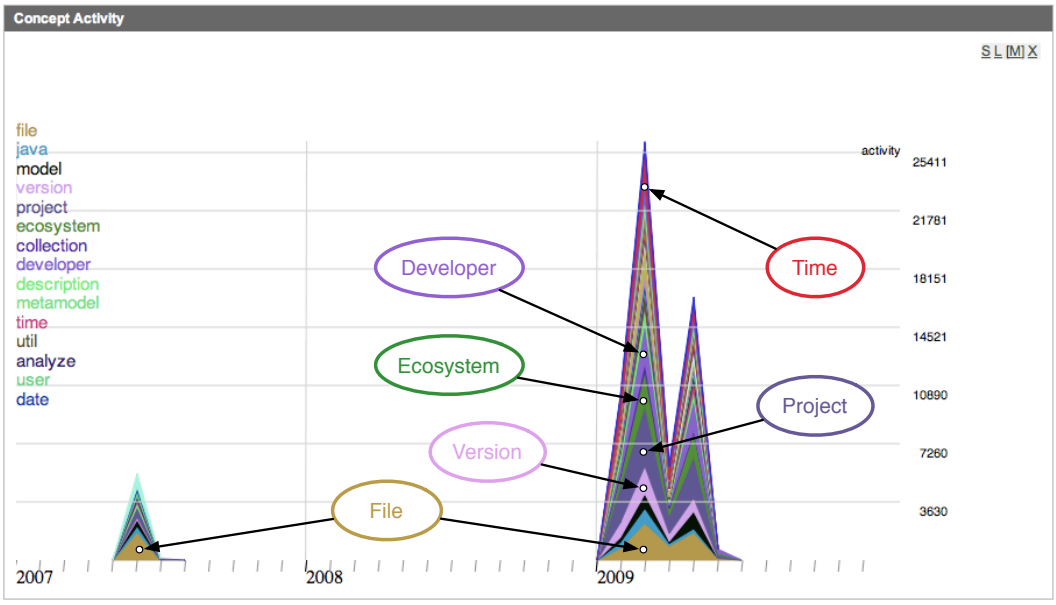

Figure 5.34: *Concepts Activity* perspective of developer *jacopo*.

in terms of `tex` documentation. His *File Extensions Tag-Cloud* clearly emphasizes it. `tex`, `pdf` and `graffle` files are all related to documentation. We know that he contributed documentation, but we still do not know to which projects and when. The *Developer Activity* perspective answer these questions.

Figure 5.35: *File Extensions Tag-Cloud* for developer *romain*.

Figure 5.36 emphasizes that *romain* contributed several revisions to many different projects. The activity on all of them last no more than two or three months each, with the exception of project *rr-phd*, his PhD thesis. It is interesting to notice that the activity on *rr-phd* started in April 2008 and is clearly visible until October. In the following two months there is a residual activity that is barely visible. After two months of no activity (January and February 2009), we notice very limited activity on this project in the third and fourth month of the year. The activity in these months, much likely, is due to limited maintenance. Figure 5.37 is the *Size by File* perspective visualizing the amount of files added to the ecosystem by this author.

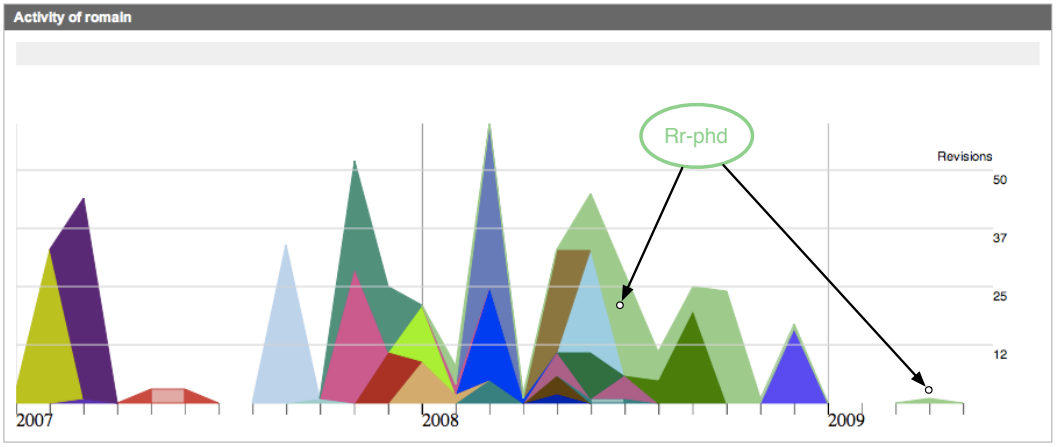

Figure 5.36: *Developer Activity* perspective of developer *jacopo*.

romain contributed only documentation, it is not surprising that the amount of `tex, pdf, png` and `graffle` files grew constantly since 2007. After September 2008 this developer stopped adding files to his projects. If we compare this fact with his *activity* on these files (Figure 5.38), we discover that he kept modifying them until April 2009. It is likely that he created the scaffolding of his thesis during the summer of 2008, creating and adding files, then he kept modifying and changing them for months.

If we want to understand the content of these changes we can have a look at his *Vocabulary Tag-Cloud*. The main concepts are *change* and *code*. According to the other important terms it is likely that he was describing, in his documentation, the *evolution* through *time* of *code*. It looks plausible that he was writing about *model*ing the *history* of *development*.

Analyzing his *Concept Activity* perspective it is clear that *software* and *changes* were important terms since the beginning of its development. It is interesting to notice that as soon as he started his PhD thesis the terms *history, development, time* and *code* appeared in his vocabulary or increased their importance.

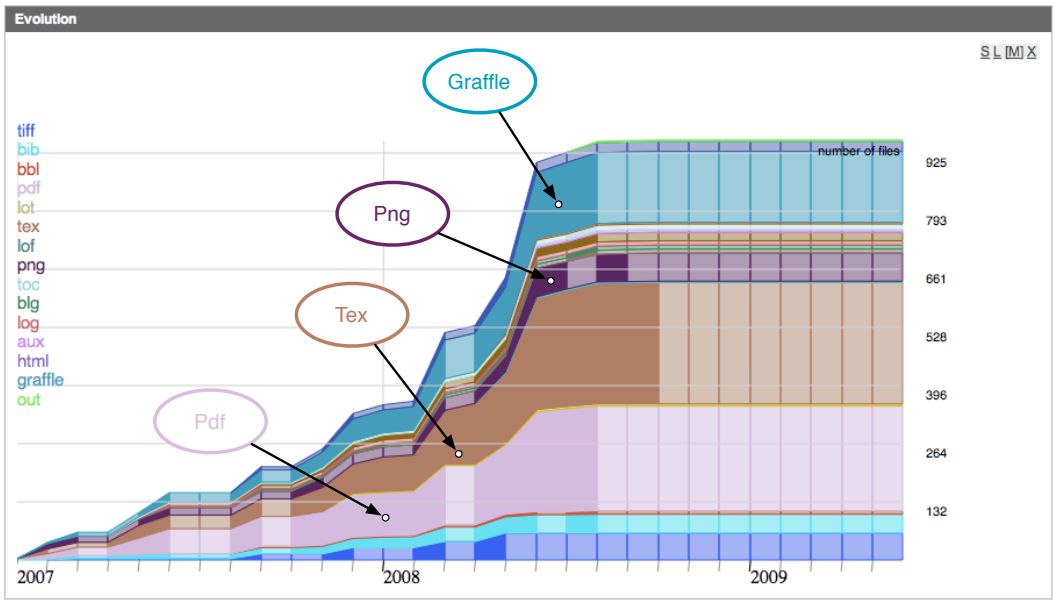

Figure 5.37: *Size by File* perspective for developer *romain*.

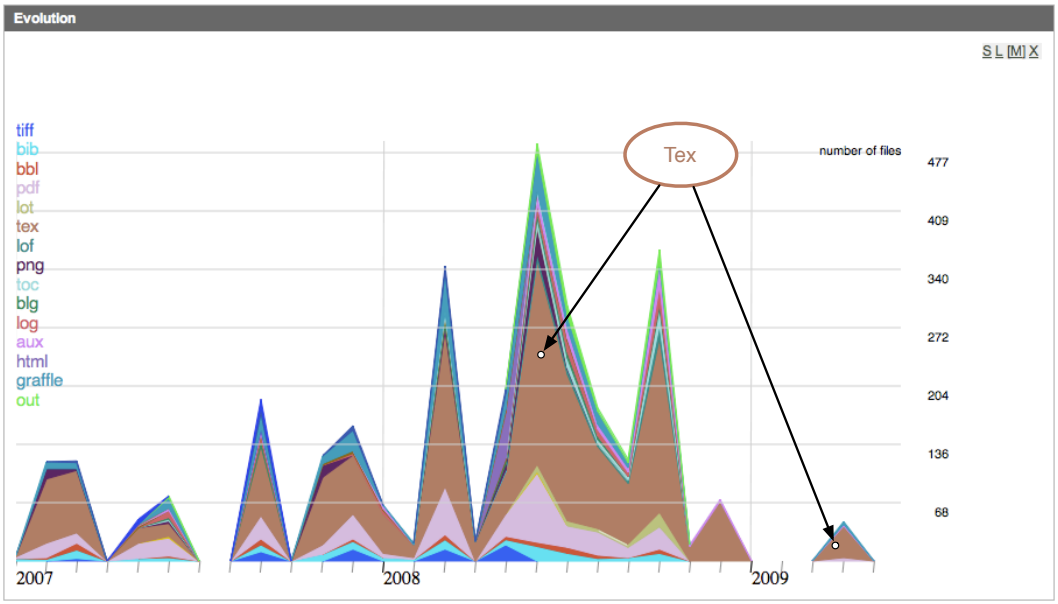

Figure 5.38: *Activity by File* perspective for developer *jacopo*.

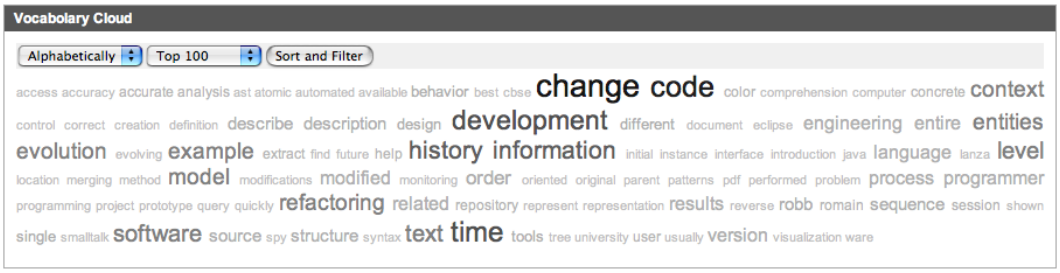

Figure 5.39: *Vocabulary Tag-Cloud* of developer *romain*.

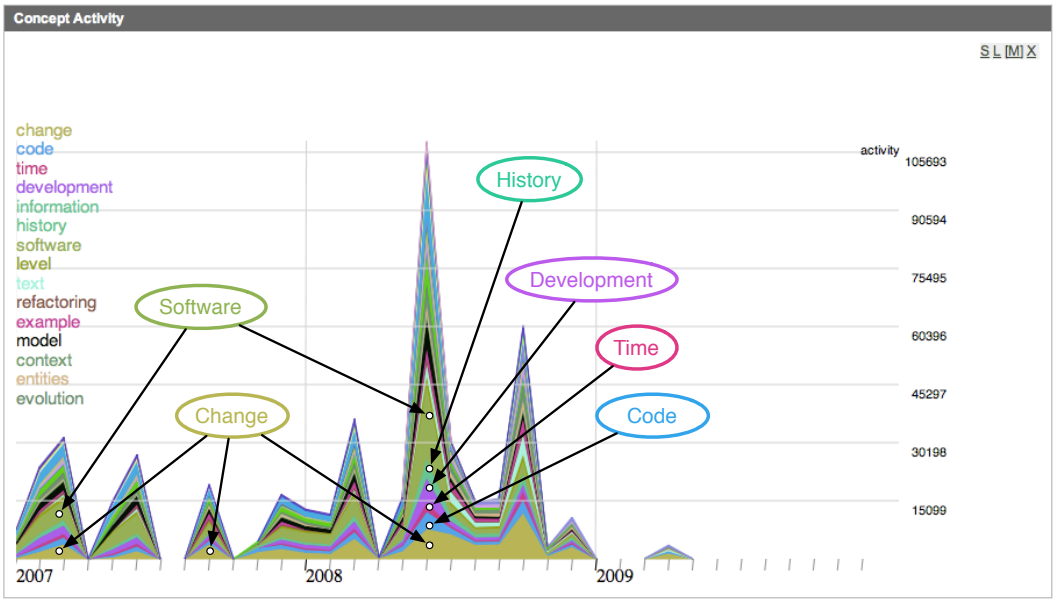

Figure 5.40: *Concept Activity* perspective for developer *romain*.

Chapter 6

Conclusions

In this thesis we present techniques and tools for analyzing and characterizing developers and their activity in the context of SVN ecosystems. Our contributions can be summarized as folows:

- We implemented SVNMole, our tool to gather SVN data, a Java library that can be easily extended or used by clients that need to cope with ecosystems. This open-source and freeware product analyzes logs and revisions of a set of SVN repositories and models them within the same context: the ecosystem. The model is built incrementally and contains information about projects, developers, commits and file changes. One of the main contributions of our approach was to analyze the added value of each file change, comparing the current version of a resource with its previous one. The difference between two versions is considered to be the result of the effort of a developer, that will be enriched by this information. On top of the added values of each file change we characterized the vocabulary of developers and projects, to better understand their evolution. Ecosystems, Projects, Developers, Commits and Changes are first class entities that encapsulate useful information that can be further exploited by clients.

- We presented a simple client of SVNMole, the MSE Extractor. This small application uses the model built on top of an ecosystem and exports it to the MSE file format. The resulting file can then be imported or used by every application that is familiar with FAMIX, a language independent meta-model.

- We extended the SPO with new perspectives that present information about the developers in an ecosystem. We then presented the new visualizations and functionalities that we developed to enhance it's developer analysis

capacities. Tag-clouds, scatterplots and stacked-graphs are used to depict the details and evolution of each developer in the ecosystems.

Analyzing the evolution of developers in a software system can be extremely useful to project managers, developers and researchers. The history and evolution of contributors that evolve in the same context is worth much more than the analysis of the the evolution of the single one. Exactly like scientists study black holes analyzing the effect they have on nearby stars and systems, we state that analyzing developers in a broader context provides useful insights and evidences of patterns and correlations that will be extremely difficult to notice otherwise. With our approach it is possible to automatically identify domain experts through their vocabulary, it is possible to identify contributors that developed or maintained a given software system. Each developer is described by his contributions, by his efforts and activity in the ecosystem. By analyzing their details it is possible to know much more than a simple list of commits or the name of the projects they worked on, it is possible to understand their history, depict the concepts they were working on and characterize their behavior over time.

6.1 Future Work

6.1.1 Data Extraction

We profiled, tested and tuned up our library as much as possible, but there are still some performance and behavioral issues that should be taken into consideration in the future releases.

Parallel Model Extraction

SVNMole, so far, is intended to be executed on a single machine. The analysis of the given ecosystem is entirely performed locally up to the point in which the results are ready to be used by clients. This sequential and monolithic computation could be split into several jobs executed in parallel on several machines. In our model, projects are isolated entities, once their content has been modeled it won't be modified any more. While extracting data from the other projects there is no need to update the model of previous projects. This property makes each projects a single unit of computation that can be assigned to different machines on a distributed network. The analysis of the ecosystem could then be divided into several, parallel, analyses of complementary subsets of projects belonging to that ecosystem.

If projects are single unit of computation, developers are not. Each developer has a set of collaborators that is incrementally update through the analysis of all the projects. Moreover, the same issue applies to the data that we decided to cache for each developer: its projects and contributions. In a parallel environment, in fact, each subset of projects will constitute a sort of partial-ecosystem that ignores information extracted by external projects.

One possible way to cope with these issues would be to use a single machine acting as a coordinator for all the machines involved in the parallel analysis of the ecosystem. The coordinator will then be in charge of collecting the information about all the analyzed projects and developers. The same developer will, potentially, appear in different partial-ecosystems. The coordinator should then merge all the aliases of the same developer inso a single, unique, one. References from projects to developers should also be updated with respect to the unique, merged, developers.

At the end of this procedure the coordinator will contain the complete, homogeneous model of the ecosystem. Figure 6.1 depicts an hypothetical scenario.

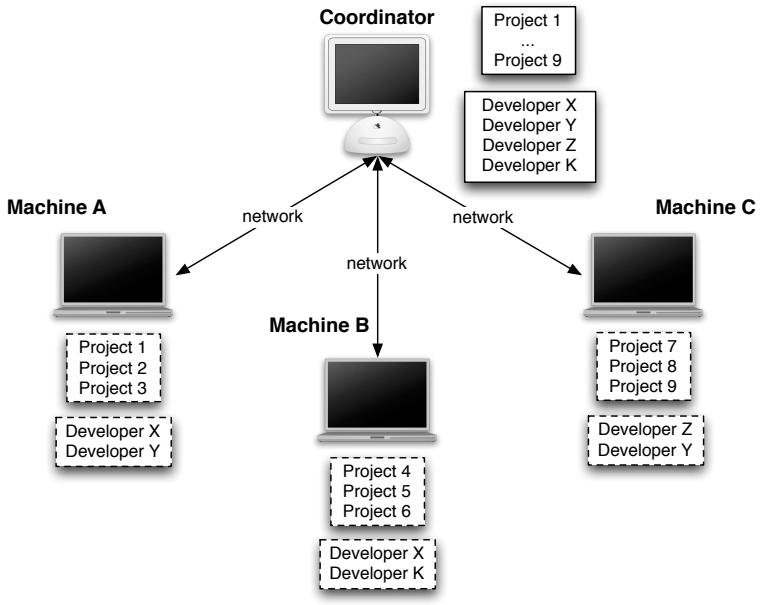

Figure 6.1: A possible scenario composed of a coordinator and 3 remote machines. The coordinator assigns 3 projects to each machine and is responsible for the refinement of the intermediate partial-ecosystems

A coordinator, on top of the (textual) description of the ecosystem that needs to be analyzed, splits the contained projects into subsets. Each subset of projects gets assigned to a different machine (A, B, or C) on the distributed network that starts to analyze them. To minimize the overhead, each machine should have a local copy of the repositories it needs to analyze. Nothing prevents the coordinator to be (also) one of these machines.

The previous is just one of the several solutions that could be implemented to increase the scalability and time performance of the model extractor. Creating the model in a parallel fashion will sensibly decrease the time required by that operation. Each machine will analyze the assigned projects and populate its partial-ecosystem that will be sent back to the coordinator (exploiting RMI or specific protocols) that will take care of the refinement and merging of the (three, in our case) intermediate data. The total time needed by the whole process is the longest of the times needed by all the partial-ecosystems to be modeled plus the time needed by the refinement of the intermediate information in the coordinator and the communication overhead.

Content Aware Modeling

Right now, each versioned file that is tokenized and analyzed with respect to its difference with respect to its previous version is treated in the same way. Let it be Java source code, HTML content or documentation, it does not matter. This has been a simplistic approach that does not consider the goal and semantic of the extracted data. We believe that it would be worth trying to apply different tokenizing techniques to different content. When we develop object-oriented concepts, for example, the names of the classes, attributes and methods describe what that entity (the class) represents, while the bodies of its methods describe what it does (what an entity does versus how is it implemented). Inside a document file, titles, chapter and section names represent important and clear concepts, while the content of its sections are, usually, more discursive.

Applying different techniques to different files implies handling the semantic information described by its content.

Assigning different weight to words extracted from semantically different contexts could provide a better description of the concepts expressed and used by developers.

6.1.2 Data Visualization

The Small Project Observatory web application has been extended and improved, but there still are many features that could be integrated in the existing application.

Developer Vocabulary Clustering

Using the added value provided by each file that gets modified and committed by developers, it is possible to infer the vocabulary of them. This information is depicted through tag-clouds and stacked-graphs. This the firsts it is possible to understand the main topics in the vocabulary of a developer while in the second one we also add temporal information, to understand *when* a term has been important.

Another interesting visualization would be the result of clustering developers according to their vocabulary over time. That is: not only use the similarity in the terms they use, but also weighting the distances in time. It will be possible, then, to spot groups of authors that developed portions of their vocabulary together. These groups will, much likely, be persons working at the same project, or (and discovering that would be more interesting) exploiting the same technologies at the same time in different projects.

Events

The life of an ecosystem is characterized by the lifecycle of its projects. This lifecycles strongly depend on external factors such as deadlines, releases, holidays, etc.. Having the possibility to add this events and depict them in each visualization in which there is a timeline will help to explain why the visualization looks like it is. Adding events should be a feature that gives the user the possibility to insert a single date or a time span and a description. This informations should then be injected in the existing visualizations as vertical bars marking a specific event or its start and end. Having the possibility to see these events will help to provide a better context to the visualized data (that is likely to be subjects to these events).

Adding this feature will make it easier to answer questions such as: how developers behave when a deadline is approaching? Why the effort is so low in this time span, was that holiday or the developer was attending some convention? What happens to the productivity after a seminar? Does the vocabulary of the developers include new terms that can be associated with that seminar? And so

on.

Animating Developer Activity

The evolution of the vocabulary of developers is provided by a stacked-graph in the *Concept Activity* perspective. The concept of time is explicitly expressed by the horizontal axis. With a single picture it is possible to describe the complete evolution of the vocabulary. We are wondering if it would be helpful to provide the same information through an animation. What we imagine is a tag cloud that starts empty and grows and changes, over time, according to the vocabulary of the developer (or project or ecosystem) in that specific moment. The approach would be similar to the one used by *code_sawrm* (Figure 2.4). We have still doubts about the additional information, if any, that this animation will provide.

Inter-Ecosystem Developer Analysis

SVNMole creates models describing ecosystems. The Small Project Observatory visualizes ecosystems and provides a broad set of perspective to infer as much information as possible out of these models. The main limitation of this approach is that ecosystems are closed systems and it is not possible to directly compare them or link their data. Let's say that the same developer contributed to several ecosystem during the same time span, on different projects. Even if we added the feature of setting aliases for developers, it is still not possible to set inter-ecosystem connections. Even if it was a single developer that contributed to different projects in different ecosystems, this information will be lost. Moreover, there are no perspective in which the visualized or analyzed entities are ecosystems, making it difficult to compare styles, content and patterns.

6.2 Closing Words

This entire work is about modeling and describing developers. More generally, contributors. Even though we believe that the contributions made with this thesis may be helpful, no matter how good a model is, it is still a finite representation of an infinite world. We did our best to represent and visualize meaningful and important details, but to understand the world, to *know* the real essence of a contributor, we need to reason in terms of all of his details. When I was visualizing some code, a wise man told me that *the truth is in the source code* even if my visualizations were capturing some of its essence. The same thing holds here. Our approach could be very useful to identify patterns and discover interesting

facts, but these observations must be validated in the real world, made by real people.

Bibliography

[DLL06] Marco D'Ambros, Michele Lanza, and Mircea Lungu. The evolution radar: Integrating fine-grained and coarse-grained logical coupling information. In *Proceedings of MSR 2006 (3rd International Workshop on Mining Software Repositories)*, pages 26 – 32, 2006.

[HL09] Lile Hattori and Michele Lanza. An environment for synchronous software development. In *Proceedings of ICSE 2009 (31st ACM/IEEE International Conference on Software Engineering - New Ideas and Emerging Results Track)*, pages 223–226. IEEE CS Press, 2009.

[LLG07] Mircea Lungu, Michele Lanza, and Tudor Gîrba. The small project observatory. European Smalltalk User Group 2007 Technology Innovation Awards, August 2007. It received the 1st prize.

[RL08] Romain Robbes and Michele Lanza. Spyware: a change-aware development toolset. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 847–850, New York, NY, USA, 2008. ACM.

[SKGD06] Mauricio Seeberger, Adrian Kuhn, Tudor Girba, and Stéphane Ducasse. Chronia: Visualizing how developers change software systems. In *CSMR '06: Proceedings of the Conference on Software Maintenance and Reengineering*, pages 345–348, Washington, DC, USA, 2006. IEEE Computer Society.

[WL07] Richard Wettel and Michele Lanza. Visualizing software systems as cities. In *In Proc. of the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 92–99. Society Press, 2007.