Università
della
Svizzera
italiana

Faculty
of Informatics

Bachelor Thesis

June 11, 2012

Visual storytelling of Software Systems

Lorenzo Baracchi

Abstract

The development of a software system is a long and complicated process: Analyzing the evolution of a software system helps to reconstruct the story of a project , which can help to understand the software itself and how it was developed.
Visualization can be a useful technique to depict this story by reducing the amount of information needed to understand software development processes.
In this bachelor project we present two tools: `Peaksight` and `Mr.Bubbly`. We developed them with the goal to create effective visualizations of data from commits, emails, and bugs. Using these visualizations we are able to tell the story of how a software system evolved over time with the goal of using this knowledge to better understand the history and the evolution of the software. We created two different type of visualizations: one static in `Peaksight` and one animated in `Mr.Bubbly`. We create a story seen from different perspectives, which combined creates a more complete vision.

Advisor
Prof. Michele Lanza
Assistants
Alberto Bacchelli, Dr. Marco D'Ambros

Advisor's approval (Prof. Michele Lanza): Date:

Acknowledgements

First of all, I would like to thank professor Michele Lanza for his helpful advices and his awesome teachings. I need to thank also Dr. Marco D'Ambros and Alberto Bacchelli, who are the best assistant a bachelor student could have.

I would like to thank my family for the big support they gave me in those years, which allows me to continue with my studies.

I have to thank my amazing girlfriend, who always believed in me and give me the strength to carry on.

Thanks to the Coffee for helping me to stay awake in the long nights of study!

Lorenzo Baracchi

Contents

List of Figures

List of Tables

1 Introduction

Software Development is a complex process that requires time and thinking. During this time a software system evolves from an abstract concept to a working system, going trough many changes, in design, analysis and implementation. Keeping track of those changes is not an easy task, especially in situations where many changes have been done [2]. Maintaining a record of changes can help remembering and understanding the design of a software system and the motivation of those changes [8]. Maintaining a "hand made" record is tedious and error prone, we need tools to analyze data relative to a project. One useful method to show large amounts of data is visualization, because it allows to "reveal stories hidden behind data" [14]. With the use of visualization it is possible to achieve a higher level of abstraction that can help to reduce the time needed to understand a lot of data about the system such as version control or bug tracker history, or even mailing lists, and create a representation of a system useful to understand it and how it is developed.
Visual representation of a project can be used to tell the story of a software system,which is useful to answer some questions, about the evolution of a software system, for example: is the development of the system constant or discontinuous? Did the development of the system got through critical periods and when?

Storytelling is a means to share experience and knowledge. Visualization is used by scientists to create simple representation of data, with the purpose to see real data that are normally not visible, giving the possibility to perceive and manipulate data with different perspectives [10].

Telling the story of a software project can be defined as depicting how the development of a software system has evolved over time. To describe the evolution of a software we have to consider some specific artifacts of a software system, because every good story has characters and in the case of a software system, characters can be found in its artifacts. Thus we need to define which are those artifacts relevant to tell the story of a project. The first artifact we consider is the source code, which is an essential elements for any software system. Software is made by people, and people need to communicate to work together. As Karl Fogel said: "Mailing lists are the bread and butter of project communications"[6], therefore we consider as second artifact the emails exchanged by developers in mailing lists of software projects. Then, in every software system, there are problems to fix, also called bugs or issues, which have an important role on the process of developing a software system, therefore the third artifact might be information about issues and bugs. Hence we can find some relevant data in these three artifacts, which can be useful to start depicting the story of a software system, thus we can consider these three artifacts the main characters in our story of a software system.

With this bachelor project we propose two different ways to tell the story of the evolution of a project based on these three artifacts. The first approach, `Peaksight`, makes usage of a static visualization, while the second, `Mr.Bubbly` use dynamic and animated visualizations.

1.1 Document Structure

In Section 2 we introduce software visualization and we analyze some related work.
In Section 3 we introduce our approach to the visualization of a software system, in particular we discuss the structure of the two tools and the visualizations that can be obtained using them.
In Section 4 we tell the story of a software system: "Apache Camel", by using the two tools: `Peaksight` and `Mr.Bubbly`.
In Section 5 we analyze the obtained results.

2 Related Work

Software Visualization

The aim of Software Visualization is to create visualizations of software systems to raise the level of abstraction and reduce the amount of information needed to understand a software system. Because of the abstract nature of software, creating a visual representation is not straightforward. Researchers have created many different metaphors to represent software systems. In this section we discuss some of those approaches to represent software systems, which are related to our work.

The Small Project Observatory

"The Small Project Observatory" is a prototype tool which aims at supporting the analysis of project ecosystems through interactive visualization and exploration [9].
The idea behind this approach is that projects are not developed in isolation, but since many of them belongs to particular ecosystems (e.g., GNOME or Apache), the visualization of the project alone should be integrated with a visualization of the entire ecosystem.
"The Small Project Observatory" allows, among other things, to identify, oldest, largest, or more active projects in an ecosystem. It also presents timelines of the evolution of projects and have also a map used to display dependencies among projects or collaborations among developers.

With this tool we understand that an important unit of measure for studying the evolution of a project is the time, indeed "The Small Project Observatory" make usage of timeleines to describe the evolution of projects. Also `Peaksight` and `Mr.Bubbly` present data relative to software system respect to a time measure.

Complicity

Complicity [11] is a tool, developed by Neu et al., that allows the analysis of the evolution of open-source software system at ecosystem level. It can be used for visualizing and understanding the evolution of single projects and contributors, but also for the analysis of entire ecosystems.
Figure 1 shows the main view of Complicity while exploring the GNOME ecosystem with a focus on GIMP.

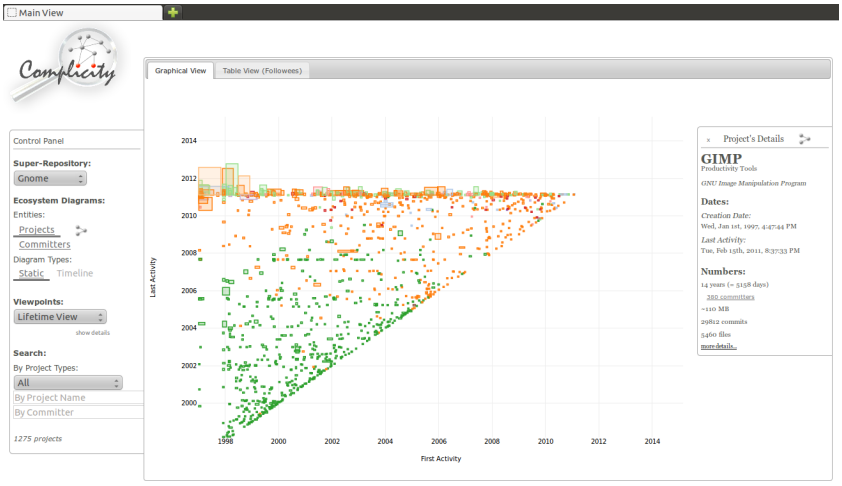


Figure 1. Complicity: Overview of the Gnome ecosystem

Regarding the visualization of a single project, Complicity presents timelines which shows the relations existing among number of commits, number of developers, lines changed and files changed. Therefore we notice that timelines are useful visualizations to compare the evolution of different artifacts. For `Peaksight` we use a similar approach, but considering commits, emails, and issues reports.

Software Evolution Storylines

Software Evolution Storylines is a tool developed by Michael Ogawa and introduced in the paper at SoftVis 2010 [12]. The goal of this tool is to show interactions between developers of a project during the development. The visualization shows a static image where each developer is represented by a line that changes in time following his/her contribution to the software. For example figure 2 shows the evolution of the popular programming language

Python.
The tool is good to perform visualization of small and medium size project while it has problem in representing large projects, for both performance issue and intricate visualization.

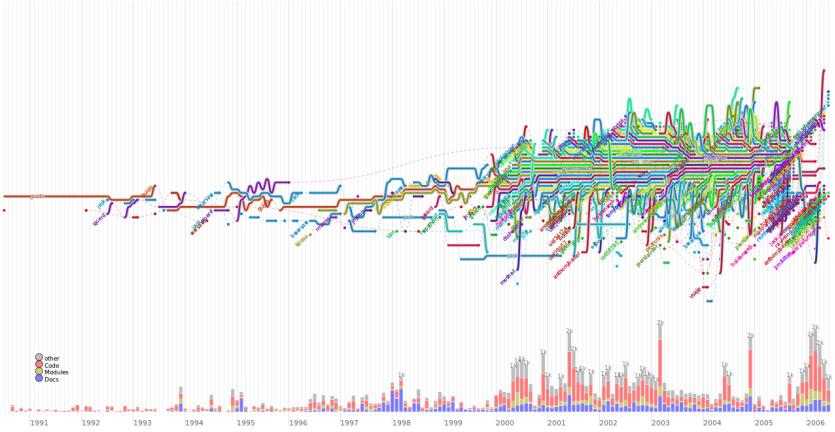


Figure 2. Software Evolution Storylines: The evolution of Python

For us, the interesting part is the bar chart at the bottom of the visualization which is used to present an overview of the number of commits made to the project over time, similar to what we obtain with `Peaksight`, but we also include emails and bugs.

code_swarm

code_swarm[1] is a tool developed by Michael Ogawa that shows the evolution of a software based on the interactions among developers. The difference from the previous approach is that the visualization is not static but animated. code_swarm shows the history of commits in a software project. Achille's heel for this tool is that users can not interact with the animated image.

Particularly interesting is the idea to use an animation to present the evolution of a project. We do something similar with `Mr.Bubbly`, with the difference that we do not consider developer interaction, but commits, emails, and issues reports.

CodeTimeline

CodeTimeline is a tool by Adrian Kuhn and Mirko Stocker [8], which creates two different visualizations. The first is a timeline visualization where each line represents one file in the software repository and the color of the line represents the ownership of the file based on commits by developers. The second visualization shows removed terms and trending terms for each release of the software.

The first visualization of CodeTimeline, is particularly interesting for the approach to create a visualization of the evolution for each file in the system, which is similar to what we do with `Mr.Bubbly`.

Visualizing Collaboration and Influence in the Open-Source Software Community

The paper by Heller et al. introduces a tool to create a visualization of the contributors to the GitHub ecosystem [7]. Their goal is to create a visualization where contributors are place in a world map according to their geographic region. Contributors are then connected, by lines across regions, based on related commits. With this visualization it is possible to create hypothesis about how the development of a software system is performed.

The approach presented is particularly interesting for the attempt to analyze the relations among developers of different regions, therefore it could be a good starting point to analyze the communication developers have. For the purpose of analyzing communication among developers we create visualizations of emails in `Peaksight` and `Mr.Bubbly`.

Visual Data Mining in Software Archives to Detect How Developers Work Together

With this paper [15] Weissgerber et al. try to obtain insights about programming behaviors, when developers uses version controlling systems likes Subversion or CVS. In particular they analyze JUnit and Tomcat as case studies.

[1] code_swarm `http://www.michaelogawa.com/code_swarm/`

Their work is motivated by three fundamentals questions: is the work in a software development equally distributed or are there some main developers? Are software system's modules developed by a single developer or are they developed by multiple developers? Is the development process constant over time, or has it some phases with higher and lower development?
They created three different type of visualization to answer those questions. One visualization shows the number of changes to the code in a certain time and the developers who did them. Another presents a matrix of file and developers where each cell of the matrix is colored depending on the number of changes the developer did to the file, the color is computed from a color scale that goes from blue to red depending on the number of changes. The last visualization shows a graph containing sequences of files which has been changed by which developer in a certain period of time.
This approach make strong accents on the changes on source code and the relation with the developer who did them, particularly interesting for us is the first type of visualization presented since it consider also the time when changes happens.

Supporting Software History Exploration

In this paper [4], Bradley and Murphy presents two tools used to explore historical information about a software system integrated in the development environment (IDE). The two tools, *Deep Intellisense* and *Rationalizer*, also includes informations about bugs. Data exploration is done with a simple view that presents the data as a long list and there is no visualization of them. Authors confirms that "scrolling through" such large amount of information can be confusing and error prone.

Even if there is no visualization of the data, these tools are interesting for our point of view since they try to create a correlation between bugs and source code, which we try to explore with our tools.

The Evolution Radar

The Evolution Radar [5] is a tool by D'Ambros et al. The authors propose a visualization based approach to analyze logical coupling information at both file-level and module-level. Evolution Radar presents a visualization in the form of a pie chart, where each sector is a module and files are represented by colored circles placed near or far each other depending on their coupling. The logical coupling is also represented with a color mapping, where blue indicates low coupling and red high coupling. Moreover the visualization is interactive so that the user can easily verify hypothesis on the software system.

This tool is particularly interesting for its approach to visualize files divided by the module they belong to, which can be used to explore the files evolution in relation with their module. With `Mr.Bubbly` we use a similar approach, were data about files is shown divided by package.

Mining Email Social Network

In this paper [3] the authors performs a study of the link that exists between email communications and source code changes in Open Source Systems development. They presents some simple diagram showing the relations between people in the mailing list and message sent and read, and a diagram about the relation between message sent and number of distinct respondent to an email.
The most remarkable result of this research is that they observed a Spearman's rank correlation of about 0.80 between the message sent by a developer and the number of changes made to the source code, which underlines the fact that the most active developer are also the most active in communication in mailing list of the project.

This work is particularly interesting for the idea of exploring link between emails and source code. The study highlight a strong link between these two artifacts. With both `Peaksight` and `Mr.Bubbly` we try to explore this correlation between emails and source code, and we add also issues.

Ohloh

Ohloh[2] is a website, similar to a social network for Open Source Developers, that offers certain statistics about many Open Source Software. In particular it has some visualization about the amount of lines of codes and different programming languages used during the time of development of the software, it also provides some simple visualization about the number of commits and number of committers. Despite the simplicity of the offered visualizations it is a useful tool to have some first insights on how a software system is evolving.

Like `Peaksight` and `Mr.Bubbly` it presents timelines visualizations of commits, but it does not consider emails or bugs.

[2]Ohloh `https://www.ohloh.net/`

REmail

REmail [1] is an Eclipse plugin aiming to integrate email communication in the development environment. REmail can automatically show you emails related to the class on which the developer is currently working on, this is accomplished by searching inside the archive of the mailing-list relative to the project for references to the class. Users can select any of the related emails and read its content. REmail also presents a basic chart representing the flow of emails relative to a class as shown in figure 3. Thanks to this simple visualization it is possible to perform

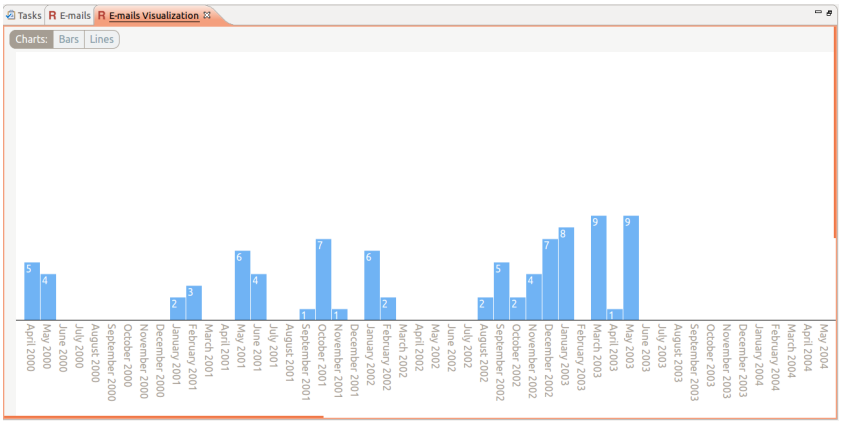


Figure 3. REmail: flow of emails relative to a class

simple analysis of how much a certain class is discussed by developers and in which period of time.

REmail explore the correlation between emails and source code, but it only considers emails relatives to a single class, therefore the visualization has a limited scope. Instead, `Peaksight` and `Mr.Bubbly` considers emails for all the time of development of a software system.

The difference of our approach

REmail (section 2) and Mining Email Social Network (section 2) are two different works, both concentrated on the retrieval and linking emails to the source code, but the latter is more a theoretical analysis of a software system, while the former provide only a visualization of emails relative to a single Java class, there is not a larger visualization.

Supporting Software History Exploration (section 2), provides some informations abut bugs, but this information is presented as lists and there is not a visualization for them, this does not allow to easily create a wide vision of the entire software system.

Except for those few exceptions, all the aforementioned tools have in common that they construct visualizations based only on the data obtained from version control and source code. Thus all the tools cited create visualizations based on a single artifact, or at maximum two artifacts. With `Peaksight` and `Mr.Bubbly` we create visualizations based on three different artifacts: commits, emails, and bugs. We believe that creating visualization based on more artifacts can help to improve the analysis of the evolution of a software system, therefore we can have the possibility to explore relations among commits, emails and bugs that were not possible to explore using other tools.

3 Telling the story of a software system with `Peaksight` and `Mr.Bubbly`

We devise two visualizations, which correspond to two different tools: `Peaksight` and `Mr.Bubbly`. `Peaksight` is focused on creating a comparison between commits and email activities, while showing the effects on bugs. `Mr.Bubbly` shows data relative to emails, commits and bugs at file level. `Peaksight` can be used to inspect the overall flow of the development process, while `Mr.Bubbly` can be used to inspect the development of specific files.

3.1 `Peaksight`

The goal of this tool is visualizing the "peaks" of code changes, communication, and bugs arising during the time of development of a software system. The idea is to provide a visualization where a user can compare the amount of commits, emails and bugs during the time of development of the software system.

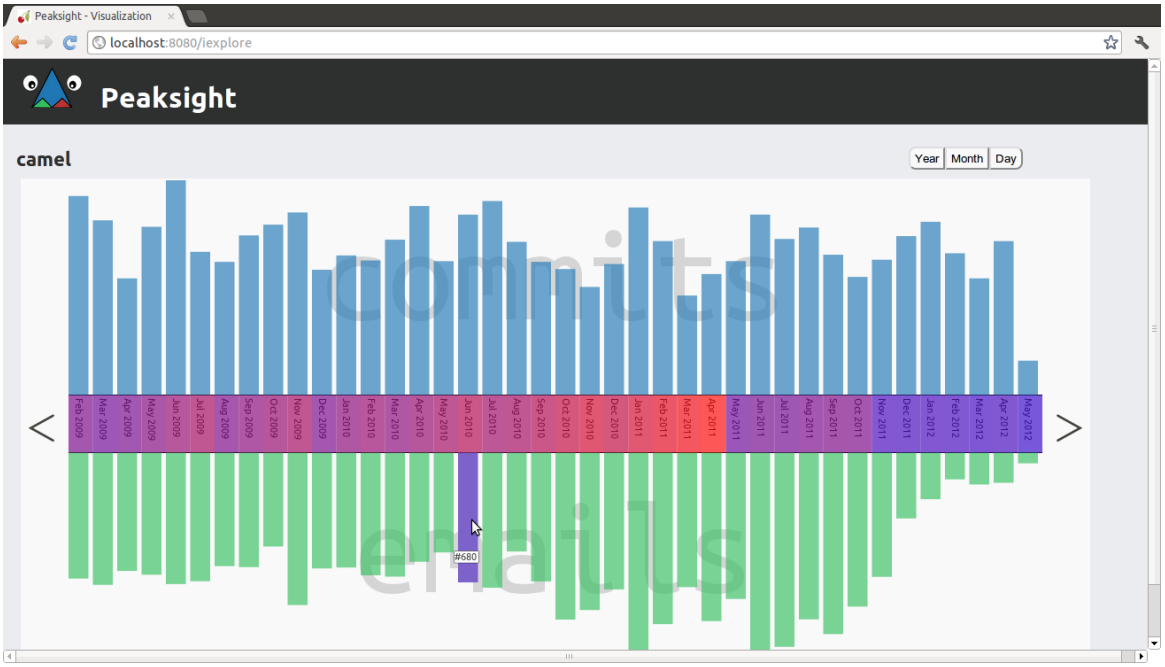


Figure 4. Peaksight: visualization by month

Figure 4 shows an example of a visualization made with `Peaksight`. The blue bars represent commits at the top, and the green bars represent emails at the bottom. Bugs are represented by the middle band, with a color scale that goes from blue to red depending on the amount of open bugs at that specific period of time, blue meaning few bugs while red meaning many bugs. When the user moves the mouse over a bar a pop-up containing the number of entities (commits, emails or bugs) for that bar is shown.
Figure 4 shows a visualization considering a month as time interval(i.e., each bar represents the number of commits, emails or bugs for that specific month). The user can choose other two time intervals: year and day. When the user goes over a bar with the mouse the number of items of that bar is shown.
For commits and emails the height of the bar is computed for each time step based on the height of the highest bar. Thus commits and emails have different scales.

Peaksight is a web application running on top of the web server CherryPy [3], which is a lightweight web server for python applications. The back-end of the tool is build in python, while the interface is done using HTML5, CSS3, and JavaScript. The data used to build the visualization is stored in a database. As database we used CouchDB: a no-sql database from the Apache Foundation.

Figure 5 depicts the architecture of Peaksight. The main element in this architecture is the server, which

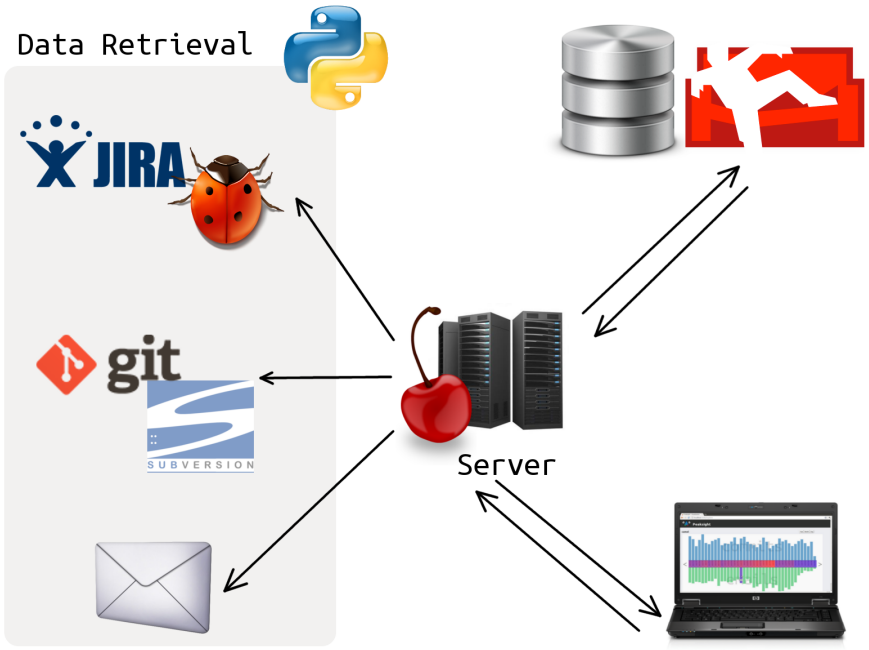


Figure 5. Peaksight: overview of the architecture

responds to user requests and communicates with the database. It is also responsible to collect the data about commits, emails, and bugs from different sources. When a user requests a visualization of a software system, the server retrieves, by means of queries, the relatives data from the database, then, if needed, it performs some transformations, and sends the obtained results to the user.

Peaksight supports two Version Control Systems: Git[4] and Subversion[5]. For getting data about bugs we use JIRA[6] as bug tracker. To collect data from mailing list we import emails from mbox files, since for many project is possible to download archives of mailing lists in this format.
The interested reader can find details about how data is imported in Peaksight in Appendix A.1.

3.1.1 Interaction in Peaksight

The first thing a user may want is more information about a specific period of time. For example one may wonder which commits were made in a month or who sent the emails relative to a month and read their content, or one can be curious about the subject of opened bugs. This could be achieved by simply clicking on the desired bar: This will show a list of commits, emails or bug relative to the period of time selected and the type of bar pressed. For example if the user clicks on the bar representing commits made in June 2010, it will appear a pop-up containing a list of commits made in June 2010. The list shows the log message for each commit, the name of the developer who made that commit and the date the changes were committed. Clicking on a bar representing emails or bugs results in the same behaviour.

[3] CherryPy: http://www.cherrypy.org/
[4] Git: http://git-scm.com/
[5] Subversion http://subversion.apache.org/
[6] JIRA: http://www.atlassian.com/software/jira/overview

Figure 6. Peaksight: pop-up showing a list of commits for a specific month

Figure 6 shows an example of a pop-up containing a list of commits for a specific month, the list of commits includes the date when each commits has been submitted, the log message of the commit and the author responsible for the commit.

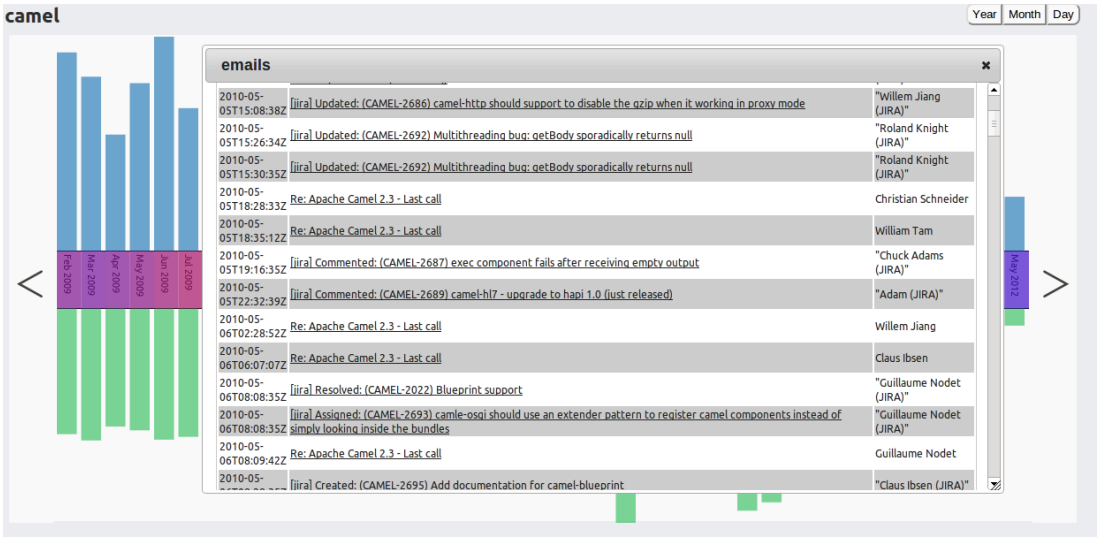


Figure 7. Peaksight: pop-up showing a list of emails for a specific month

Figure 7 shows an example of a pop-up containing a list of emails for a specific month, the list of emails includes the date on which each email has been send, the subject of the email and the sender.

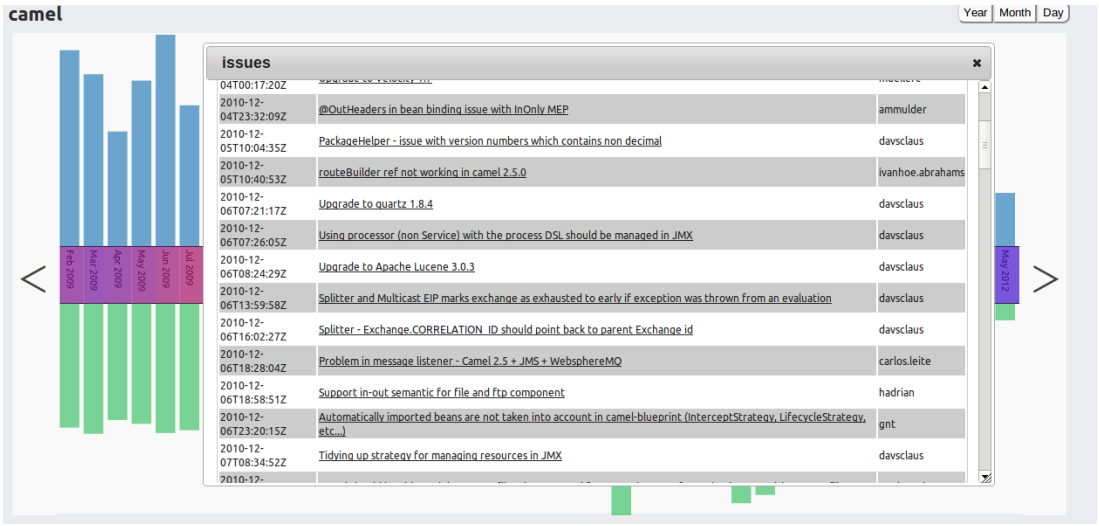


Figure 8. Peaksight: pop-up showing a list of bugs opened in a specific month

Figure 8 shows an example of a pop-up containing a list of issues (bugs) opened in a specific month, the list of issues includes the date on which each issue has been reported, the summary of the issue, i.e a short sentence describing the problem, and the name of the developer, or user, who reported it.

From these pop-ups is possible, by clicking on the log message for a commit, on the subject for an email, and on the summary for an issue, to open a page where more information about those objects is available. Figure 9 shows

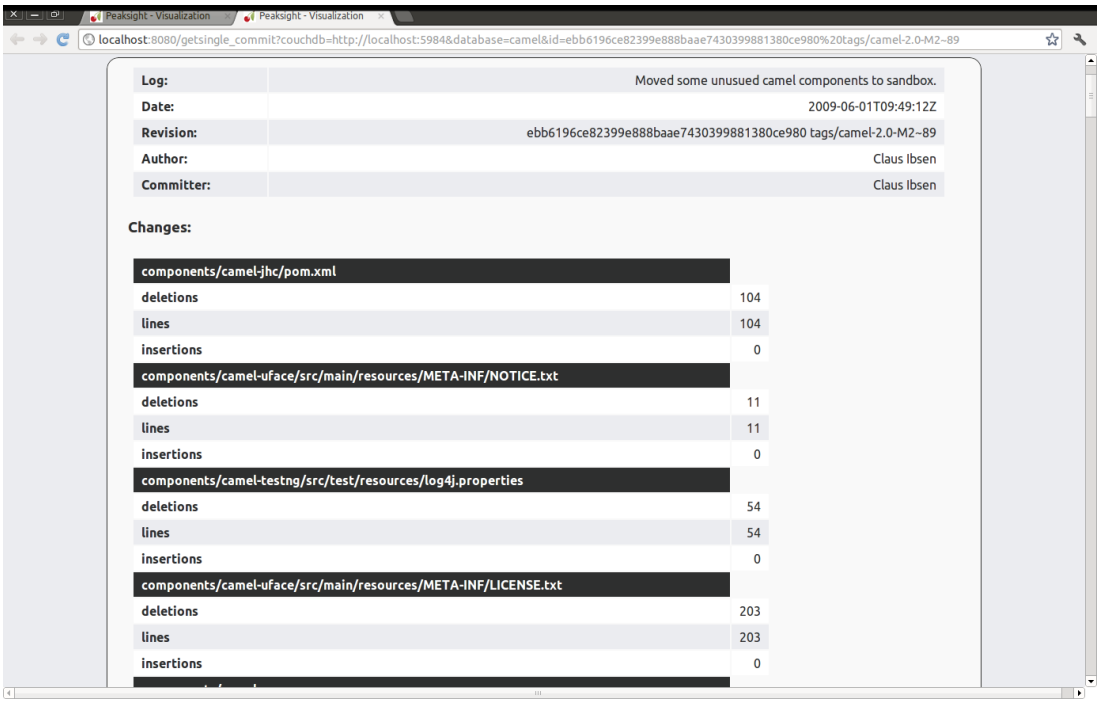


Figure 9. Peaksight: information about a commit

an example of information which can be visualized for a single commit, in particular there are the usual information about the author, the revision the log message, and the date, plus there is the list of changes reporting the types and number of lines modified for each file that has been modified with the commit.

Figure 10. Peaksight: information about an issue

For issues, as shown in figure 10, the visualization is more simple. It shows the summary of the bug, the date in which the bug has been reported, and the last date it has been updated, then it shows the name of the bug reporter and the developer to which the issue has been assigned. Then there is information about the resolution status of the issue and its priority.

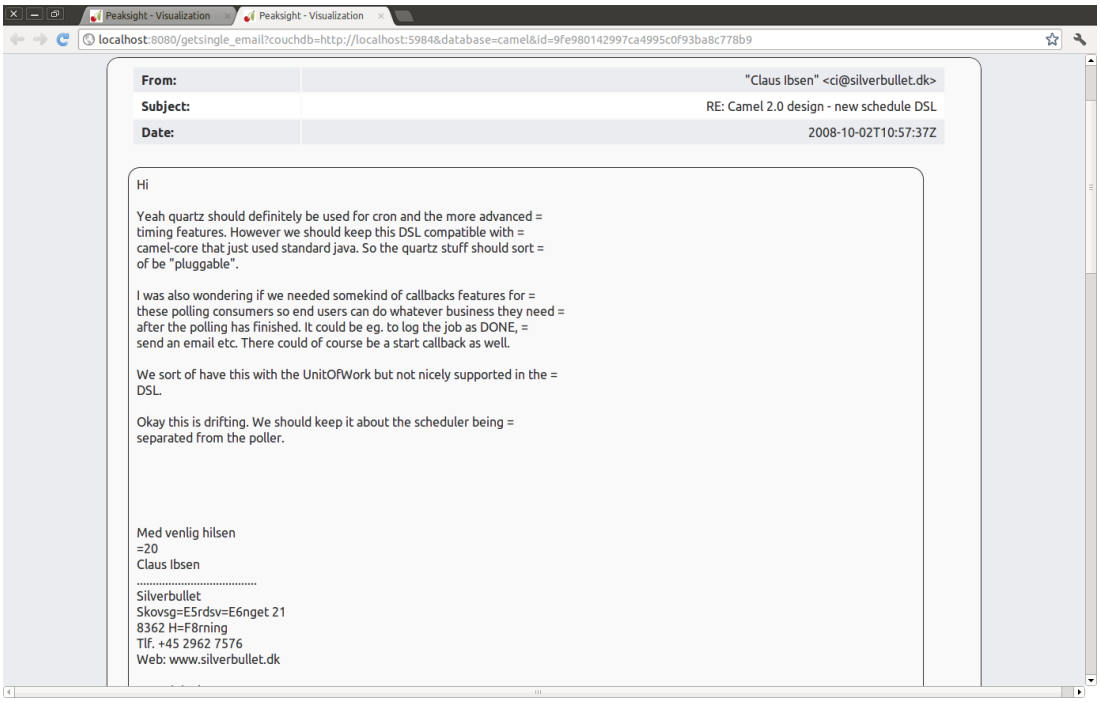


Figure 11. Peaksight: information about an email

It is also possible to read every email present in the archive as shown in figure 11. By clicking on an email it will be shown the correspondent email, with some basic information about the author, the date the email was sent and the subject of the email.

Filtering:
One important interaction available in `Peaksight` is data filtering, which allows the user to obtain a visualization with only commits, emails and issues respecting some desired properties. For example, with this ability it is possible to show in the visualization only commits made by an author, or bugs assigned and opened by the same developer, etc.
`Peaksight` includes the possibility to filter data because we believe that such a feature can result in great benefits when using `Peaksight` to explore a software system with the purpose to understand its evolution.
To perform filtering the user have available an interface, which is positioned below the main visualization of the software system being explored, Figure 12 shows how the "filtering interface" appears.

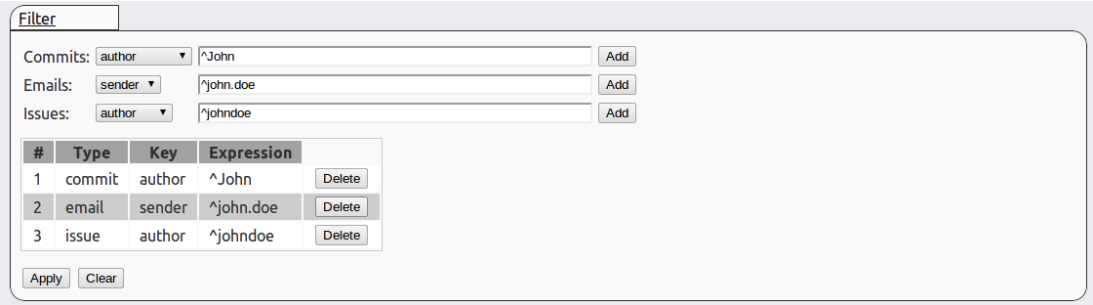


Figure 12. Peaksight: interface for filtering data

For each of the three artifacts, commits, emails, and bugs, the user can specify any number of filters. Each artifact has some properties on which is possible to add filters: For commits it is possible to apply filters based on author, commiter[7] and log message; for emails it is possible to apply filters based on sender or subject; while for issues it is possible to apply filters on author or summary. The filter is expressed by a regular expression, which must be in the format Python uses for regular expressions. For example the expression `^johndoe` means "everything that starts with the string *johndoe*". When a user adds a new filter it will be add to the table below the three artifacts, which shows a summary of the filters currently added. Once all the desired filters have been added to the list it is possible to click on the "Apply" button to see which effect they have on the visualization of the software system.
If two or more filters are applied to the same artifact they are concatenated with a logical AND; for instance if we add two filters on the property author for commits that states `^john` and `^jack`, i.e. authors which names starts with "john" AND "jack", the result of the combination is null.

[7]The difference between author and committer is subtle. In Git could be possible for anybody to wrote a patch and submit it to the project repository, thus the person who submitted the patch will be denoted as the author of that patch. The committer is the developer, which has the right to modify the project repository, and therefore applies the patch to the project. Therefore if a developer sends a patch and applies it, he/she will be both the author and the committer.

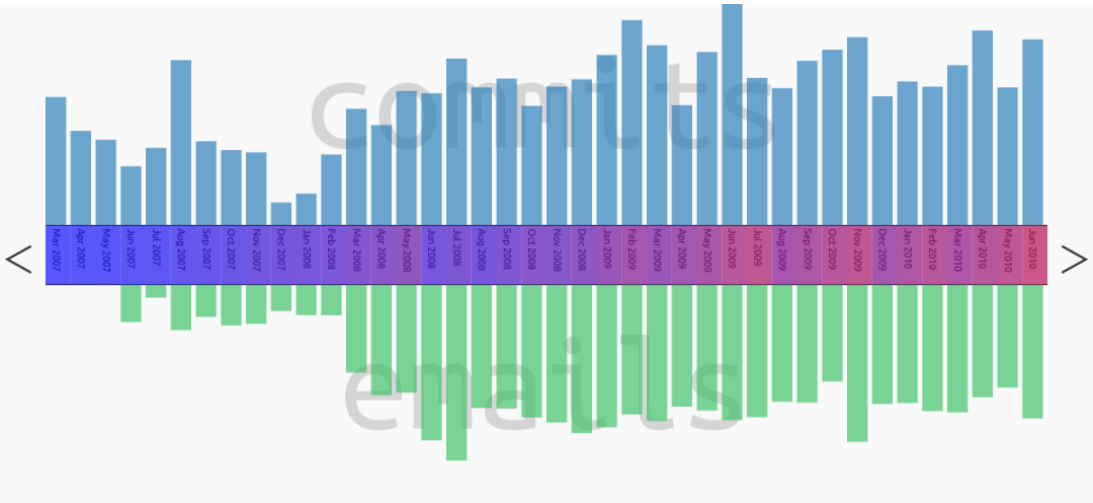


Figure 13. Peaksight: example of visualization before apply filters

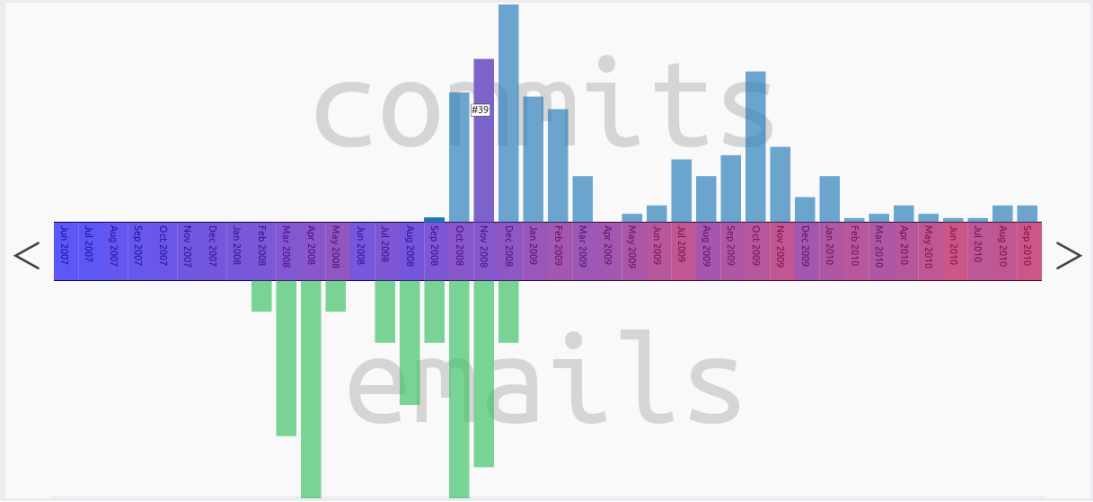


Figure 14. Peaksight: example of the result of filtering data

Figures 13 and 14 shows an example of the visualization without filters and the visualization resulted by filtering the data relative to a single developer, i.e. all the commits made by the developer, the bugs he opened, and the emails he sent. In this example the developer is *Jonathan Anstey*, who collaborates to the *Apache Camel* project, a software system that we will analyze in detail in section 4 of this document.

Filtering data is not a trivial task, the interested reader can find details about the filtering implementation in Appendix A.2.

3.2 Mr.Bubbly

The second tool we present is Mr.Bubbly. Its visualization is inspired by the famous TED talk "Wealth & Health of Nations"[13] by Hans Rosling in 2006, where he shows the evolution of nations based on their life expectation and their income per capita.

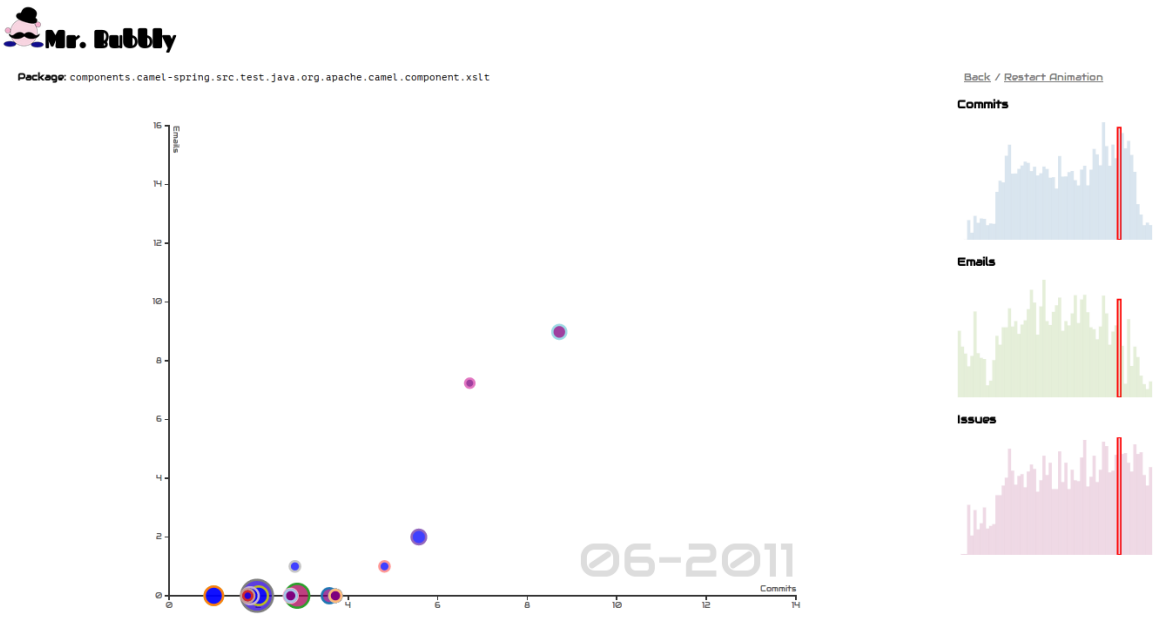


Figure 15. Mr.Bubbly: a momentum of visualization

Figure 15 shows an example of a momentum in a visualization offered by Mr.Bubbly. We distinguish four charts: The three bar chats on the right, which represents the number of commits, emails, and issues for each month of the development of a software system; and the "main chart" which shows the bubbles. Each bubble represents a file in the system, while its position is given by the number of commits that affect the file on the x-axis, and the number of emails that mention the file on the y-axis. The number of commits and emails are cumulative, for example if for the file "foo.java" there are 4 commits on May 2012 and 3 on June 2012, the value used to construct the visualization are 4 for May 2012 and 7 for June 2012 ($3 + 4 = 7$). The size of the bubble is determined by the number of lines of code of the file, the more lines it has the bigger the bubble is. About the color of bubbles, there are two things to mention: the border and area of the bubble. For the border we use different colors for each files, so that it is more easy to watch the behaviour of a file during the animation. For the area we use a color scheme based on the bug opened which are relative to a file. The color scheme goes from blue to red depending on the number of issues for each file, where blue identify the presence of few bugs and red identifies the presence of many issues.
At the bottom right of the "main chart" a big label shows the date, month and year, corresponding to the current period in the animation. The same date is also shown in the small charts on the right, where a red border is drawn on the corresponding bar. This is used to show where the current visualization is positioned with respect to the overall time of development of the considered software system.

The visualization of Mr.Bubbly is dynamic. The "main chart" shows an animation of the changes made to files during the time of development, starting from the first date until the last date available for them.

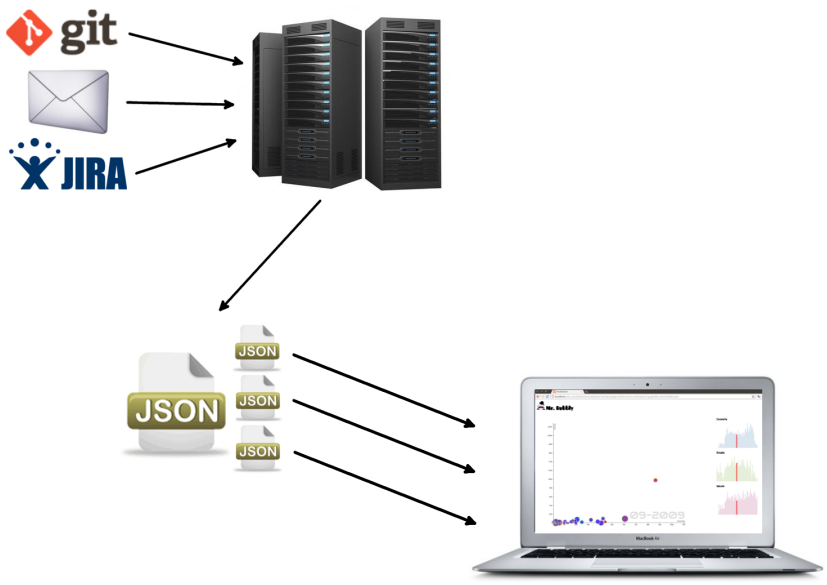


Figure 16. Mr.Bubbly: overview of the architecture

Figure 16 depicts the architecture of `Mr.Bubbly`. Here is possible to see how the design of `Mr.Bubbly` is simplified. It just consists of a set of utilities that collect the desired data for commits, emails, and bugs; some other utilities that are responsible to construct the file containing the data ready to be used for the visualization, in JSON format; and the function used for the visualization itself.
With this simple design we are able to build visualizations much faster than if we were to use a database to get the data and then send them to the application, but there is also the drawback that using precomputed data restrict the possibility of interaction with the visualization, for instance, is not yet possible to create a filtering function like the one for `Peaksight`.
Data are again collected using python and we decided to use Git as the source for commits, mbox files as source for emails and JIRA as bug tracker information source. Once data has been collected it is then processed to create JSON file containing data ready for the visualization. In particular there are three kind of json files used for the visualization: the file containing the number of commits, emails and bugs for each month, which is used for the three little bar-charts, a file containing the structure of the package of a software system presented as a tree structure, and a set of files containing the informations about commits, emails, issues, and lines of code for each file of a package of the software system (there is one file for each package in of the system).
To create the visualization we take advantage of the modern web technologies, in particular we use HTML5, CSS3, JavaScript and the d3.js[8] library to draw the charts and perform the animations.

3.2.1 Interaction in `Mr.Bubbly`

The welcome screen of `Mr.Bubbly` presents just a selection of the available software systems the user can currently explore with `Mr.Bubbly`. After having selected the desired system, `Mr.Bubbly` shows a "bubble representation" of the packages in the system chosen, where each package is a bubble. Packages which are sub-package of a bigger package are identified by the same color and placed one near the other. For example, the packages *"org.apache.tools.ant.util.facade"* and *"org.apache.tools.ant.util.depend"* are two sub-package of the package *"org.apache.tools.ant.util"*.
Thus, is possible to easily spot related packages by looking for clusters of bubbles of the same colors. When the user goes with the mouse over a bubble it is shown the full name of the packages.
Bubble size is based on the number of files the packages contains, i.e. if the package contains many files the bubble will be large, while if the package contains few files the bubble will be small. For instance, there are some big bubbles which corresponds to the packages containing a lot of files, while there are many packages which have small size. It is possible to easily recognize these characteristics without having any knowledge about the project. We believe that this is an important feature, because the possibility to spot the biggest packages in the system can give insight about which packages contain a lot of source code.

[8] d3.js: `http://d3js.org/`

When user selects a bubble by clicking on it, the visualization changes. A new page is opened with the visualization of files for the package selected.

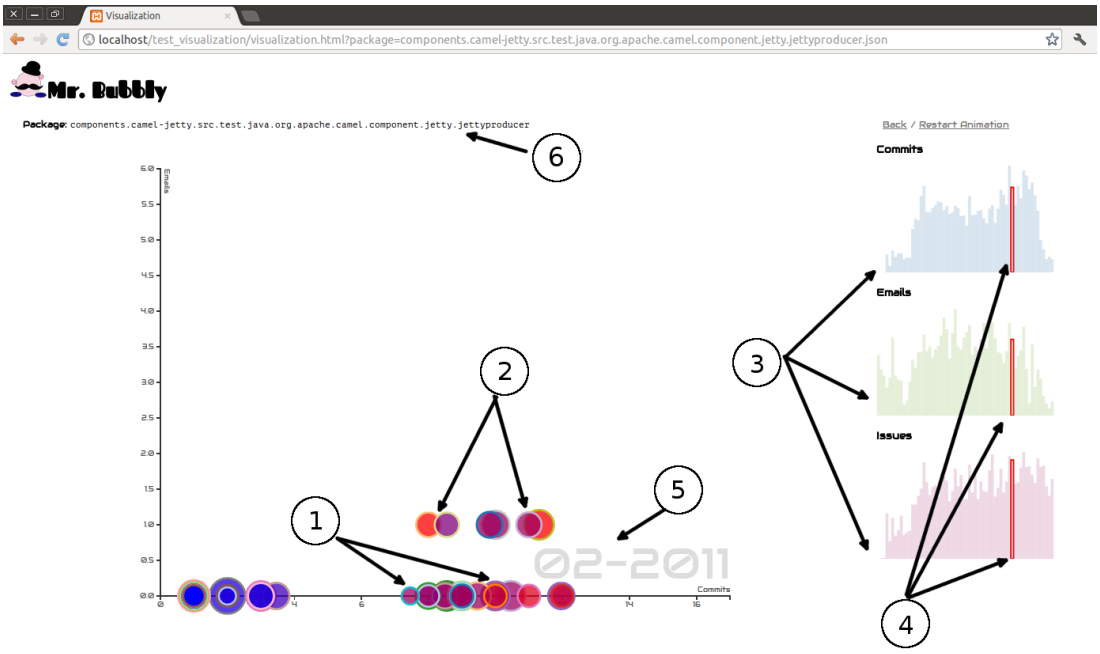


Figure 17. Mr.Bubbly: visualization of files in a package (with legend)

Figure 17 shows the visualization of files which belongs to a specific package, next we describe some components of this visualization. The components of the visualization are:

1. The bubbles, each representing a different file. Their position is given by the numbers of commits for that file (x-axis) and the number of emails that make references to that file (y-axis). The size is given by the number of lines of codes, while the color represents the bugs opened.

2. Each border of the bubbles should have, in theory, a different color depending on the file associated with it. Of course there are a finite numbers of colors in this world, and we use a color scale of 20 colors, while the number of files that belongs to a package could be very big, therefore there some color may repeat. Nevertheless, we believe that having a different color on the bubble border helps to distinguish different files during the animation.

3. The three bar-charts at the right of the visualization are used to show an overview of the amount of commits, emails and issues for each month of the project.

4. Associated with the bar-charts there is an animated red border for the bars. It is used to highlight the month displayed in the main chart. This border is intended to be a pointer to where the position of the current month, in the overall time of development of the project.

5. Inside the main chart it is also displayed the date relative to the month currently visualized. This date has two purpose, the first is obviously, to inform user about which month the visualization is about, while the second purpose is to have the ability to choose the month to visualize after the animation is completed. When the animation is finished the user can move his mouse pointer on the date and trigger the visualization, movement to the left will show preceding months, while movements to the right will show succeeding months (if presents).

6. At the top of the page, below the log of the tool, is displayed the complete package name, so that is possible to know exactly which package is shown in the animation.

Other than use the date label to trigger the visualization, the user, can also click on a specific bar of one of the three bar-charts. When doing this the main chart will show the files and the date relative to the month selected.
The main difference between these two ways of trigger the visualization is that by clicking on a bar makes the main chart “jump” directly to the selected month, while if the user moves the mouse on the data label, the chart smoothly shows the transitions of the bubbles from the current month until the selected month.

4 Telling the story of Apache Camel

Apache Camel[9] is an open source framework written in Java with the purpose of making integration of transport APIs easier and more accessible to developers. To accomplish this goal, Camel provides many concrete implementation of widely used EIPs (Enterprise Integration Patterns), allows connectivity to a great variety of APIs (Application Programming Interface) used as transport layer. Camel provides a DSL (Domain Specific Language) which is used to wire EIPs and transports together. Therefore it makes possible to use different APIs and EIP, but using a single framework.
Apache Camel can support type-safe smart completion of routing rules in many different IDEs using regular Java code without having to create huge amounts of XML configuration files.

4.1 The story by `Peaksight`

We start exploring the evolution of the "Apache Camel" software system by the overview of the development it had divided in year from its beginning in 2007 until the last data we collected (begin of May 2012). Figure 18 shows commits, emails, and bugs of Camel divided by years. We see that 2009 was the year when the highest number of

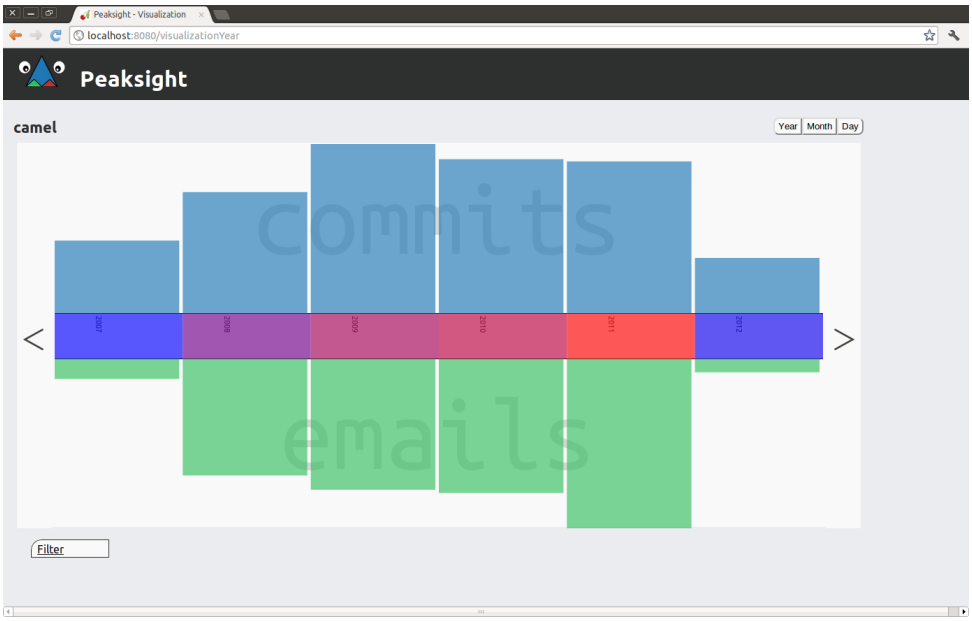


Figure 18. Peaksight: years of development of "Apache Camel"

changes have been done to the project, while in 2011 a lot of communication happened among developers, also the number of bugs opened is higher in 2011 respect to the other years, but drastically reduced in the following year. We can, then suppose that many commits done in 2011 were made to correct issues.

Year	Commits number	Emails number	Bugs number
2007	1173	1190	280
2008	1954	6942	1193
2009	2728	7794	1611
2010	2484	7988	1795
2011	2448	10088	2317
2012 (until May)	894	799	407

Table 1. Peaksight: camel development by year

Table 18 shows the number of commits, emails and bugs divided per year, as shown by visualization made by `Peaksight`.

We notice that the commits have, more or less, a regular behavior, i.e. they grow fast in the first three years of development, then they remains stable. But we notice that there is a very high number of emails, especially in year 2011, when there is also the peak of open bugs. If we take a closer look to the list of emails we can see that

[9] `http://camel.apache.org/`

many of them are emails generated automatically by the bug tracker JIRA, to warn developers that a new bug as been opened, closed or commented. Therefore we would like to remove them from the visualization to see how many emails have been effectively written by developers in the mailing list. To do this, we can use the filtering functionality. We open the filter interface, select the Email attributes and add a new regular expression on the "subject" property like (?!^\[jira]\).

Figure 19. Peaksight: emails filtered to remove automatically generated emails of jira

Figure 19 show the result once the filter is applied. Visually there is little difference. But if we move our mouse pointer over the bars representing emails we see some interesting values which are reported in table 2.

Year:	2007	2008	2009	2010	2011	2012 (until May)
Emails no filter:	1190	6942	7794	7988	10088	799
Emails filtered:	310	1272	1616	1370	2475	799

Table 2. Peaksight: emails filtered

Except for the current year (2012), for all the other years, emails sent by developers are nearly $\frac{1}{4}$,$\frac{1}{5}$ of the total emails, all the rest are automatically generated emails by JIRA.
It is interesting the behavior of 2012 respect to the other years, where there are not present any emails automatically generated by JIRA. After a small research in the official website of the project, we discover that starting from November 2011, the developers of Camel decided to create a separate mailing list, from the developer mailing list, specific for issues. Perhaps, they found disturbing having a lot of emails relative to issues in the developer list, which may interferer with a normal discussion of problems during developing.

Another thing we can explore thanks to filtering is the number of commits which are relative to bugs. Camel developers are used to identify commits done to solve a specific bug by pre-pending to the log message of the commit the identifier of the issue, which is something like *CAMEL-2325*. Therefore is really easy to make a regular expression which acts on the log message of the commit and returns only the log messages that starts with *CAMEL-*. The regular expression will look like ˆCAMEL−. Figure 20 show the result of this filter. Again, the visualization is similar to the

Figure 20. Peaksight: commit filtered to display only commits relatives to issue

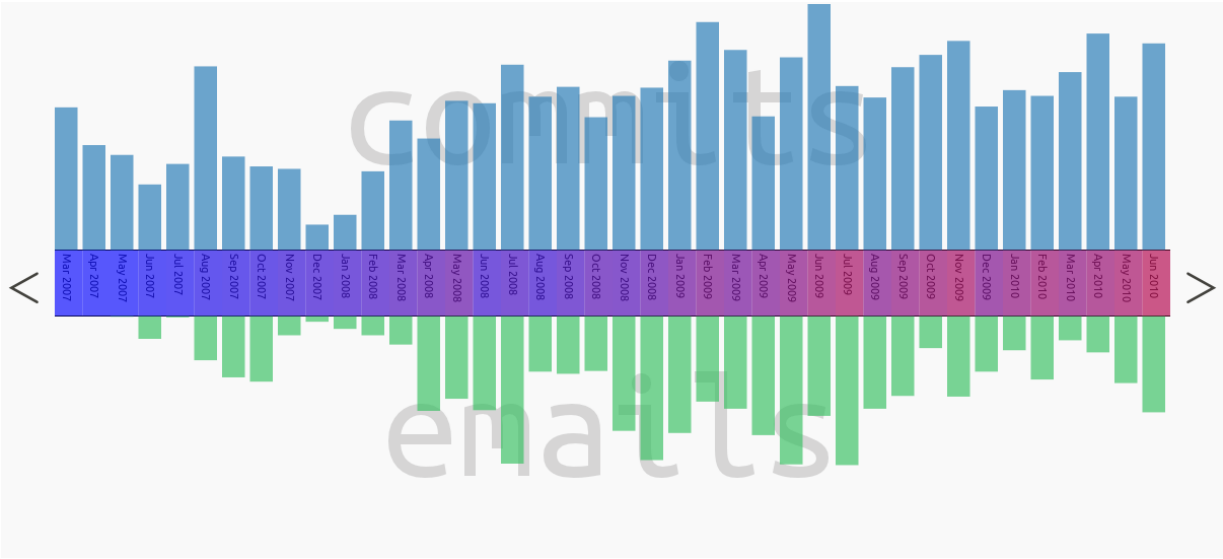
precedents, but if we look at the values displayed we can see that, as happened for emails they are very different, as shown in the table 3.

Year:	2007	2008	2009	2010	2011	2012 (until May)	2012 (prevision)
Commits no filter:	1173	1954	2728	2484	2448	894	2145.6
Commits filtered:	9	1079	1498	1462	1529	521	1250.4

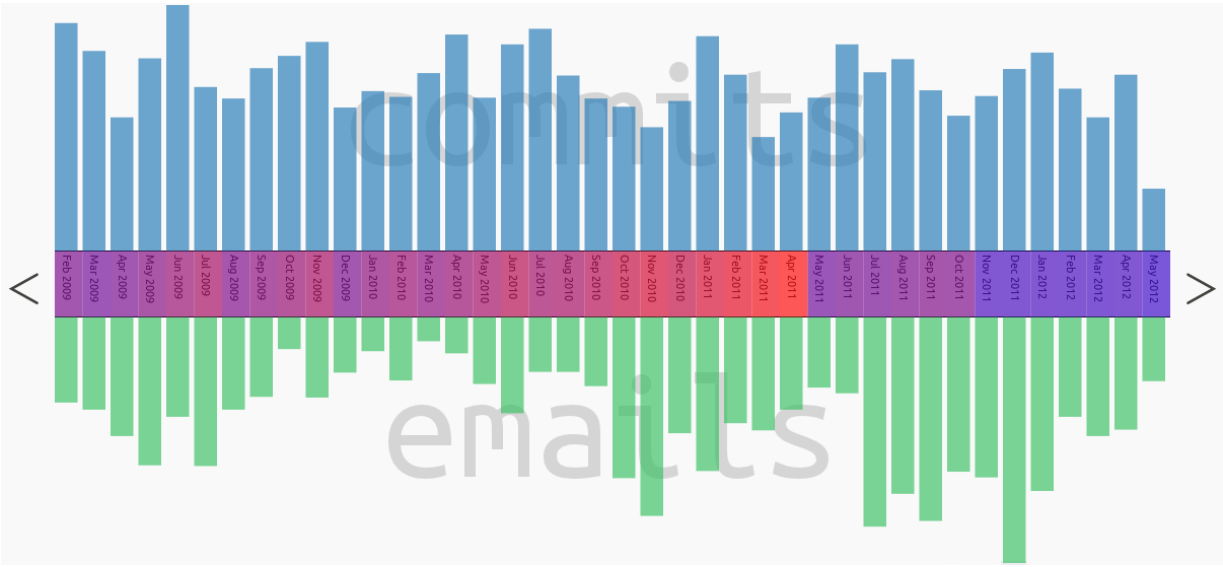
Table 3. Peaksight: commits filtered

We can see that, except the first year of development, for every other year the number of commits relative to bugs are more or less half of the overall commits. This means that half of the development efforts are to correct issues. We would have expected a higher number of commits to correct bugs on the year 2011, but since the number respect the overall behavior we suppose that in 2011 there were many issues easy to solve that didn't require many changes to the code, or some commits solved more than one issue. This behaviour might be an interesting topic for further investigation.

We have seen some interesting things about the project by looking at the the development just by considering years as time intervals. Now we go deep and consider months as time interval, to see if we can obtain other informations about the system.



(a) From March 2007 to June 2010



(b) From February 2009 to May 2012

Figure 21. Peaksight: development of "Apache Camel" in month

Figures 21 shows the development of "Apache Camel" considering as time intervals months. In there we have already "filtered out" the emails automatically sent by JIRA, because they are not useful for our purpose at this moment, since we want to inspect only emails sent by developers to the development mailing list of the project. The first thing we notice is that there is a big period of time where relatively few emails are exchanged among developers, while the number of commits does not have many changes. This period goes from August 2009 to September 2010, that is, more than a year. Interestingly, in the months exactly after this period of time there is the most critical span of time for the project, considering the number of bugs opened for each month. In particular the "problems" starts from the 1194 issues opened in October 2010, until reaching the peak of 1546 issues in April 2011. Table 4 contains the data about emails and commits from August 2009 to September 2010 of "Apache Camel". The last two columns presents a ratio between, respectively, numbers of emails for a month and the maximum number of emails sent in a month during the development; and numbers of commits for a month and the maximum number of commits sent in a month during the development. The maximum number of emails sent was 345 on December 2011, while the maximum number of commits was 300 on June 2009. These numbers confirms the hypothesis we made by inspecting the visualization, i.e., the number of emails sent in this period of time is very low: always less than half the emails sent in the month were the most communication happened, while the number of commits remains more

month	emails	commits	emails relative to maximum	commits relative to maximum
Aug 2009	130	186	0.38	0.62
Sep 2009	112	223	0.32	0.74
Oct 2009	45	238	0.13	0.79
Nov 2009	113	255	0.33	0.85
Dec 2009	78	175	0.23	0.58
Jan 2010	48	195	0.14	0.65
Feb 2010	89	188	0.26	0.63
Mar 2010	34	217	0.10	0.72
Apr 2010	51	264	0.15	0.88
May 2010	94	187	0.27	0.62
Jun 2010	135	252	0.39	0.84
Jul 2010	77	271	0.22	0.90
Aug 2010	77	214	0.22	0.71
Sep 2010	97	186	0.28	0.62

Table 4. Peaksight: ratio among emails and commits form August 2009 to September 2010

or less the same: if we compare the number of commits for each month in this period to the number of commits had in the month were the most commits were done, we have very similar results.
We believe that this behaviour of lack in communication, but continuously changes in the project source code, could have influenced the "production" of issues in the months following this period.

Inspecting this visualization we notice another interesting behaviour about emails and bugs. We notice that when the number of bugs increases from a month to the subsequent month, often we have the same behaviour of emails, i.e when the number of bugs grows also the number of emails grows. Some example of this behaviour are:

- March 2008 to April 2008 where bugs goes from 411 to 474 and emails goes from 40 to 133.
- June 2009 to July 2009 where bugs goes from 899 to 987 and emails goes from 140 to 209.
- October 2009 to November 2009 where bugs goes from 894 to 981 and emails goes from 45 to 113.
- May 2010 to June 2010 where bugs goes form 976 to 1086 and emails goes from 94 to 135.
- October 2010 to November 2010 where bugs goes from 1199 to 1276 and emails goes from 226 to 279.
- February 2011 to March 2011 where bugs goes from 1392 to 1474 and emails goes from 149 to 159.

We can hypothesize that when the number of issues relative to the project increases, developers are stimulated to create discussion in the mailing list, to try to solve these issues.

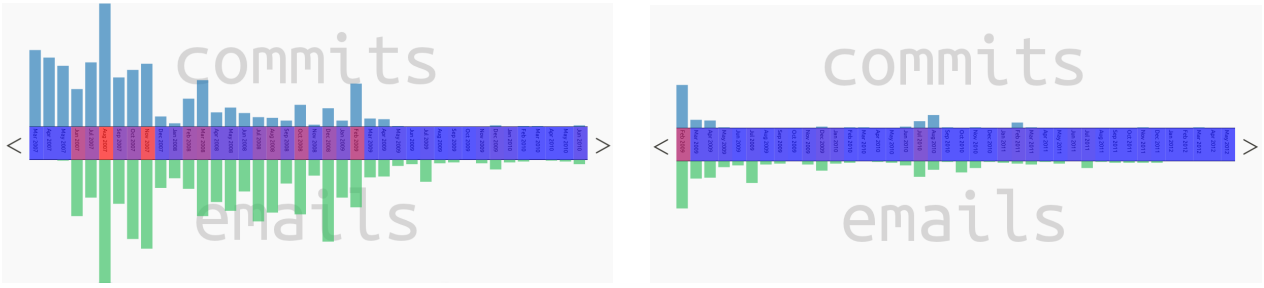
Developers

An important thing that we did not considered yet in this evaluation is the impact of developers in the development process. If commits, emails and issues are important artifacts, then developers are the people responsible for them. Therefore we believe that is important to have also a little look at the contribution made by some developers. Therefore we present some analysis about a small group of developer. We choose three developers: *James Strachan*, *Claus Ibsen* and *Hadrian Zbarcea*. We inspect their work because when exploring the visualization, by clicking on commits bar, their names are often displayed among the names of the authors of commits. Therefore we believe that they are meaningful example to create visualizations for developers of a software system. Moreover, since we can often see their name in the list of commits, probably they are among the developers which contributes the most to "Apache Camel", therefore having a closer look to their work can help to reconstruct the story of the project. To create visualizations about specific developers, we can use the filtering function by specify the name used by the developer in the regular expression for each of the three artifacts: commits, emails and issues. In particular, for the three authors we choose, we have to search for the following names: *James Strachan* uses this name for commits and emails, while for issues is identified by *jstachan*, *Claus Ibsen* uses this name for commits, for emails uses the same plus *davsclaus*, which is also used for issues, at last *Hadrian Zbarcea* uses this name for commits and emails, while for issues uses the name *hadrian*. These names are relatively easy to find just by comparing names in the list of commits, emails and bugs. The "Apache Camel" official page of committers and contributors[10] to the project

[10] `http://camel.apache.org/team.html`

confirms that the name we found are correct.
For each developer we create a different visualization of the contributions he made to “Apache Camel”, in the visualization is possible to see the changes to source code by amount of commits sent by month, the number of emails he sent and the open issues that they have created for every month. Again, for issue red means that the developer in that month has the highest number of issues assigned to him while blue means that in that month the developer has the lowest number of issues assigned to him.
In the following paragraphs are discussed the contribution made to “Apache Camel” by the three developers.

James Strachan:
Figure 22 shows the contributions made by James Strachan to the “Apache Camel” project.

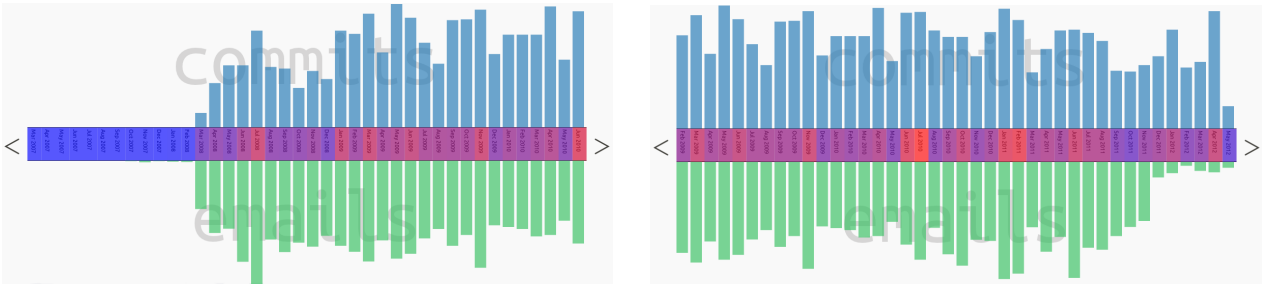


(a) From March 2007 to June 2010 **(b)** From February 2009 to May 2012

Figure 22. Peaksight: contributes of James Strachan to “Apache Camel”

We immediately notice that Strachan was very active at the beginning of the projects, in terms of commit. We can say, that probably he was the main developer for the first year of developing “Apache Camel”, since if we compare the number of his commits to the overall commits, from March 2007 to March 2008, he made most of them. In this period of time he also had assigned the most number of bugs while working to this project and was also active in the mailing list. We can guess that Strachan was among the developers who designed the initial implementation of “Apache Camel”.
After this first year, the contributions of Strachan starts decreasing, and from May 2009 Strachan made only few occasional commits usually when he opens some issues. He is a little more active in the front of emails communication, even if he usually does not send more than about a dozen of messages.

Claus Ibsen:
Figure 24 shows the contributions made by Claus Ibsen to the “Apache Camel” project.

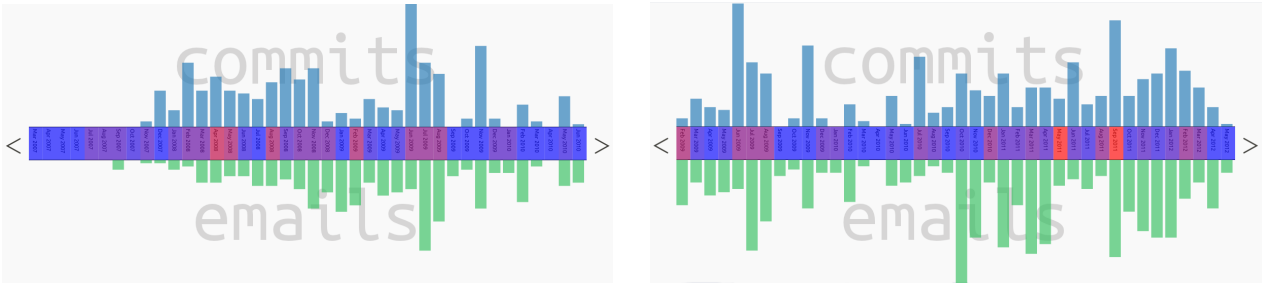


(a) From March 2007 to June 2010 **(b)** From February 2009 to May 2012

Figure 23. Peaksight: contributes of Claus Ibsen to “Apache Camel”

Basically Ibsen continued the work of Strachan, he joined the project in March 2008, after having sent some few emails messages in the development mailing list in the preceding months.
Hence, exactly when the main developer of the project was decreasing his contribution, a new developer joins and from the beginning starts to heavily contribute. Ibsen is pretty constant in his contributions to the project, and if we compare his number of commits per month to the overall number of commits, we can see that more than half the commits for each month are made by him. We can then think that, by now, he is the main developer of “Apache Camel”, with more than 4 years of heavily contributions. Other than commits, Ibsen has also quite constant number of emails sent and bugs opened.

Hadrian Zbarcea:
Figure 24 shows the contributions made by Hadrian Zbarcea to the "Apache Camel" project.



(a) From March 2007 to June 2010 **(b)** From February 2009 to May 2012

Figure 24. Peaksight: contributes of Hadrian Zbarcea to "Apache Camel"

Zbarcea joins early to the project, his first commits was on November 2007, while his first email to the mailing list of the project was on July 2007, such as the first bug report. Compared to the two precedent developers, Zbarcea is a more ordinary contributor. In fact he constantly makes a small amount of commits for each month, except rare exceptions where he didn't make commits or the two "peaks" at 44 (June 2009) and 38 commits (September 2011). He also sends a considerable amount of emails for each month, which tells that he is rather active in the communication with other developers. Occasionally he have opened some bugs, but never a lot for each month, indeed his maximum number of bugs opened per month is 8 (May 2011 and September 2011).

It is possible to create visualization for all the developers who contributed to "Apache Camel", but this would require a lot of space and time, thus we are not going to do it here, since the purpose of this section was to create an example of how it is possible to inspect how some developers contributed to the project. Anyway we succeed in find some meaningful examples of developers, in particular Strachan and Ibsen, which are, in two different period of the project the main developers of "Apache Camel".

And this, concludes the story of the evolution of "Apache Camel" seen by the perspective of `Peaksight`.

4.2 The story by `Mr.Bubbly`

We start the story of "Apache Camel" by inspecting how the source code is organized in packages. Figure 25 shows

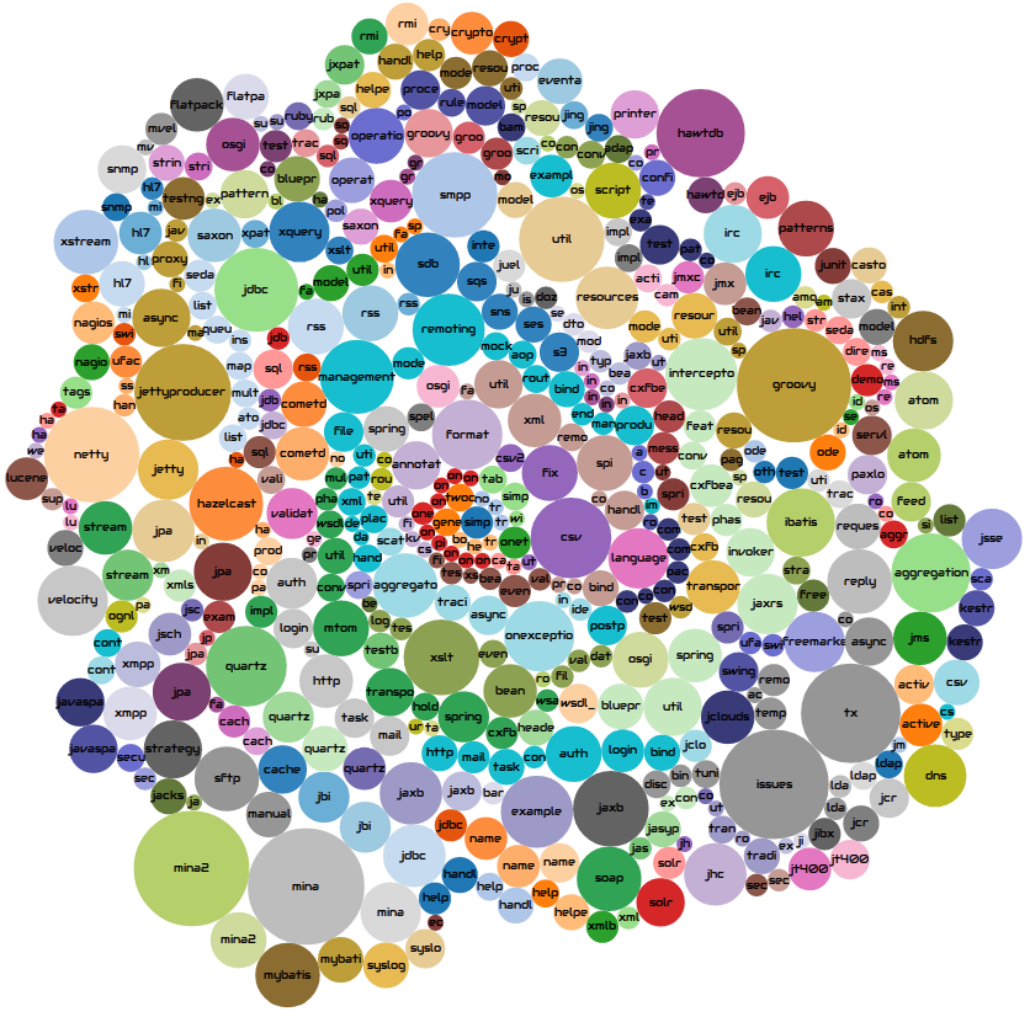


Figure 25. Mr.Bubbly: the packages of "Apache Camel"

a visualization of the packages of "Apache Camel", made with `Mr.Bubbly`. Every bubble represents a package. We can notice that "Apache Camel" has a huge amount of packages, going from packages with small size, i.e. with few classes or files, to packages with big size, i.e. with many classes or files. We then notice that usually, packages with similar size are, displayed near each-others and with the same color, this means that they are sub-package of the same package. We deduce that developers of "Apache Camel" have the tendency to create clusters of packages which have packages that contain similar amount of files or classes. Particularly there is a large amount of small packages, like those represented by the bubbles in the center of the visualization in figure 25, while very big package are more rare and they are close to other big or medium packages or even alone.

When we move our mouse over the bubbles we can see a pop-up showing the full name of the package. Thanks to this we can see another characteristic behaviour of "Apache Camel", which is that most of the bigger bubbles are of packages containing test classes. We know this, because in the name of the packages we find the word "test", which is usually used to identify a test!

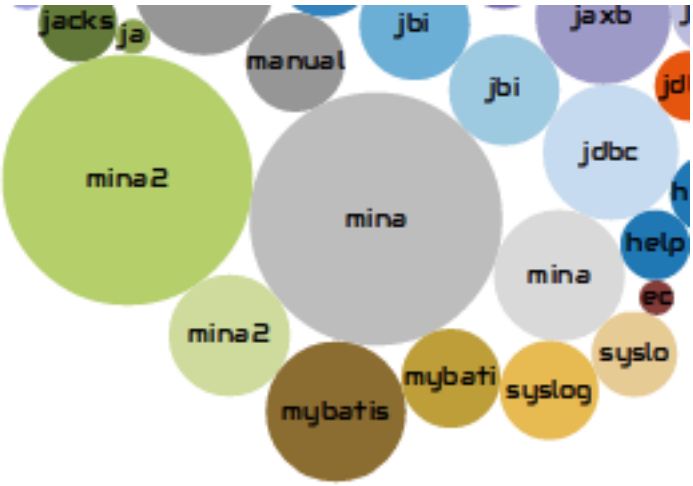


Figure 26. Mr.Bubbly: comparison of the size of a package and the correspondent test package

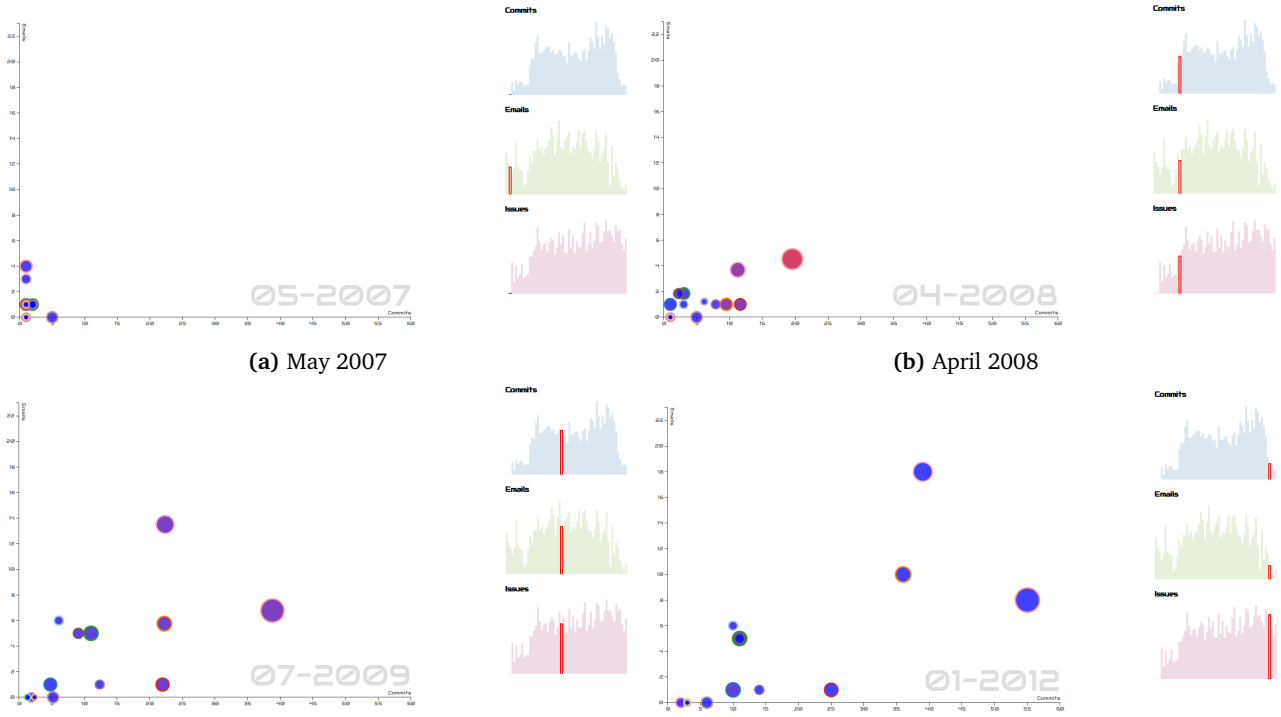
For example figure 26 shows the difference between a package and the corresponding test package, we can see that both the package *"mina"* and *"mina2"* have modest dimension, while the corresponding test packages which are casually displayed near in darker color, are at least twice or three times as big. Also the packages *"mybatis"* and *"syslog"* and their respective test packages present a similar behaviour, even if less evident. Since these packages have a large number of tests files we can hypothesize that developers of "Apache Camel" are careful in providing tests files for the classes they develop. This is a very good thing, because it produces a good impression of solidity about "Apache Camel", that considering the fact that it is a framework which provides a unified interface for many APIs, this is a really good thing.

Next we look at some visualization of packages we think are interesting for the purpose of exploring the system. We present some snapshot of their visualizations (4 snapshot for each package) and we analyse what we obtain form them. For obvious reason of time and space we can not perform here a complete analysis of the system, but we try to analyse some packages we believe that are important. The packages that we analyse are eight in total and are:

- *"components.camel-mina.src.main.java.org.apache.camel.component.mina"*,
- *"components.camel-mina.src.test.java.org.apache.camel.component.mina"* (the test file for the previous package),
- *"components.camel-web.src.main.java.org.apache.camel.web.util"*,
- *"components.camel-smpp.src.main.java.org.apache.camel.component.smpp"*,
- *"components.camel-lucene.src.main.java.org.apache.camel.component.lucene"*,
- *"components.camel-rss.src.main.java.org.apache.camel.component.rss"*,
- *"components.camel-rss.src.test.java.org.apache.camel.component.rss"* (the test file for the previous package),
- *"components.camel-jetty.src.main.java.org.apache.camel.component.jetty"*.

mina:

The package *"mina"*, should refer to the "Apache MINA" project, which is an application framework for high scalability network applications. This package should contains classes and method to build the integration with MINA, therefore should be interesting to explore in detail. Figure 27 shows 4 snapshot of the development of the package



(a) May 2007 **(b)** April 2008

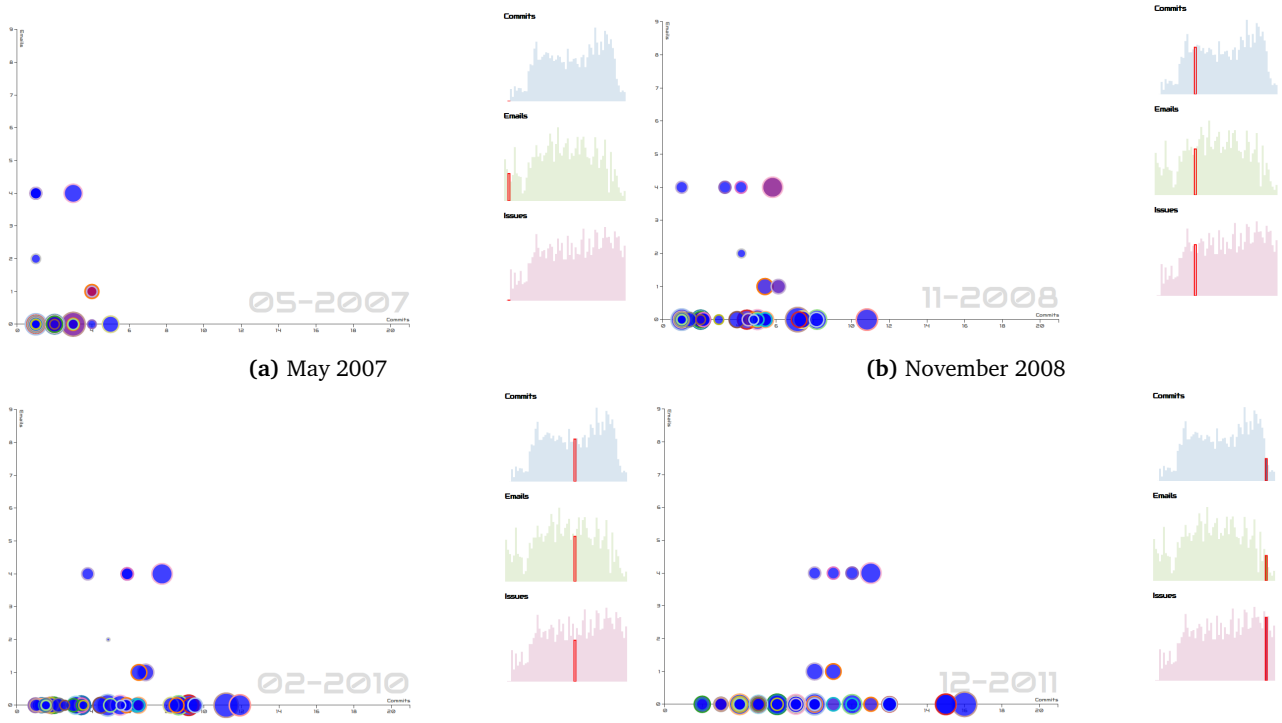
(c) July 2009 **(d)** January 2012

Figure 27. Mr.Bubbly: the "mina" package for "Apache Camel"

"mina". Each bubble is a file. We notice that the development of the considered files starts on May 2007, therefore early in the overall time of development of the project; here there are few commits and emails for each file. At this moment there is *MinaComponent.java* which has 4 emails and only 1 commit, and *MinaUdpProtocolCodecFactory.java* which has 5 commits and no emails. Bugs are really few for all the files as denote by the blue color of the bubbles. The next snapshot shows *MinaComponent.java* and *MinaProducer.java*, which was just below the first in the previous snapshot, which receive a lot of commits reaching respectively 19 and 11 commits, while remaining constant in the amount of emails. It is important to notice that many files, in the months that passed between the two snapshot, received an important number of commits, but very few emails, while in the meantime some bugs were opened on them, indeed the bubble of *MinaComponent.java* changed color from blue to red. The third snapshot shows *MinaComponent.java* that continues to receive a lot of commits and slightly increases the number of emails, and decreases a little bit the number of bugs. While *MinaProducer.java* receives less commits but an impressive numbers of emails, which makes it reach height 14 emails, also for it the number of bugs decreases. Another file, *MinaConsumer.java*, which is the bubble with orange border receives a good number of commits and emails, which means that some work has been done on it. The last snapshot shows three files that received much more commits and emails respect to the other, and are *MinaComponent.java* with 55 commits and 8 emails, *MinaConsumer.java* with 36 commits and 10 emails, and *MinaProducer.java* with 38 commits and 18 emails. This fact, denotes that these three files are the core part of the package "mina". We also notice that all the bubble are blue, therefore doesn't have many bugs, hence we believe that at this time the package "mina" can be considered stable. Furthermore we want to highlight the fact that this packages has begun to be developed early in the history of the project, and it is still object of modifications nearly five year later.

mina (test):

This package is the package that contains classes used to test the package analysed before, hence we believe that it could be interesting to take a look to what happens to the package containing tests, and if there are some correlations between it and the package tested. We can see with just a quick look that this package has a completely different behaviour respect to the precedent, indeed the balls have pretty constant linear progress, where the commits increases, while the emails remains constant. Also bugs are more or less the same for all the time, except for a



(a) May 2007 **(b)** November 2008

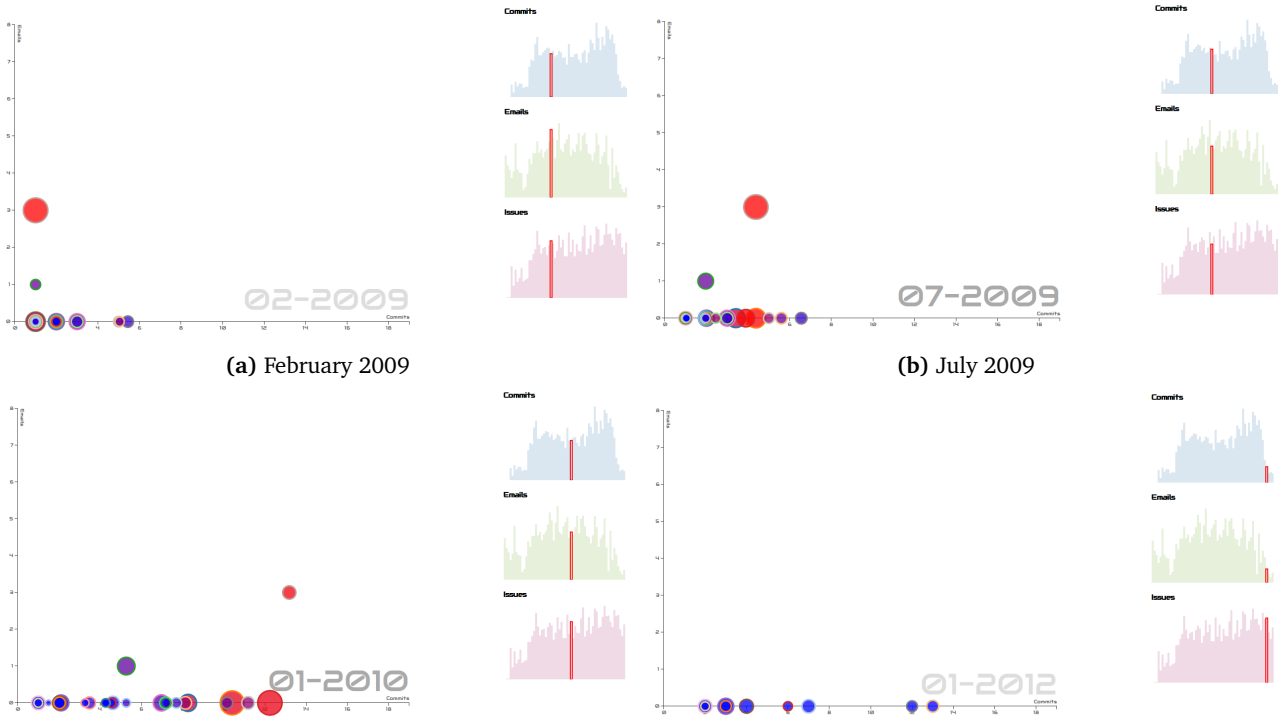
(c) February 2010 **(d)** December 2011

Figure 28. Mr.Bubbly: the "mina" test package for "Apache Camel"

few rare exceptions, like those on snapshots two and three.
The time of development is similar to the time we see in the "mina" package, and, as the package visualization suggested, there are much more files in the test package than on the other package. Also we notice that there are no files with a name which explicitly refer to the files we see in the visualization of "mina". Therefore we think that developers of "Apache Camel", names test files with the functionality they need to test, and not with the name of the file they test. In other words developers don't makes test based on the method of a class but, they make them based on the functionality of a certain action. For instance the file *MinaConverterTest.java* can contain tests for converters functionalities, or the file *MinaDisconnectTest.java* can be used to test a function that perform a disconnection from something on MINA.

util:

We decide to inspect this package because it is among the bigger bubbles which doesn't represent a test package. The development of this packages starts at February 2009, therefore when "Apache Camel" is in development for almost two years. Figure 29 present four snapshot of the visualization for the package "util". In this package few



(a) February 2009 **(b)** July 2009

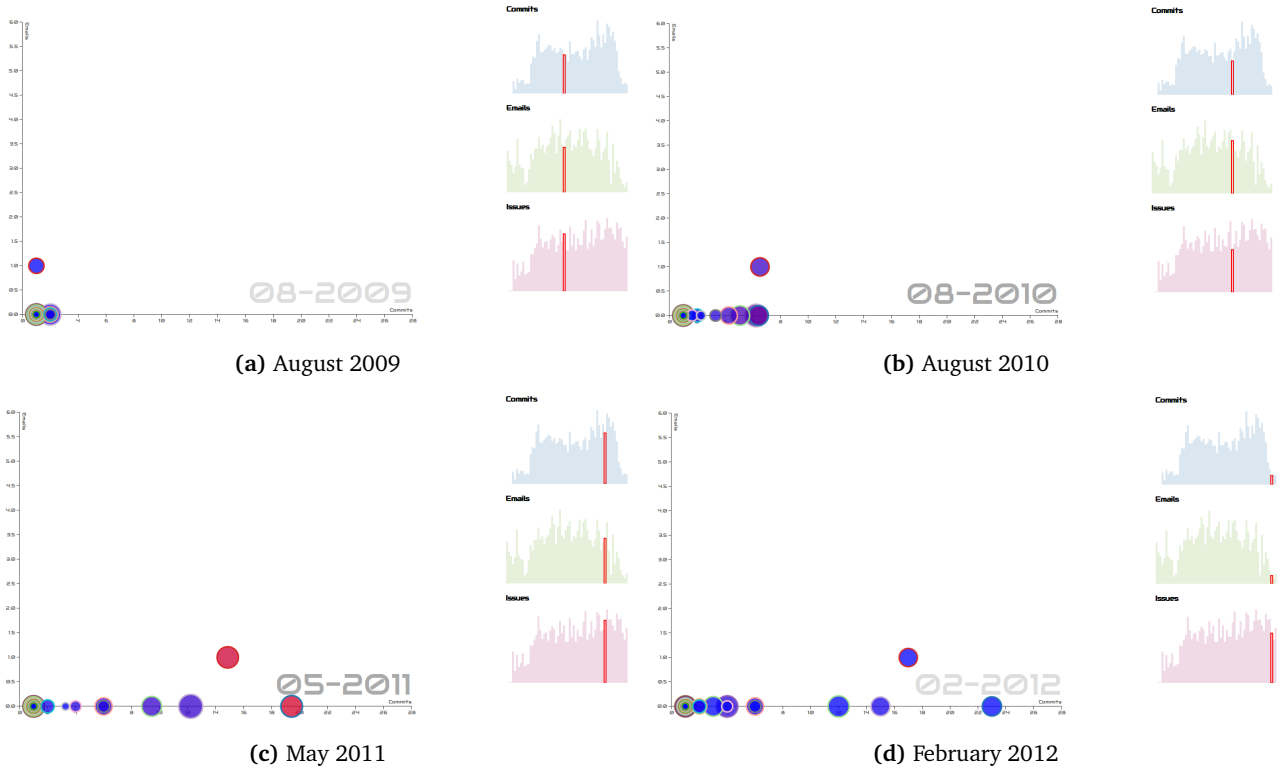
(c) January 2010 **(d)** January 2012

Figure 29. Mr.Bubbly: the "util" package for "Apache Camel"

communication happens, indeed the maximum value for emails is 3 and is at the beginning. We can see that many files during the time of development on this package present a red color, which means that they have a significant amount of bugs opened on them. In particular the file *GroovyRender.java*, i.e. the big red ball which has higher number of emails, has a red color for all the time. Is interesting to see that in the last snapshot there are present fewer bubbles respect to the beginning, this means that some file have been removed. Even *GroovyRender.java* disappears, indeed its size continues to decrease over time, while the number of bugs remains constant, until it is definitely removed.

smpp:

smpp is another big package that is not a test package, thus is interesting to explore it. Figure 30 present four



(a) August 2009

(b) August 2010

(c) May 2011

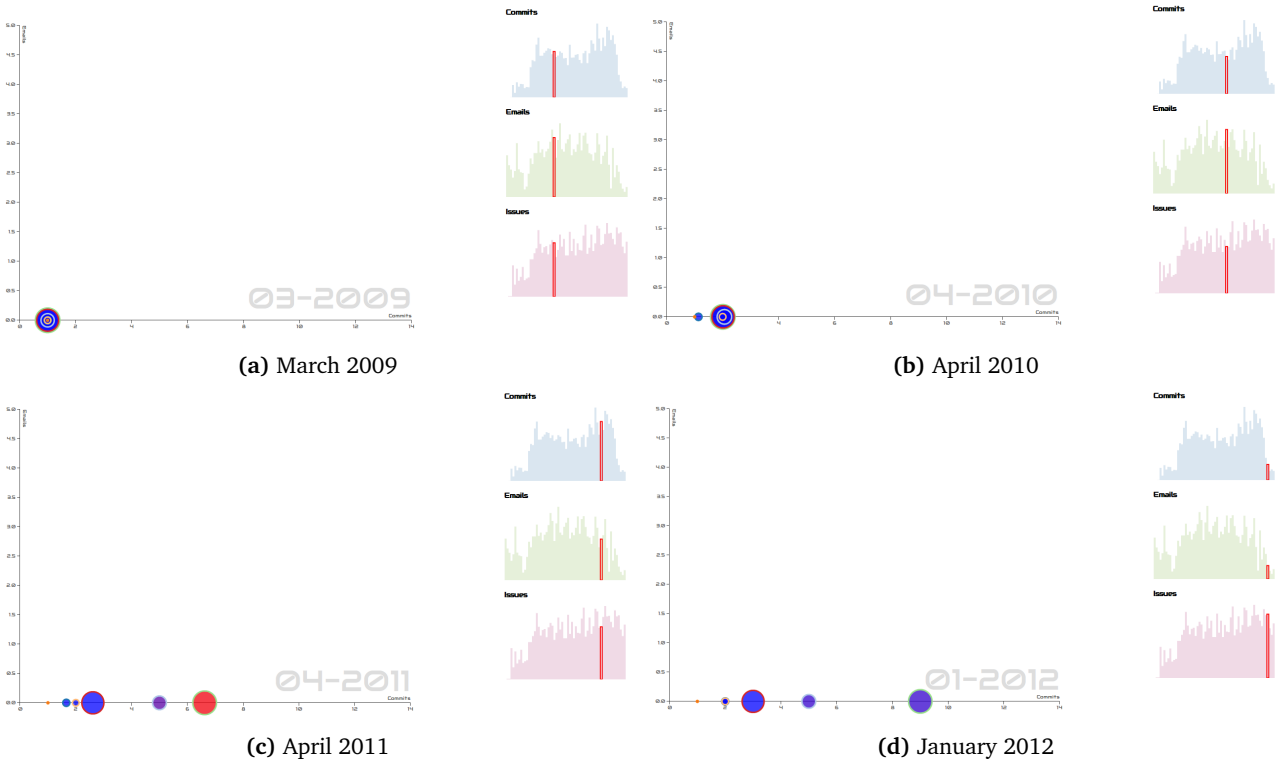
(d) February 2012

Figure 30. Mr.Bubbly: the "smpp" package for "Apache Camel"

snapshot of the visualization for the package "smpp". This package present some similarities with the precedent, indeed the files has a linear behaviour in time, i.e. there is an increasing of commits, but not for emails. The latter are also very few: only one file has a single email associated with it. Around May 2011 there is the presence of few bugs in some of the files, but they are quickly solved in the succeeding months. We also notice that there are many files created at the beginning of the development of this package and then developers do not make changes on them, hence the file remains around zero for commits and emails count.

lucene:

The package “lucene” is an example of a medium size package inside the “Apache Camel” project. Then it is probably related to the project “Apache Lucene”, a famous text search engine written in java. Figure 31 present four



(a) March 2009 **(b)** April 2010

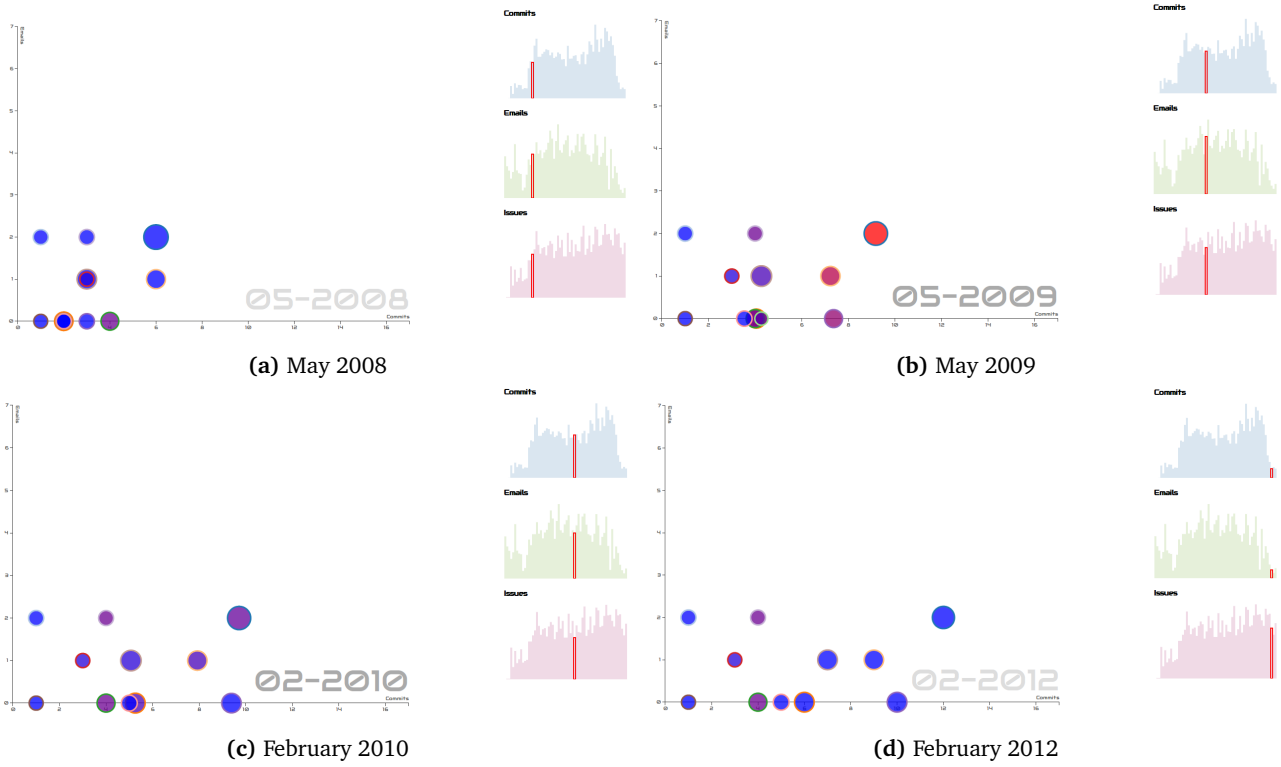
(c) April 2011 **(d)** January 2012

Figure 31. Mr.Bubbly: the “lucene” package for “Apache Camel”

snapshot of the visualization for the package “lucene”. We notice that overall, this package has very few commits and absolutely no emails. This makes us think that the package “lucene” is a marginal package for the “Apache Camel” project. The first two snapshot shows that in a year there was basically no development on this package, while the third snapshot shows that in the following year there have been some commits, and also some open bugs, particularly on one file. Last snapshot shows that only the file on which there were open bugs have been committed and the bugs were resolved.

rss:

The package rss seems to be particular because it has an associated test packages which has similar dimension. Therefore we think that is interesting to inspect both of them. Figure 32 present four snapshot of the visualization



(a) May 2008 **(b)** May 2009

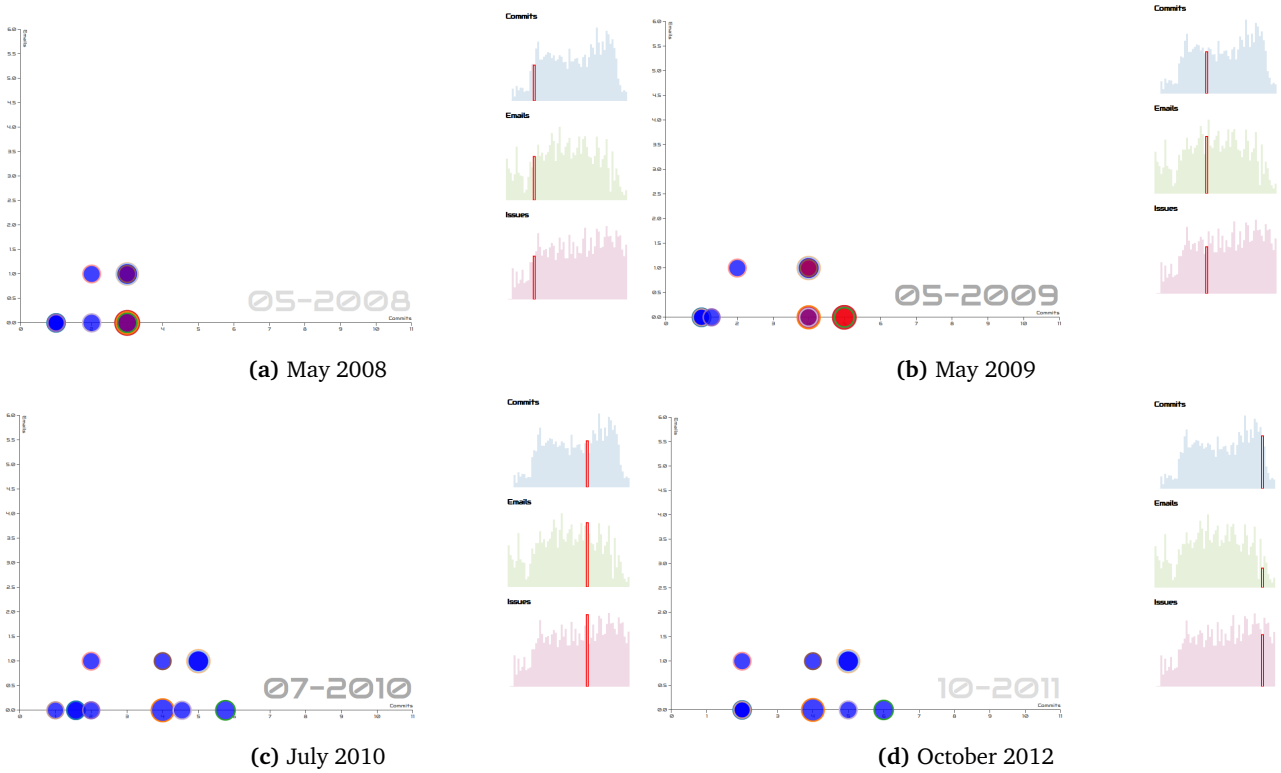
(c) February 2010 **(d)** February 2012

Figure 32. Mr.Bubbly: the "rss" package for "Apache Camel"

for the package "rss". We see that the development for this package start in an heterogeneous way, indeed there are files with a moderate amount of emails and commits, while others have some commits, but no emails, and the last have few commits and no emails. The second snapshot shows that the development continues, with increasing in commits, but nothing happens on emails, while some bugs were open on few files. The last two snapshots shows that commits are made mostly on the files containing bugs, and those bugs are slowly solved, which can be seen by the colors that goes from red to purple and then to blue.
This package is the umpteenth example of a package that "evolves horizontally", i.e. the number of commits increases while the number of emails remains constants.

rss (test):

This package is the test package relative to the "rss" package analysed before. Figure 33 present four snapshot of the visualization for the test package "rss". We can see that this test packages presents many similarities with the package been testes, i.e. the package "rss". For instance the files starts with heterogeneous amount of commits and emails, as in the previous package, and as before at May 2009 there is the presence of open bugs for both the package. Some of them can be related like the file *RssEntryPollingConsumer.java* in the package "rss" and the file *RssPollingConsumerTest.java* in the test package. Thus we can hypothesize that the presence of bugs in a package can affect the presence of bugs in the relative test package, especially if the files have high relations.



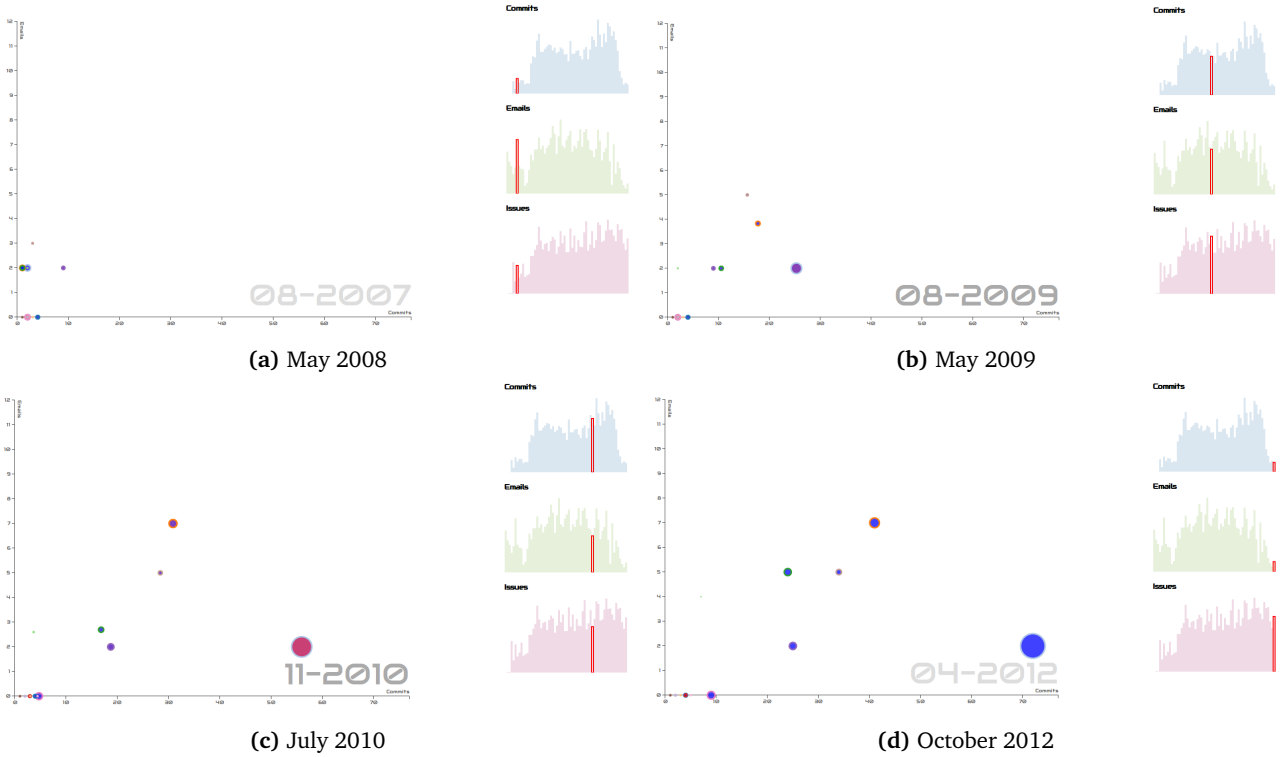
(a) May 2008 **(b)** May 2009

(c) July 2010 **(d)** October 2012

Figure 33. Mr.Bubbly: the "rss" test package for "Apache Camel"

jetty:

This package is medium sized package in terms of number of files inside the "Apache Camel" project. Figure 34



(a) May 2008 **(b)** May 2009

(c) July 2010 **(d)** October 2012

Figure 34. Mr.Bubbly: the "jetty" package for "Apache Camel"

present four snapshot of the visualization for the package "jetty". The package "jetty" has a different behaviour respect to the packages considered before. We can see that the files starts with small dimension, around 1000-5000 lines of codes. Some of them have already two or three emails discussing the class. In the following months of

development for many files increases both the number of commits and emails, but the dimension remains small. The third snapshot show also the presence of some bugs, especially in the file *JettyHttpComponent.java* which is the biggest file and has a lot of commits, but only two emails. The last snapshot show the increasing number of emails for some files and the correction of the bugs previously opened.

These eight packages show the overall behaviour of the "Apache Camel" software system. Except for the packages "mina" and "jetty" which presents increment on both emails and commits, the other packages shows that mostly the number of commits increases, while the number of emails remains constant. If we consider the four snapshot as four picture of the status of the packages which divides the time of development of that package in quarters, we see that the number of bugs have the tendency to increase in the second or third quarter. We believe that mostly, bugs start to appear after a significant amount of commits done on a file, and they are more probably to appear when there are few or no emails. While files that increases the number of commits and emails at the same time are less prone to bugs, like in the case of the package "jetty", or decrease the number of bugs, like in the case of the package "mina".

We also observe two different relations between a package and its test package, like in the case of "mina" and "rss". The former has many test files, much more respect to the package it tests, while the latter has nearly the same amount of file respect to the package it test, and present some correlations among files and test files. We can hypothesize that one package, "mina", is considered more important, for the purpose of the project, respect to the other therefore, to ensure its correctness, developers and testers writes more test cases and spread them among different files. While, in the other case, developers and testers tries to ensure correctness by testing if methods, present in classes been tested, work as expected.

4.3 The story combined

By using the two tools presented in this document, `Peaksight` and `Mr.Bubbly`, we are able to inspect a software system in two very different way. Hence the story of the evolution of a project is very different depending on which tool is used, as depicted by the two previous sections. Moreover each of the two tool has is peculiarity and focus the narration of the story on different aspect of the project, while not considering others. For instance, `Peaksight` construct an overall story of the project having no informations about specific files or packages, while `Mr.Bubbly` does not have any information about developers.
The purpose of this section is to construct a story of the evolution of "Apache Camel" by combine some aspect of the two stories made with the point of view of `Peaksight` and `Mr.Bubbly`.

The Apache and the Camel

Once upon a time, there was a community of developers known as the Apache Foundation, or the "Apache Group". Among the developer of this group there were few people, who were used to many transport messaging models, but they realized that using different transport model, would require them to use many different APIs, one for each transport. They dreamed to have a unique API that would be able to group all the transport, so that would be possible to to use same methods and architectures regardless the transport used. Away back in the March of 2007, this group of Apache developers started to work on a project which will be able to provide such an unique interface for each transport, which will be then called Camel. Among those pioneers of Camel, there is James Strachan, who is remembered by his strong contribution to define and build the fundamentals of the project.
For the first two year Camel were growing fast, the commits made to the project were growing and also the emails communications among contributors to the project were increasing. Developers were able to keep under control, the big threat of Bugs. (figure 21)
But it was just one year, after the project was started that, the biggest contributor, James Strachan, was decreasing is activity, until nearly stop almost any contribution to Camel. (figure 22) Those days, could have been marked the starting of a difficult period for Camel, where the development could have been decreased. Fortunately, exactly when James Strachan was decreasing his activity, a new Apache "warrior" joins the project to bring his contribution, he was Claus Ibsen.
With the contribution of this new developer, Camel has a new period of highly contribution, commits were increasing and emails were almost the same, while bugs were few. (figure 21)
The development of Camel, was proceeding, many files were committed, but few of them were discussed in mailing list. (figures 31, 32, 30, 29).
Camel, was getting bigger and bigger, many small packages were part of the project and many big packages were used to test if the software system was correctly working. (figure 25)

Then, it was August 2009, when email communication is subjected to a strong decreasing in the number of emails exchanged, which last until September 2010. Right after this period, the number of bugs opened highly increases. It's a terrible time for the developers of Apache Camel. The "horizontal evolution" of files where many commits are made, but really few emails are related to them increased the possibility to introduce bug. (figures 31, 32, 30, 29) Luckily from May 2011, developers managed to control the volume of open bugs, that in about an year returns to an acceptable number. (figure 21).

The moral of the story is that, continuously make changes by commits on file, with few discussion on them in emails is probably a good way to augment the chances to introduce issues in the system being developed.

5 Conclusions

With this bachelor project we created two tools: `Peaksight` and `Mr.Bubbly`, which performs two different visualizations. Both visualizations are created considering three artifacts: commits, emails and bugs. Using these two visualization is possible to create a story of the evolution of software systems.
With `Peaksight` is possible to see an overview of the system respect to the three artifacts, and is possible to obtain interesting information, to study the evolution of the system, by using the filtering functionality. For example is possible to inspect the contribution of single developer or exclude from the visualization artifacts which has some characteristics that are not useful for the analysis.
With `Mr.Bubbly` is possible to explore a software system by looking at package level. For each package present in a software system is possible to see how each file belonging to the package has evolved over time, by means of commits to it, emails relative to it, bugs opened on it and the size of the file in terms of lines of code. The animated visualizations help to see the evolution of the considered software system over time.

By combining the visualizations offered by `Peaksight` and `Mr.Bubbly` is possible to create significant stories of the evolution of software system, seen by different perspectives. Those stories can be used to understand how software systems have evolved and where could be possible problems in the development. For instance in the case study of "Apache Camel" (Section 4), by exploring the visualizations of `Mr.Bubbly` we noticed that many files present an increasing number of bugs when they are committed but not discussed, by email, for long time. On the other hand, with `Peaksight` we discovered that after a period of poor communication in terms of emails, there is a period where bugs increase rapidly. We also able to discover that Claus Ibsen is probably the main developer of "Apache Camel".

Future work:

"Apache Camel" is not the only software system it is possible to study with `Peaksight` and `Mr.Bubbly`. As future work, we would like to study other software systems and compare the stories it is possible to create. Hence, it would be possible to compare how different software systems evolves and it could be possible to find some relations and patterns among them. Particularly interesting would be to see if the relation of bugs and emails saw in "Apache Camel" is present also in other software systems.

`Peaksight` and `Mr.Bubbly` could be expanded to create visualizations centred on the developers of software system, for example in `Peaksight` could be created a visualization which compares the contributions of developers, while for `Mr.Bubbly` could be interesting to create a visualization where bubbles do not represent files, but developers.

A `Peaksight` Appendix

In this appendix we describe in more detail how `Peaksight` works internally. These details were not suited to be discussed in the previous sections which were intended to give just a brief overview of the system and its functionalities, but not to clarify all the implementation details. Anyway we believe that some readers could be interested in a deepen explanation of what makes `Peaksight` working, and this can be found in here.

A.1 Importing Data

In this section is discussed the implementation of the functions used to import data relatives to commits, emails and bugs in the database of `Peaksight`. We will begin considering how data are extracted from a Version Control System to have data for commits and how are saved into the database; then we will discuss about mailing list and at least about issues.

Version Control

It is possible to import data about Version Control Systems, from Git and Subversion. To import data about them, users have an apposite web form, shown in figure 35, where is possible to insert all the needed information about the version control. In particular the users have to specify the name of the project, the address of the repository which will be used to checkout, if the repository is a git repository or a Subversion repository, the address where is the database they want to use, and the name of the database they want to use.
The process of importing data about version control system can be very long, therefore users are informed to wait some time before try to use them. In the meantime, in the server an independent process to get the data has started. The process as first thing, checkouts the repository in a local folder, then when finished it retrieves all the informations about commits from the local copy. We used two python libraries to get the informations: *GitPython* for git and *pysvn* for subversion. The information retrieved are then stored in the database, in particular we store the following informations for each commits: the revision identifier, the date in which the commit was made, the author and the committer, the log message, and the list of changes. The latter is a list of each file changed in the commit with the number of deletion, insertions and lines changed.

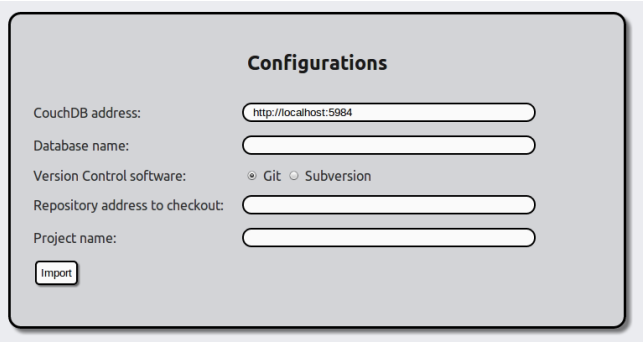


Figure 35. Peaksight: The interface to import data about Version Control System

Mailing List

To import data from mailing lists we decided to import emails form mbox files. Mbox files are simple text files containing list of emails separated by an empty line. Users can upload their mbox files with a form similar to the one for version control, as shown in figure 36. The only difference is that users have to specify the name of the mailing list and choose the mbox file to upload. The process of importing data of an entire mailing list can be very long, therefore users are informed to wait some time before try to use them. In the meantime, in the server an independent process to get the data has started. To parse them we used an apposite python library that deals with mbox files. The only problem with importing emails using this library is that many dates are not recognized and parsed correctly. This is due to the fact that, even if there is a specified standard for the format of the date emails should have, many emails client doesn't respect it. Therefore we had to write ourself some specific parser to recognize many different date format. We are then able to correctly parse date with an error of about 4%.

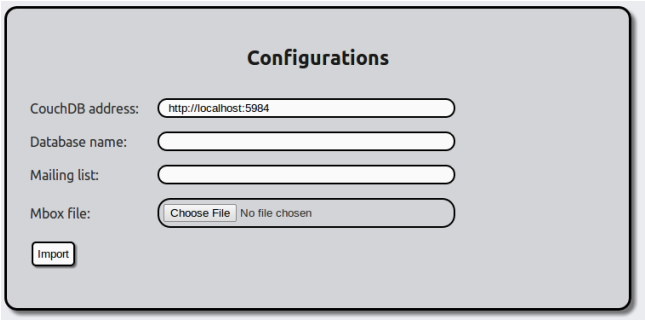


Figure 36. Peaksight: The interface to import data about mailing lists

In the database we store information about: the author, date when the message was sent, the body of the message and a list of headers.

Bug tracker

To import issue we chose to focus on the JIRA bug tracker. The interface shown to users to import data from JIRA, 37 is very similar to the two described before. The two different filed are: the field where it must be specified the address where the JIRA web interface is and the filed where must be specified the name of the project as identified by JIRA. Once the parameters are inserted we then need to crawl the specified JIRA address to search for issue relative to the specified project. The process of crawling JIRA for retrieving issues can be very long, therefore users are informed to wait some time before try to use them. In the meantime, in the server an independent process to get the data has started. In the database we store information about: who reported the issue and to who it is assigned, the date when the issue was reported, the date of the last update to the issue, the summary, which is a sentence describing the bug, the priority of the issue, and the resolution status.

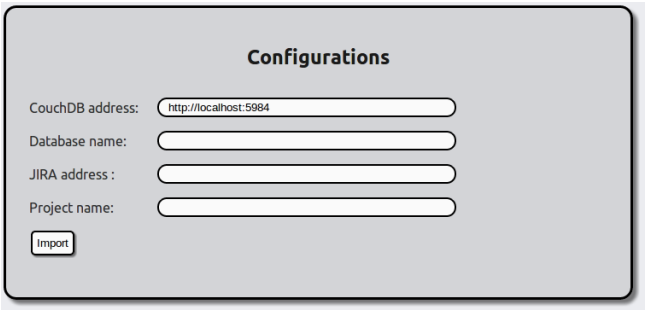


Figure 37. Peaksight: The interface to import data about issue

A.2 How is filtering performed?

In this section is explained in detail how the filtering function is implemented. Here in particular, we will explain the problem of performing filtering and we will present our solution.
With filtering, we intend that users have the possibility to choose which data have to be included in the visualization, by specify some common property the data should have.

The Problem

When users submits a list of filters to `Peaksight` we need to get from the database the data that respects the properties specified with the filters. Unfortunately, retrieving filtered data couldn't be done by directly query the database, due to the structure of CouchDB. Or, to be more precise, it could be done in theory, but in practice retrieve data with custom query created on the fly from parameters passed by the user would be too slow. This because CouchDB, performs queries with functions the call view, those are basically some static documents, stored along the databse, which contains the query and the result of it. Result of views will be updated any time a document is added to the database, modified or deleted. Since those views are static there is no possibility to create them based on arbitrary parameters given by an user, other that create a new view for each filter, which will turn in an overpopulation of the database which will then cause all access to database to be slow. CouchDB offers also the possibility to create the so called "temporary views". These are not static and does not persist in the database, but are not suitable to be used in production. In fact their purpose is only to experiment with view functions during development. Hence, if we base filtering on "temporary view" we will have as result very poor performance and a non usable tool.

The Solution

The only possible solution to the problem described above is to retrieve all the data from the database and the apply the filtering functions on them before send the results to the user. Of course doing this thing on a large amount of data can require a lot of time if not done in the correct way, hence exploiting parallelism. Figure 38 presents an

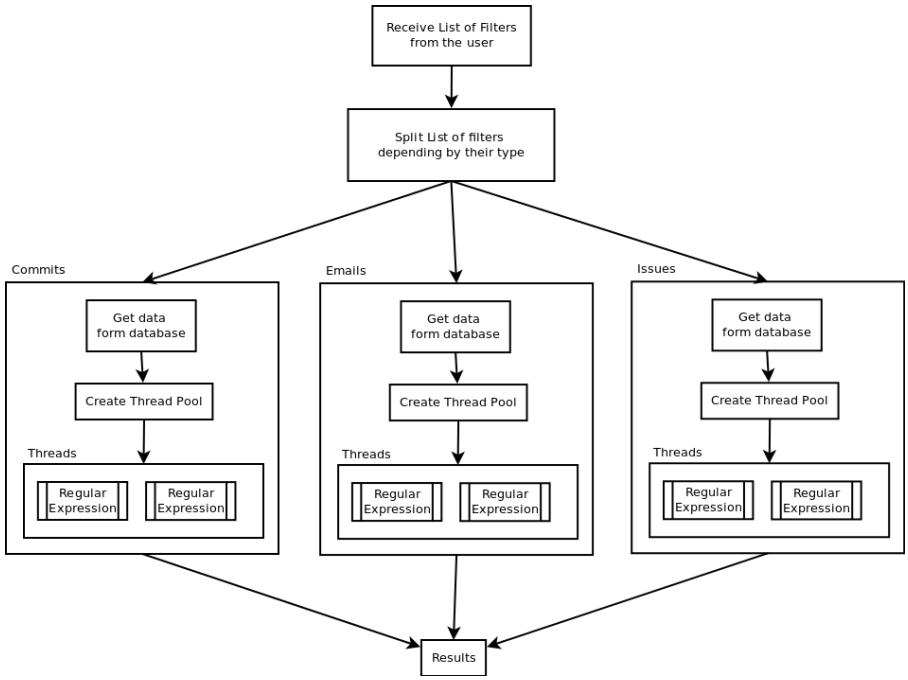


Figure 38. Peaksight: The filtering process

overview of the process we used to implement filtering. Here we explain it in detail.
The process start with the user which sends the list of filters he/she want s to apply to the visualization. This list of filters is then divided in three sub-lists depending on the artifacts on which their have to be applied: commits, emails or issues. These three sub-list are then passed as many threads which will then proceed to apply the filters. To do that they create a thread pool of maximum size of four threads for each. A thread pool is basically a group of thread which are created to perform a number of tasks, which are usually organized in a queue. Each thread starts executing a task, then as soon as a thread completes its task, it will request the next task from the queue until all tasks have been completed. In our case tasks are filters that have to be applied to the data collected from the database. Thus each thread executes the regular expression corresponded to each filter and save the result into a list. Once all the threads have finished, the results are collected and aggregated with a logical and.

B Mr.Bubbly Appendix

In this appendix we describe the format of the JSON files used in Mr.Bubbly to store the data necessary to create the visualization.
There are two kind of JSON file: the file containing information about packages structure, and the file containing information about files in a single package of the project.

JSON for package structure

Figure 39. Mr.Bubbly: JSON file for package structure

Figure 39 show an example of JSON file used to store the information about the structure of packages present in a project.
The packages structure is stored inside a tree where each node represent one package.
For example if we have the package *foo.bar.baz* the tree will have *foo* as root, *bar* as child node of the root, and *baz* as child of *bar* and leaf of the tree.
Every leaf in the three will contain also the information about the size of the package in terms of number of files which belong to it.

JSON for package information

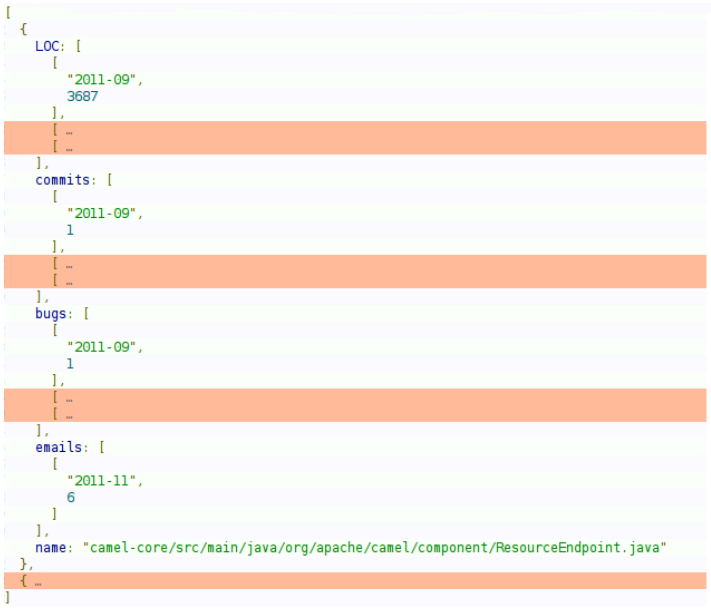


Figure 40. Mr.Bubbly: JSON file for package information

Figure 40 show an example of JSON file used to store the information relative to files in a package.
The information about files are stored in a list of dictionaries, where each element is identified by the name of the file. Then for each of these dictionaries there are four lists which contain information about lines of code in the file, changes to the file by commits, emails relative to the file, and bugs which affect the file. Each of these lists is actually a list of pairs: $(\text{date}, \text{number})$, where the date in the format "year-month" represent the date when some changes in commits, email, bugs or lines of codes occurred, while the number represent the quantity of them. For commits, emails and bugs this quantity is cumulative, while for lines of codes respects the real number of lines of a file at the specified date.

References

[1] A. Bacchelli, L. Baracchi, and M. Lanza. Remail-blending talk and work in eclipse. In *In Proceedings of Eclipse-IT 2011 (6th Workshop of the Italian Eclipse Community)*.

[2] A. Begel, K. Y. Phang, and T. Zimmermann. Codebook: Discovering and exploiting relationships in software repositories. In *32nd International Conference on Software Engineering, Cape Town, South Africa*, May 2010.

[3] C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan. Mining email social networks. In *Proceedings of the 2006 international workshop on Mining software repositories*, MSR '06, pages 137–143, New York, NY, USA, 2006. ACM.

[4] A. W. Bradley and G. C. Murphy. Supporting software history exploration. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 193–202, New York, NY, USA, 2011. ACM.

[5] M. D'Ambros, M. Lanza, and M. Lungu. The evolution radar: visualizing integrated logical coupling information. In *Proceedings of the 2006 international workshop on Mining software repositories*, MSR '06, pages 26–32, New York, NY, USA, 2006. ACM.

[6] K. Fogel. *How to Run a Successful Free Software Project*. O'Reilly Media, October 2005.

[7] B. Heller, E. Marschner, E. Rosenfeld, and J. Heer. Visualizing collaboration and influence in the open-source software community. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 223–226, New York, NY, USA, 2011. ACM.

[8] A. Kuhn and M. Stocker. Codetimeline: Storytelling with versioning data. In *34th International Conference on Software Engineering*, June 2012.

[9] M. Lungu, M. Lanza, T. Girba, and R. Robbes. The small project observatory: Visualizing software ecosystems. *In Journal of Science of Computer Programming (SCP)*, Vol. 75:pp. 264 – 275, 2010.

[10] K.-L. Ma, I. Liao, J. Frazier, H. Hauser, and H.-N. Kostis. Scientific storytelling using visualization. *Computer Graphics and Applications, IEEE*, 32(1):12 –19, jan.-feb. 2012.

[11] S. Neu, M. Lanza, L. Hattori, and M. D'Ambros. Telling stories about gnome with complicity. In *In Proceedings of VISSOFT 2011 (6th IEEE International Workshop on Visualizing Software For Understanding and Analysis)*, pages 14–21. IEEE CS Press, 2011.

[12] M. Ogawa and K.-L. Ma. Software evolution storylines. In *5th international symposium on Software visualization*, New York, NY, USA, 2010. ACM.

[13] H. Rosling. Wealth & health of nations `http://www.ted.com/talks/hans_rosling_shows_the_best_stats_you_ve_ever_seen.html`, 2006.

[14] E. Segel and J. Heer. Narrative visualization: Telling stories with data. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)*, 2010.

[15] P. Weissgerber, M. Pohl, and M. Burch. Visual data mining in software archives to detect how developers work together. In *Mining Software Repositories, 2007. ICSE Workshops MSR '07. Fourth International Workshop on*, page 9, may 2007.