Telling Evolutionary Stories with Complicity

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
Software Design

presented by
Sylvie Neu

under the supervision of
Prof. Michele Lanza and Lile Hattori

June 2011

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.

Sylvie Neu
Lugano, 29 June 2011

Nothing great in this world has ever been accomplished without passion.

Christian Friedrich Hebbel (1813-1863)

Abstract

Software systems change when for example new requirements arise, their environment changes or bugs are fixed. During this process they grow in size and complexity making their maintenance a time-consuming and thus costly process. Software evolution analysis, focuses on the changing process of software systems. Its main goal is to analyze this process to better understand how a system evolves and to eventually reveal aspects that have an impact on the maintenance tasks.

Traditionally, the target of software evolution research has been single software systems. A number of approaches were proposed to understand the evolution of a software system, exploiting various sources of information, *e.g.,* versioning system data, bug databases, and mailing lists. However, during the last years researchers observed that software systems are often not developed in isolation, but within a larger context, *e.g.,* company, or open source community. We call this context ecosystem. Analyzing software evolution at ecosystem level allows a better understanding of the evolution phenomenon, as the entire development context can be studied. Nonetheless, software ecosystem analysis is challenging because of the sheer amount of data to be processed and understood.

We present Complicity, a web-based application that supports software ecosystem analysis by means of interactive visualizations. Complicity breaks down the data by offering two abstraction levels: ecosystem and project or contributor. To support a thorough exploration and analysis of ecosystem data, the tool provides a number of fixed views and the possibility of creating new views with given software metrics. We illustrate in a case study how Complicity can help to understand the GNOME ecosystem. In a bottom-up approach we start from a single project and contributor and show their impact on the ecosystem. In a top-down approach, we start our analysis at ecosystem level before we move to a lower level of abstraction for some selected projects.

Acknowledgements

I would like to express my gratitude and appreciation to the following people:

First of all, I would like to thank my supervisor, Michele Lanza. I am grateful for the opportunity to work on this project within the REVEAL research group. Your suggestions, feedback and ideas pushed and promoted me throughout the entire master thesis.

A special thanks to my co-advisor, Lile Hattori, for her continuous support and reviews. You were always available for discussions, for helping me overcome my blockades, or for just having an ice cream together. I will definitely miss our ice cream degustation tour through Lugano.

To all REVEAL group members from the University of Lugano – thank you for giving me the feeling of being part of the group, *e.g.,* having little and longer chats, sharing your office with me when possible, and providing me any kind of support. You all made it a special experience I do not want to miss. Many thanks to Lile Hattori, Marco D'Ambros and Michele Lanza, for your guidance and support in writing my first paper.

A special thanks to my mother Annette Klein for her regularly talks and encouragements but especially for her love, and for kicking me when required. Furthermore, I want to thank my sister Tessy Neu and her boyfriend Laurent Breyer for always believing in me. I also want to thank my grandmother Catherine Klein for just being you.

To everybody I met in Lugano and who made my time here unforgettable in a way or another. I thank Verena Dollberg for our many fights in Unihockey and Vera Lill for our few sessions together in Capoeira and Badminton.

Contents

Figures

Tables

Part I

Prologue

Chapter 1

Introduction

Software systems change, and during this process they grow in size and complexity, and incrementally move away from their initial design. This phenomenon, known as *software evolution* [Leh80], makes it difficult to maintain a software system, which claims a share estimated up to 90% of total software costs [LS81, Erl00], of which 60% is spent in understanding the system [Cor89].

The high cost of software maintenance results from many factors: Documentation is often not updated, or not existing [KC98]; because of the continuous turnover of developers, changes to a software system are often performed by developers with a limited knowledge of the system. Reverse engineering deals with these problems and its aim is to ease software maintenance. Chikofsky and Cross [CC90] defined *reverse engineering* as:

"*The process of analyzing a subject system to (1) identify the system's components and their interrelationships, and to (2) create representations of the system in another form or at a higher level of abstraction.*"

Most reverse engineering research is mainly concerned with satisfying these goals using different abstraction levels (*e.g.,* code level [ESS92], and design level [LDGP05, WL07]). The problem with following the above definition is that it takes into account a single software system focusing on either the project or its contributors. However, software systems are rarely developed in isolation, but in so called ecosystems. Lungu [Lun09] defines a *software ecosystem* as:

"*A collection of software projects, which are developed and co-evolve together in the same environment.*"

In this context, the environment is a company, a research group, or an open source community. The data about the changes executed on projects from a same ecosystem are often kept together in one same location, called super-repository. Lungu [LLGR10] defines a *super-repository* as:

"*A collection of version control repositories of the projects of an ecosystem.*"

To analyze software ecosystems at any abstraction level, techniques are required to cope with the huge amount of data available. Two analysis techniques have been effectively used to convey the results to the end user: metrics [CK94, LFRGBH06, LDGP05], and visualization [ESS92, GKSD05, WL07].

The advantage of software visualization over pure metrics is that it uses the brain's ability of remembering images [Die07] and extracting patterns and anomalies from the data that are unlikely when data or numbers are presented in tables or text [PSB92]. Diehl [Die07] defines *software visualization* as:

"*The visualization of artifacts related to software and its development process [...] including for example program code, requirements and design documentation, changes to source code, and bug reports.*"

In order to support the analysts in their tasks to better understand the evolution of software systems within their ecosystem a framework can be beneficial. We present Complicity, a web-based interactive application that visualizes –at different abstraction levels– the data extracted from the Git web interface of super-repositories. Complicity makes use of metrics and visualization to analyze software ecosystems in a top-down or bottom-up approach. The user can start at ecosystem level and move down to the entity level or vice versa, start analyzing a project or a contributor, and go to the ecosystem level to give a higher level context for the individual analysis. Using the GNOME super-repository as a case study, we demonstrate the use of Complicity to better understand the ecosystem and its entities. For this purpose, we show how an individual project affects the events at ecosystem level, and follow a contributor's involvement in different projects.

1.1 Contributions

The contributions of this thesis can be summarized as follows:

- Provide a super-repository independent meta model for software ecosystems.
- Make this work available to others as an interactive web-based application where the users can explore and follow up the software systems they are interested in.
- Provide a catalogue of visualizations illustrating different views of software systems and contributors at entity and ecosystem level, which are beneficial for understanding the evolution of software systems on their own and within their ecosystem.
- Supply a methodology of how to use the different views to understand the evolution of the software systems and their ecosystem.

1.2 Structure of the Document

This document is structured into three main parts as shortly described in the following.

Part I: *Prologue* is the introductory section that gives an overview of the subject research field, introducing the main terminology, the problem at hand and some examples depicted from the state of the art.

- **Chapter 1** (p.3) introduces the research field and the main terminology, which will be used in the remaining chapters. It describes the goals we want to reach and outlines the document structure.
- **Chapter 2** (p.7) describes the related work and how it distinguishes from or leads us to our work.

Part II: *Understanding Software Ecosystems* is the central part of the thesis. It provides the main contributions of this work, describing the process and tool support for software evolution analysis at ecosystem level.

- **Chapter 3** (p.13) introduces Complicity, a web-based and interactive tool that we developed in order to support the user in his task to analyze and understand software projects and its contributors within a software ecosystem. We explain the interface with its available features and most importantly the meta model behind it.
- **Chapter 4** (p.29) describes our main contributions to this research field, a catalogue of different views, which have been integrated into Complicity, and which can be interactively explored. We explain how they are constructed, what data they provide, for whom they are useful, and how they can be interpreted.
- **Chapter 5** (p.55) we evaluate the views implemented within Complicity by providing scenarios in how they can support the user in its analysis and learning tasks.

Part III: *Epilogue* is the closing part of the work, in which we take a step back to review and conclude our work.

- **Chapter 6** (p.79) reviews and concludes the work. We discuss what has been done so far and evaluate our work by introducing what could be done in the future.

Chapter 2

State of the Art

Software visualization is mainly concerned with three different aspects of software systems [Die07]:

1. *Behavior*, which refers to the execution of the program, i.e. how the program state or its output changes when executing instructions;
2. *Structure*, which is about the static parts and relations of the system. It can be computed without running the program;
3. *Evolution*, which involves the changes that have been done on a software system over time whereas this information is generally extracted from other sources than the code.

Visual analysis of the evolution of software ecosystems is a rather underexplored area of research. Thus, we place our work in a broader context and relate it to (1) software evolution visualization, and (2) software ecosystems analysis.

2.1 Software Evolution Visualization

Visualization is "the process of transforming information into a visual form, enabling users to observe the information" [Ger94]. Software evolution visualization specifically deals with the abstract data available from different sources that keep track of the changing process of software systems. It enables the analysts to visually perceive features, which are hidden in the data but are necessary for the data exploration and analysis.

Much of the research work in software evolution visualization has been targeting single software systems. They either focus their analysis on the subject system itself at different abstraction levels (see Section 2.1.1), or on social structures and how they influence the evolution process (see Section 2.1.2).

2.1.1 Single Software Systems

According to Lungu [Lun09] static visualization tools work at three different abstraction levels:

- *Code-level*, which refers solely to the text.
- *Design-level*, which uses classes or files.
- *Architecture-level*, which considers modules and their relationships.

We present some examples that make use of software visualization to analyze the evolution of a single software system at code or design level.

One of the first tools to visualize software evolution was Seesoft [ESS92]. It visualizes the change history of files by mapping a single line of code to a pixel line, where the color of the pixel line is an indication of interest: red, most recently changed versus blue, least recently changed. Seesoft enables a comparison of multiple files with up to 50 KLOC at the same time.

The Evolution Matrix [Lan01] displays the evolution of a software system at class and system level. Within this matrix a class is represented as a two dimensional box with number of methods as width and number of instance variables as height. Every row of the matrix represents the evolution of a different class of the subject system and every column a different version of the system. This matrix makes it easy to extract information about the size of a system, the addition, removal and changes of classes, and thus to identify different evolution stages of a system.

The Evolution Radar [DLL06] visualizes logically coupled files and modules. The subject module to be analyzed is placed in the middle of a pie chart, all remaining modules are placed in sectors around the center. A file is represented by a colored disk placed according to its logical coupling: the more coupled two files are, the closer they are placed to the center. The higher the value of the logical coupling the more the color of the file changes from blue to red. The goal is to identify entities that have often changed together in the past and are thus likely to change together in the future. Furthermore, misplaced entities can be detected if files are often changed together but placed in different modules.

By means of a three dimensional approach, Wettel *et al.* [WL07] uses the city metaphor to visualize the current state of an object-oriented software system and its evolution over different versions. Mapping software metrics like number of methods (NOM) on the height and number of attributes (NOA) on the base size of a building, they could depict different types of classes, *e.g.,* brain classes, god classes, brain & god classes, and data classes.

2.1.2 Social Network Analysis

Software does not evolve without external impact. Humans are those who make a software system change. As a consequence, the analysis of social aspects (*e.g.,* activity, communication structure, and knowledge flow) becomes a valuable source to better understand the evolution of software systems or of those who develop them. We illustrate on some examples what information can be extracted from different data sources using different visualization techniques.

The Ownership Map [GKSD05] identifies the owner of every single file within a software system. Every file is represented with a pixel line, a disc defines a file change, and the color of the line and the disc defines the owner and the committer, respectively. The owner of a file is the developer who made most changes to a file in terms of number of lines.

Ogawa *et al.* use discrete sankey diagram to show the flow of people between clusters, which are generated using the Markov-Cluster (MCL) algorithm[1]. Clusters represent people that are highly connected based on a high amount of email exchange among them: large clusters represent core conversations and small clusters represent side conversations [OlMB+07]. A single person is represented by a disk placed on a horizontal line grouped together by their cluster. Every horizontal line represents the email list conversation state at one time step. Time flows downwards in monthly stages and edges are drawn between two stages if people continue to participate in the mailing list. The width of an edge is proportional to the number of people staying within the two clusters. The exiting rate can be

[1] http://micans.org/mcl/

depicted from the graph by subtracting the number of people of the next stage from the current stage. Applying this approach on large data sets increases the number of clusters and edge crossings, which might decrease the readability of the visualization.

In Maispion [SMG09], the authors analyze the activity within the mailing lists and versioning systems of a single software system in order to reveal communication behavior within a project. They answer questions such as "Is there a main driver?" and "When are the developers most active?". The tool has been evaluated on three different well-known open-source projects: Moose –project developed at the University of Bern–, Drupal and Phyton. With the help of Maispion, they found out that most developers are likely to communicate only with a small number of other developers. This phenomenon is called *clique* and is an "indication of redundancy of communication links around a developer" [LFRGBH06].

Oezbek *et al.* [OPT10] check whether the onion communication model is applicable to open-source software systems using mailing lists as data source. They find out that the core developers (active since more than eight months) are highly interactive and tightly connected to each other. The co-developers are mostly core-oriented and only few connections exist between them. The periphery is either connected to the core or not connected at all. They name the model resulting from their work earth-moon-stars with earth representing the core, moon the co-developers and the periphery as stars.

2.2 Software Ecosystems Analysis

The second main research field we are considering in our work is the analysis of software ecosystems. To the best of our knowledge, little has been devoted to this topic. In the following we present some works that are directly or indirectly related to software ecosystem but which do not necessarily use visualization techniques.

FLOSSMole [CHC05] is a project that aims at mining free, libre, and open-source software (FLOSS) super-repositories (*e.g.,* sourceforge, github, and google code) and making general information about these projects, *e.g.,* project name, description and developers available for researchers and anyone who is interested in it. The data from the version control systems or any other data source is not mined.

López-Fernández *et al.* apply social network analysis techniques to community-driven libre software projects, such as Apache, GNOME and KDE. What the authors call large projects, we define as software ecosystems [LFRGBH06]. Their goal is to gain knowledge about the coordination structures prior to understand how large open-source projects can function without formalized organizational structures. Using different metrics (*e.g.,* weighted clustering coefficient, distance centrality, and betweenness centrality) they found out that committers within the GNOME and KDE are more tightly connected than the ones of the Apache ecosystem because of the GNOME's and KDE's projects technical proximity. They also discovered that these ecosystems have a good structure for knowledge flow and the absence of centers of power.

Ohloh[2] is an online directory of free and open source software (FOSS) projects and its developers. It retrieves data from version control repositories (*e.g.,* CVS, SVN, Git, *etc.*) and provides a minimal number of visualizations showing the evolution of language usage, number of commits, number of committers, and number of lines of code versus number of lines of comments versus number of empty lines. Ohloh focuses mainly on supplying the necessary information for its users to find the best OSS projects, which they are looking for and, which best fits with the tools they are currently using.

[2] See `http://www.ohloh.net/`

In a short summary they list the most crucial information about a single project: (1) project's main language, (2) project's team size, (3) how well the code is documented, (4) how active a project is in terms of commits, and (5) how well the code base is established based on its lifetime. The most significant difference between ohloh and other tools or approaches is the underlying information exchange platform with discussion forums; the possibility to mark, save and follow up only the projects and people you are actually interested in; rating and reviewing functionality for project and committers; and facilities to read and write journal entries about projects even from some messenger outside the ohloh platform.

Lungu focused his work on reverse engineering software ecosystems [Lun09]. He created the Small Project Observatory (SPO), a tool in which analysts, developers and managers can interactively analyze the evolution of software ecosystems. SPO mainly differentiates between two aspects, project and developer, and for which the ecosystem can play one of two different roles: *focus* –to better understand the ecosystem– or *context* –to understand a single entity of the ecosystem [LL10].

Seichter *et al.* introduced a new approach of knowledge management by developing a social network of software artifacts in which the knowledge is attached to a software artifact rather than an actor (contributor) [SDPH10]. This has the advantage that if a developer leaves an open-source software project, the knowledge remains within the project. This form of network allows new forms of interactions: actor to actor (collaboration), artifact to actor (ownership), actor to artifact (interest), and artifact to artifact (dependency).

Goeminne and Mens provide a framework to mine version control systems, mailing lists and bug tracking databases, to analyze and to visualize mainly the mailing and commit activity of open-source software ecosystems [GM10]. Contrary to us, they define an ecosystem as "the source code together with the user and developer communities surrounding the software".

2.3 Conclusion

We introduced some of the existing work devoted to either software evolution visualization and/or software ecosystem analysis. In these example multiple data sources (*e.g.,* source code, email lists, version control systems *etc.*) have been consulted. The extracted data has been applied in different approaches using distinct visualization techniques and metrics to support the comprehension of the evolution process.

In our work, we consider the data available in the Git web interface of super-repositories. We combine software visualization with software metrics, by focusing not only on a single project or contributor, or only on the ecosystem level, but enabling the analysts to switch between the ecosystem and project or contributor level, and between projects and contributors. We provide different views, in which we use some of the presented visualization techniques (*e.g.,* two dimensional boxes to encode different metrics and visualizes contributors' activity) to support the understanding of software ecosystems. We combined all this in a single tool called Complicity, which is introduced next.

Part II

Understanding Software Ecosystems

Chapter 3

Complicity

Our goal is to support those who are interested in analyzing and understanding the evolution of software ecosystems. However, as software ecosystems consists of many projects, the amount of data available is huge. Even though we focus on a single data source, the web interface of Git version control systems, we have to handle a large data collection. Therefore, we use visualization of software metrics, as visualizations can cope with large data sets and make it possible to extract patterns and anomalies from them that would not be possible otherwise. Furthermore, we break down the data into different high levels of abstraction, ecosystem and entity, so that only a limited amount of data is visible at a time.

In order to share our work with others, and allow them to interactively explore, analyze and understand the evolution of super-repositories, we developed a tool called Complicity [1]. The term complicity is defined as "the fact or condition of being an accomplice, especially in a criminal act"[2]. In this work, we are not interested in any criminal act but instead our aim is to identify collaboration, dependencies and relationships between projects, between contributors and between projects and contributors. For this reason, the tool is highly interactive allowing a smooth transition from one abstraction level to another and from project to contributor or vice versa.

Complicity is web-based to make it easier for the analysts to use it, as no installation is required and they have direct access to the newest version of the tool without the necessity of an update. In addition, it supports collaboration as everybody accesses the same data set. The advantage of this approach to us is that we have full control over who sees what and when. Furthermore, we are able to gather usage data.

In this chapter, we define different stakeholders (see Section 3.1 (p.14)), who might have interest in the analysis of software ecosystems. We present Complicity by giving a short overview of its user interface in Section 3.2 (p.15) before explaining the architecture of complicity and its backend in Section 3.3 (p.17). We introduce the underlying data model in Section 3.4 (p.19) and the metrics that could be extracted from the data in Section 3.5 (p.21). Based on these metrics we generate different views. In Section 3.6 (p.22) we present how the different views are constructed and what interaction they allow. In Section 3.7 (p.26) we present the super-repositories, from which the data has been extracted.

[1] See `http://complicity.inf.usi.ch`
[2] See `http://www.thefreedictionary.com/complicity`

3.1 Stakeholders

The evolution of software ecosystems is of interest to different stakeholders. In the following we introduce some stakeholders and describe their role, needs and responsibility, which eventually lead to their interest in software evolution.

Project Managers' (PM) main responsibility lays on developing project plans and managing project schedules, stakeholders, teams, as well as project risks, budget, and conflicts.

For a project manager to do these tasks, he needs information about the evolution of each single project as well as their current status. This enables him to keep the stakeholders informed about the advancements in the development. This also reveals possible delays based on which the schedules and project risks can be recalculated and updated. In case of delays and conflicts, a project manager must be able to react fast and allocate resources with adequate expertise to avoid further delays and minimize risks and conflicts in order to stay on budget.

In critical situations, project managers need the information about contributors' expertise and the relationships among them to adequately allocate resources to projects. Knowledge about relationships of contributors within single projects but also in software ecosystem is crucial in managing projects as they can give information about the information flow within a single project [LFRGBH06].

Software Quality Assessment Teams (QA) aim to ensure the quality of a single software system or a software ecosystem throughout the software development life cycle.

As members of the software quality assessment team also do change management, they are highly interested in the past changes of single projects but also how they evolve in an ecosystem. The information available about ecosystem can reveal critical projects of which they have to keep track more consciously than others and in consequence they can allocate resources accordingly. Information at software ecosystem level could also reveal relationships between projects which were not obvious or known beforehand.

Beside metrics, statistics and general facts about the projects, software quality teams might also be interested in information about the contributors, their expertise in the software ecosystem (on which project they have worked in the past for how long and executing what kind of tasks) and since when they are working on a specific project. This information is crucial for the quality analysis, as new contributors might be more critical in introducing new errors because of shortcomings in understanding the system at hand or because of missing expertise.

Contributors especially in the open source environment are always looking for interesting projects they can contribute to.

When searching for new projects, developers require information like which projects are available in a super-repository and which among them are still alive. Generally developers want to contribute to projects but also ecosystems or super-repositories which are relatively active compared to others because they have a higher relevance and are probably better supported.

Once they have selected a project, they need further information about the past of the projects (e.g. find the critical parts and the best entry point) but also about their past or current contributors. The latter can be beneficial for developers, especially newcomers, as they run into trouble or as questions come up, but also to get basic help about the best entry point.

Researchers and other Explorers are interested in exploring super-repositories, ecosystems and single projects to reveal new insights, facts, and patterns of which no one has thought before, and

which might be useful and crucial for the software evolution process or software engineering field.

Researchers and explorers do not always follow given paths or have a predefined goal but instead they usually try to find something new. In this context, their focus is either on the analysis of projects, contributors, relationships between projects, between contributors or between both, revealing some patterns or behaviors. They might follow a predefined methodology or just explore the data at random. For this purpose providing an interactive tool that allows exploration of undefined views is necessary.

To best support the need of these stakeholders, we developed Complicity. In the following we shortly describe how the information and facilities are organized in the user interface.

3.2 User Interface

The user interface is kept simple to avoid distracting the user from the analysis tasks, whereas the shapes in the visualizations are colored to attract their attention.

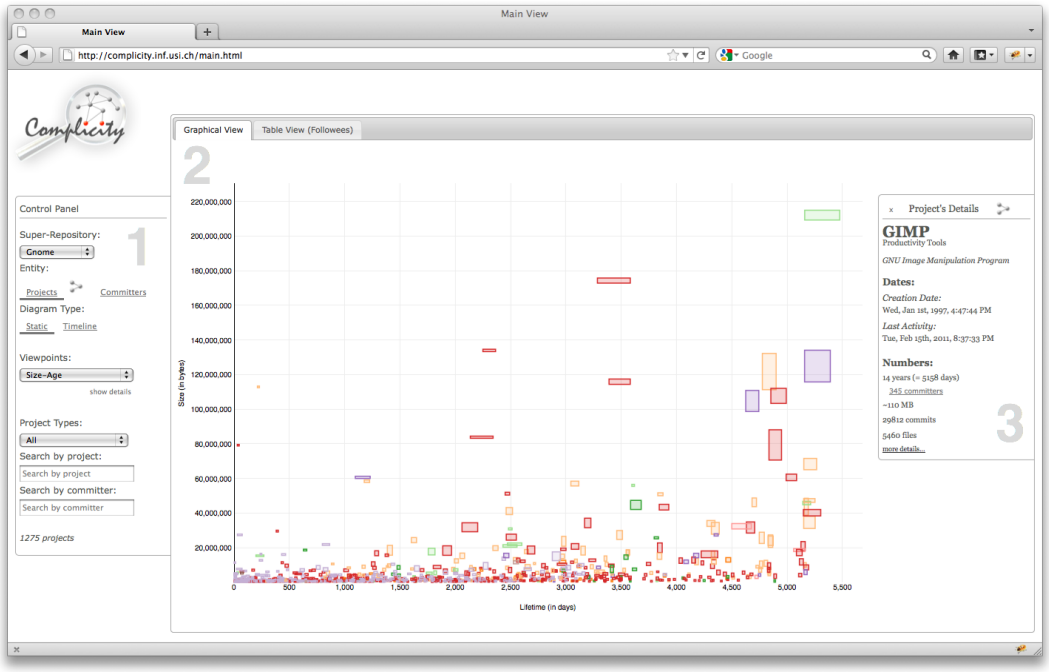

Figure 3.1. Main View of Complicity visualizing the GNOME projects at ecosystem level, and the details of the Gimp project

The **main page** (see Figure 3.1) is divided into three main parts:

1. The control panel on the left,
2. The main panel in the center of the page with the graphs of the ecosystem and the table view,
3. The quick view panel of a project's or contributor's details on the right, which appears by clicking on a shape in the graph.

Control Panel

It gives the user the possibility to (a) analyze different super-repositories, (b) choose between two entity types (project and committer) for the visualization at ecosystem level and filter out projects or committers that have not been marked, (c) navigate through predefined views or change their settings, and (d) search for projects or contributors of interest by project type, project name or contributor name.

Main Panel

Graphical View visualizes the available data for the selected super-repository and the selected entity type as scatter plots. Every box represents either a single project or contributor, depending on the entity type selected, and reflects up to five different metrics: position on x and y axis, width, height and color. By clicking on a shape a detail panel on the right appears.
Table View lists marked projects and contributors in form of a table. It gives the analysts the opportunity to compare these projects and contributors against each other based on some metrics, *e.g.,* number of commits, number of projects or contributors, *etc.* Projects and contributors can be marked from the detail panel.

Entity Detail Panel

It provides general information about the selected entity, which goes from name and date of the first and last commit, up to number of commits, number of contributors or projects, *etc.* From this panel the user has the possibility to get more details and further analyze the selected project or contributor at the detail page.

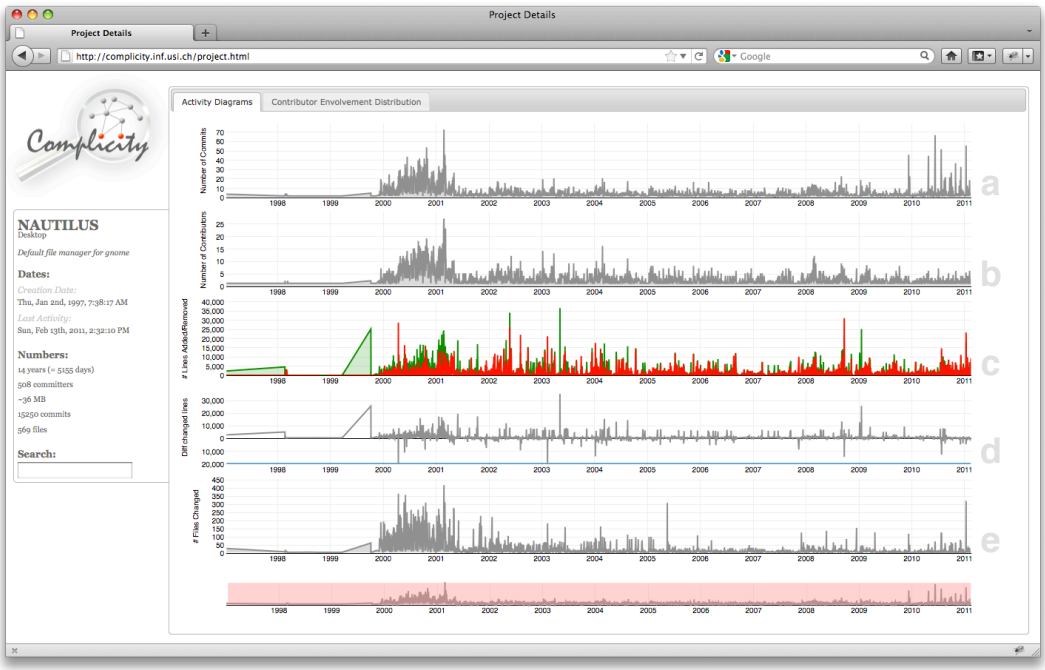

Figure 3.2. Project Detail Page (here: Activity per day over the entire lifetime of the *Nautilus* project (a. number of commits, b. number of contributors, c. number of lines added (green) vs. number of lines removed (red), d. difference between number of lines added and removed, e. number of files changed))

The **detail page** of a project or contributor (see Figure 3.2) has a similar layout as the main page, with the exception that the control panel on the left is replaced with an overview of the selected project's or contributor's details.

In the center of the page, the user can choose between two views: (a) activity diagrams and (b) projects/contributors involvement distribution. The Activity Diagrams View allows the user to compare the activity of the selected entity in terms of number of commits, number of contributors/projects (depending on the selected entity type), number of lines added versus number of lines removed, and number of files changed. In the Involvement Distribution View, the analyst gets an idea about how many contributors have been involved, when, and for how long, or –in case of a selected contributor– how many projects he worked on and what his speciality is.

3.3 Architecture

Complicity is a web-based tool that allows analysts to visually explore and understand different software ecosystems. In the following section we introduce the architecture of Complicity and its backend as illustrated in Figure 3.3.

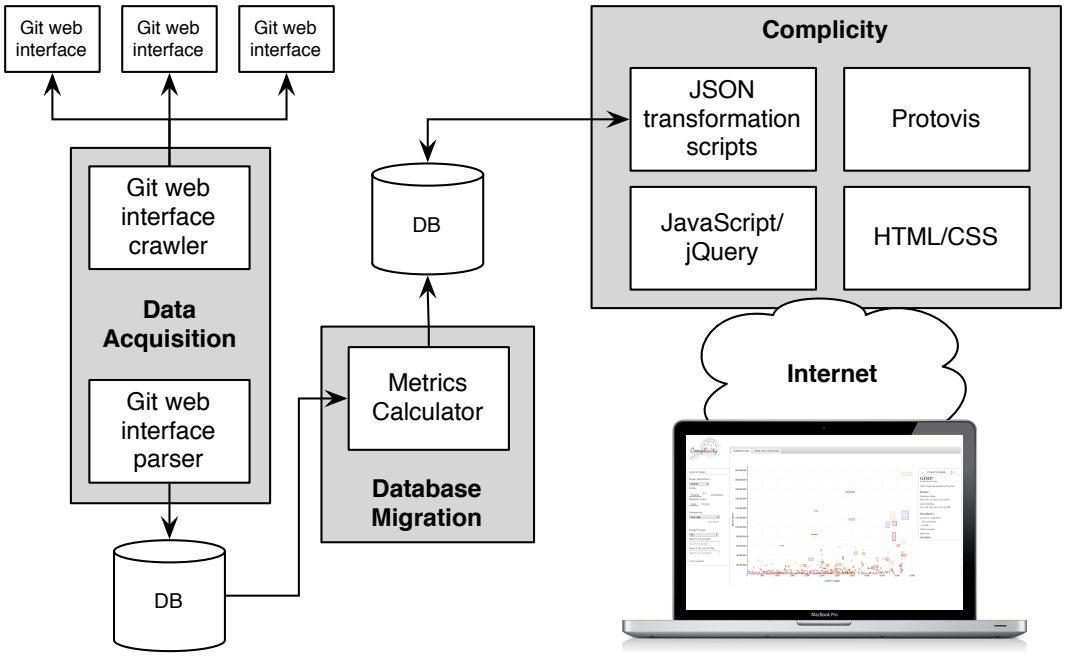

Figure 3.3. Architecture of Complicity and the backend

Data Acquisition

The data required for the visualization of the ecosystem is extracted from super-repositories that support the *Git web interface*. To fetch the relevant data from the Git web interface into our database, we developed a number of Java programs. The *Git web interface crawler* stores a copy of the web pages from the web interface of the Git super-repositories locally. To make sure that the web pages

are well-formed when stored we use HTMLCleaner[3]. It is an open source parser written in Java that cleans up web pages for example by reordering tags or closing missing tags.

The *Git web interface scraper* extracts the data from the stored web pages, and stores it in a database. For the data extraction we use the XML Path Language (XPath)[4], a language that allows an easy navigation through elements and attributes in XML or HTML files. This framework makes it easy to select only the attributes we are interested in. However, XPath is only applicable if the web pages are well-formed, which explains the usage of the HTMLCleaner when crawling the web pages.

In the Appendix A we describe our experience in crawling and scraping the data using two slightly different approaches from two different super-repositories.

Database Migration
The *Metrics Calculator* takes the extracted data, prepares and stores it in such a way that Complicity can easily retrieve the data necessary for the visualization without having to calculate the metrics on the fly. The dates of the first and last commit of a project and contributor have to be identified as their creation and death dates. The metrics that are pre-calculated are the lifetime of a project or a contributor, the total number of files a projects contains, how many commits have been done, how many and which contributors have executed at least one commit to a project and vice versa, on how many and which projects has a contributor worked on. How the data is stored is explained in more detail in Section 3.4.

During this step of the process, we also analyze the contributors in order to find redundant people based on their email address or names. Different approaches have been applied in the literature, such as the fuzzy string similarity [BGD+06], or the levenshtein distance [GKSD05]. We discuss the approach we have chosen in more detail in Section 3.8.

Complicity
We define Complicity as the web-based visualization tool with the underlying data model. The layout of the user interface of Complicity is solely written in the web standards *HTML 5* and *CSS 3*. The main interactions of Complicity are implemented in *JavaScript* using *jQuery*[5] for some user interface components (*e.g.,* slider, and tabs) and interactions (*e.g.,* asynchronous calls of the PHP scripts). *PHP scripts* retrieve the data from the database and convert it into JSON objects, the format required for the visualizations with Protovis.

The graphical visualization of the data is done using an external toolkit, called *Protovis*[6]. Protovis is a free and open source graphical visualization toolkit written in JavaScript. It has a good browser support as it renders the visualizations into Scalable Vector Graphics (SVG). In addition, Protovis provides many visual features and basic interaction facilities, which make it easy to develop different types of visualizations that are aesthetically appealing at the same time. Any graph developed with Protovis is written using the underlying declarative language, which allows to write short code compared to other visualization toolkits [BH09].

[3] See `http://htmlcleaner.sourceforge.net/`
[4] See `http://www.w3.org/TR/xpath/`
[5] See `http://jquery.com`
[6] See `http://vis.stanford.edu/protovis/`.

3.4 Data Model

Our goal is to develop a data model that is *simple* and *generic*, which leads to the following requirements that have to be met:

- *Adaptability*: As our goal is to model and analyze software ecosystems, the data model should applicable to different super-repositories.
- *Flexibility*: The data has to be stored in such a way that can serve different purposes, *e.g.,* different visualizations.
- *Extensibility*: If in the future more data and different visualizations become available that require more tables, the data model should be easily adaptable to new circumstances.
- *Scalability*: For its usage in web-based applications, the data has to be stored in such a way that requests can be served in an acceptable time window.
- *Reduce Redundancy*: Even though the data should be accessible within a short time, we should try to avoid the storage of redundant data. The reasons are that it reduced the required storage capacity and, most importantly, updates of the data can be executed more easily.

The resulting data model, which deals with the identified requirements in order to meet our goals, is illustrated in Figure 3.4.

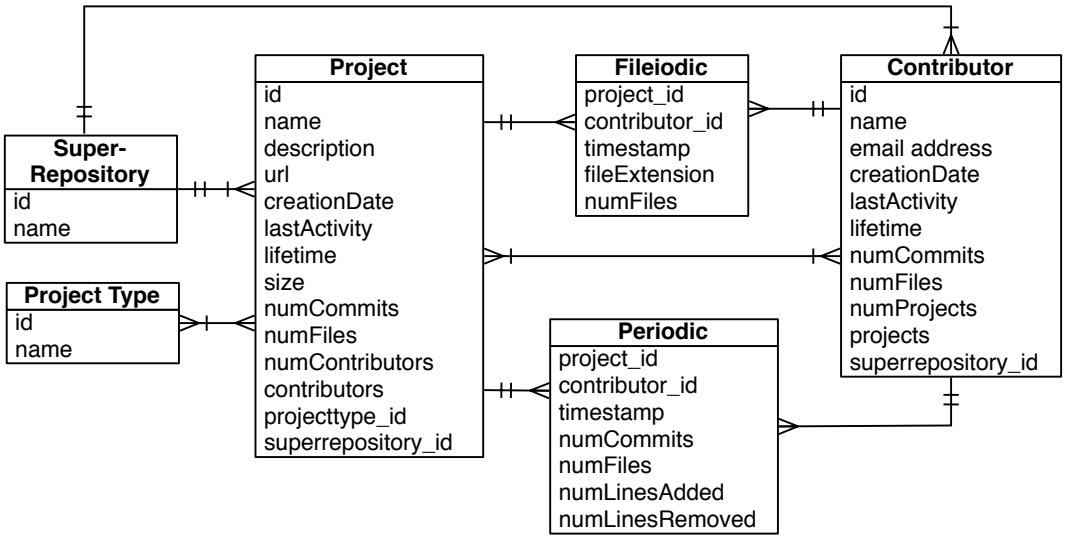

Figure 3.4. Data model behind the Complicity visualizations

In the following, the different tables are introduced and their purpose and dependencies among each other are explained in detail.

- The table **Super-Repository** contains the names of the super-repositories, for which we provide the data.
- **Project Type** lists all the categories into which the projects are grouped within their ecosystem. This categorization is done by the developers of the projects and does not result from our analysis.
- Each **Project** is attached to a super-repository and to at least a project type. For the projects from the GNOME super-repository each project has exactly one project type. The project table contains the general information, *e.g.,* name, and description, the date of the first and the last commit based on which the lifetime is identified, but also the pre-calculated metrics, such as number of commits (numCommits), number of files (numFiles), and number of contributors (numContributors). A special field of this table is contributors. It contains all the IDs of the contributors that committed at least once to the project
- The **Contributor** table is a central table as it contains the data about the second most important entity, the contributors. Same as the projects, each contributor is attached to one super-repository. This allows a fast access to the contributors of the selected ecosystem. Compared to other version control system, Git differentiates between author and committer[7]. The author is the person who originally wrote the work, and the committer is the person who last applied the work. In our work, we choose the author and stored its data in the Contributor table, ignoring the committer. This might lead to different conclusions if the committers differ from the authors most of the time. Beside the general information such as name and email address, this table also holds the dates of a contributor's first and last commit as well as pre-calculated metrics (*e.g.,* number of commits, number of projects and the lifetime, which is calculated based on the dates of his first and last commit). Same as the Project table it contains a special field, projects, which contains all the IDs of the projects a contributor committed to during his lifetime in the ecosystem. In conclusion, a project can have many contributors and a contributor can work on multiple projects.
- The tables **Periodic** and **Fileiodic** contain information about the changes that have been done to the project by the contributors at any time. The difference is that Periodic contains the general changes based on pre-calculated metrics (*e.g.,* number of commits, number of files), whereas Fileiodic contains the data necessary to visualize the expertise of a contributor or project based on the number of files that have changed with specific extension. These two tables are of main interest for time-based visualizations, *e.g.,* time series or animations.

[7] See http://progit.org/book/ch2-3.html

3.5 Metrics

As previously mentioned we focus on software metrics and visualization. Table 3.1 lists all metrics that we are used subsequently in the views. These metrics are not all applicable to both abstraction levels, ecosystem (E) and entity (N), nor can they all be used for both entities, project (P) and contributor (C). The value of a metric can be a summation over the entire lifetime (L), or at a specific timestamp (T).

Table 3.1. An overview of the metrics used in the views

Metric	Shortcut	Abstraction Level	Entity	Period
Number of Commits	NOC	E, N	P, C	L, T
Number of Files	NOF	E, N	P	T
Number of Projects	NOP	E, N	C	L, T
Number of Contributors	NOD	E, N	P	L, T
Lifetime	LT	E	P, C	L
Date of First Commit	FDATE	E	P, C	L
Date of Last Commit	LDATE	E	P, C	L
Number of Lines Added	NLA	N	P, C	T
Number of Lines Removed	NLR	N	P, C	T
Difference between Number	DIFFL	N	P, C	T

A metric that can represent the current state but not the entire lifetime is for example number of files. It counts the files that are currently available for a project and ignores the files that have been deleted. The same is true for the metric project type. It represents the category a project has been grouped in at the moment of the data extraction.

All these metrics are used in visualizations. The graph types used for the visualizations are presented next.

3.6 Views

In this section we introduce the different graph types that are used for the visualization of the data. We explain briefly their general construction principles and what interactions they support in our implementation.

3.6.1 Scatter Plots

All our views at ecosystem level are of type scatter plot. The main difference between each of them is the data they visualize: different entity type (projects or contributors) and different metrics.

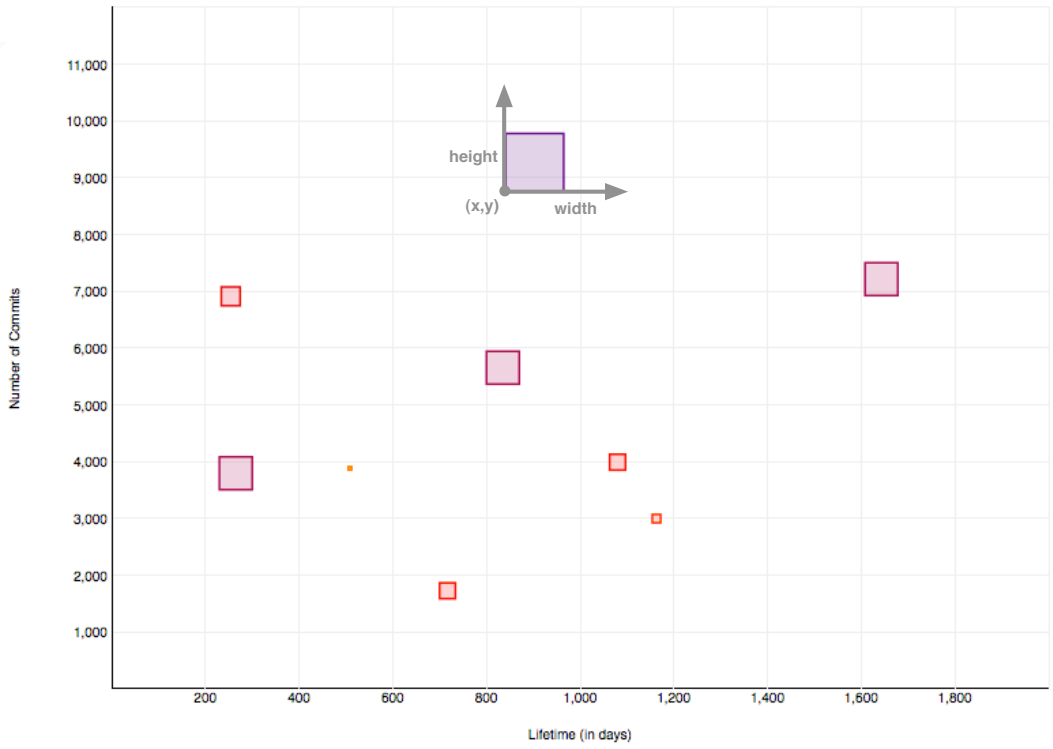

Figure 3.5. An example of a scatter plot illustrating how the position, width and height of the boxes are calculated

The coordinates, x and y, as well as the width, height and color of each shape is calculated based on the selected metrics. The right bottom corner of the shape is positioned at the calculated coordinates and they are expanded to the right and to the top according to the calculated width and height, respectively.

On mouse over a shape, a pop-up window appears with the basic data about the underlying entity. By clicking on the shape a detail panel opens, which provides a summary.

3.6.2 Stacked Area Charts

Two views at entity level are constructed using a stacked area chart. Each item (*e.g.,* project, contributor, or file extension) in this graph is displayed with an area stacked over each other and using one of the predefined colors at random. The sequence in which they are stacked depends on the sequence the items are retrieved from the database. We do not oppose any fix ordering.

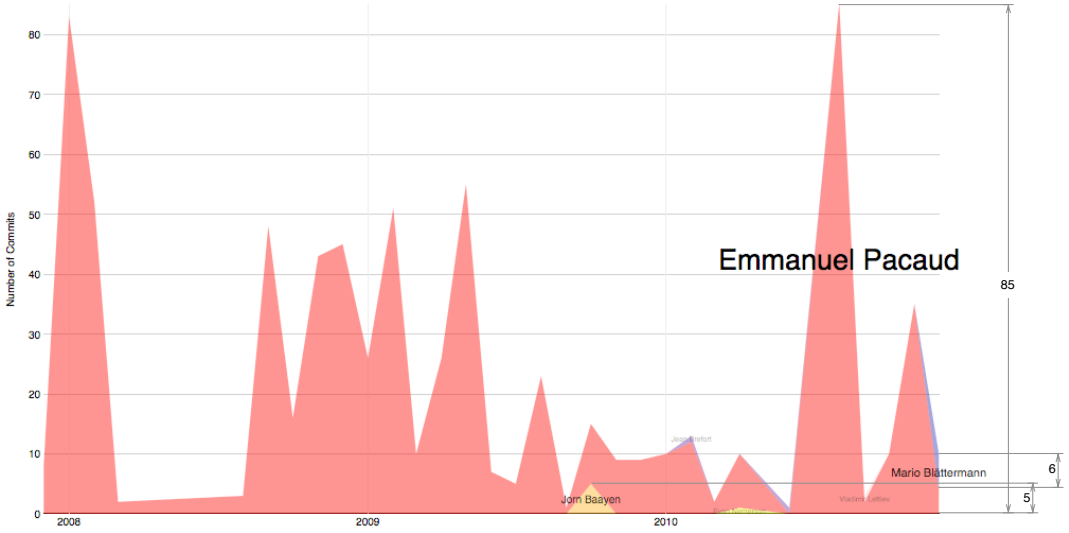

Figure 3.6. An example of a stacked area chart illustrating the difference in number of commits and the relative size of the labels

The information in the stacked area charts is displayed at monthly basis. For each item (*e.g.,* project, contributor, or file extension) we identify the month where most change have been done. Based on this maximum value (max) the size, position, and color intensity of the label text is calculated. The size (in pixel) of the label is equal to the rounded value of:

$$5 + \sqrt{max}$$

whereas the alpha value for the font color intensity is calculated as follows:

$$\frac{\sqrt{max}}{7}$$

On mouse over an item, a pop-up window appears with the total number of changes for the selected item. By clicking on an area, a pop-up window opens that provides a link to the detail page of the selected entity.

3.6.3 Area Charts

Area Charts are used to visualize some metrics of a single project or contributor at entity level. They visualize the change of a metric over time having the date on x axis (see Figure 3.7). Area charts connect two subsequent points with a line and fill the area between the line and the axis with a color.

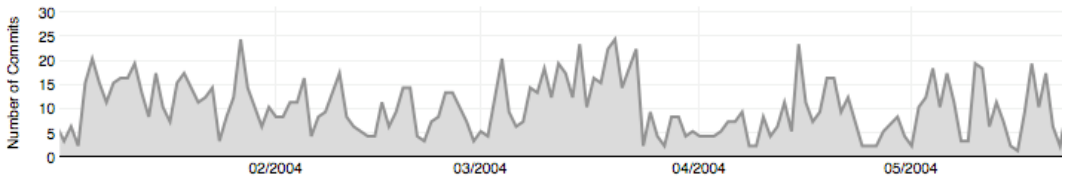

Figure 3.7. An example of an area chart illustrating the number of commits over some timestamps

Area charts connect two subsequent points. However, this can cause a problem, if no data is available for each timestamp. If we assume, as illustrated in Figure 3.8, two commits have been done on each of the two dates June 20 and June 23, but no one committed on the two days in between. The area chart connects the point of June 20 with June 23 instead of assuming a zero value for June 21 and 22. Examining the resulting graph, the analysts might take wrong conclusions by assuming that two commits have also been executed on the 21. and 22. June. One solution for this problem is to add zero values for the days, on which no commits have been executed. The problem with this approach is the enormous data overhead that results from adding zero values for every day no commit has been executed by any contributor to any project. For this reason we decided to stay with the initial data set and accept the just explain drawback.

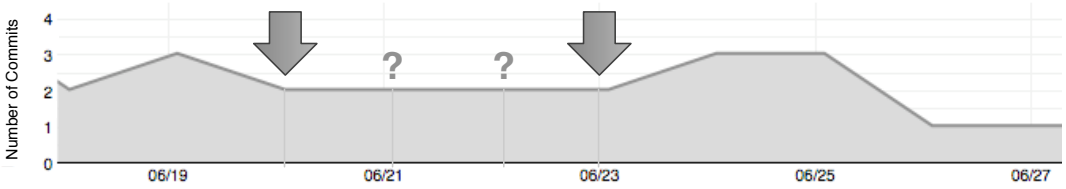

Figure 3.8. An example illustrating a general problem of area charts

We do not provide any direct interaction for this type of chart. Instead we provide a context chart that allows the modification of the time period that is displayed.

3.6.4 Line Charts

Line Charts are used at project and contributor entity level. The data is visualized by a line, which connects two subsequent points. We use the line chart to visualize the change of a single metric, the difference between number of lines added and number of lines removed, over time having the date on x axis. The user is free to choose between the time unit of day and month, respectively.

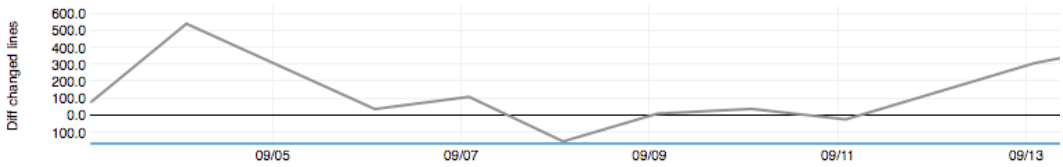

Figure 3.9. An example of a line chart illustrating the difference between number of lines added versus number of lines removed over some timestamps

Same as the area chart, no interaction is implemented directly in the area chart. Instead they make use of the context chart to define the time period for which to display the data.

Note that the last three chart types (stacked area, area and line chart) require data for at least two different timestamps (which have to be two different months for the stacked area chart) in order to be able to show any visualization.

3.7 Case Studies

Two different ecosystems are considered for the validation of our work. They have been chosen according to the following criteria:

- **Availability**: Both ecosytems contain open source and free software systems and so the required information is freely available.
- **Git Web Interface**: This work focuses on the data available in the Git web interface. In consequence, only projects that use this interface are considered.
- **Size**: One of the ecosystem is of smaller size, which allows a fast investigation due to a smaller crawl and scrape time. The second ecosystem's size is one order of magnitude larger and makes it possible to countercheck whether the observations made for a small super-repository can be generalized for larger ecosystems.

The following Table 3.2 provides an overview of the two ecosystem case studies with its key numbers:

Table 3.2. An overview of the two ecosystem case studies

Repository	Projects	Contributors	Active Since	Version Control
Gnome	1,292	> 4,800	1997	Git
SourceForge	> 260,000	> 2,700,000	1999	CVS, SVN, Git

The two super-repositories, which we are considering for our investigation are the following:

- **GNOME** is a desktop environment for GNU/Linux and Unix computer. It contains exclusively free software that integrates smoothly into the rest of the Unix or GNU/Linux desktop. GNOME allows contributions in different programming languages (C, C++, Python, Java, *etc.*) and fields (usability, QA teams, *etc.*). GNOME 3, has been launched in April 2011. The numbers presented in the above table are based on the data we scraped from the Git web interface. We stopped the scraping process in February 2011.
- **SourceForge** is one of the largest open-source applications and software directory currently available. It also contains exclusively free and open-source software systems of which the data, including code, software details, and version control information is freely available. We consider only the projects that make use of the Git version control system and make the data available via the Git web interface. Due to the limited time, only a subset of the projects is considered in this work.

In the subsequent chapters we only make use of the data from the GNOME ecosystem as at the time of writing this document only that data was available.

3.8 Aggregation of Contributors

The data collection of open source software ecosystems available on the internet is huge but also dirty (see Appendix B) in the sense that fields are likely to be misused, *e.g.,* a contributor's field. In addition, contributors are often active using different email addresses and names, *i.e.,* they may contain typos, special characters, shortcuts, *etc.*, to commit to a repository.

This is a common problem, known as aliasing [BGD+06].This problem might be especially bad in an open source environment where people do not care or no common guidelines exist. Different solutions have been proposed to tackle this problem, such as using the Levenshtein distance [SMG09] or fuzzy string similarity, domain name matching, clustering, and heuristics [BGD+06].

We tried to reduce the number of redundant committers of the GNOME ecosystem focusing mainly on the fuzzy string similarity distance.

3.8.1 Email address

Committers within a project or super-repository can be identified by their name and/or email address. Email address have the advantage over names that they are unique, and contain only few special characters, *e.g.,* @._- Therefore, we consider two contributors to be the same if they use the same email address independent of their names.

3.8.2 Contributor names

Names compared to email addresses are not limited in special characters and can be displayed using different character sets. In addition, people tend to sometimes write there names in different forms: with special character or without; with middle name(s) or without; with abbreviations or without; with typos or without. This makes it hard to identify a same contributors solely based on their name.

The main disadvantage of names over email addresses is that they are not unique. Two contributors with exactly the same name independent of their email address can be two different people in reality. However, we consider two people to be the same person if they have the same name.

We go even further in applying the *fuzzing string similarity* algorithm to our collection of contributors to find a same person based on similar names.

3.8.3 Simple Fuzzy String Similarity (SFSS)

To identify whether two strings are the same, the simple fuzzy string similarity algorithm first converts each string into a collection of bigrams. Then, it compares the two collections, counting the total number of bigrams that are equal in both collections (NbrOfMatches). Based on the resulting sum of matching bigrams the simple fuzzy string similarity index is calculated as follows:

$$\frac{NbrOfMatches}{NbrOfBigramsInCollection1 + NbrOfBigramsInCollection2}$$

For this index we get a value between 0 (no bigram in common) and 1 (two equal bigram collections).

Using a SFSS index of strictly larger than 0.8 we managed to reduce the total number of contributors by 1,227 from initial 6,064 to 4,837 contributors. Surprising was that approximately 71% of eliminated contributors had the same name but different email addresses (876 out of 1,227). 26 out of 351 (approx. 7.4%) duplicates without error could be successfully identified for an index strictly larger than 0.95. For the remaining 325 contributors we found that 29 misclassifications (error of

8.9%). Most misclassified contributors (25 out of 29) could be identified for an SFSS index between [0.9 ; 0.8[. All classification results have been checked manually.

3.9 Conclusion

This chapter, we identified different stakeholders (*e.g.,* project managers, software quality assessment teams, developers, and researchers/explorers) for which Complicity but also the analysis of software ecosystems in general is of interest.

We introduced Complicity a web-based tool that supports analysts in understanding the evolution of software ecosystems. We presented its user interface and how the data is distributed within it to facilitate the usage of the tool. Furthermore, we explained the architecture of Complicity and its backend and how it supports the process from the data acquisition to the data visualization. We showed the underlying data model, which illustrates how the data used within the visualizations is stored in such a way that it is applicable to different super-repositories and can be accessed easily.

As already mentioned several times, our focus lays on visualizing the evolution of software ecosystem. We introduced the metrics that have been extracted from the collected data. We explained what type of graphs (*e.g.,* scatter plot, stacked area chart, area chart and line chart) are used within Complicity, how they are generally constructed, and how we can interact with them.

We ended up with the introduction of the ecosystems, from which we extracted the data for the visualizations and analyses, and which are available within Complicity. We also showed on the GNOME ecosystem how we eliminate duplicate contributors by applying the fuzzy string similarity distance to contributor's names. We were able to reduce our collection of contributors by almost 80%. This aggregation process is especially important as with the elimination of duplicates we reduce the number of elements, which in consequence increases the performance in generating the visualizations. In addition, if the aggregation is applied correctly we gain data of higher quality, which results in more accurate analyses and visualizations, and eventually in better understanding.

In the next chapter we introduce different views. We explain how they are constructed, which metrics and graph type they use and what data can be extracted from them.

Chapter 4

A Catalogue of Views

Using basic metrics extracted from the data scraped from the web pages of the Git web interface, different views have been constructed. These views make use of some or all metrics available for a single entity or for all entities at ecosystem level. In the following, we describe the different views, the metrics they consider, how they are constructed and the general idea about them, illustrating it on one or more examples. The views are grouped by their abstraction level and are introduced in the following order:

Ecosystem Level Views

- Lifetime View
- Affectional Bond View
- Popularity View
- Activity Fire View
- Extremes View

Entity Level Views

- Activity Diagrams View
- Involvement Distribution View
- Expertise View

4.1 Views Map

The map in Figure 4.1 provides an overview of the available views. Each of them is either a view at ecosystem or entity level, but can be available for either project or contributor, or for both. Those views that exist for both are placed on the line separating the project from contributor views.

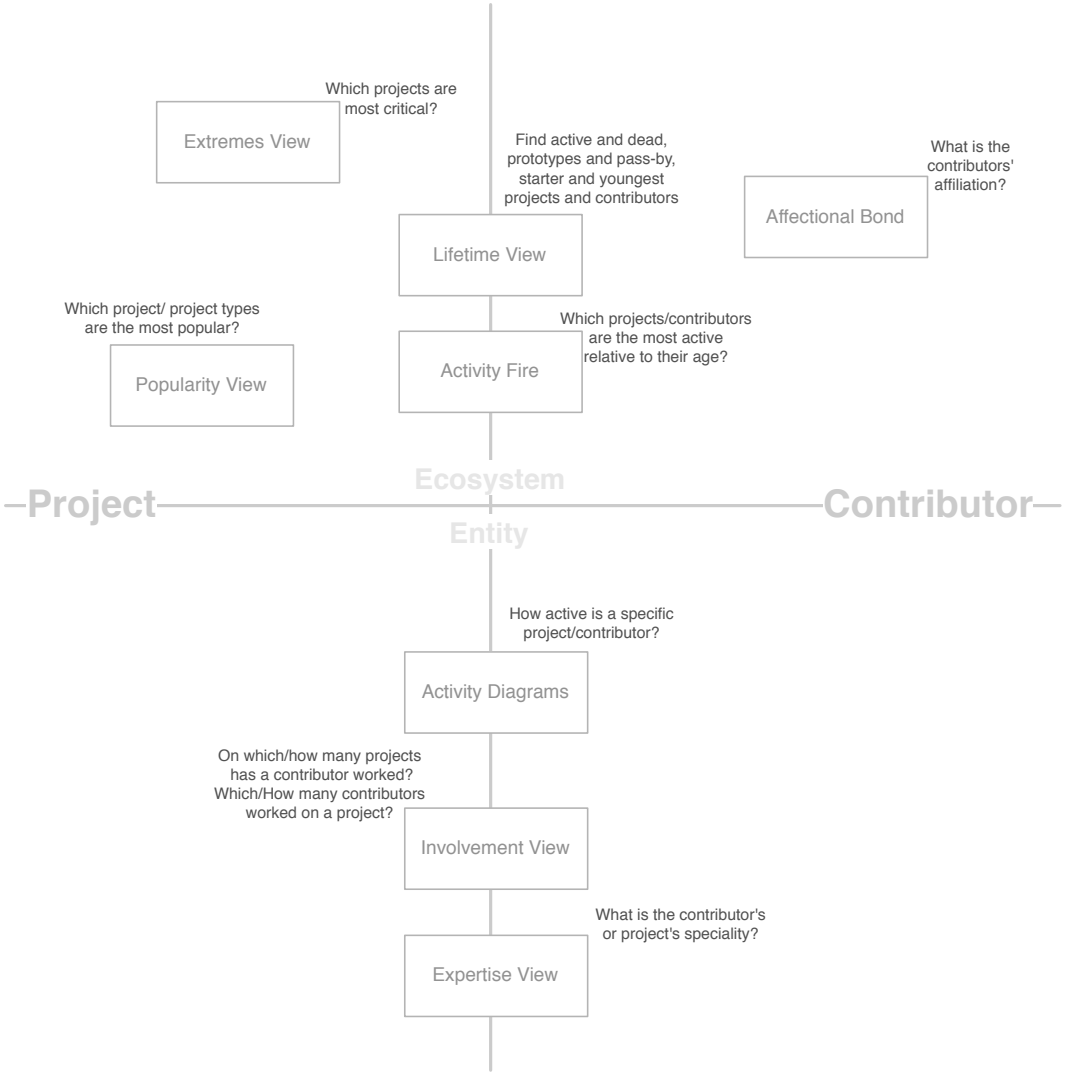

Figure 4.1. Views Map gives an overview of the predefined views and provides information about which view is interesting for which groups

Independent of the group of concerns, anybody should start with the Lifetime View of the projects or contributors of the subject ecosystem (top center of Figure 4.1) as they provide information about the projects or contributors that are still alive and active, respectively. The steps that will follow depends on the interest of every single person.

The analysts can move from any view to another independent of the abstraction level or underlying

entity type. In other words, from any view at ecosystem level, we can go to the entity detail page and analyze a single project or contributor. From the entity detail page we have again the possibility to move back to the ecosystem level. This results in an highly interactive tool without restricting the users to any fixed path.

Selecting some projects at ecosystem level and then changing to the contributors at ecosystem level illustrates only the contributors that worked on at least one of the selected projects. The same is true the other way around, selecting some contributors and then moving to the projects' ecosystem level, shows only the projects, on which any of the selected contributors worked in the past.

4.2 View Description Template

In the following section we introduce our template of the views. Each of them is described by the following properties:

- **Name**: The name of the view is specified as the title of the respective subsection.
- **Goal**: A short goal will be defined for each view, which makes it easy for the reader to find out whether that view is of interest for him/her.
- **Abstraction Level**: It states the level of abstraction of the data that is visualized: project(s) or contributor(s) at ecosystem or entity level.
- **Stakeholder(s)**: Not every view is relevant for each interest group. Here we will specify for whom it might be beneficial.
- **Metrics**: It will be listed which metrics could be applied in this view. To state that any numerical metric could be applied, we write: *num**. *PT* is standing for project type. The metrics are abbreviated as listed in Table 3.1.
- **General Idea**: In this part we describe what information we expect to extract from the visualizations and for which of the stakeholders this might be of interest.
- **Construction**: For each view we will demonstrate how they are constructed: type of graph and metrics.
- **Examples**: For each view at least one example is taken from our case studies using Complicity. They are discussed in details including our observations.
- **Discussion**: At the end we make some statements about the strengths and weaknesses of the graphs.

4.3 Ecosystem Level Views

4.3.1 Lifetime View

Goal	Get an idea about the overall lifetime of the projects and contributors in the ecosystem and about how many of them are still active.	
Abstraction Level	Projects' or Contributors' Ecosystem	
Stakeholder(s)	Project Manager, Contributors, Quality Assessment Team, and Researchers & Explorers	
Metrics	**Metric Name**	**Metric(s) Used**
	x-axis	FDATE
	y-axis	LDATE
	width	num*
	height	num*
	color	PT, num*

General Idea

This first view provides a general picture of the overall projects' or contributors' lifetime in the ecosystem. In addition, we can reveal the projects or contributors which are dead and active, respectively, but also prototypical projects or pass-by contributors. In case, the number of active projects or contributors decreases continuously, it is likely that the subject ecosystem is about to die. Vice versa, if the number of projects and contributors is continuously increasing, it is an indicator for a growing and active ecosystem. These are important information for any stakeholder especially when they are about to decide whether or not to contribute to a specific ecosystem.

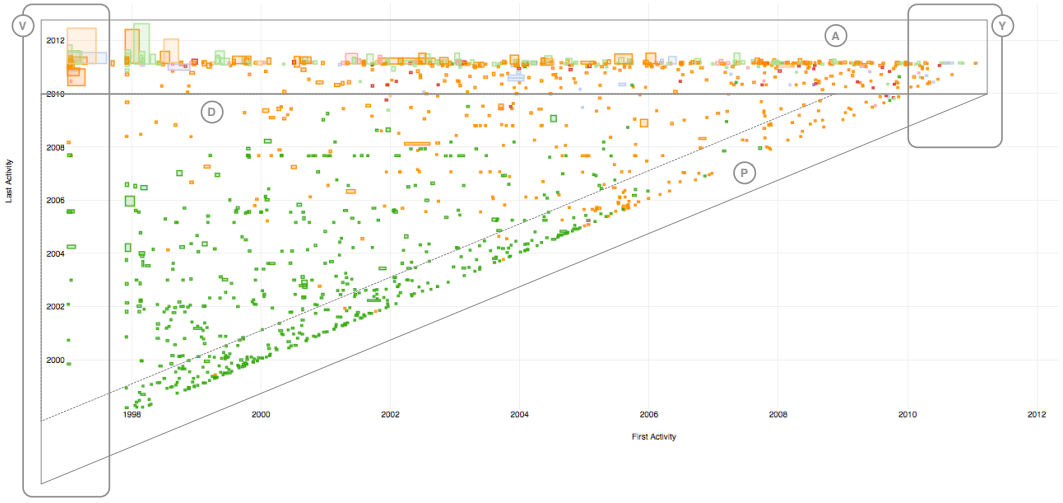

Figure 4.2. Projects' Lifetime View of the Gnome Super-Repository (height: NOC, width: NOF, color: PT)

Construction

The following view is drawn as a scatter plot with each box representing a project or contributor. The boxes are placed on the x and y axes according to their first and last activity dates at a monthly basis. Depending on the settings of width, height and color, different groups of projects or contributors can be identified that are likely to die or contrary to survive.

Examples

GNOME. Figure 4.2 shows an example of the projects' Lifetime View. The x and y axes are set as explained above. Each box represents a project and the width is equal to the total number of files and the height is equal to the total number of commits. Each color symbolizes a different project type.

The shapes on the top horizontal line represent the projects that are still active (marked as A). We define a project as active if at least one change has been done within the last year. The projects positioned on the most left vertical line of the triangle area are the projects that were created with the ecosystem in 1997 (marked as V) whereas those on the top right corner illustrate the newly created once (marked as Y). Generally, it is observable that many projects of GNOME are dead (marked as D) and either of color green (project type: Z_Archived) or orange (project type: Other). Many projects die within a year, which we classify as prototypes (marked as P). They are positioned on the diagonal line closing the triangular area.

In the example of the contributors' Lifetime View (see Figure 4.3), every entity represents a single contributor. For the width and height of each box we used the total number of commits. The number of projects a person contributed to is illustrated by the color gradient from orange (few projects) to purple (many projects).

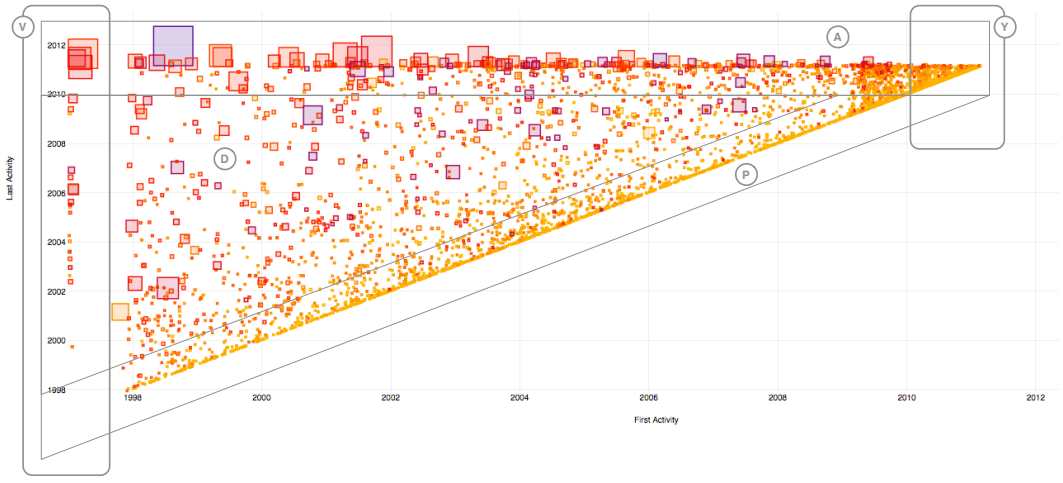

Figure 4.3. Contributors' Lifetime View of the Gnome Super-Repository (height/width: NOC, color: NOP)

In 1997 the first contributors started the GNOME project (marked as V), positioned on the left most vertical line, of which few people are still active today. Only around 1998 new people joined the starting group (boxes positioned on the vertical line around 1998). Same as for the projects, the contributors on the top most horizontal line are still active (marked as A). Contributors that have not committed during the last year are considered dead (marked as D). We can observe that some potential

experts, who either did a lot of commits (large box) or worked on many different projects (purple box), are dead and their knowledge might be lost. On the other side, many contributors are pass-by committers (marked as P) staying only for a short time (less than a year), positioned on the diagonal line of the triangular shape, and doing only a few commits (orange colored). The contributors in the top right corner are the youngest who joined within the last year (marked as Y).

Discussion

From this view we can extract many different and relevant information about the projects' and contributors' lifetime in the ecosystem. For both entity types, we can identify the dead, the still active as well as the veterans and youngsters. At the project ecosystem level, we can also identify the prototypes and the project types with the majority of dead projects. At contributors' ecosystem level, we can specify the pass-by committers and the possibly lost experts.

A visual shortcoming of this view is that due to its triangular shape, we end up with a lot of unused white space. From this view we can extract a contributor's or project's lifetime based on the first and last commit but we do not know how active a single contributor has been between these two dates or how often and how much a single project has been changed during this time. For this purpose we introduce the Activity Diagrams View at entity level in Section 4.4.1. Additionally, we have seen that many contributors have a short lifetime within the ecosystem. Some of them have a relatively high number of projects (purple colored box) and become an expert within a short time. From the Lifetime View, we cannot say what type of expert we lost, if he was a developer or a translator. The Affectional Bond View, introduced in Section 4.3.2, analyzes this shortcoming.

4.3.2 Affectional Bond View

Goal	Find a contributors' affectional bond to either development or translation work.	
Abstraction Level	Contributors' Ecosystem	
Stakeholder(s)	Project Manager, Contributors, Quality Assessment Team, and Researchers & Explorers	
Metrics	**Metric Name**	**Metric(s) Used**
	x-axis	NOC
	y-axis	NOP
	width	LT
	height	LT
	color	NOP

General Idea

It depicts the affectional bond of a contributor to either development or translation work, or both, based the total number of commits a person has done or the total number of projects a person has contributed to over his lifetime in the ecosystem. Our assumption is that a contributor is likely to be a developer if he committed many times to only a few projects. Contrary, we think that a contributor is likely to be a translator, if he worked on many different projects but has executed a relatively small number of commits to the ecosystem.

Construction

The Affectional Bond View is visualized as a scatter plot with each box representing a single contributor. They are distributed based on their total number of commits or total number of projects. The size of the shapes is defined by their lifetime in number of days.

Examples

GNOME. The shapes in the Affectional Bond View are distributed according to the total number of commits on the x-axis, and the total number of projects on the y-axis and for the color of the shapes. The width and height are defined by the contributor's lifetime.

By looking at Figure 4.4 we can observe a highly intense left bottom corner, where the shapes cannot be clearly differentiated any more (marked as N). The remaining shapes tend to split into two groups: one with contributors who have a high number of projects (marked as T) and one with the contributors who have a high a number of commits (marked as D). Furthermore, we notice that the boxes in T are of different sizes whereas the one in D tend to be relatively large compared to most others. By interactively checking the nodes, we found out that the boxes in T are likely to be translator and that the boxes in D are likely to be developers that have possibly also done some translations in the past.

Discussion

From this view we get an overview of the contributors affection to either translation or development work. We get a feeling for the distribution of the contributors within a single project or at ecosystem, if there are more translators than developers and allrounders or vice versa.

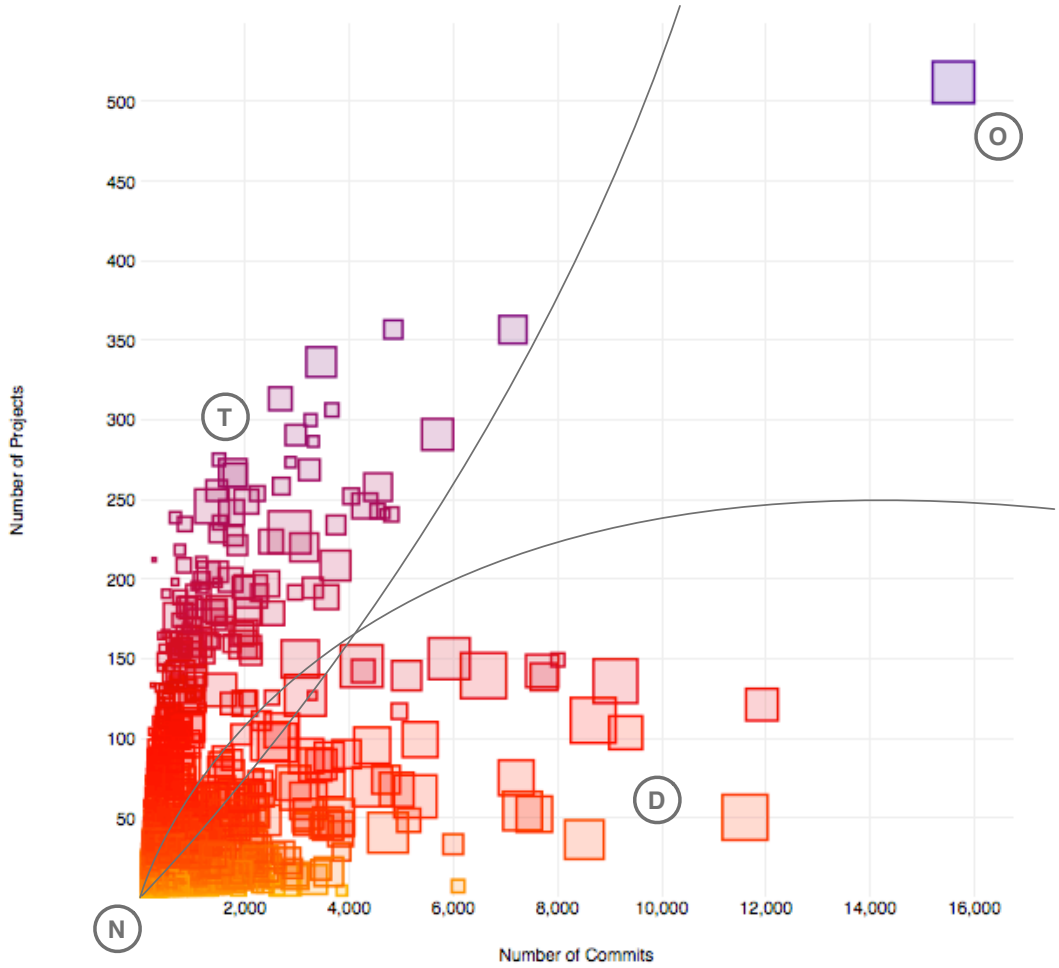

Figure 4.4. Affectional Bond View of the contributors within the GNOME ecosystem (x-axis: NOC, y-axis: NOP, height/width: LT, color: NOP)(T: translators, D: developers, N: no-mans land, O: outlier)

The main drawback is that people who started more recently but contributed a lot relative to others are getting lost in this visualization. For this purpose, we introduce the Activity Fire View in Section 4.3.4.

4.3.3 Popularity View

Goal	Find the most popular projects based on the total number of commits or number of people that contributed to it over its entire lifetime	
Abstraction Level	Projects' Ecosystem	
Stakeholder(s)	Project Manager, Contributors, Quality Assessment Team, and Researchers & Explorers	
Metrics	**Metric Name**	**Metric(s) Used**
	x-axis	NOC
	y-axis	NOD
	width	LT
	height	LT
	color	PT, num*

General Idea

It Identifies the most active and popular projects in the ecosystem based on the number of commits and number of contributors, respectively. The more people contributed to a project or the more changes has been done to a project, the higher its popularity degree but also the higher the risk of introducing new bugs into the system.

Construction

The Popularity View is a scatter plot with each shape representing a single project within the ecosystem. It is constructed the same way as the Affectional Bond View in Section 4.3.2 with the boxes spread according to their total number of commits on the horizontal axis and the lifetime defining the width and height. Contrary to the Affectional Bond View it has the total number of contributors on the vertical axis.

Examples

GNOME. In Figure 4.5 we can observe that most projects tend to have less than 200 contributors and less than 5000 commits in their entire lifetime (marked as G). The remaining projects follow a logarithmic distribution (marked as L), which means that they need a large number of contributors in order to produce many changes. There are only few outliers, that follow an almost linear line (marked as O).

Discussion

The view depicts the projects that are popular according to their number of commits and/or number of contributors. We observed that a project in the GNOME ecosystem needs at least 200 contributors or more than 5,000 commits to become popular.

But the main drawback is that even if we used the lifetime as width/height of the boxes, the projects with a short lifetime are likely to disappear in the crowd, even though they might be quite active or popular relative to their age. For this purpose we have the Activity Fire View presented in Section 4.3.4.

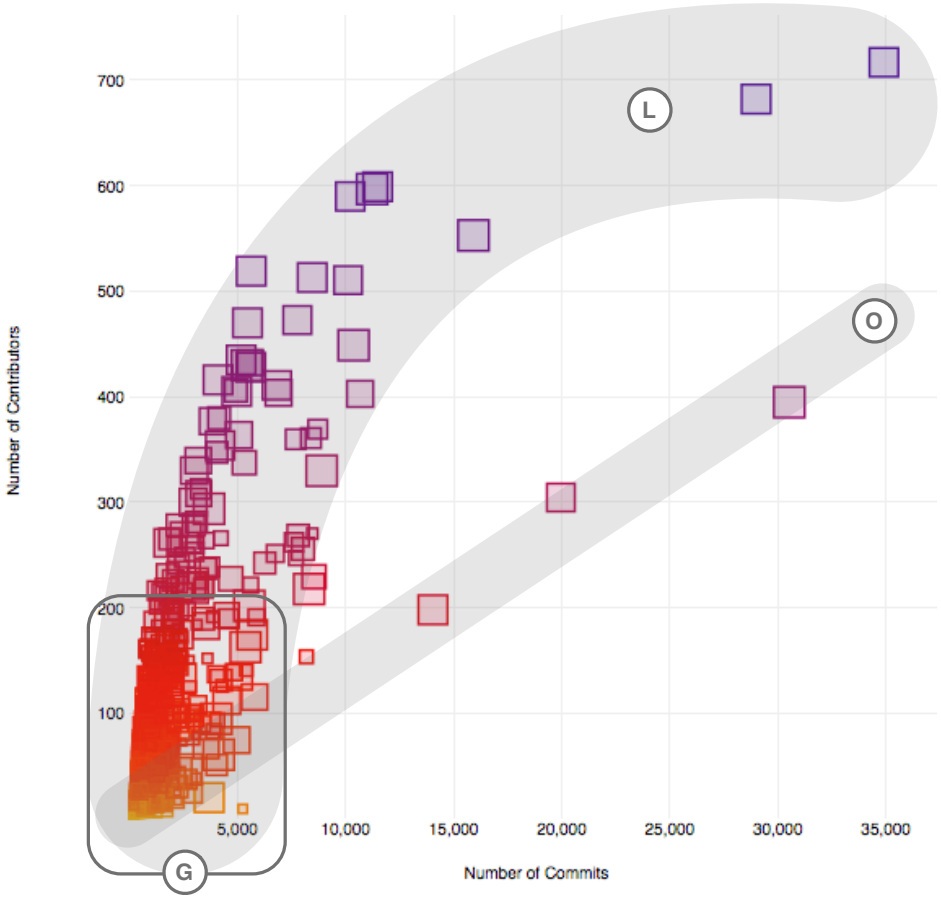

Figure 4.5. Popularity View of the projects within the GNOME ecosystem (x-axis: NOC, y-axis: NOD, height/width: LT, color: NOD)

4.3.4 Activity Fire View

Goal	Detect outstanding projects or contributors based on a high activity rate (*e.g.,* many commits, many projects, *etc*..)	
Abstraction Level	Projects' or Contributors' Ecosystem	
Stakeholder(s)	Project Manager, Contributors, Quality Assessment Team, and Researchers & Explorers	
Metrics	**Metric Name**	**Metric(s) Used**
	x-axis	LT
	y-axis	num*
	width	num*
	height	num*
	color	num*

General Idea

Depending on the numerical metric chosen on the y-axis, outstanding projects and contributors relative to their age can be identified that would not be possible otherwise. The metrics chosen for width, height and color can further classify the outstandings into different categories.

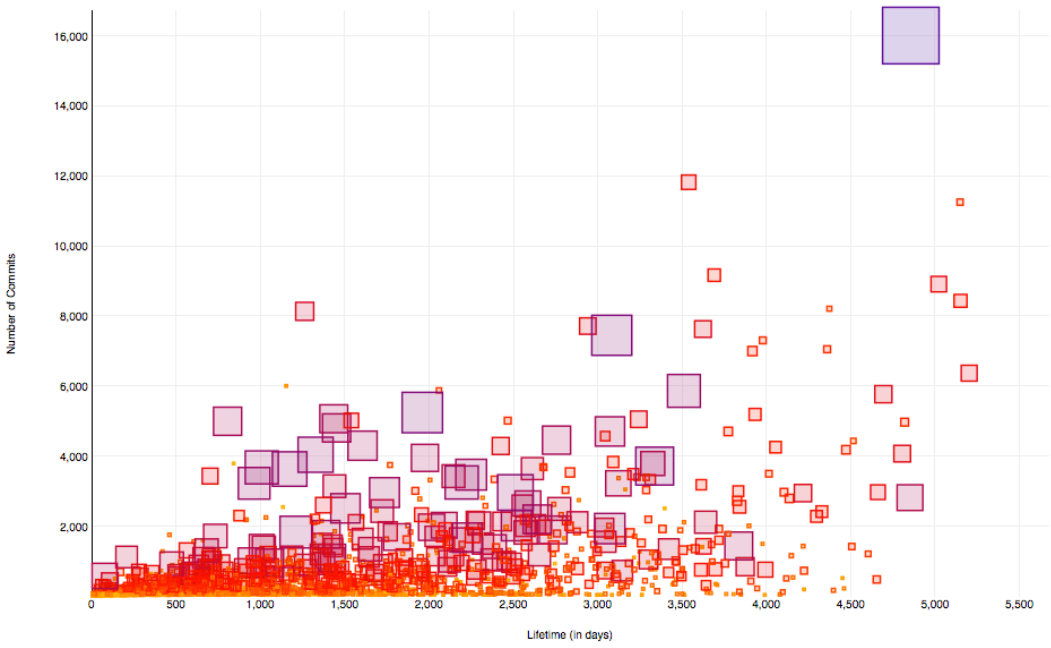

Figure 4.6. Activity Fire View of the contributors within the GNOME ecosystem (x-axis: LT, y-axis: NOC, height/width/color: NOP)

Construction

The Activity Fire View is drawn as a scatter plot distributing the shapes according to their lifetime in days (x-axis) and the defined metric for the y-axis. Width, height and color of the shapes can be freely chosen.

Examples

GNOME. In Figure 4.6 we show the Activity Fire View for the contributors at ecosystem level. The width, height and color of the shapes is defined by the number of projects and distributed according to the lifetime (x-axis) and the number of commits (y-axis).

Using number of commits on the x axis allows us to find the most active people according to their lifetime. We can spot that the density of the shapes is higher for lower lifetime values, which can either mean that the subject ecosystem has many youngsters, or alternatively, that many contributors have a short life in the ecosystem.

Discussion

This view allows us to identify the most active people based on selected metric for the y axis and their lifetime. In other words, we have the possibility to spot contributors that are highly active relative to other peers of the same age.

Using lifetime on the x axis may end up in confusion because analysts might think that there are many newcomers (with a small lifetime) whereas they forget that not all contributors with a short lifetime are still active. In other words, this graphs shows all people that contributed to the ecosystem at least once in their lifetime. The same problem occurs at projects' ecosystem level. Consulting the Lifetime View, which we introduced in Section 4.3.1 can clarify the situation.

4.3.5 Extremes View

Goal	Identify extreme projects that have either a high number of commits or a high number of files.	
Abstraction Level	Projects' Ecosystem	
Stakeholder(s)	Project Manager, Contributors, Quality Assessment Team, and Researchers & Explorers	
Metrics	**Metric Name**	**Metric(s) Used**
	x-axis	NOC
	y-axis	NOF
	width	num*
	height	num*
	color	num*

General Idea

It detects different categories of projects based on a high number of commits or a high number of files or a high number for both metrics. Especially projects with a high number of commits can be an indicator for critical projects as it means that they change often and so new errors can be introduced.

Construction

The boxes in the scatter plot of the Extremes View are spread according to their number of commits (x-axis) and number of files (y-axis). Each shape represents a single project of the ecosystem. Any numerical metric can be chosen for the width, height and color of the shapes.

Examples

GNOME. In Figure 4.7 we illustrate the Extremes View of the GNOME super-repository with the lifetime defining the width and height of the shapes and the color representing the number of contributors.

We can observe that most projects have less than 1,000 files and less or equal to 10,000 commits (marked as N). Those projects that have a high number of files are marked with F whereas those with a high number of commits are marked with C. We can observe that projects need a longer lifetime (large boxes) in order to enter group C compared to group F, which contains also little boxes (short lifetime). In addition, it becomes visible that projects with more than 5,000 commits tend to have a higher number of contributors (purple color) than projects of group F. With this view we can also spot extreme outliers (marked as O) having both a large number of commits and files, and also a quite long lifetime (large box) and a relative large number of contributors (light purple).

Discussion

We can spot different groups of extreme projects with either a high number of files or high number of commits whereas only few (in our case a single one) projects fit in both categories.

Figure 4.7. Extremes View of the Gnome Super-Repository (x-axis: NOF, y-axis: NOC, height/width: LT, color: NOD)

4.4 Entity Level Views

4.4.1 Activity Diagrams View

Goal	Identify the vitality of a project or contributor based on their changing or activity rates.	
Abstraction Level	Project or Contributor Entity Level	
Stakeholder(s)	Project Manager, Contributors, Quality Assessment Team, and Researchers & Explorers	
Metrics	**Metric Name**	**Metric(s) Used**
	x-axis	Date
	y-axis	NOC, NOD/NOP, NLA, NLR, DIFFL & NOF
	width	-
	height	-
	color	NLA (green), NLR (red), otherwise (gray)

General Idea

With the activity diagram, one can get an idea about the vitality of a project or the activity degree of a contributor based on the number of commits, number of lines added versus number of lines removed, and number of files changes. Based on the different metrics we can identify whether a person is a regular contributor or not, whether he tends to add more lines, then he removes, whether he is working on many projects in parallel or concentrates on one project at a time, etc. The vitality of a project can be similar analyzed: the project under continuous change can be an indicator for continuous development, expansion, maintenance or bug fixing. Alternatively we can identify whether a project is about to die if no changes or only few changes are done very rarely.

Construction

The Activity Diagrams View consist of five diagrams that show the project's or contributor's activity at daily or monthly basis. Four of them are illustrated as area charts whereas the metrics number of lines added (green) and number of lines removed (red) are drawn in the same diagrams for better comparison. Only the metric of the difference between lines added and lines removed is drawn as a line chart. A sixth diagram at the bottom allows the user to interact and change the period for which he wants to analyze the changing or activity rate.

Examples

GNOME. In Figure 4.8 we show the Activity Diagrams View of the Nautilus project. We can identify different phases of the software development life cycle, *e.g.,* the birth of the project (1997-2000), the first major release (2000-2001), maintenance phase (2001-2010), and another major release (2010-2011).

In Figure 4.9 we illustrate the Activity Diagrams View of Christian Rose. We can observe that he is continuously highly active since end of 2000. Only by 2006 Christian Rose diminishes his activity in the GNOME ecosystem.

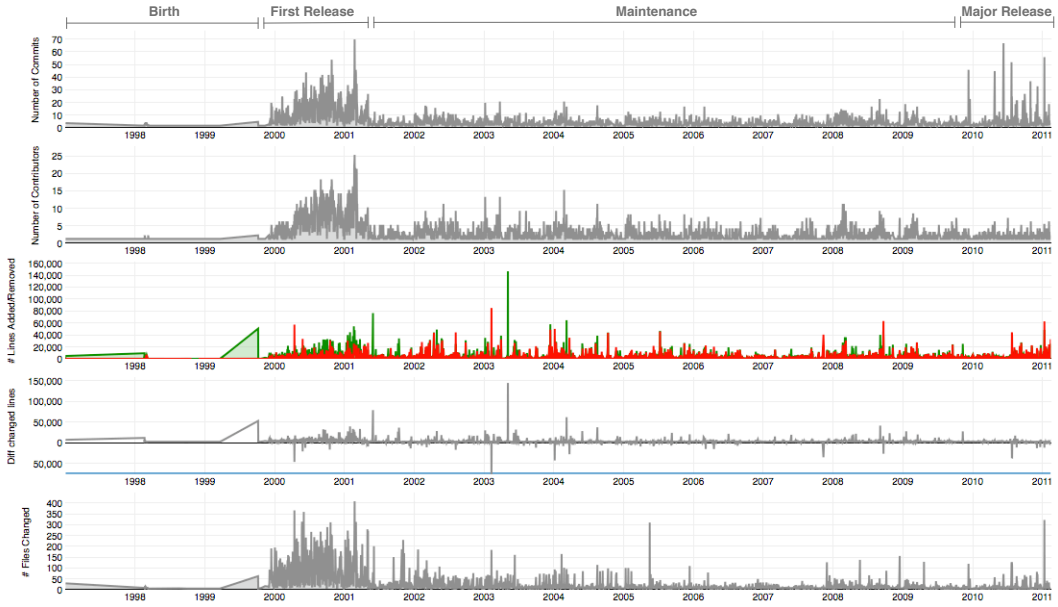

Figure 4.8. Activity Diagrams View of the Nautilus project within the GNOME ecosystem (x-axis: Date at daily basis, y-axis: NOC, NOD, NLA, NLR, DIFFL, and NOF)

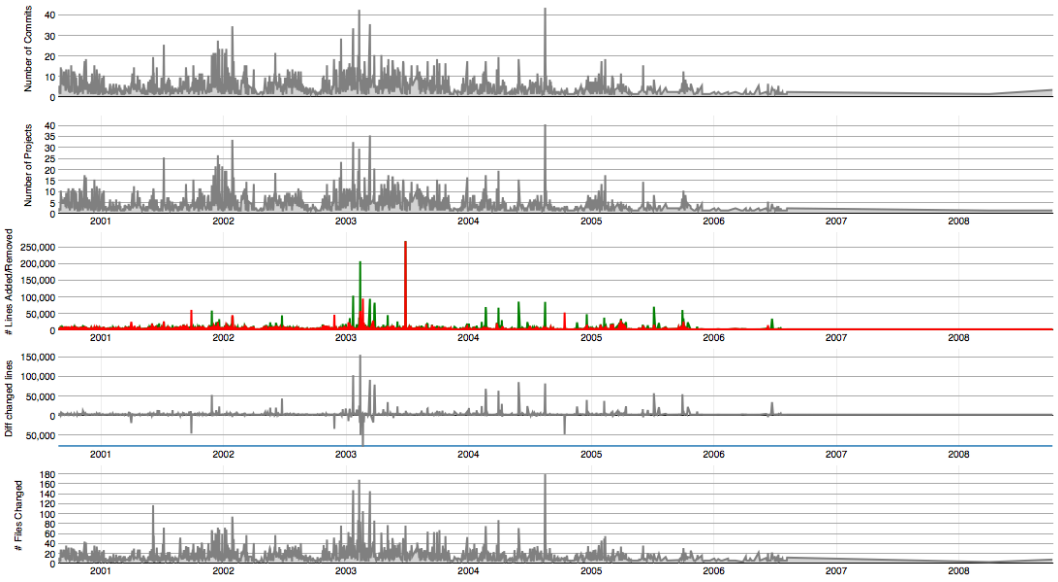

Figure 4.9. Activity Diagrams View of Christian Rose within the GNOME ecosystem (x-axis: Date at daily basis, y-axis: NOC, NOP, NLA, NLR, DIFFL, and NOF)

Discussion

Using the Activity Diagrams View we can observe the activity of a single project and contributor and detect extreme activity peaks or long activity lows in the changing rates at both entity levels. They also allow us to depict development stages of a single project.

The activity diagrams may lead to wrong conclusions depending on the affiliation of the contributor or the changes that have been executed on a project. A contributor committing or deleting graphical files (*e.g.,* icons, images, *etc.*) to or from the repositories results in many lines being changed. So, by analyzing only the activity diagrams, we might come up with a wrong conclusion. To act to the contrary, and reduce the risk of misinterpreting this data, we added the Expertise View introduced in Section 4.4.3.

Another drawback of this view are the general limitations of the area charts and missing data values for some timestamps as described in section Section 3.6.3.

4.4.2 Involvement Distribution View

Goal	Identify the projects a contributor worked on over his entire lifetime or the contributors that worked on a project over its lifetime.	
Abstraction Level	Project or Contributor Entity Level	
Stakeholder(s)	Project Manager, Contributors, Quality Assessment Team, and Researchers & Explorers	
Metrics	**Metric Name**	**Metric(s) Used**
	x-axis	Date
	y-axis	NOC
	width	-
	height	-
	color	Project, Contributor

General Idea

Depending of the entity type, project or contributor, different information can be extracted. If the entity at hand is a contributor, we can identify the projects he worked on over his entire lifetime in the ecosystem. We know whether he worked on many different projects or whether he focused his work on some few projects. Eventually, we could get a feeling whether the person is a translator or not depending on the involvement graph.

If the subject entity is of type project, we get an overview of the contributors that worked on it over time. In this view, we can also spot outstanding contributors, that committed more to the project than others over a short period or the entire lifetime. Eventually, we could identify whether a project has a good knowledge flow or not.

Construction

The Involvement Distribution View is drawn as a stacked area chart with the dates at monthly basis on the horizontal line and the number of commits on the vertical line. Every stack represents a different project or contributor and has one of the eleven specified colors at random.

Examples

GNOME. John is a contributor of the GNOME ecosystem since 1997 when he started working on the librep project (see Figure 4.10). In 1999 he started working on the rep-gtk project and shortly after on sawfish. During his three years at GNOME, he mostly focused his work on librep and sawfish before he quit the ecosystem.

Compared to John, Daniel Nylander worked on many different projects during his five years' lifetime at GNOME (see Figure 4.11). Most of the time the commit rate per month is more or less constant between the projects, as most project names are barely readable because of their small size and low color intensity. The topology of this graph is often an indicator for a translator as it is almost impossible for a single person to do development work on so many different projects at a time.

As a first example of the Involvement Distribution View at project level, we choose the F-Spot project. In Figure 4.12 we can spot four main actors: Ettore Perazzoli, Larry Ewing, Stephane Delcroix, and Ruben Vermeersch. Ettore is the main initiator lauching the project end of 2004. Directly after, he left the project and Larry Ewing takes over. Stephane Delcroix enter the scene in

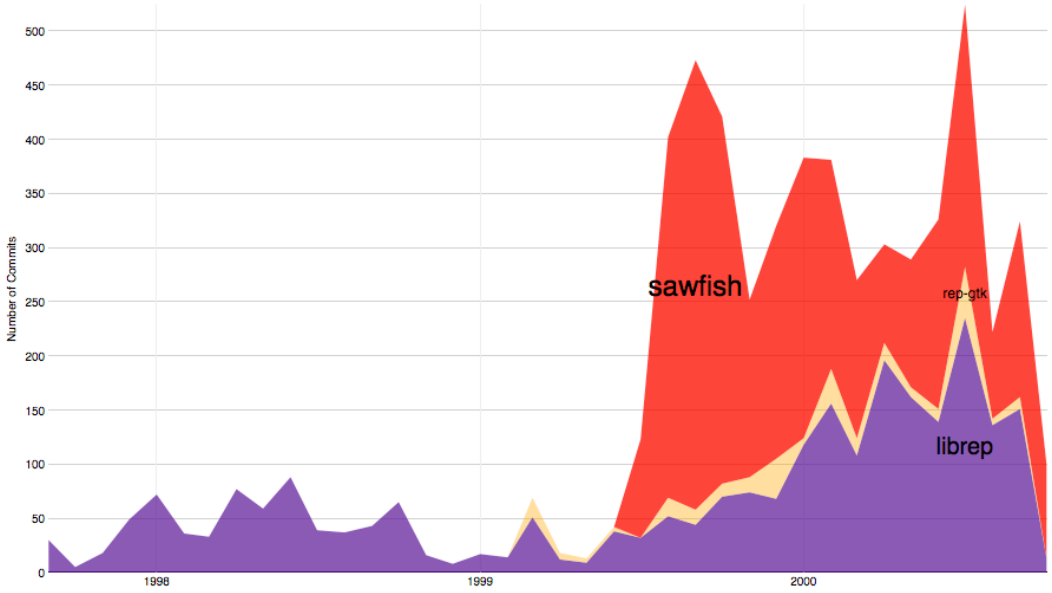

Figure 4.10. Involvement Distribution View of John a contributor of the GNOME ecosystem (y-axis: NOC, x-axis: Date at monthly basis, color: project)

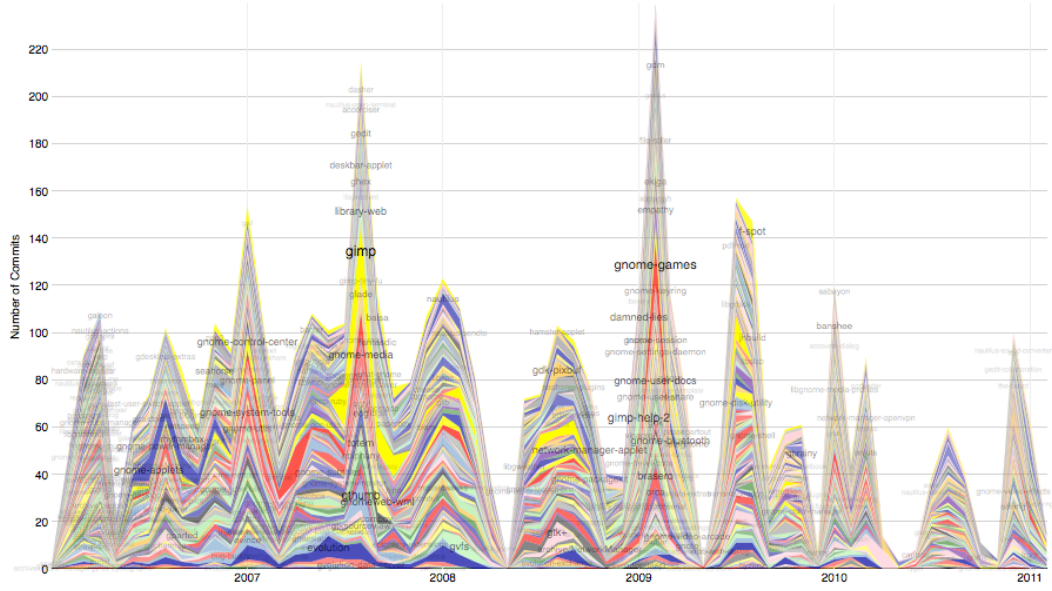

Figure 4.11. Involvement Distribution View of Daniel Nylander a contributor of the GNOME ecosystem (y-axis: NOC, x-axis: Date at monthly basis, color: project)

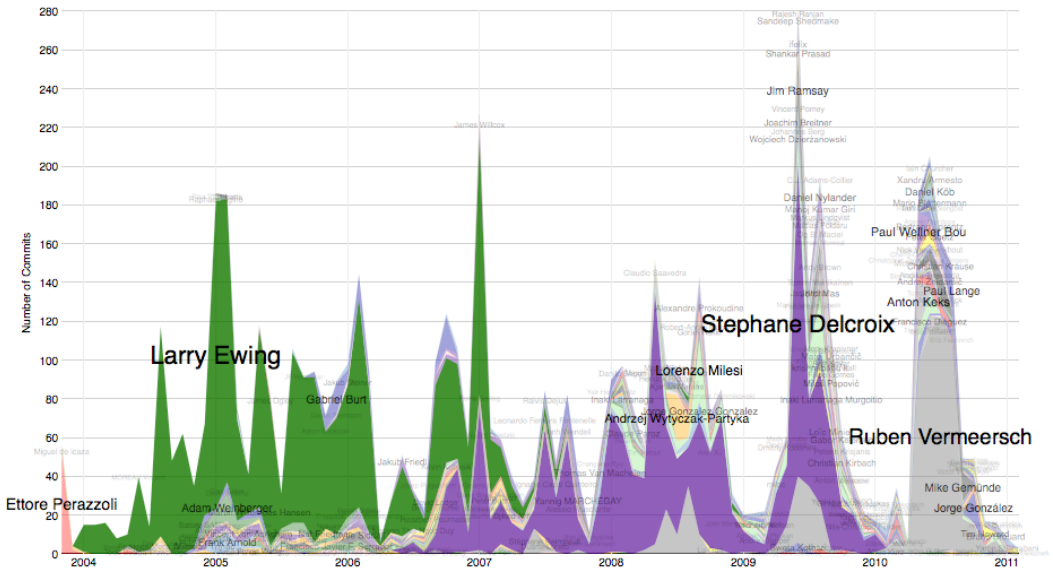

Figure 4.12. Involvement Distribution View of the F-Spot project within the GNOME ecosystem (y-axis: NOC, x-axis: Date at monthly basis, color: project)

2006 and slowly becomes the main contributor within this project until he left beginning of 2010 and Ruben Vermeersch takes over. We can observe smooth takes overs between the last three main contributors, which is an indicator for a good knowledge flow within F-Spot. It also shows that there are periods when only a single person is contributing, which might be an indicator for a center of power. In other words, if the central person within a single period would leave the project, the knowledge would get lost.

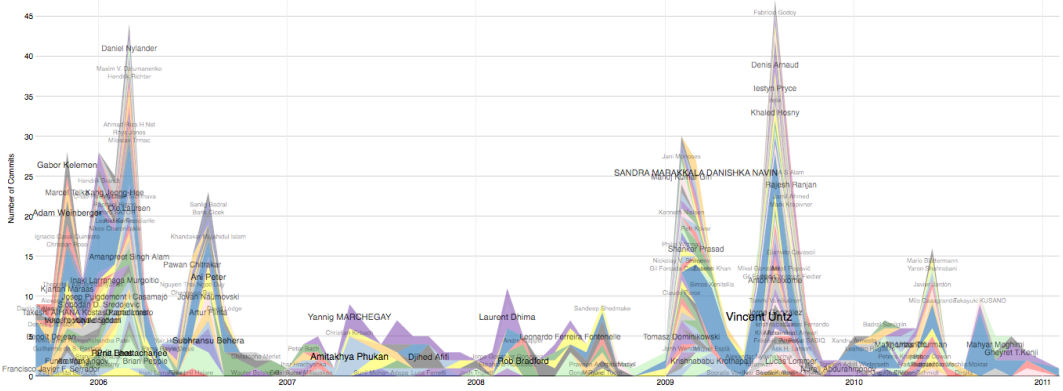

Figure 4.13. Involvement Distribution View of the Pessulus project within the GNOME ecosystem (y-axis: NOC, x-axis: Date at monthly basis, color: project)

A project, for which the contributors are more tightly connected, is illustrated by the Pessulus

project in Figure 4.13. This graphs shows no period where only one person is the center of power, which is an indicator for a safe communication structure as knowledge is less likely to get loss, after a single person leaves the project.

Discussion

This view provides an overview of the contributors involved in a project or the projects a contributors has committed to. But it also reveals the knowledge flow within a project and eventually whether a contributor tends to translation work or not. The analysis of a single user's affection to translation or development work, or both, is supported by the Expertise View in Section 4.4.3.

This view makes it is easy to spot "outliers" that have a high number of commits compared to the others. The main drawback is the limitation of the stacked area chart: If a project has many contributors or alternatively a contributor worked on many projects, the single items illustrated in the stacked area chart cannot be clearly differentiated. To weaken this limitation we added a select interaction, that enables to focus on a single project by clicking on it, or by searching for an entity by entering its name in the search panel. In addition, we provide a slider to enlarge the graph, so that it is possible to spot details which would not have been possible otherwise.

4.4.3 Expertise View

Goal	Illustrate the contributor's expertise or a project's main language basis.	
Abstraction Level	Project or Contributor Entity Level	
Stakeholder(s)	Project Manager, Contributors, Quality Assessment Team, and Researchers & Explorers	
Metrics	**Metric Name**	**Metric(s) Used**
	x-axis	Date
	y-axis	NOF
	width	-
	height	-
	color	File extension

General Idea

Extract a contributor's expertise based on the number of files he changed with a specific extension. This way, we can conclude whether he is a developer with an affiliation to a specific language (*e.g.,* C, Python, *etc.*), whether he is a translator, or whether he is both, or whether he changed the interest field over time, or whether he is a designer.

If the entity at hand is a project, we can identify the project's basic nature: manual/documentation, tool, or a graphical project based on the file extensions that have been changed more often. A graphical project provides for example the icons and themes of the software systems.

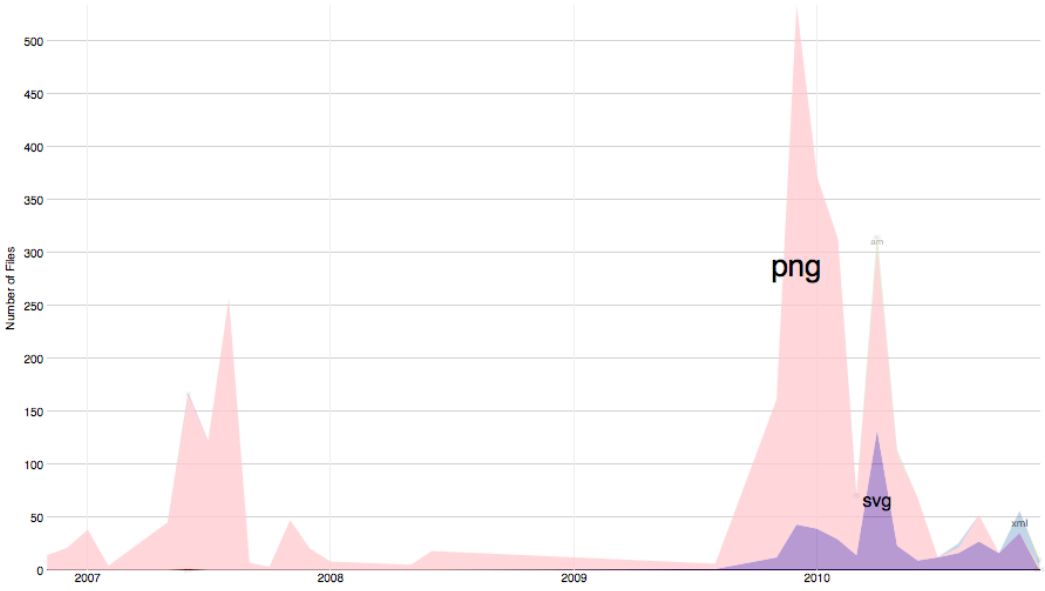

Figure 4.14. Expertise View of Lapo Calamandrei a Designer of the GNOME ecosystem (x-axis: NOF, x-axis: Date at monthly basis, color: file extension)

Construction

Same as the Involvement Distribution View, the Expertise View is drawn as a stacked area chart with the dates at monthly basis on the horizontal line and the number of file changes on the vertical line. Every stack represents a different file extension and has one of the eleven specified colors at random.

Examples

GNOME. Figure 4.14 shows the expertise graph of Lapo Calamandrei, a contributor of the GNOME ecosystem. During his lifetime in the ecosystem he mostly added, changed, or deleted .png (pink) or .svg (purple) files. In other words, we can conclude from this view that he is a designer within GNOME.

Figure 4.15 shows the extensions of the files that Daniel Nylander changed during his lifetime. We can observe that he mainly changed .po files (blue). As .po is the extension for translation files within the GNOME ecosystem, we can conclude that Daniel is a translator, which supports our expectation based on Figure 4.11.

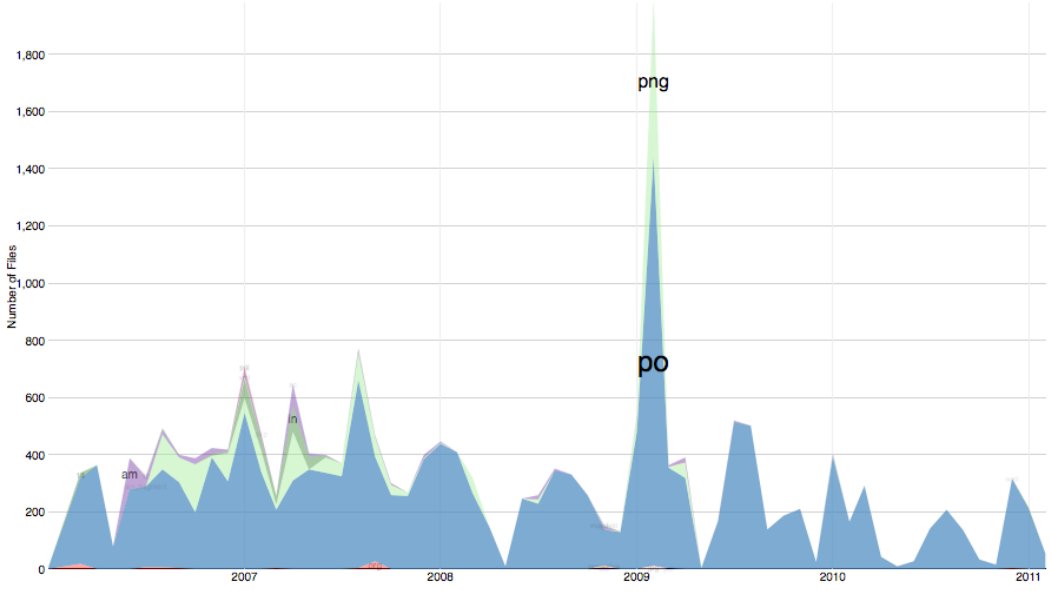

Figure 4.15. Expertise View of Daniel Nylander a Translator of the GNOME ecosystem (x-axis: NOF, x-axis: Date at monthly basis, color: file extension)

Michael Natterer, is a developer in the GNOME ecosystem. We come to this conclusion by looking at his Expertise View in Figure 4.16. He mainly changed .c (green) or .h (light gray) files.

Not all contributors can be classified in a single category. There are contributors who do different types of work, *e.g.,* translation and development. Matthias Clasen in Figure 4.17 is such a candidate. He was initially a C programmer (blue) in 2001 before he started doing translation work (lightgreen) in 2004.

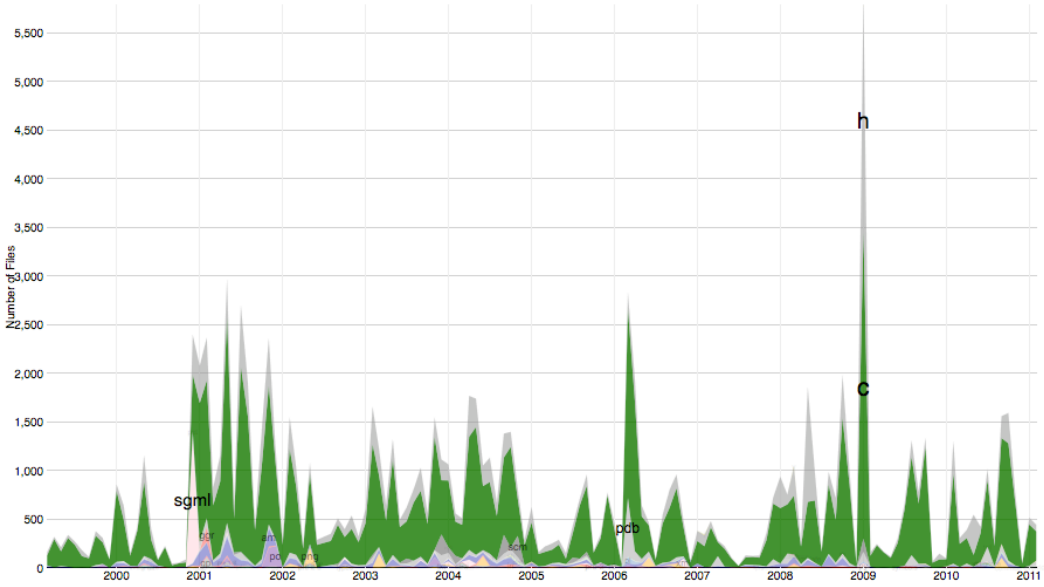

Figure 4.16. Expertise View of Michael Natterer a Developer of the GNOME ecosystem (x-axis: NOF, x-axis: Date at monthly basis, color: file extension)

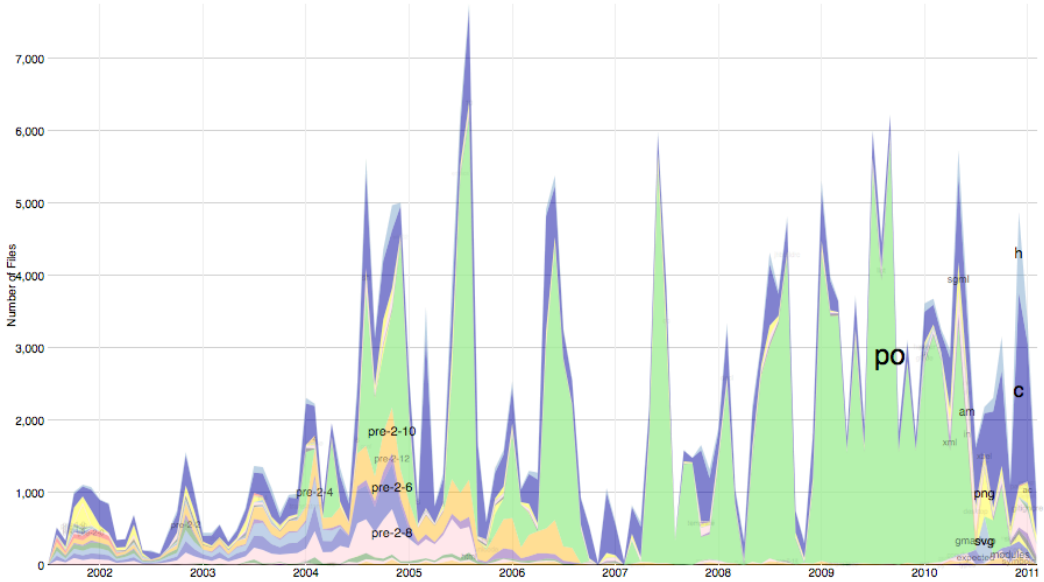

Figure 4.17. Expertise View of Matthias Clasen an Allrounder of the GNOME ecosystem (x-axis: NOF, x-axis: Date at monthly basis, color: file extension)

Discussion

This view allows us to depict the affiliation and expertise of a contributor in more detail based on the number of times he has changed a certain file type. We can differentiate between translators,

developers, allrounders or designers.

For projects, we can identify the project types: documentation, software system, and projects providing graphics and themes.

A problem which this view shares with the Involvement Distribution View in Section 4.4.2 is the problem of stacked area charts, that if many files with different extensions have been changed, it might be difficult to differentiate. The same facilities as for the involvement view (*e.g.,* slider, select event and search by file extension) are also provided for this view to overcome this problem.

4.5 Conclusion

In this chapter we introduced a catalogue of different predefined views. We described how they are constructed, for whom they might be of interest and what information can be revealed with them. Even though we used basic metrics we were able to depict different circumstances at ecosystem level:

- Identifying starters, youngsters, active and death projects and contributors, as well as project prototypes or pass-by contributors
- Spotting the most active projects or contributors relative to their age
- Pointing out the most popular and extreme projects
- Differentiating between contributors that have an affinity to translation work or alternatively are developers or allrounders

At entity level we could depict:

- contributors of project and, vice versa, the projects of a contributor at monthly basis, which can be an indication about the contributors' profession within the ecosystem
- the contributor's expertise (*e.g.,* developer, translator, designer or allrounder) or the project's language or file basis.

Every visualization contains only a limited number of information. Combining the different views, allows us to reveal information that would not be visible by looking at a single one. In the following chapter, we illustrate, on different examples, how we can make use of these predefined views to reveal valuable information about the evolution process and about the position of a single or a group of entities within the ecosystem.

Chapter 5

Telling Stories of the GNOME Project with Complicity

Visualization plays a central role in the analysis of software ecosystems. We have introduced Complicity as a tool support, which implements the catalogue of different views explained in depth in the previous chapter.

Every depicted view focuses on a single problem and so visualizes only a limited amount of information. Combining the different views on the other hand, makes them a valuable source to analyze the evolution of ecosystems.

In the following, we evaluate the different views, by combining them in stories about the GNOME super-repository with the aim to reveal insights into the changing process of this ecosystem, its projects and contributors. GNOME is a desktop environment for GNU/Linux/Unix composed of many free and open-source software systems. It was created in 1997 by two students, and since then it has grown in popularity. We show how Complicity can support software analysis at ecosystem level.

5.1 Bottom-up Approach

Using a bottom-up approach, we first analyze a single project regarding its activity and community support before moving to the ecosystem level and trying to reveal some patterns in the contributors' affection to either translation or development work. In a second example, we analyze a contributor's activity and expertise before examining at ecosystem level how he affected the GNOME project. In a first example we focus on Nautilus, GNOME's default file manager.

5.1.1 The Nautilus project and its impact on the ecosystem

Project Activity Diagrams View are a good starting point to get a first idea of Nautilus' evolution beyond the basic information available in the project's details. They illustrate the project's daily activity comparing six different metrics: number of commits, number of contributors, number of lines added and removed, difference between number of lines added and removed, and number of files changed. We use an area chart for all activity metrics, except the line-difference metric, which is drawn as a line chart. All of them have the time on the x-axis and one of the six metrics on the y-axis. The number of lines added (in green) and removed (in red) are drawn in a same diagram for better comparison. These diagrams allow us to identify different phases of a project. In Figure 4.8 we observe that Nautilus was created in 1997 (birth), when also GNOME was created. Between 2000/2001 its popularity increased in terms of contributors, in number of commits per day and in number of lines added/removed (first major release). Afterwards the number of commits and contributors decreased and stabilized. Also the number of lines added/removed equalized (maintenance). In 2010, the number of commits increased again until Feb. 2011 (new major release).

Beside the different phases, we observe that there is continuous development taking place almost every day, which is an indicator for an active development team. The next step is to get familiar with its contributors: Who worked, when, and for how long on the project? This information is presented by the contributors' *Involvement Distribution View* in Figure 5.1.

Contributors' Involvement Distribution View. This view gives information about the number of commits, but it does not allow a comparison of different metrics at the same time. Instead, it provides a more detailed view on the people who contributed to the project over time, so that the drivers of the different phases can be identified. This information is crucial to understand the flow of knowledge (Is there a center of power? Are there smooth take overs?), which is a good basis for a project to evolve well and have a long lifetime. For this visualization, we use a stacked area chart using colors to identify different contributors. The size, the density and the position of the contributor's name is based on the maximum number of commits a person has done at once. Clicking on a single contributor hides all the others, so that the contribution timeline of a single person can be better analyzed. This view shows the evolution on a monthly basis. In Nautilus' first major release phase (see Figure 5.1), Ramiro Estrugo (light pink), as well as Andy Heitzfeld (yellow), Darin Adler (red), and John Sullivan (purple) are the main initiators of the project. Darin Adler remains the main driver at the beginning of the maintenance phase until the beginning of 2002, when Alex Larsson (blue) takes over. He remained the driver for almost the whole maintenance period. Only by 2008 a new actor enters the scene, Cosimo Cecchi (gray), who takes over the project. The above diagrams contain information about a single project, how it and its community evolve over time. As Nautilus is only one software system of the GNOME project, it is important to analyze how it impacts the ecosystem.

Affectional Bond View - Ecosystem Diagram. This view shows the contributor distribution at

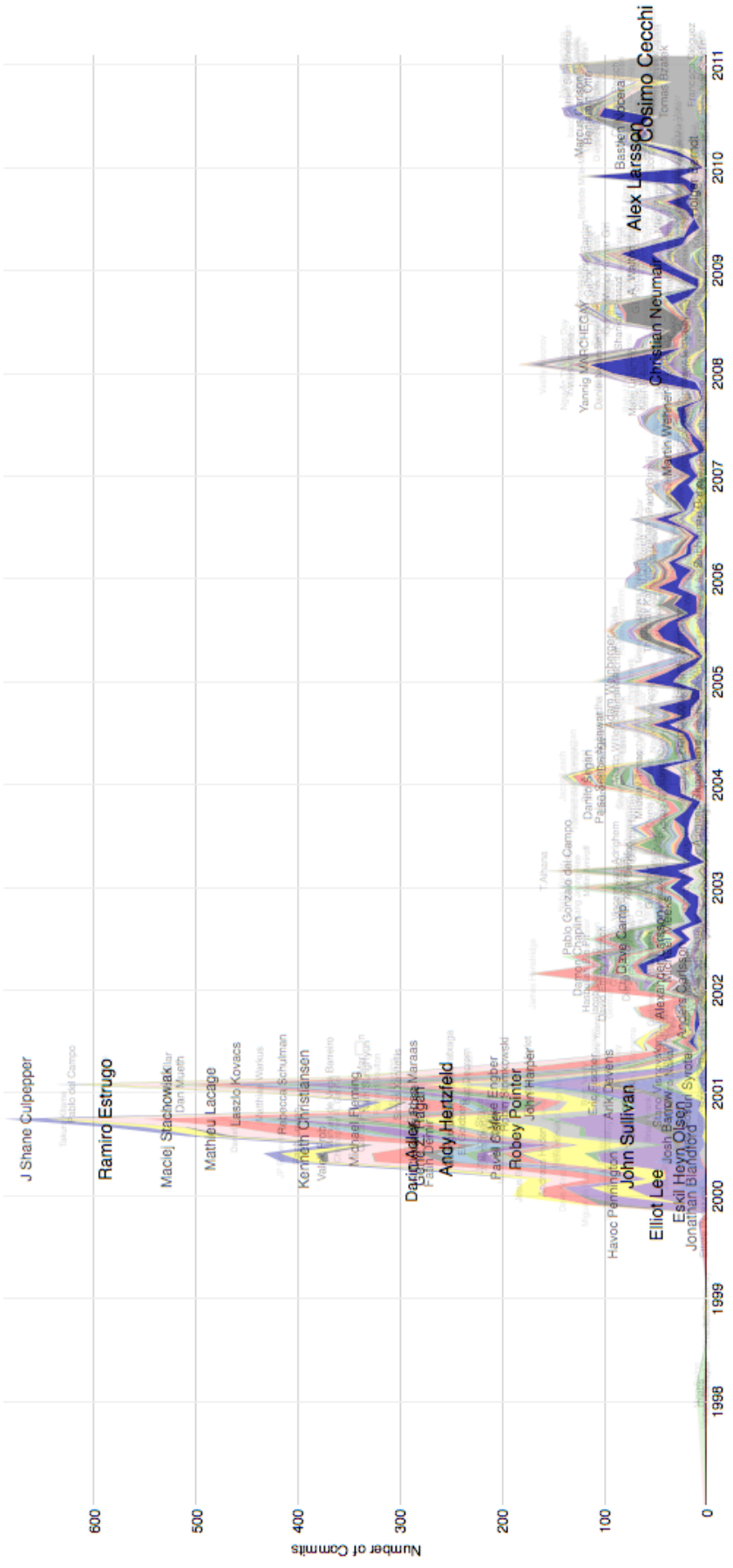

Figure 5.1. Contributors' Involvement Distribution View of the Nautilus project

ecosystem level, with number of commits on the x-axis, and number of project on the y-axis and as color metric. The width and height of each shape is defined by a contributor's lifetime in days. This view can be used to visualize all contributors of the entire ecosystem or filters can be applied to show only the contributors of a specific project.

This visualization presents a piece of information that cannot be revealed at project level from any of the two views described above: the contributors' general affectional bond to either development or translation work. In this graph the contributors are split into these two groups under the assumption that people who contributed a lot but only to a relatively small number of projects are likely to be developers. Conversely, people who committed less often but to more projects are likely to be translators. This results into the following distribution: people located under a logarithmic-like curve are defined as developers and the ones placed above an exponential-like curve are considered to be translators.

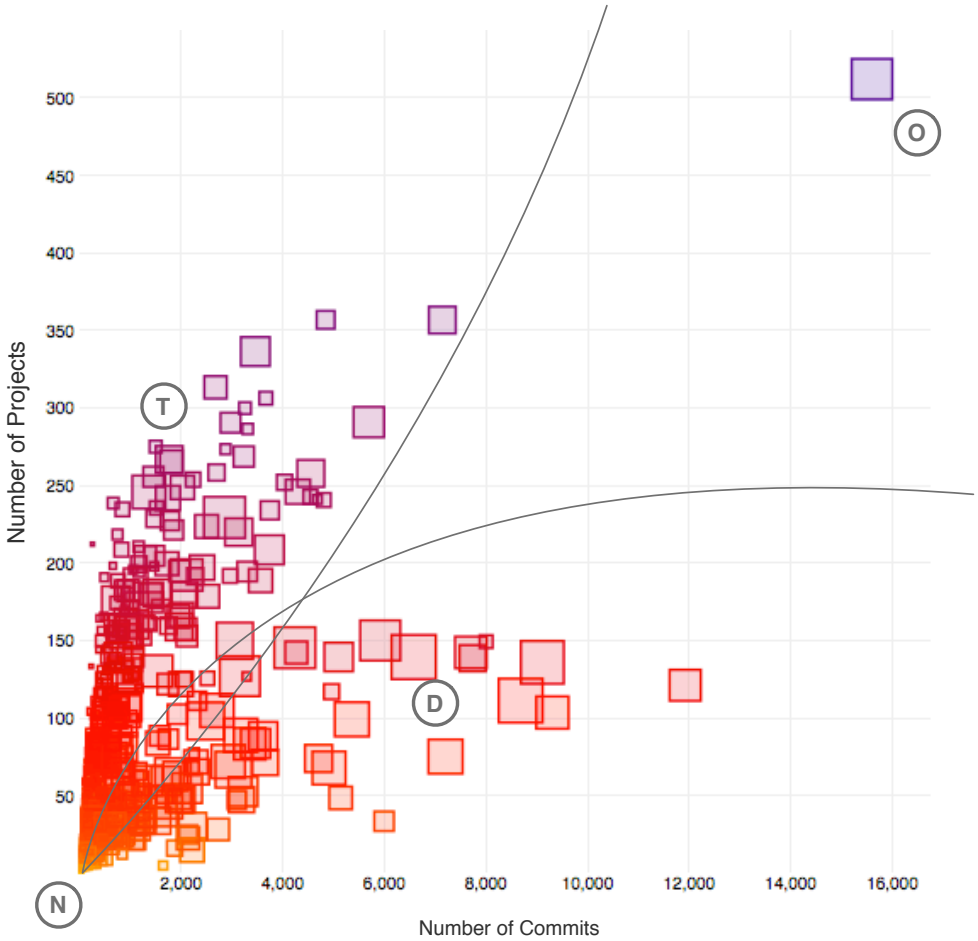

Figure 5.2. Affectional Bond View at ecosystem level. Each square represents a committer, where the x position maps the number of commits, the y position and the color the number of projects, and the size the lifetime in number of days (T: translators, D: developers, O: outlier, N: no man's land)

Figure 5.2 illustrates the contributor distribution of the Nautilus project. It reveals that many people have been working on the project over time and that the distribution between developers (marked as D) and translators (marked as T) is roughly equalized. The contributors located within the high density bottom left part (marked as N) might not be clearly differentiable. It seems that developers need a longer lifetime (larger boxes) in order to be clearly differentiable from the translators, who have a more varied lifetime (collection of large and small boxes). The outlier on the top right (marked as O) did a huge amount of changes (positioned to the right) and contributed to many projects (positioned to the top) over a long lifetime (relatively large box). O is Kjartan Maraas, both a developer and a translator within the GNOME project.

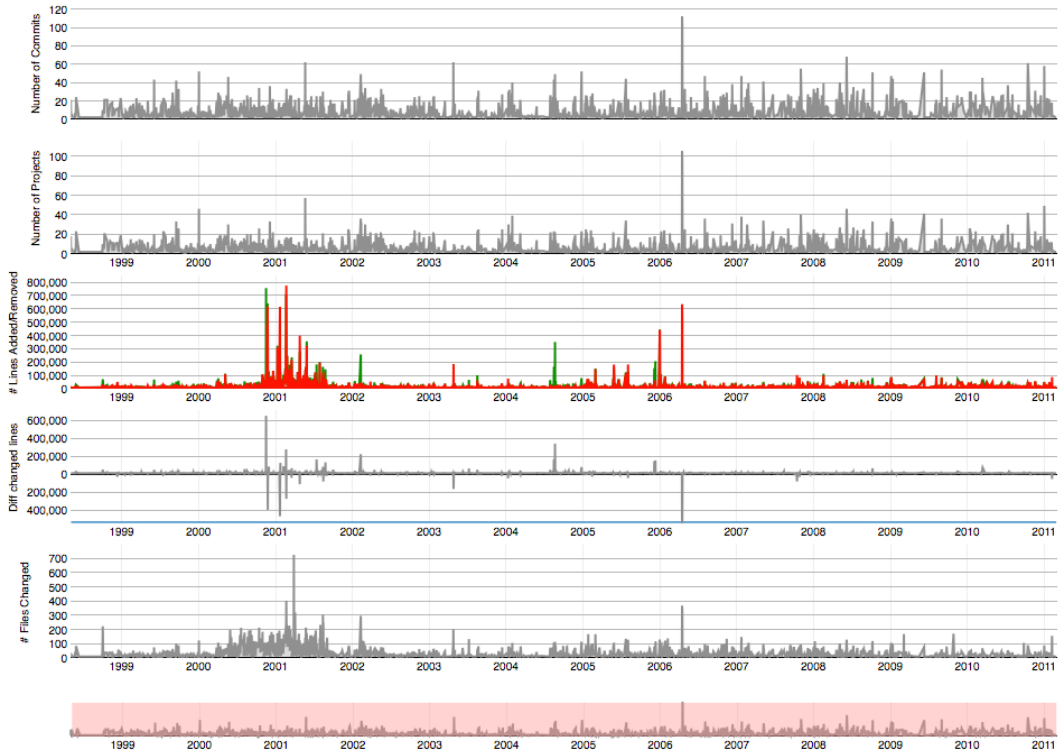

Figure 5.3. Activity Diagrams View of Kjartan Maraas

5.1.2 Impact of the contributor Kjartan Maraas on the ecosystem

As a second example we have chosen Kjartan Maraas as the contributor for the bottom-up analysis. Since 1998 he has been working on 499 different projects, having done more than 15,000 changes by February 2011. During 4,689 days (*ca.* 13 years) he had an average rate of three commits per day.

We know when a contributor did his first and last commit and we can calculate his average commit rate. However it is not possible to know whether he has been active all the time or there have been some gaps of inactivity during his lifetime. This information can be extracted from the developer Activity Diagrams View.

Developer Activity Diagrams View gives an overview of the contributor's daily activity within an ecosystem. It is constructed the same way as the Activity Diagrams View at project level with the only difference that the metric number of contributors is replaced by number of projects. In Figure 5.3 we observe that Kjartan Maraas has been continuously active with only one exception of few months. Particularly interesting in his past activities is a period of time in 2001, during which he added or removed up to almost 800,000 lines, performed only few commits and changed up to 700 files. Another interesting short period (one day), can be spot in 2006, where he did more than 100 commits on more than 100 projects changing more than 600,000 lines on 300 different files. A piece of information missing in this view is on which projects Kjartan Maraas did these changes. To this aim, we have introduced the project Involvement Distribution View, presented next.

Project Involvement Distribution View. This view shows the projects to which a person committed monthly. As for the contributors' Involvement Distribution View, we use a stacked area chart with number of commits on the y-axis and time on the x-axis. Figure 5.4 shows that Kjartan Maraas has been active in many different projects.

The number of projects he worked on increased after the first few years, which can be revealed by the fact that the colors representing the projects in this view become less differentiable towards the right part of the visualization. Also the color intensity and size of the labels become more similar, which means that he committed with similar rate to these projects. At this point of our analysis it might be interesting to understand how it is possible for a single person to do so many commits and changes while working on so many projects. To answer this question we devised the Expertise View.

Expertise View. This visualization yields information about a contributor's expertise based on file extensions. It is drawn as a stacked area chart, counting the number of files that has been changed within a month aggregated by their file extension.

Figure 5.5 shows the Expertise View applied to Kjartan Maraas. He is a translator and C developer, as he changed a large number of .po files (gray) as well as many .c files (green).

The combination of the three previous visualizations explains how Kjartan Maraas does so many changes in so many projects: He regularly worked on the GNOME ecosystem over a very long period. Exploring the projects' Involvement Distribution View, we see that he commits to most of the projects more than once but with a low monthly commit rate.

Knowing that beside being a translator, which makes it possible to work on many projects simultaneously, he is also a developer, we can conclude that he is likely to be a maintainer trying to fix bugs on different projects. Indeed, by checking his profile on LinkedIn[1] we found out that he has been member of the Bugquads at GNOME for almost ten years beside being a member of the release team and of the GNOME foundation.

The views at contributor level provide details about a single person's activity level and expertise. However, to better understand how a contributor affects the GNOME ecosystem, we need visualizations at ecosystem level.

The **Activity Fire View - Ecosystem Graph** is constructed with number of commits on the y-axis, lifetime on the x-axis, and number of projects as width, height and color of the boxes. It illustrates the distribution of the contributors according to their activity over their lifetime in the GNOME project.

Kjartan Maraas is an outlier compared to the contributors of Nautilus (see Figure 5.2) and within the GNOME ecosystem (marked by an arrow in Figure 5.6). He performed by far most commits

[1] http://www.linkedin.com/in/kjartanmaraas

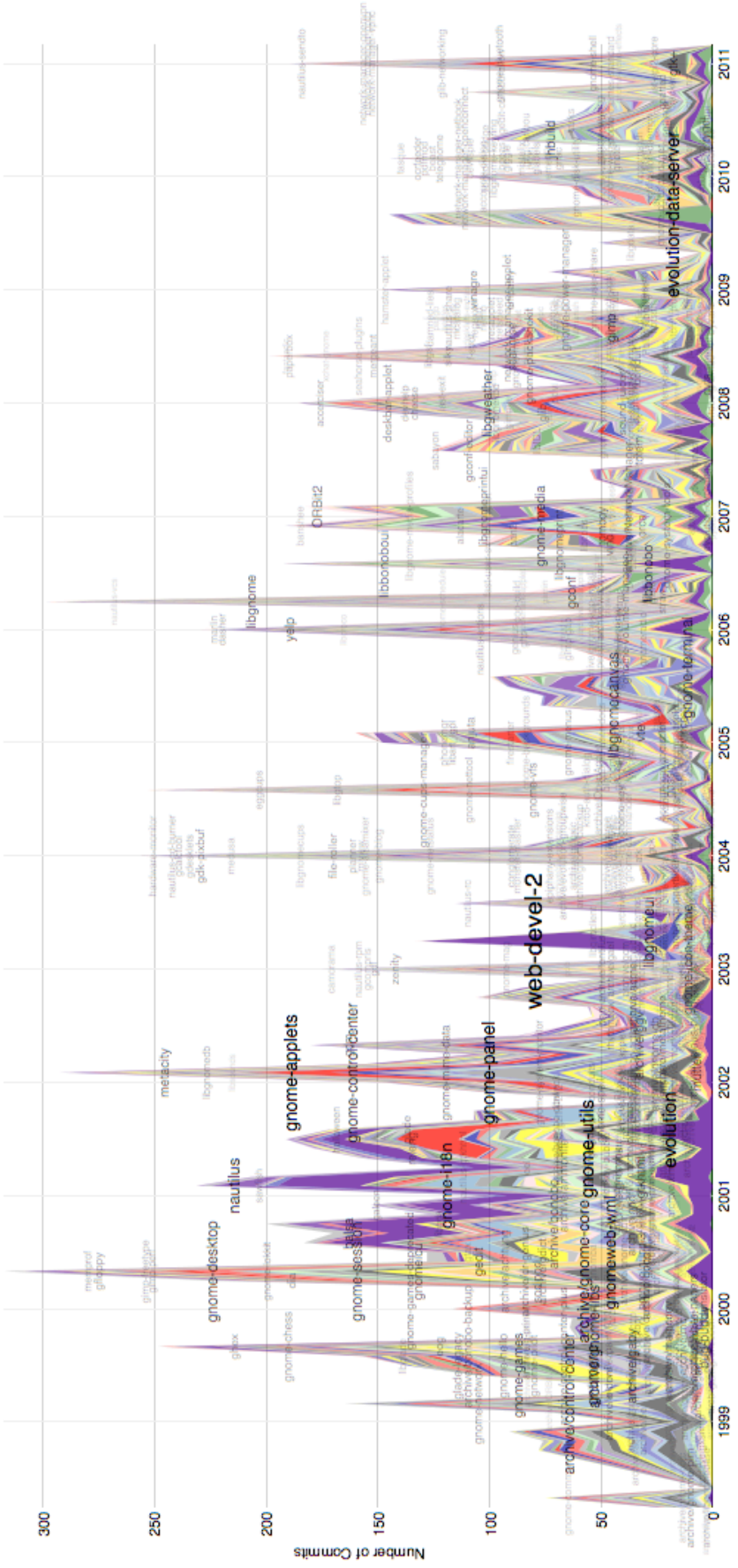

Figure 5.4. Projects' Involvement Distribution View of the Kjartan Maraas

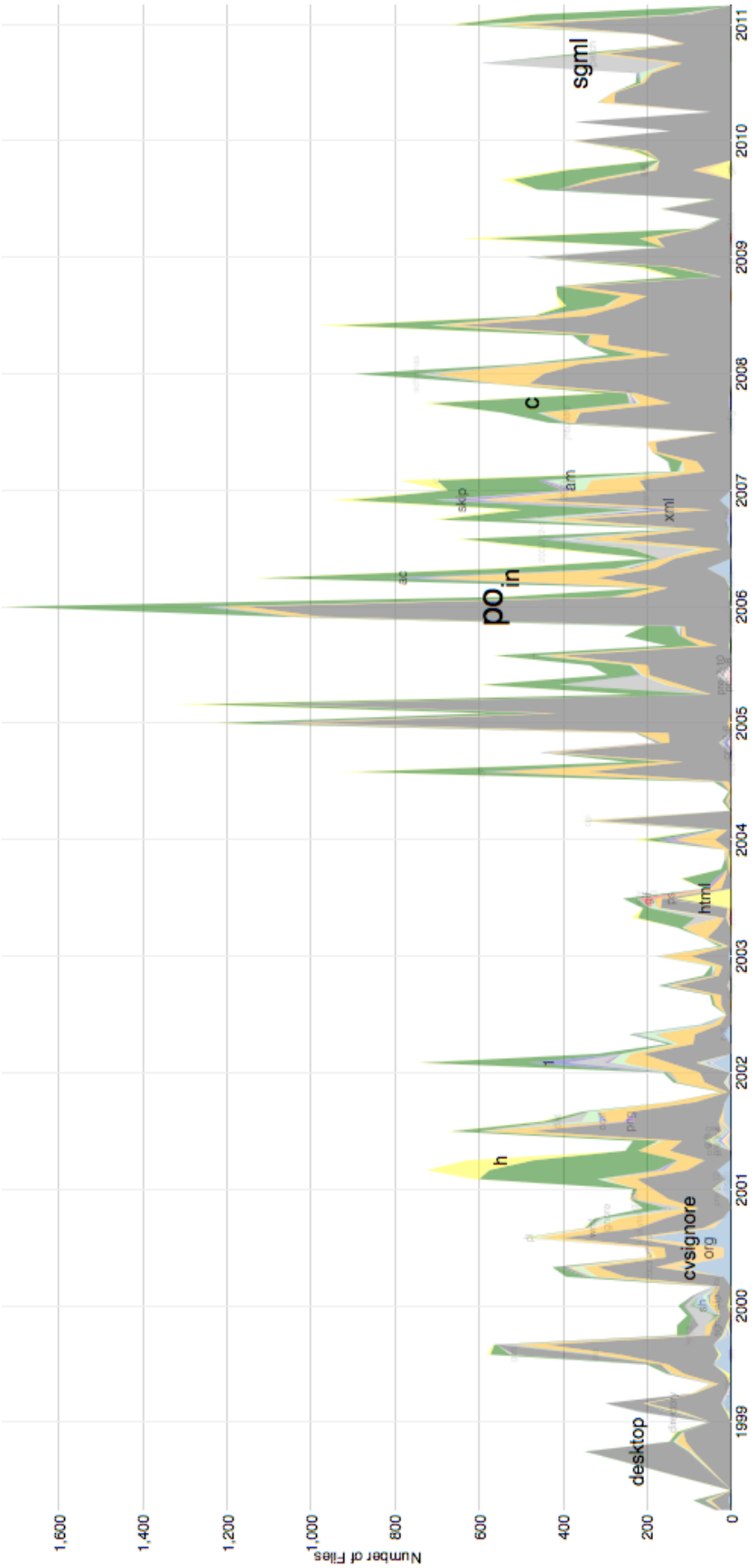

Figure 5.5. Expertise View of Kjartan Maraas

(positioned to the top) and worked on more projects than anybody else (large box) in the ecosystem. This might be related to his long lifetime (positioned to the right) at GNOME but also to the fact that he is both a translator and a developer.

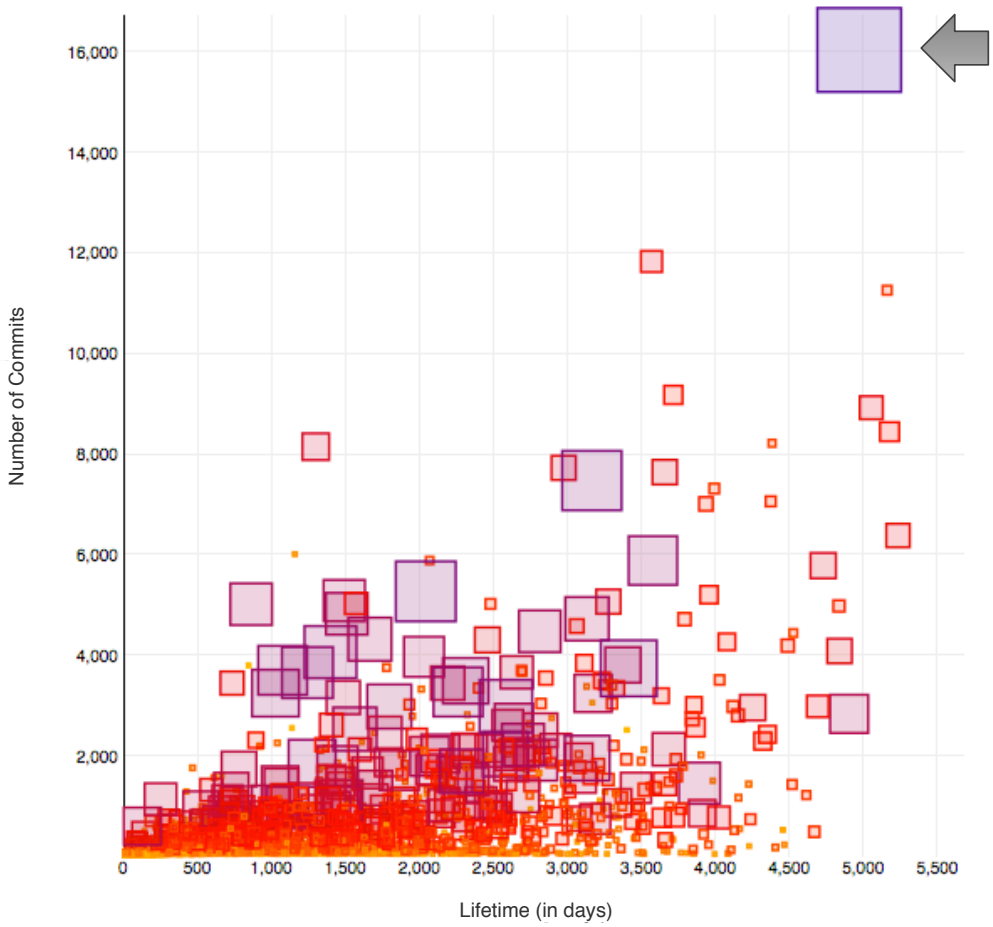

Figure 5.6. Kjartan Maraas (marked by the arrow) compared to all other contributors of the GNOME project within the Activity Fire View

Projects do not always have long lifetimes. As consequence, Kjartan Maraas worked on many projects but probably not on all at the same time. The following view provides clarification by illustrating the projects' lifetime.

Projects' Lifetime View - Ecosystem Graph. This view shows the projects' life duration. The width and height of each box are defined by the total number of commits and the total number of contributors, respectively. The color represents the different project categories.

Figure 5.7 shows that many of the projects Kjartan has contributed to have died (D). Most of them are colored red (archived), green (others), or orange (desktop). The projects within this area placed at the diagonal line (P) are likely to be prototypes or projects that have been integrated into other

projects. They have a very short lifetime and die almost as soon as they are created. Only the projects on top (A) are still alive, and actively under change. Maraas never worked on all projects at the same time as some of them died before others were created.

Figure 5.7. Projects' Lifetime View showing only the projects Kjartan Maraas contributed to with A: active projects, P: prototypes, and D: dead projects (Color scheme: projects of category archived (red), others (green), desktop (orange), bindings (light blue), development tools (pink), and platform (purple))

5.2 Top-Down Approach

Using a top-down approach, we first analyze projects at ecosystem level. We select at random two projects with different life expectations, a veteran and a youngster, and compare both projects an their contributors against each other at ecosystem level. Then we analyze the subject projects' evolution and some of its contributors at entity level.

5.2.1 Veteran project Balsa versus youngster project TBO

As a first example of the top-down approach, we have chosen two projects of different age group, veteran versus youngster.
The *Balsa* is a mail client. It was created in January 1997 and is one of the first projects created within the GNOME ecosystem (veteran). It is still active after more than 14 years and counts more than 200 committers so far.
TBO is an easy and fun program to draw comic and make funnier presentations. It was created in December 2009 (youngster) and is nowadays still active. After slightly more than one year of activity it counts 21 committers.

Projects' Lifetime View - Ecosystem Graph. This view shows all projects within the GNOME ecosystem. They are distributed according to their first and last change. The size of the box is defined by the number of commits (height) and number of files (width) and the color represents the number of contributors (orange to purple).

Figure 5.8 supports the above statements. It shows *Balsa*, as one of the veteran projects in the top left corner, which is still active and *TBO* positioned to the top right corner, as a youngster project, which is still active since end of 2009. We can observe that the veteran projects that survive are likely to have a large number of contributors (color gradient from red to purple) and a large number of commits and/or files (large boxes). This phenomenon is likely due to their long lifetime.

Unfortunately, we cannot clearly spot the subject projects as they are overlapping with others. For this purpose, we use the Activity Fire View, which allows us to spot outstanding projects compared to others of the same age.

Activity Fire View - Ecosystem Graph. The view illustrates the outstanding projects according to their number of contributors (y axis and color of boxes). The width and height of the boxes is defined by the number of files and number of commits, respectively.

Generally Figure 5.9 shows that the number of contributors of a project is likely to increase linearly the older they get. Many projects have a quite short lifetime (intense color in the bottom left corner. This is due to the fact this area illustrates prototypes as well as new projects.

Balsa (on the right) has only a moderate number of contributors compared to the projects of a similar age. Compared to some projects of the same age, it has a moderate to small number of files and commits. *TBO* (bottom left) is on a good way to become an outstanding project relative to the projects of the same age. Two outstanding and quite old projects, *evolution* and *gtk+* can be spotted on the top right of the figure. Compared to most of the others in the ecosystem, they have a relatively high number of commits besides the high number of contributors.

This view illustrates that *Balsa* has many more contributors than *TBO*, which is most probable due to the age difference. What we do not know is the distribution of the contributors in each of the two projects based on their affiliation.

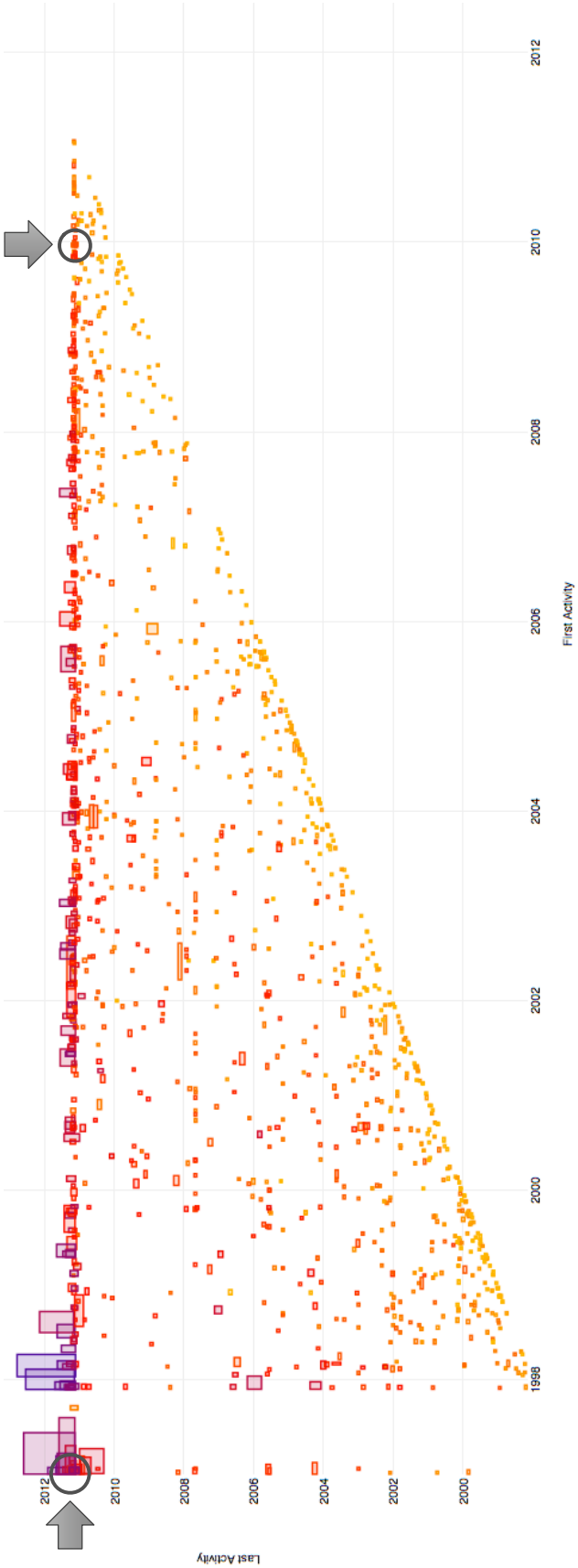

Figure 5.8. Projects' Lifetime with TBO marked on the top right and Balsa on the top left corner

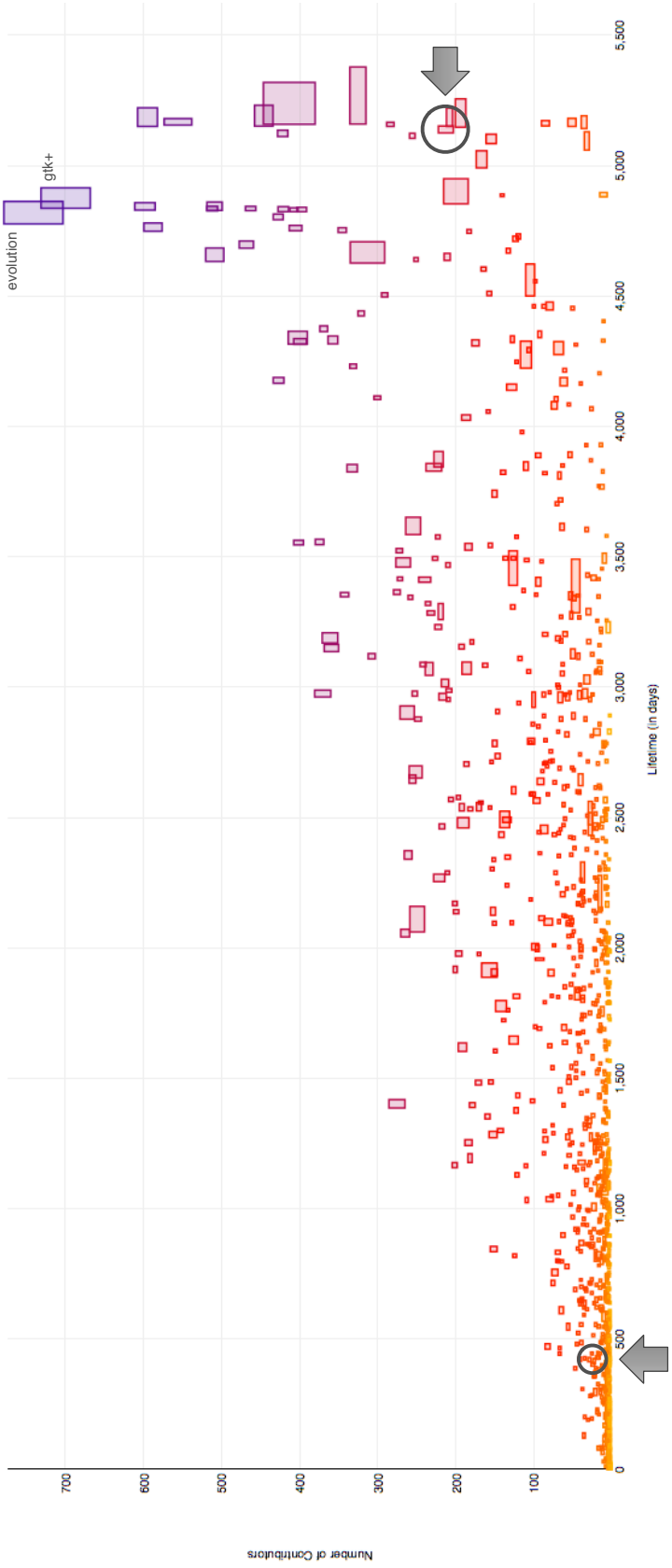

Figure 5.9. Projects' Activity Fire View with TBO marked on the bottom left and Balsa on the right (x: lifetime, y/color: NOD, width: NOF, height: NOC)

Affectional Bond View - Ecosystem Graph. The contributors of the two subject projects are distributed according to the number of commits on the x axis and the number of projects on the y axis is also used for the color of the boxes. The size of the box is relative to their lifetime. Figure 5.10 shows the Affectional Bond View of the contributors of *TBO* on the left and of *Balsa* on the right.

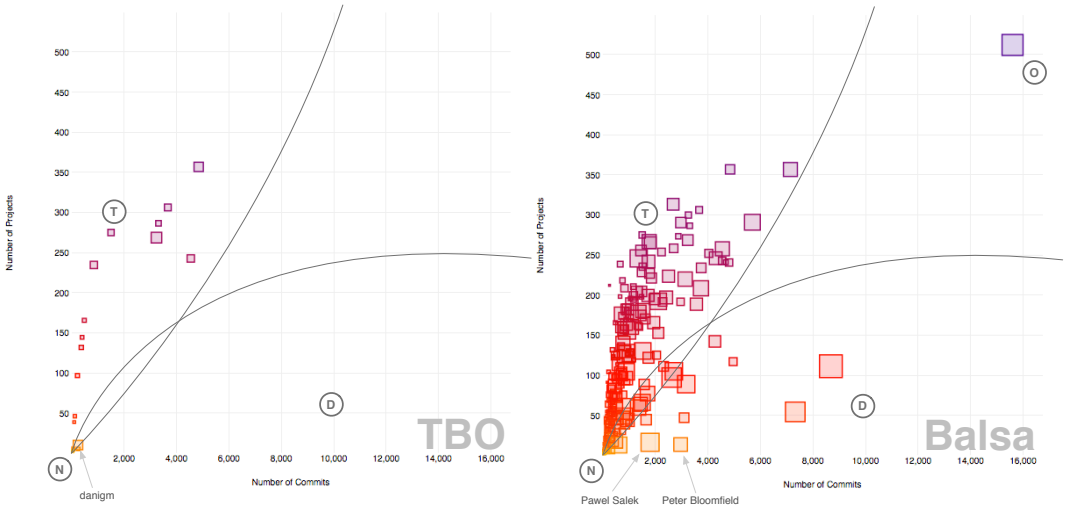

Figure 5.10. TBO's contributors' affectional bond (left) versus Balsa's contributors' affectional bond (right)

Despite the fact that *Balsa* has more contributors than *TBO* we can observe that both projects have a similar distribution of their contributors. In both projects the number of translators is much higher than the number of developers and all-rounders, respectively. Surprisingly is that the youngster seems to have no clear defined and experienced developer compared to the veteran project.

To get clarification about this observation, we require information about the nature of the project and its contributors. For this purpose, we use the contributors' Involvement Distribution View and the project's Expertise View at project level.

Project Expertise View - Project Level. It gives information about the project's language and file basis based on the most commonly changed file extensions. The Expertise View is introduced and described in detail in Section 4.4.3.

According to the Affectional Bond View our expectation would be that *TBO*'s main activity is translation work. By analyzing Figure 5.11, we can see that this is not the case. The most occurring file extensions are .svg, .c and .h. In other words, the basis of *TBO* is a C program that makes use of SVG images. Furthermore, we can observe that only recently (end 2010) they became active in translation work.

In order to find out whether this project has a specialist in development, we need information about *TBO*'s contributors. For this purpose we make use of the contributors' Involvement Distribution View, which we introduced and described in Section 4.4.2.

Contributors' Involvement Distribution View - Project Level. The contributors' Involvement Distribution View is a stacked area chart with number of commits on the y axis and data at monthly basis on the x axis. Each stack represents a different contributor.

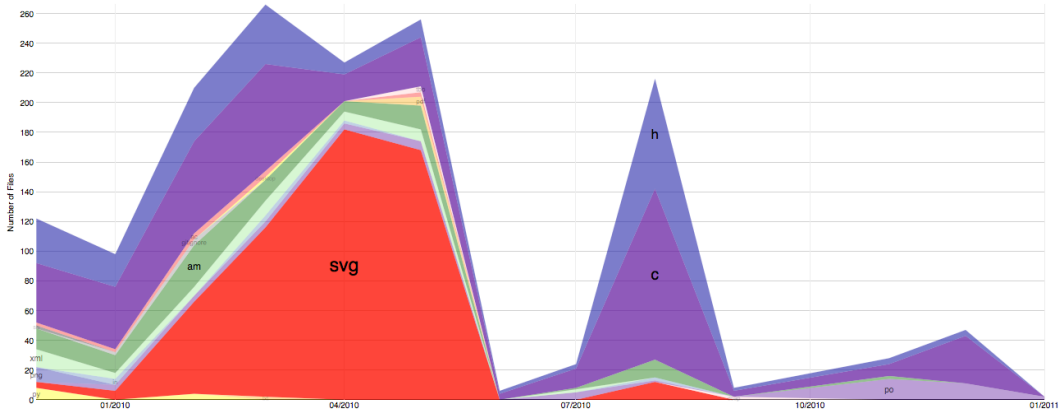

Figure 5.11. TBO's Expertise View

Looking at Figure 5.12 we can observe that with the increase in .po file changes at the end of 2010 (see Figure 5.11) also the number of different contributors increases. Additionally, we can spot a center of power, *danigm*, the main contributor over *TBO*'s entire lifetime.

By interactively checking Figure 5.10 we find *danigm* in the bottom left corner of the graph, the area in which developers and translators cannot be clearly differentiated. Comparing the two graphs, TBO's Expertise View and TBO's Involvement Distribution View, we expect *danigm* to be a C programmer and also having changed many .svg files in the past. However, to be sure about the nature of *danigm*'s work expertise within the TBO project, we consult his Expertise and Involvement Distribution View at contributor level.

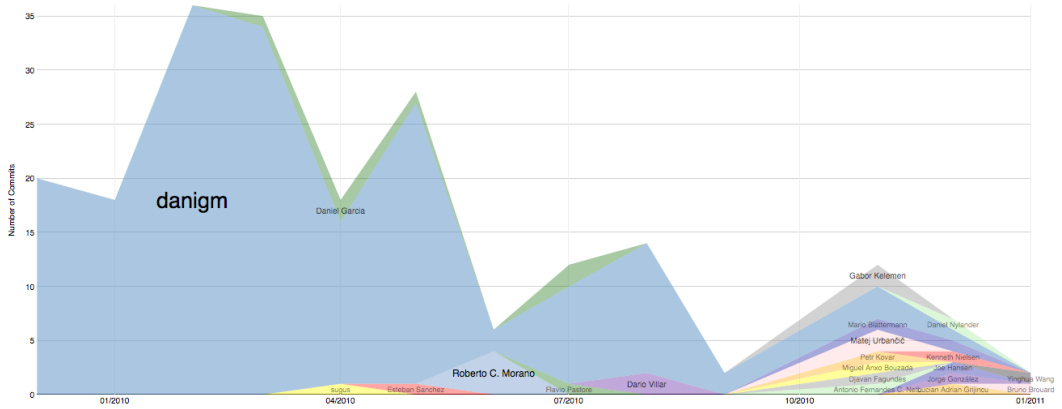

Figure 5.12. TBO's contributors' Involvement Distribution View

Projects' Involvement Distribution View - Contributor Level. Figure 5.13 illustrates clearly that *danigm* worked only on the *TBO* project with the exception of three last commits, which have been devoted to two other projects, evince and gtk+.

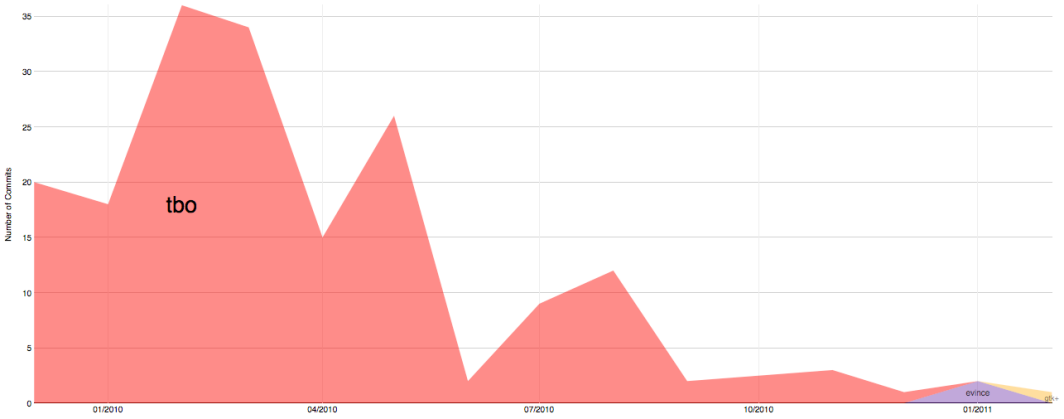

Figure 5.13. Danigm's projects' Involvement Distribution View

Expertise View - Contributor Level. Knowing that *danigm* only worked on the *TBO* project (with the exception of the last three commits) results in an Expertise View that visualizes only the files he changed on that project. This observation would not be possible otherwise. In other words, by looking at Figure 5.14 we now know for sure that he is a C programmer and SVG developer and that it is him who changed most of the files in the TBO project (see Figure 5.11).

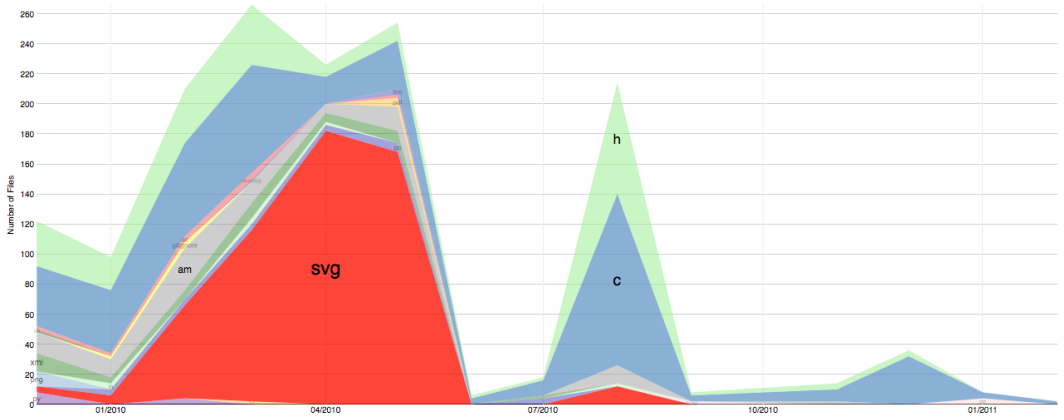

Figure 5.14. Danigm's Expertise View

Now that we analyzed the *TBO* project in depth, we now focus on the *Balsa* project and study it in details.

Expertise View - Project Level. Figure 5.15 shows that *Balsa*, similar to TBO, is a C program but different from TBO, it has a higher .po file changing rate. This is likely the reason for the large number of translators compared to developers in the Affectional Bond View in Figure 5.10.

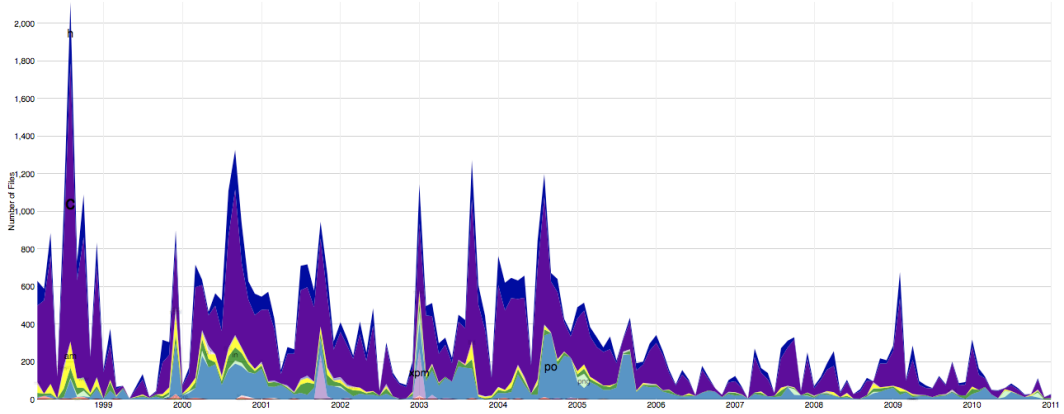

Figure 5.15. Balsa's Expertise View

Contributors' Involvement Distribution View - Project Level. As identified at ecosystem level, *Balsa* has a larger lifetime and a higher number of contributors then *TBO*. In consequence, the contributors' Involvement Distribution View is likely to have a more diverse structure. By analyzing Figure 5.16, we can clearly see that *Balsa*, different from TBO, does not have a single center of power over the entire project's lifetime. Instead, *Balsa* has a good knowledge flow as it has smooth take overs from one main contributor to another. The main actors in this view are: Stuart Parmenter, the initiator of the project in 1997; Peter Williams, a pass-by contributor who ensured a smooth take over from Stuart to Pawel Salek in 2000. Pawel remains active until today and is one of the main contributors together with Peter Bloomfield who joined the project in 2003.

From *Balsa*'s Expertise and Involvement Distribution View we could identify the language basis and the main drivers of the project but we cannot generalize only by looking at these views that all three experts are C programmers. For this purpose we make use of the Expertise and project Involvement Distribution View at contributor level.

Contributor's Expertise View - Contributor Level. Analyzing the main actors' expertise in Figure 5.17 (Stuart Parmenter), Figure 5.18 (Pawel Salek), and Figure 5.19 (Peter Bloomfield), we can conclude that all of them are mainly C programmers. It should be remembered that the colors in the stacked area charts are assigned randomly. Consulting the projects' Involvement Distribution View of the three in the following allows us to identify the projects, which they contributed to, and ensure that they also mainly worked on the Balsa project.

Projects' Involvement Distribution View - Contributor Level. Figure 5.20, Figure 5.21, and Figure 5.22 illustrate that Parmenter, Salek and Bloomfield are not only the main actors of the *Balsa* project but that they also focus their activity around this project.

Taking a look back to the Affectional Bond View, we can interactively spot two of the four main contributors of the *Balsa* project. Surprisingly, none of the experts within the GNOME ecosystem is highly active in the *Balsa* project, even though the project is quite old and generally popular in terms of number of contributors (see Figure 5.9).

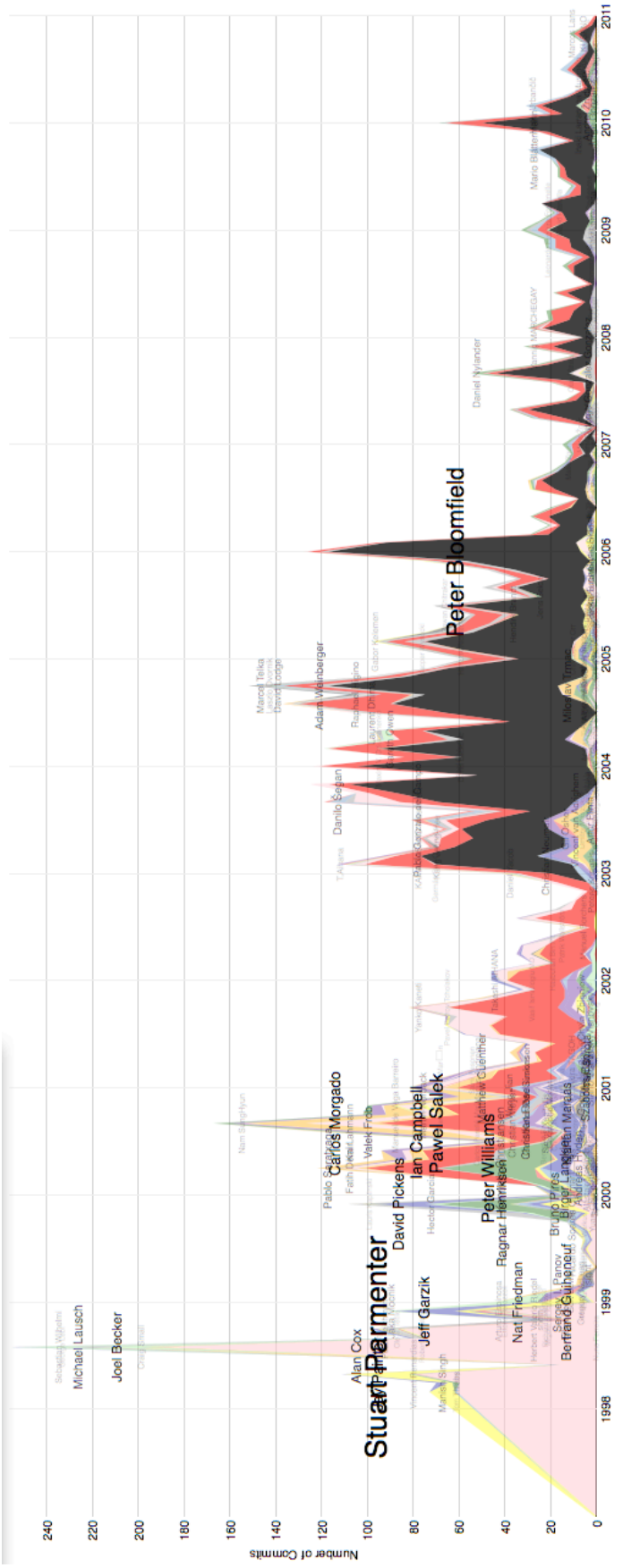

Figure 5.16. Balsa's contributors' Involvement Distribution View

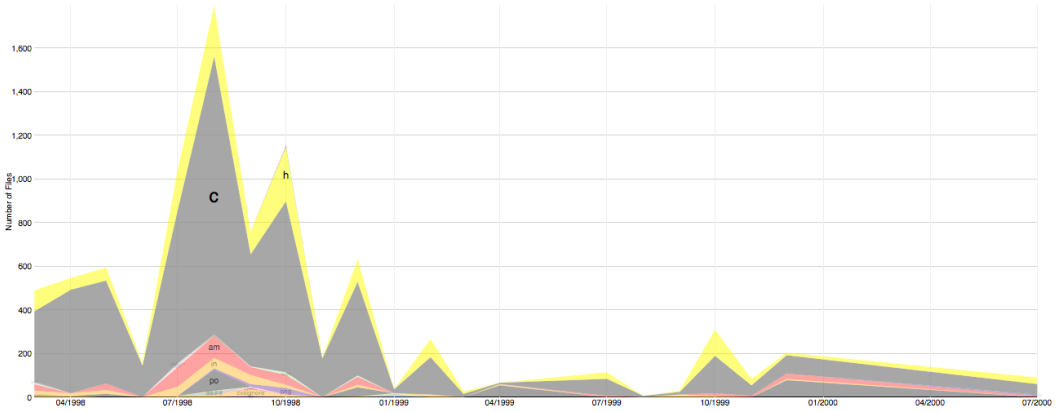

Figure 5.17. Stuart Parmenter's Expertise View

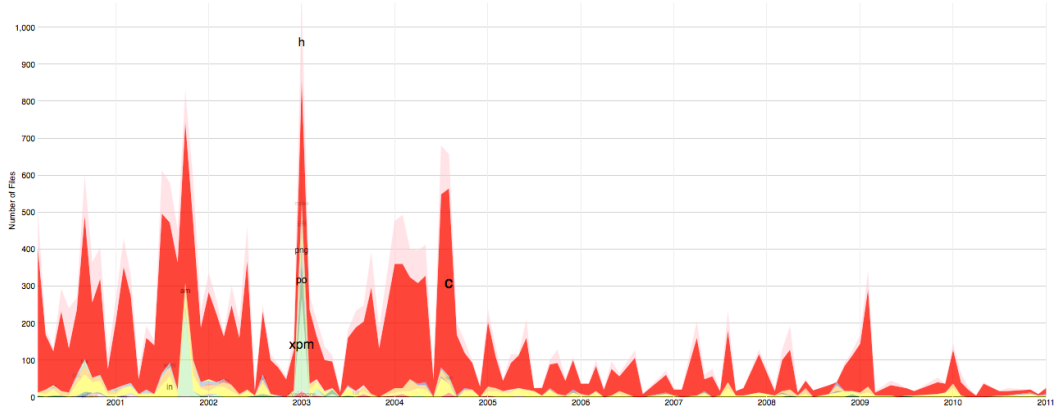

Figure 5.18. Pawel Salek's Expertise View

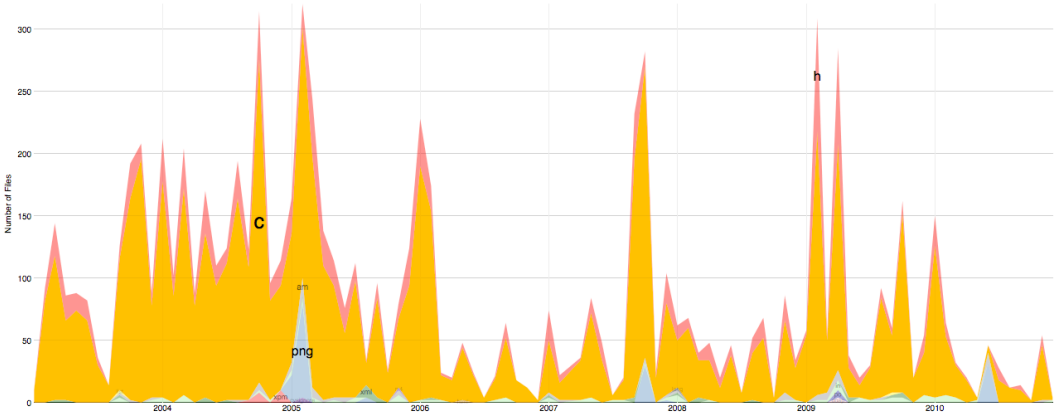

Figure 5.19. Peter Bloomfield's Expertise View

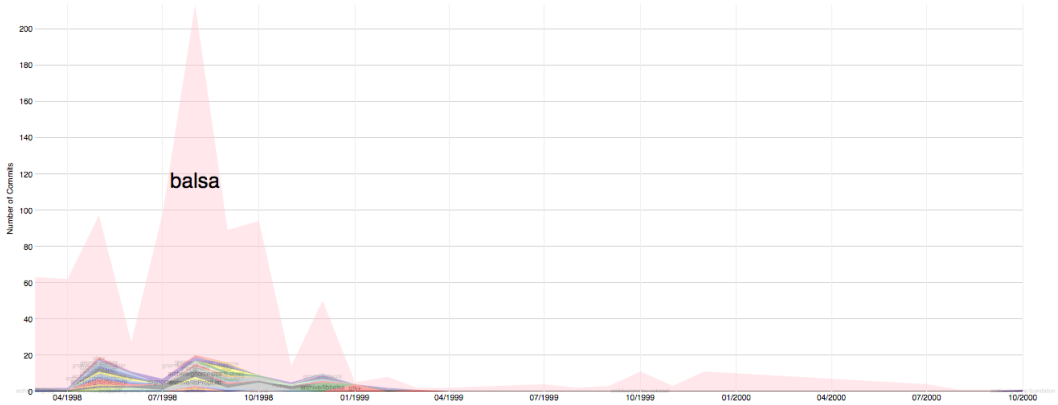

Figure 5.20. Stuart Parmenter's projects' Involvement Distribution View

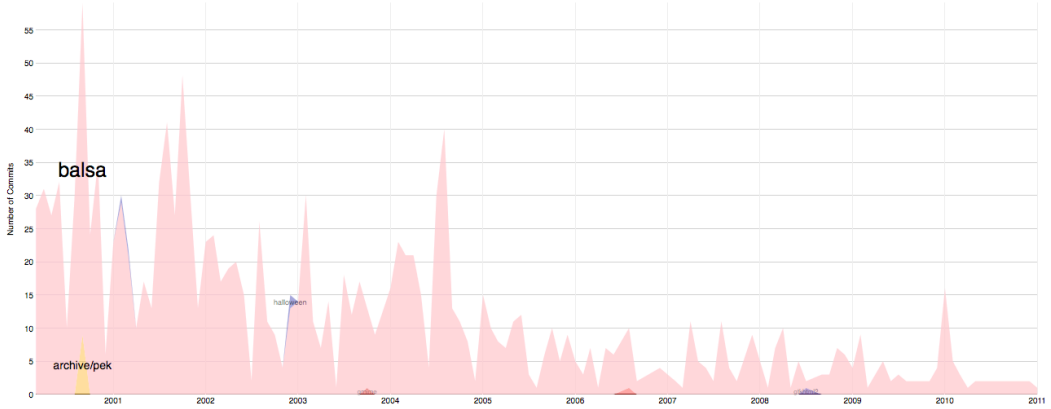

Figure 5.21. Pawel Salek's projects' Involvement Distribution View

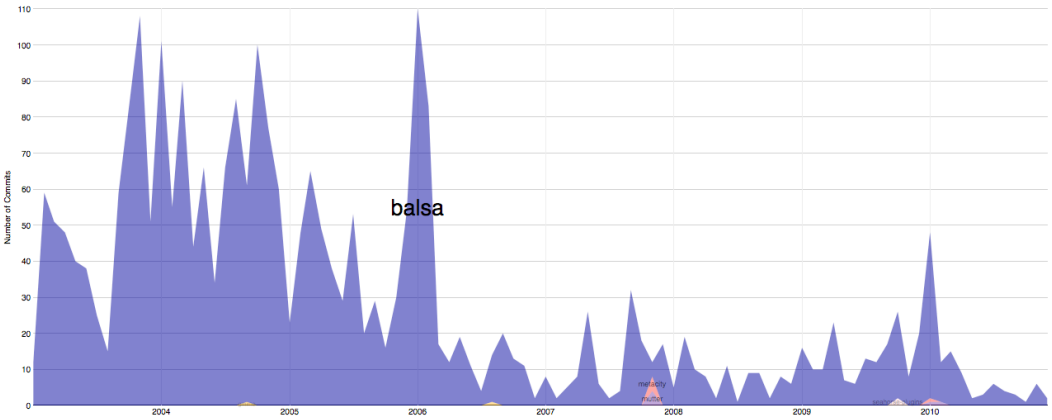

Figure 5.22. Peter Bloomfield's projects' Involvement Distribution View

5.3 Conclusion

We illustrated in this chapter how we can make use of the different views in a bottom-up and top-down approach.

Summarizing, we can say that Kjartan Maraas is a very active person with a long lifetime as translator and developer with affinity for bug fixing. He is the most important person according to the number of commits and the number of projects he has been involved over his lifetime. He is an outstanding person not only compared to the contributors of the Nautilus project but within the entire GNOME ecosystem.

For the Nautilus project, we can conclude that it is under continuous change with active contributors, who are equally distributed into number of translators and developers. In addition, it has at least one main driver in each phase with smooth take-overs, which are essential for a good knowledge flow.

The Balsa project, the selected veteran project, as well as TBO, the selected youngster project, have different life expectations. Both of them show a similar distribution of their contributors to either development and translation work in the Affectional Bond View. However, TBO has no contributor who is a recognized expert developer in the ecosystem but analyzing the data in depth we found out that it contains one developer, who is the center of power at the same time.

During the evaluation process, we showed that a single view is not sufficient to make conclusions but combining different views of different abstraction levels make it possible to tell stories about projects (*e.g.,* Nautilus and TBO) and about contributors (*e.g.,* Kjartan Maraas) within their ecosystem (*e.g.,* GNOME). These stories extracted from information available within the presented views support the understanding of the evolution of software ecosystems.

Part III

Epilogue

Chapter 6

Conclusion

In this chapter we take a step back to review and discuss the work we have done. Furthermore, we give some ideas about some open issues or extensions of Complicity and the research field.

6.1 Summary

Software systems are rarely developed in isolation but in the same environment, the ecosystem. The aim of this work is to model and visualize software ecosystems as the state of the art shows that this field is rather under-explored and that the ecosystem has often been ignored in the software evolution analysis. We summarize our contributions in the following.

- **Introduction of a tool support for the analysis of software ecosystems.** We developed Complicity, a web-based visualization tool that allows interactive exploration and analysis of software ecosystems evolution. It provides the facilities to move between different levels of abstractions. Our case study shows the importance of different abstraction levels – ecosystem and entity level – and the ability to interactively move from one level to another, as they complete each other. These abstraction levels allow analysts to identify dependencies between projects, contributors, and between both. In addition, by integrating both of them in one analysis task, one can illustrate the impact of a single project or contributor on the entire or part of an ecosystem. The data used in the analysis has been collected by reverse engineering super-repositories. We wrote Java programs that allowed us to scrape the data from repositories that are available through a Git web interface.

- **Introduction of an ecosystem meta model.** We introduced a super-repository and language independent meta model. By implementing it in Complicity, we showed that it is practically applicable.

- **Introduction of a catalogue of different views.** Furthermore, we presented a catalogue of different views at different abstraction levels. We described how they are constructed, the general idea behind them on some examples and discussed their advantages and limitations. We implemented the different views in Complicity so analysts can make use of them. We showed that a lot of information can be extracted by applying only basic metrics, *e.g.,* number of commits, number of projects, *etc.* but it has to be considered that these metrics might be misleading and should be considered carefully. For instance, the fact that one person commits

more than another does not necessarily mean that the former does more work than the latter. Instead, it depends on the committer's attitude on performing large or small commits.

- **Introduction of a methodology to evaluate the views using a bottom-up and top-down approach.** We analyzed the history of a project and a contributor, and showed their impacts on the ecosystem using predefined views at two different abstraction levels. We showed that Complicity is applicable to analyze software ecosystems in a bottom-up approach. We illustrated the usage of Complicity in a top-down approach, by comparing two projects of different lifetimes at ecosystem level and moved down to a lower level of abstraction to support the observations made at ecosystem level. Furthermore, analysts can use Complicity to visually explore the provided data on their own using other views and approaches.

6.2 Future Work

In this thesis, we devised a meta model and a catalogue of views that have been implemented into Complicity. In the following we tackle some issues and ideas on how our work can be extended and improved.

Exploration of more Ecosystems with Complicity. First, we intend to explore other ecosystems with complicity and compare the results. This is necessary to illustrate that Complicity, its views and meta model are applicable to other super-repositories.

Supply of an Application Programming Interface (API). So far only a single super-repository has been explored using Complicity. In order to make Complicity more popular, an API can help. It facilitates the expansion of the data collection in that everybody can provide data of any ecosystem, which consequently makes it more interesting to a larger range of people.

Extension of the User-based Queries. In case future super-repositories are larger than GNOME, we have to consider user-based queries in order to reduce the number of displayable elements. This is especially important for a good performance of the tool as well as smooth and fast data visualization, but also for the good user experience and flexible data analysis as the user can easily influence the data to be visualized and.

Extension of the Metrics List. We aim at incorporating in the analysis other metrics (*e.g.,* number of files addition versus removal or using other statistical functions) and data from other types of archives (*e.g.,* email archives, bug tracking tools and forums). Providing more metrics allows the user to explore into new directions and gain new insights into the evolution process of open source software ecosystems.

Addition of a Connection between Involvement Distribution View and Expertise View. At entity level, we provide the Involvement Distribution View and the Expertise View for both entities. In the current version, these are two independent graphs that support each other in the analysis tasks of the subject entity. An interesting extension of these graphs would be to connect them. An example scenario would be: while analyzing a contributor's details and select a project in the Involvement Distribution View, we would like to see what contributions he did to that specific project; Did he mostly changed .po files and so translation work? Or was he part of the development team and changed mostly .c, .h, .py or similar files. We could think of a similar scenario for the project entity level. In

consequence, we could identify or filter out the projects on which a contributor did only translation work.

Supply of an Information Exchange Platform. Another interesting direction would be to provide an information exchange platform where people can comment their own evolution process within an ecosystem, or alternatively, analysts can share and discuss some patterns or some anomalies they discovered during their analysis. This way, users may come up with new insights by combining their knowledge with their discoveries and the comments from real contributors. The input from real contributors can furthermore help to clean the data about the contributors, by stating that they actually worked on the different projects or to reveal that there are more people with the same name.

Exploration of the Contributor Aggregation based on Frequency and Projects. If we do not have the contributors themselves who give a statement about the correctness of the aggregated people, we have to come up with other ways to aggregate. We plan to improve our technique to automatically eliminate duplicate contributors. We have focused on aggregating people based on their email addresses, and names using the Fuzzy String Similarity distance. Identifying people based on their commit rate and the projects they worked on at a same time could identify and aggregate contributors with different names and email addresses.

Exploration of Time-based Visualizations. Lastly, time-based visualization might be a good technique to illustrate the evolution of some processes. So far it has only been treated little in literature. In the affectional bond view for example, this could illustrate how the contributors move from an affiliation to translation work to an affiliation to development work. In consequence, new trends within an ecosystem could be revealed. Generally, animations of any kind have to be used with care, as any kind of movement can easily distract humans from their main analysis tasks.

Further Investigation of Social Network Analysis. In our work, we showed how we can identify a good knowledge flow and center of power within a single project and how we can identify the distribution of the contributors at ecosystem level to either translation or development work. This work can be extended by analyzing how tightly the different contributors of the ecosystem are connected. This is a challenging job, as with the number of contributors, the number of connections between them is likely to increase. In consequence, metrics and filters have to be identified that limit the number of item but provides some new insights.

6.3 Closing words

The software life cycle is a process driven by humans. Thus, the analysis of the social structure behind a single software system became more and more popular in the last years. Especially with the hype around open source systems, analysts became curious in understanding how it is possible to develop popular tools in an unstructured environment.

Most of the past research in software evolution focuses on single software systems. However, software systems are rarely developed in isolation but in software ecosystems, a valuable source of information. So far only little has been contributed to the research field of analyzing and visualizing software ecosystems. Lungu is one of the first initiators who explored this new research direction during his PhD.

With this work, we tried to contribute to the research field of software visualization and software evolution of ecosystems integrating parts of social network analysis. We focused on simple metrics that can be extracted from the data available in the Git web interface of open source software systems and we visualized data at a higher level of abstraction (ecosystem and project/contributor level). We hope that Complicity and the provided views help on the one hand analysts in understanding software ecosystems and on the other hand attract researchers' interest in software visualization, evolution and ecosystems.

Appendix A

Data Collection

This chapter in the appendix is mostly concerned with my experience during the thesis with crawling and scraping the data from the web pages of super-repositories, which make use of the Git web interface.

A.1 Technologies Used

For the crawling and scraping of the data I decided to choose the Java programming language as it is the one I am most comfortable with. Before crawling I analyzed the structure of the super-repositories so I can scrape only the pages I am interested in without having to mirror the whole super-repository.

For crawling the web pages I used **HTML Cleaner**[1], an open-source HTML parser written in Java, which cleans the ill-formed and unsuitable web pages in such a way that it can be further processed. This cleaning is necessary for the use of **XML Path Language** (XPath), which allows the extraction of exactly the attributes from a well-formed XML or HTML file that I am interested in.

A.2 Crawling and Scraping GNOME

The first super-repository we have crawled and scraped is GNOME. In the following section, we describe our approach and the experience we made by collecting the data from the Git web interface. The crawling and scraping of the web pages of GNOME has been done on a MacBook Pro notebook with a 2.53 GHz Intel Core 2 Duo Processor and 4 GB RAM.

A.2.1 Procedure

We developed a Git web interface crawler written in Java that uses the HTML Cleaner to make sure that the pages are well-formed. This Java program enables us to crawl the super-repository in four separated steps:

[1] http://htmlcleaner.sourceforge.net/

1. project list
2. file tree structure of all projects
3. commit list of all projects
4. commit details of all projects

The mirrored pages are saved in the local file system.

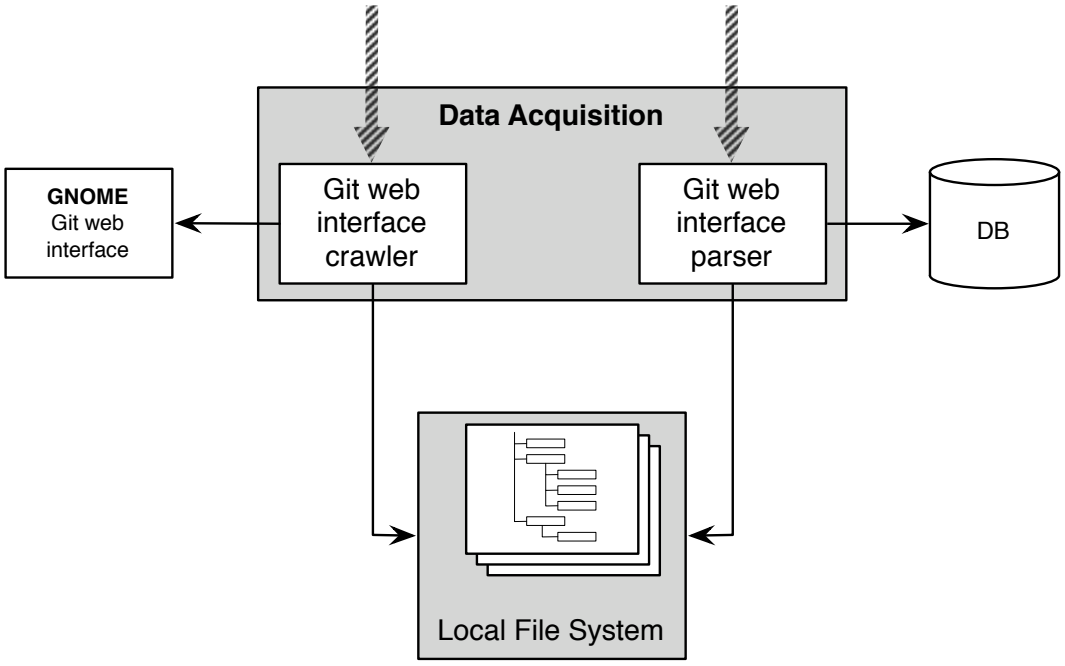

Figure A.1. Illustration of the crawling and scraping approach for GNOME

The Git web interface parser loads the well-formed HTML files stored in the local file system, extracts the relevant information from them using XPath and stores them into the database. Same as the crawler it scrapes the data in four steps and each of them has to be started separately.

In both processes, crawling and parsing, the sequence of the actions plays a central role. Step 1, crawling and parsing the project list, has to be carried out before all the other steps as it contains the data about the projects. This data is necessary for the link construction to access the file tree structure and commits. Step 2 and 3 (crawling and scraping) has to be executed before Step 4 as they contain important information to get the url to the detail page and to facilitate the linkage between commits and files.

A.2.2 Problems

During the crawling and scraping process some problems occurred, which we discuss in the following subsections.

Stable Internet Connection

As we are crawling the data from web pages, a stable internet connection is required in order to be able to collect the data smoothly. In other words, the location from where we access the internet and crawl the web pages should be chosen with care.

Access Denied to GNOME Git web interface

By scraping the data from web pages we run the risk of getting banded from the server on which the Git web interface is running. This is what happened to me shortly before I finished scraping all the data from the GNOME super-repository. In order to solve this problem, I built a work-around making use of **The Onion Router** (TOR)[2], which defends against any form of network surveillance. TOR provides a distributed, anonymous network where the client, which requests a web page is routed over so called TOR nodes to the destination server instead of building a direct connection to it. This allows an anonymous connection and allowed me to finish crawling the GNOME super-repository. **TorLib** (TOR Java Library)[3] is an implementation of this approach in Java. This approach has three drawbacks. First, you are dependent on external products. Secondly, you have to trust the TOR nodes that routes you to your destination page. Lastly, it slows down the crawling process.

File Size

A main drawback of using external libraries and toolkits is that we have to be aware of their limitations and shortcomings. During the crawling process the file size caused problems as HTML Cleaner cannot process very large files (here: commit detail pages with commits that made changes to over 30,000 lines). I solved this problem in crawling only the commit detail pages with less than 30,000 lines changes within a single commit. I ended up with a loss of 2,000 to 3,000 out of more than 950,000 web pages that have not been crawled. In consequence, at maximum 0.3% of the commit detail pages are missing.

Performance

The collection of data from super-repositories is a time-consuming process. The main issue of this first approach is its performance in speed. The fact that the pages are stored locally before scraping them improved a little the performance. The performance can be improved by running multiple processes concurrently or to run the programs on a machine that is faster than a notebook. However, whatever approach we choose to improve the performance we have to consider the risk of getting banded by the GNOME server on which the Git web interface is running.

Storage capacity

The fact that we store the web pages locally has the advantage that we can scrape the data from them over and over again without running the risk of getting banded or without requiring any internet connection. The main problem with this approach is that it requires a large storage capacity. To work to the contrary, we changed our crawling and scraping approach to collect the data of a second super-repository.

[2] http://www.torproject.org/
[3] http://web.mit.edu/foley/www/TinFoil/

A.3 Crawling and Scraping SourceForge

For comparison purposes, we decided to scrape another super-repository and evaluate Complicity on a different data set. In this section, we share our experience about scraping SourceForge. This second approach has been executed on another notebook with a 2 GHz Intel Core 2 processor and 2 GB RAM using Microsoft Windows as operating system.

A.3.1 Procedure

As the first version of the crawler and parser requires too much local space, we developed a new one to collect a subset of SourceForge projects. Same as for the previous approach, both, the crawler and the parser are written in Java.

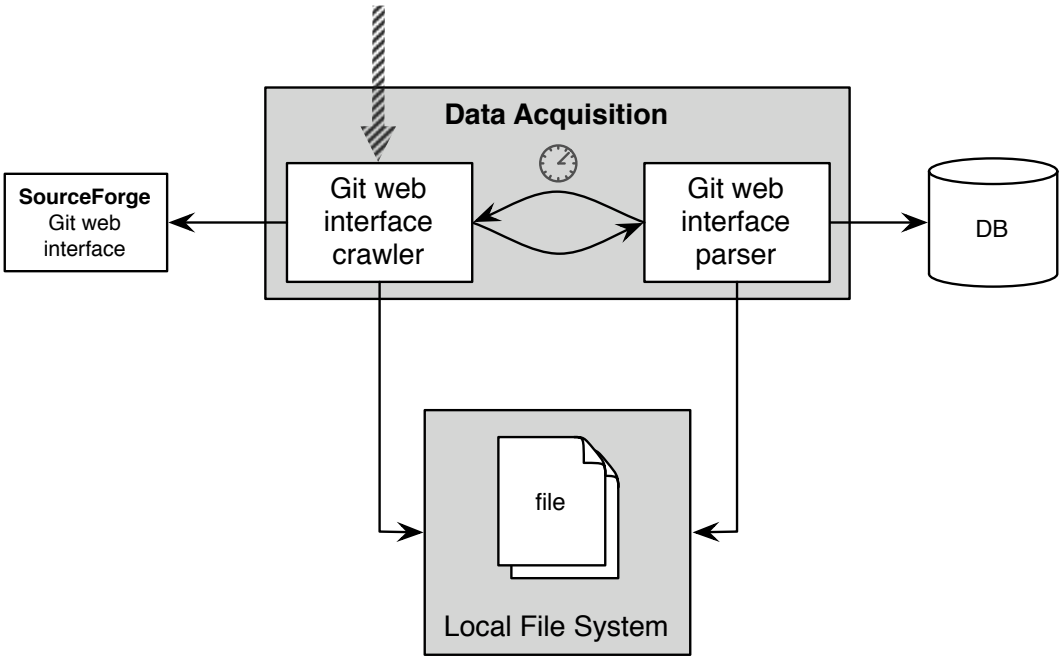

Figure A.2. Illustration of the crawling and scraping approach for SourceForge

The Git web interface crawler copies a web page from the SourceForge super-repository using the HTML Cleaner, and stores it as a well-formed HTML file in the local file system. Then the crawler triggers the parser, which loads the HTML file, extracts the relevant information using XPath and stores it in the database. Afterwards, the crawler copies a new web page from SourceForge overwriting the old HTML file in the local file system before triggering the parser for the data extraction *etc.*

Instead of mirroring the whole super-repository this approach crawls and parses a single page before crawling and parsing the next one. As it always overwrites the previous HTML file in the local file system it requires less storage capacity than the previous version. Furthermore, we added a method of random waiting time (between 5 seconds and 1 minute) before each page crawl. This reduces the crawling speed and increases the total time to collect the data but according to James Howison [HC04] it reduces the risk of getting banded from the SourceForge server.

Same as in the previous approach, we divided the data acquisition in the same four steps. However this approach requires only a single trigger for each of these steps as the crawling and parsing process are tightly connected. The different steps have to be executed in the fixed sequence from step 1 to 4 whereas step 3 and 4 are processed at the same time: for each commit that we extract from a project's commit list, we scrape the details subsequently before we move on to the next commit in the list.

A.3.2 Problems

In this second approach we tried to solve the storage problem and to avoid getting banded from the SourceForge server. But some problems remain the same. In the following we share our experience of this second approach.

Remaining problems

As we use the same technologies for the second than for the first approach, the limitation of the HTML Cleaner remains. However, for a subset of 88 SourceForge projects only 2 out of 24,629 commits could not be crawled, which are approximately 0.008% of missing commits. The problem with the requirement of a stable internet connection continue to exist. In addition, our second crawling process becomes even slower due to the waiting method.

Access Denied to SourceForge Git web interface

In order to avoid getting banded a second time we introduced a random waiting time of 5 seconds to 1 minute. Due to the limited time available for scraping data from SourceForge, I decided not to add too much waiting periods. I crawled the projects' file tree structure with a waiting period between 10 seconds to 1 minute between each project instead of each file crawl. Due to this mistake I got banded a second time. However, this time they banded me only from accessing the web interface through a web browser. The Java programs continued running without the need of TOR. For the crawling of the commits and commit details I added

A.4 Conclusion

We can conclude that the technologies chosen for the data acquisition are not the fastest but still too fast to get banded twice from the two super-repository servers. We experienced that scraping is a time-consuming act, especially because of the additional waiting periods between each page crawl. However, there is no guarantee that this actually protects us from getting banded. Using XPath instead of Regular Expressions made it easy to extract only the data we wanted.

The main disadvantage of the chosen approach for the data collection is that the Java programs have to be updated for every super-repository. This makes it inflexible and time-consuming venture.

A possible solution is to mine the Git version control system instead of scraping the data from the web pages available through the Git web interface. First, it is independent of the super-repository in that the structure for the data extraction remains the same for every project using a Git version control system. Secondly, it retrieves the data of a single repository with one single command instead of crawling many pages and thus reduces the risk of getting banded.

Appendix B

Data Analysis

Data plays a central role in the analysis of software evolution. With the hype about open source software systems a huge amount of data becomes available and we are no longer dependent on the data gathered from industry. But it hides some problems, which we discuss in the following chapter.

B.1 Huge Amount of Data

With the increase in popularity of open-source systems, more and more software systems and data about their evolution become available. With the introduction of new data sources, new problems have to be faced.

First, the data analysis becomes more time-consuming, as more data has to be analyzed. In order to be able to analyze the data, we require techniques for the data collection. This in consequence is again a time-consuming task the more data becomes available and more different sources and formats are defined.

Another problem, that occurs is the modeling of the data. As with the development of new data sources, old data models have to be adapted or new ones have to be generated.

The huge amount of data also influences the the analysis and visualization tasks. For this purpose, new tools and techniques have to be developed that improve the readability. Text-based is rarely a good choice for analysis and using a table-based approach is only partly applicable if the goal is to do quantitative comparisons on a rather small amount of data. Visualizations on the other side can cope with a large amount of data especially when the goal is to extract patterns or anomalies from the data set. Unfortunately also visualizations may reach their limits when its comes to visualize a huge amount of data, as either the graph is becoming unreadable due to too many elements that overlap and flood the graph.

Lastly, scalability is another factor that comes into play when dealing with a huge amount of data. Many tools are available that cannot cope with a large data set. For this purpose, new techniques and filters become necessarily.

B.2 Dirty Data

As already mentioned in the previous chapter Appendix A, one problem that occur when dealing with data from the internet are *ill-formed web pages*. The solution we used to solve this problem is the

HTML Cleaner as it cleans the underlying tree structure by closing missing tags. But as mentioned before it does not perform so well in terms of speed as other approaches.

Another problem we have to cope with are the different *character sets and special characters*, in which the data is provided. This is a well-known problem and no perfect solution is available. We decided to crawl, scrape and save the data using the utf-8 character set. It is the most commonly used character set within the web and so covers most special characters.

An issue that might be less common in industry than it is in free and open-source projects is the misuse of the data fields. For example, many committers used the "author"-field of a commit also as message or date field. Another problem that arises is the fact that committers use different usernames or nicknames or some abbreviations of their names or any other combination of their names, which makes it very hard to match the same users into one single committer.

B.3 Redundant committers

The fact that users misuse the author and committer field but also due to the usage of different names with and without special characters and nicknames representing the same person makes it difficult to match the same person. In consequence, being able to combine different committers in the same person, reduces the number of people and so the complexity. More importantly the elimination of redundant committers provides a more correct image of the state, communication or the evolution of the contributors and the involving projects. More about this topic in Appendix 3.8.

B.4 Attic files

In this work, as already mentioned before, we focus on the data available about projects using the Git web interface. These pages contain general information about the project, its commits and file tree structure. While they keep track of all changes executed on any file that existed at any point in time in the commit messages, the file tree structure represents only its current state. In other words, all information about files that have been deleted or whose names or location changed is only available in the commit messages. This can cause problems if one wants to illustrate the evolution of the whole project's file structure, as only a limited amount of data about these *attic files* is available within the commits. For example, we have no information about the attic file size. A work around this problem is to use the number of lines instead of the file size [GKSD05].

Bibliography

[BGD+06] Christian Bird, Alex Gourley, Prem Devanbu, Michael Gertz, and Anand Swaminathan, *Mining email social networks*, Proceedings of the 3rd International Workshop on Mining Software Repositories (MSR 2006), ACM, 2006, pp. 137–143.

[BH09] Michael Bostock and Jeffrey Heer, *Protovis: A graphical toolkit for visualization*, IEEE Transactions on Visualization and Computer Graphics **15** (2009), 1121–1128.

[CC90] Elliot Chikofsky and James Cross, *Reverse engineering and design recovery: A taxonomy*, IEEE Software **7** (1990), no. 1, 13–17.

[CHC05] Megan Conklin, James Howison, and Kevin Crowston, *Collaboration using ossmole: a repository of floss data analyses*, SIGSOFT Softw. Eng. Notes, 2005.

[CK94] Shyam Chidamber and Chris Kemerer, *A metrics suite for object oriented design*, IEEE Transactions on Software Engineering **20** (1994), no. 6, 476–493.

[Cor89] T. A. Corbi, *Program understanding: Challenge for the 1990s*, IBM Systems Journal **28** (1989), no. 2, 294–306.

[Die07] Stephan Diehl, *Software visualization - visualizing the structure, behaviour, and evolution of software*, Springer Verlag, Berlin, April 2007.

[DLL06] Marco D'Ambros, Michele Lanza, and Mircea Lungu, *The evolution radar: Visualizing integrated logical coupling information*, Proc. of Mining Software Repositories (MSR 06, 2006, pp. 22–23.

[Erl00] Len Erlikh, *Leveraging legacy system dollars for e-business*, IT Professional **2** (2000), no. 3, 17–23.

[ESS92] Stephen G. Eick, Joseph L. Steffen, and Eric E. Sumner, *Seesoft - a tool for visualizing line oriented software statistics*, IEEE Transactions on Software Engineering **18** (1992), 957–968.

[Ger94] N. Gershon, *From perception to visualization*, Scientific Visualization (L. Rosenblum, R.A. Earnshaw, J. Encarnacao, H. Hagen, A. Kaufman, S. Klimenko, G. Nielson, and D. Thalmann F. Post, eds.), Academic Press, London, UK, 1994, pp. 129–139.

[GKSD05] Tudor Gîrba, Adrian Kuhn, Mauricio Seeberger, and Stéphane Ducasse, *How developers drive software evolution*, Proceedings of IWPSE 2005, IEEE, 2005, pp. 113–122.

[GM10] Mathieu Goeminne and Tom Mens, *A framework for analysing and visualising open source software ecosystems*, Proceedings of IWPSE-EVOL 2010, ACM Press, 2010, pp. 42–47.

[HC04] James Howison and Kevin Crowston, *The perils and pitfalls of mining sourceforge*, In Proceedings of the International Workshop on Mining Software Repositories (MSR 2004, 2004, pp. 7–11.

[KC98] R. Kazman and S. J. Carrière, *View extraction and view fusion in architectural understanding*, Proceedings of ICSR 1998, IEEE, 1998, pp. 290–299.

[Lan01] Michele Lanza, *The evolution matrix: Recovering software evolution using software visualization techniques*, Proceedings of IWPSE 2001, ACM Press, 2001, pp. 37–42.

[LDGP05] Michele Lanza, Stéphane Ducasse, Harald Gall, and Martin Pinzger, *Codecrawler — an information visualization tool for program comprehension*, Proceedings of ICSE 2005 (27th IEEE International Conference on Software Engineering), ACM Press, 2005, pp. 672–673.

[Leh80] Meir M. Lehman, *Programs, life cycles, and laws of software evolution*, Proceedings of the IEEE, vol. 68, 6, no. 9, September 1980, pp. 1060–1076.

[LFRGBH06] Luis López-Fernández, Gregorio Robles, Jesus M. Gonzales-Barahona, and Israel Herraiz, *Applying social network analysis techniques to community-driven libre software projects*, International Journal of Information Technology and Web Engineering **1** (2006), no. 3, 27–48.

[LL10] Mircea Lungu and Michele Lanza, *The small project observatory - a tool for reverse engineering software ecosystems*, Proceedings of ICSE 2010 (32nd ACM/IEEE International Conference on Software Engineering), ACM Press, 2010, pp. 289–292.

[LLGR10] Mircea Lungu, Michele Lanza, Tudor Girba, and Romain Robbes, *The Small Project Observatory: Visualizing software ecosystems*, Journal of Science of Computer Programming (SCP) **75** (2010), no. 4, 264–275.

[LS81] Bennet P. Lientz and E. Burton Swanson, *Problems in application software maintenance*, Commun. ACM **24** (1981), 763–769.

[Lun09] Mircea Lungu, *Reverse engineering software ecosystems*, Ph.D. thesis, University of Lugano, Switzerland, October 2009.

[OlMB+07] Michael Ogawa, Kwan liu Ma, Christian Bird, Premkumar Devanbu, and Alex Gourley, *Visualizing social interaction in open source software projects*, In APVIS'07: Proceeding of the 2007 Asia-Pacific Symposium on Visualisation, IEEE Computer Society, 2007, pp. 25–32.

[OPT10] Christopher Oezbek, Lutz Prechelt, and Florian Thiel, *The onion has cancer: Some social network analysis visualizations of open source project communication*, Proceedings of FLOSS 2010, ACM Press, 2010, pp. 5–10.

[PSB92] Blaine A. Price, Ian S. Small, and Ronald M. Baecker, *A taxonomy of software visualization*, Proceedings of the 25th Hawaii International Conference on System Sciences (Kauai, HI, USA), vol. 2, January 1992, pp. 597–606.

[SDPH10] Dominik Seichter, Deepak Dhungana, Andreas Pleuss, and Benedikt Hauptmann, *Knowledge management in software ecosystems: software artefacts as first-class citizens*, Proceedings of the Fourth European Conference on Software Architecture, ECSA '10, ACM, 2010, pp. 119–126.

[SMG09] François Stephany, Tom Mens, and Tudor Gîrba, *Maispion: A tool for analysing and visualising open source software developer communities*, Proceedings of IWST 2009, ACM Press, 2009, pp. 50–57.

[WL07] Richard Wettel and Michele Lanza, *Visualizing software systems as cities*, Proceedings of VISSOFT 2007, IEEE CS Press, 2007, pp. 92–99.