Bachelor

June 14, 2012

Facets of Software Evolution

Aggregation and Visualization

Carol Alexandru

of Zurich, Switzerland (07-926-744)

supervised by

Prof. Dr. Harald C. Gall

Giacomo Ghezzi

University of Zurich UZH

s.e.a.l. software evolution & architecture lab

Bachelor

Facets of Software Evolution

Aggregation and Visualization

Carol Alexandru

University of Zurich^UZH

s.e.a.l. software evolution & architecture lab

Bachelor

Author: Carol Alexandru, carol.v.alexandru@gmail.com

Project period: 20.12.2011 - 20.06.2012

Software Evolution & Architecture Lab
Department of Informatics, University of Zurich

Abstract

SOFAS is a service oriented platform for analysing software projects, which can be reached over the internet. It consists of several different services, each of which is able to analyze a different aspect of the source code, such as its structure, size and complexity as well as the quality of its design. The services produce raw data stored in RDF graphs and it is up to the user to process the data, for example to produce Visualizations or to draw conclusions. The *Facets* application fills this gap by offering an easy to use web interface where people can submit the URL to their code repository, upon which *Facets* will start a complex workflow involving several SOFAS services to create a comprehensive analysis of the software project. Once the analysis is complete, the user can use a web browser to explore the results using a number of visualizations which offer an insight on several facets of software evolution: The large-scale shape of a project, the quality of its design, the metric properties of each and every entity of the source code and history-related information such as the changes in size and developer activity. While traditionally, developers are required to invest time and effort into the setup of analysys software and the preparation of analyses, *Facets* offers a simpler and more straight-forward approach for people to analyze their software projects with very little effort on their own part.

Zusammenfassung

SOFAS ist eine Service-Orientierte Plattform zur Analyse von Software-Projekten, welche über das Internet erreichbar ist. Sie besteht aus mehreren Diensten, wobei jeder Dienst einen anderen Aspekt des Quell-Codes analysieren kann, wie zum Beispiel dessen Struktur, Grösse, Komplexität oder die Qualität des Designs. Die generierten Daten werden in RDF-Graphen gespeichert und es ist dem Nutzer überlassen, die Daten zu verarbeiten, um zum Beispiel Visualisierungen zu erstellen oder Schlüsse zu ziehen. Die *Facets*-Applikation erfüllt diese Aufgabe, indem sie eine einfach zu bediendende Webapplikation zur Verfügung stellt, wo der Benutzer die URL zu einer Code-Repository eingeben kann, worauf *Facets* einen komplexen Arbeitsprozess startet, welcher eine Vielzahl an SOFAS-Diensten involviert, um eine umfangreiche Analyse des Software-Projekts zu erstellen. Sobald die Analyse abgeschlossen ist, kann der Benutzer den Web-Browser verwenden, um die Ergebnisse mittels einer Auswahl an Visualisierungen zu erkunden, welche einen Einblick in mehrere Facetten von Software-Evolution geben: Die grobe Form des Projekts, die Qualität des Designs, die metrischen Eigenschaften aller Bestandteile des Quellcodes sowie den Verlauf des Projekts, zum Beispiel im Bezug auf Grössenveränderungen oder Entwickleraktivität. Während traditionelle Lösungen dem Entwickler einiges an Zeit und Arbeit abverlangen, um Werkzeuge zur Analyse von Software zu installieren und Analysen durchzuführen, bietet *Facets* eine einfachere und direktere Möglichkeit für Entwickler, ihre Software-Projekte mit geringem Aufwand zu analysieren.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Overview

1.1 Problem Description

SOFAS[1] is a service oriented platform for analyzing software projects. It provides various services for extracting source code from repositories, building object models from the source code, measuring code metrics and running even complex analyses that can make a statement about the quality of the code. However, SOFAS only generates raw data and until now there has been no tool that took the data produced by SOFAS and turned it into visual representations that can be interpreted without much effort.

The goal of this bachelor thesis is to develop an easy-to-use web application that can be used directly via a web browser. It should enable the user to easily analyze their own code without any help from third parties such as the SOFAS developers or dedicated personnel. The user should simply be able to supply the URL of a code repository and after some time, be presented with the final results. It should cover not only one aspect of software evolution but rather involve several different *facets*, such as code complexity, ownership, version control history and quality.

1.2 Introducing «Facets»

The *Facets* application proposes a solution to this problem. It is a web service designed for end users and it can be visited using a modern web browser upon which the user is greeted with a simple submission form. The user enters the URL of a publicly accessible git or SVN repository and after a number of analyses have been performed, the user can browse and view a variety of visualizations. Visualized are a large number of metric properties such as the size, complexity, internal coupling and hierarchy of the code as well as version control properties of the project, such as how many lines of code were modified in any given commit or how many developers were active during a certain month. *Facets* also yields some information on problematic structures of the code: Code disharmonies or *smells* are structures in the code that may pose a problem to the smooth evolution and growth of the project [20] and *Facets* will point out all disharmonies for each release in the project. Finally, a PDF report can be generated and downloaded from the web application and a printer-friendly version of the interactive web application is available as well.

The *Facets* application server back-end sports a simple, yet extendable implementation that allows developers to write additional visualizations.

[1]An acronym for "SOFtware Analysis Services". SOFAS has also been developed at the software evolution and architecture lab at the University of Zurich.

1.3 Thesis Outline

Within the following chapters we first unravel at the reasons that warrant the implementation of such a tool, after which the solution will be presented in detail. Chapter 2 discusses the work this thesis builds on, explains some of the difficulties with existing solutions and finally formulates the requirements for the newly created tool. In chapter 3, the front-end of the web application will be discussed, describing general ideas and giving detailed descriptions of each visualiztion. We will discuss the background and purpose of each visualization. After the surface of the application has been explained, chapter 4 will look at the back-end. We will elaborate on the work flow that leads to the final product and reason for the essential design choices that have been taken. There will also be a brief explanation of how *Facets* can be extended and we will look at different methods of creating PDF reports from the analysis data. In chapter 5, we will evaluate the solution, establishing how it differs from existing ones and explain why it may be advantageous, before coming to a conclusion in chapter 6. The appendix contains a more detailed description of the architecture, installation instructions and a guide on how to develop additional visualizations. The development guide includes code samples and a sample visualization, both of which illustrate the process.

Chapter 2

Introduction

Software visualization is a well established field of research [19] and a valuable aid for software development and maintenance [15]. There exists however a significant problem in the area of software analysis: Most tools used for analyzing software are devised as independent programs written by various authors. Some are written as plugins for IDEs and many are stand-alone tools that need to be installed on the computer of the developer. It is rather difficult to make different tools inter-operate as parts of a bigger framework or towards a new super-type of analysis. [18]

In this chapter, we will look at a possible solution for this problem - called SOFAS - and explain its limits and how we can build on top of it. Then we will outline the concepts behind the *Facets* application and the requirements the application needs to fulfill.

2.1 Software Analysis as a Service

The creators of SOFAS set out to solve the aforementioned problem of high dispersiveness and lacking compatibility of software analysis tools: They created a distributed, collaborative platform on which software analysis tools can be accessed over the internet. All services are registered with an analysis broker and the data they produce is described by ontologies which are valid across any category of analyses. Each piece of information is part of an RDF graph - retrievable via a URL - and it is possible to query data using SPARQL[1] queries. SOFAS enables engineers to easily compose analysis work flows, where the output of one service may serve as the input for another service. Developers can even create and publish new analysis services to be used by others as part of their desired work flows. SOFAS services provide a RESTful API, which is kept simple and practical [18].

SOFAS however does not concern itself with the presentation of the data which is generated by the different services. While the services are already being used to analyze software projects and develop further analyses, there hasn't existed any project that makes the data presentable. *Facets* uses the SOFAS framework by creating a complex work flow, starting with just a code repository URL and ending up with an extensive visualization of several important aspects of a the given software project. Within this work flow, several SOFAS servers are inter-operating in order to create the required data. As such, this project also serves as a practical trial of the service oriented software analysis architecture.

[1]SPARQL is a query language for querying and modifying data stored in RDF graphs.

2.2 Facets of Software Evolution

Software evolution is a multifaceted topic: It's possible to focus on the historic aspects, describing how the code base changes over time. Tools like SvnStat[2] or oloh[3] already that do this - Code Swarm[4] and Gource[5] are more extravagant examples. It's also possible to look at how the quality and maintainability of the project rises and falls over the lifetime of the product. Squale[6] and Sonar[7] are example for such tools. Finally it's possible to gain insights on the structure of the code itself, looking at code metrics. There are many tools which do this, often as plugins for the more popular IDEs, such as Metrics[8] for Eclipse. The *Facets* application is trying to bring together the different facets of software evolution in one place, harnessing the functionality and flexibility of SOFAS. We shall take a quick look at the individual facets proposed by the initial thesis description:

- The *metrics* facet: Code metrics are often used in an attempt to directly derive the quality of the given project. However they can only ever be used as indicators for potential issues with the design and should be used with care. The upper and lower thresholds for a metric to be considered reasonable will certainly differ depending on several factors: It matters which programming language is used to write the code, what kind of entity was measured and what the semantic purpose of the code is. Metrics can however still be useful to gain an overview over the project and identify hot spots which need to be investigated more thoroughly [20].
- The *code smells* facet: When analyzing a software project, one of the ultimate answers being sought after is whether or not the code lives up to certain quality expectations. Code smells are an indicator for possible problems in the design and they can be derived from code metrics [20]. This facet is somewhat special in the context of this thesis because it is not very visual. Of course it is possible to plot the number of disharmonies for a class, but ultimately, the information conveyed is purely semantic. In chapter 3 it will become clear, how the code smells do not only appear in a single visualization but stretch across several visualizations. As the user is not only interested in the presence of code smells, but more importantly their causes, the reasons for each smell are also being presented to the user.
- The *project history* facet: Analyzing the history of a project can yield interesting results. For example, plotting the number of commits per developer per month can give an impression of developer activity and involvement in a project during a certain period. In larger projects, developers may leave or join a project during its lifetime. As such, the code ownership of any entity in the software project may change over time. Both of these aspects are important in order to know which developer has the required knowledge on a given entity if questions should arise [6, 13].
- The *projects future* facet: Developers and project managers alike are longing for methods that allow them to predict and avoid problems before they happen. One example is the *Yesterdays Weather* algorithm, which attempts to predict the likelihood of changes happening in any given file. The base for the algorithm is the empirical observation that entities which have changed the most in recent time are more likely to undergo important changes in the future [16].

[2]http://svnstat.sourceforge.net/
[3]http://www.ohloh.net/
[4]http://code.google.com/p/codeswarm/
[5]http://code.google.com/p/gource/
[6]http://www.squale.org/
[7]http://www.sonarsource.org/
[8]http://metrics.sourceforge.net/

2.3 Requirements

Software Analysis as a Service[9] was the initial thesis idea that sparked this project to develop a modern, interactive web application that visualizes the data gathered by SOFAS, giving the user the chance to get an overview over the shape, history and quality of a given software project. Under this pretext, the application should set itself apart among the software analysis tools designed for end users by a number of advantages:

- It should be profoundly easy to use. The user should not need to install the software or be required to do pretty much any work at all other than visiting a website and entering the URL to a repository. In other words: The tool should pose minimal entry barriers for users to participate.
- The tool should focus not on only one specific type of analysis or visualization but rather encompass a larger number of facets of software evolution.
- Given that SOFAS can grow in the future, additions to both the *Facets* data aggregation services as well as the visualizations should be possible. There should be an easy way for developers to write additional visualizations or configure the back-end services for different SOFAS services later-on.
- In addition to the interactive front-end it should be possible to generate a report that can be printed for use in meetings or which can be sent via Email.

[9] In this case, service does not refer to a software service but to the idea of outsourcing a workload.

Chapter 3

The «Facets»Web Application

3.1 Overview

In this chapter, we look at the features and usability features of the *Facets* web interface. There exists a multitude of visualizations that are being used to visualize different aspects of software evolution [17]. For this thesis, the majority of visualizations were already part of the thesis description. Nevertheless, we take a closer look at the individual visualizations, justifying their use and illustrating their purpose. As an extra feature, *Facets* allows users to download or print a PDF report on their project by the click of a button. Since the details on this feature are almost entirely of an implementational nature, they will be discussed in chapter 4.

3.1.1 Grouping of Visualiztions

Early on into the design of the *Facets* application, quite a few realizations took place, indicating that a split into the facets outlined in the thesis description is not particularily beneficial to the clarity of the presentation and that a slightly different grouping should be used. The actual contents of the aforementioned *semantic* facets is still present in the final application, but they have been distributed differently, because the facets are actually interconnected more strongly than anticipated. In particular, it is beneficial to display the code disharmonies along with the entities when viewing their metric information. This is equally valid for pointing out per-entity code ownerships. The project future facet has been dropped entirely very early in the project because it became clear that the library providing the predictions is not working reliably yet and re-implementing the algorithm would be outside of the scope of this project. All this leads to the following views inside the application:

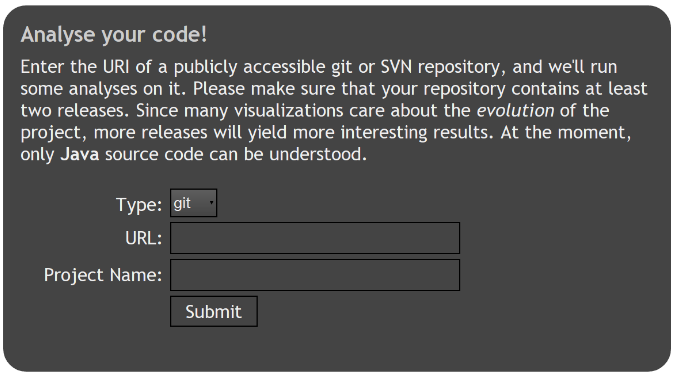
- *Overview:* This is the entry-point when visiting the analysis of any project. It includes the *Overview Pyramid*, which gives a concise view over the complexity, coupling and hierarchy of a system, the *Overview Pyramid Evolution*, which is a flat representation of all pyramids of all releases and the *Code Disharmony List*, which displays the code disharmonies of each type for all releases.

- *Entity Metrics:* This view contains two different browsers, namely the *Treemap* and the *Treebrowser*. Both browsers work on an entity level, meaning that the user can explore individual packages and classes, viewing their metrics and disharmonies. However, both browsers also display the code disharmonies of the entities. The Treebrowser also includes the display of the ownership history of each entity.

- *Project Evolution:* This view displays the commit history and the changes in lines of code for the entire project from the very first to the last commit.

3.1.2 Issuing New Analyses

When visiting the main web page of the *Facets* front-end, the visitor is presented with a simple submission form. The user enters the URL to a repository, selects if it's a git or SVN repository, and enters a name for the project. The analysis request is submitted by the click of a button. The back-end first does a quick check if the repository actually exists and if there are valid release tags present. This is done because the first SOFAS service being used - the git or SVN importer - does not check for this condition and if there are no release tags, later services will fail and the analysis will have been started for naught in the first place.


Analyse your code!

Enter the URI of a publicly accessible git or SVN repository, and we'll run some analyses on it. Please make sure that your repository contains at least two releases. Since many visualizations care about the *evolution* of the project, more releases will yield more interesting results. At the moment, only **Java** source code can be understood.

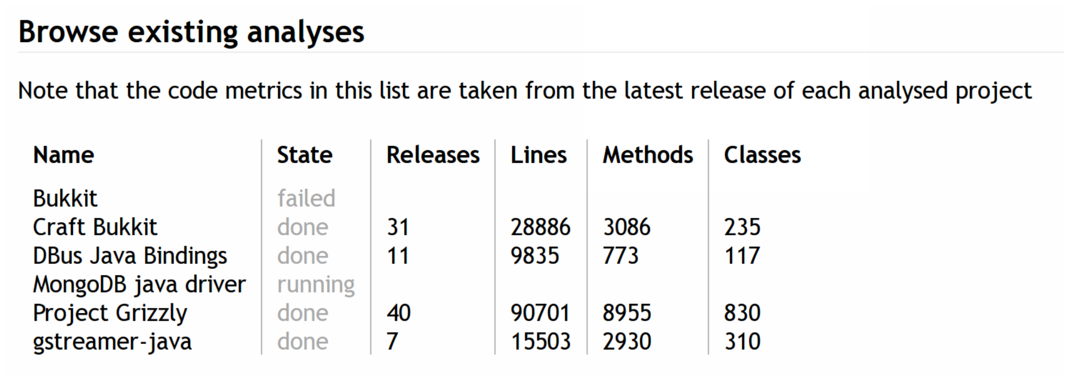
Type: git
URL:
Project Name:
Submit

Figure 3.1: The submission form that is placed on the main page. Its simplicity should entice visiting users to give *Facets* a try.

If the preliminary checks pass, the user is supplied with two URLs. Both lead to the analysis page, however one of the links contains an additional argument that contains a token. Using the link with the token will allow the visitor to delete the analysis. By this simple method, only the creator of an analysis can delete it, without the need for a user registration form or a user database. The token is generated using a simple but randomly salted hash.

3.1.3 Browsing Existing Analyses

All analyses on *Facets* are publicly viewable. The *Browse* button takes the user to a list of existing analyses, which includes the most important data points that allow the user to quickly glance over the list and find out what might be interesting. It also includes analyses that are still running or which have failed. Figure 3.2 displays a screenshot of such a list and illustrates that it is easy for a browsing visitor to get an impression of the existing analyses without visiting each one of them.



Browse existing analyses

Note that the code metrics in this list are taken from the latest release of each analysed project

Name	State	Releases	Lines	Methods	Classes
Bukkit	failed				
Craft Bukkit	done	31	28886	3086	235
DBus Java Bindings	done	11	9835	773	117
MongoDB java driver	running				
Project Grizzly	done	40	90701	8955	830
gstreamer-java	done	7	15503	2930	310

Figure 3.2: List of existing analyses in *Facets* with their most important cornerstone data points.

3.2 Cross-Visualization Features

Some of the features in *Facets* stretch across many visualizations, either because they are usability features that apply to most visualizations or because the information conveyed may be useful in more than one context. These features shall now be explained in more detail.

3.2.1 Basic Features

Duplication of Visualizations

The architecture of the web application makes it possible to easily duplicate each visualization. Each interactive visualization initially has a small [+] symbol on their right edge. Clicking it will open another instance of the visualization. Duplicates also have a small [-] symbol which can be used to destroy the additional instance. Each copy is fully functional. This feature enables the user to investigate different entities at the same time, for example in order to compare them. When opening the print view for an analysis, the duplicates and their state will remain present, which means that a user can set up several visualizations portraying a selection of the most relevant and interesting entities and then print those visualizations.

Contextual information

Most visualizations provide context sensitive information in the form of tooltips[1]. This is information usually eases the understanding for new users who are not familiar with the given visualization. On the other hand, tooltips can sometimes be used to reveal more details about an entity.

3.2.2 Indicating Code Disharmonies

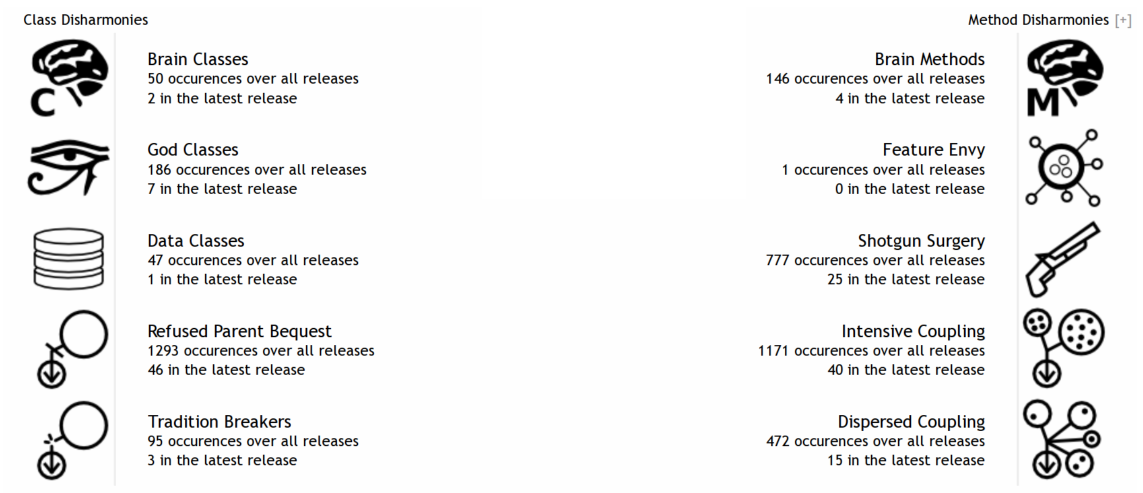
A software system has to evolve in order to maintain a level of usefulness and as a consequence, it is bound to become more complex [5]. The number of people working on it and their capabilities as well as the tools used during development and the constraints of the project affect the design of a system [8,9]. Because of these factors, it is very difficult if not impossible to have a clear picture

[1]A tooltip is a small box containing text that will appear near the mouse pointer when the user hovers over an element.

of the entire design. Lanza et al. propose a method of assessing the soundness of the design of object-oriented software systems using code metrics.

They define three types of *harmonies* which certain entities in an object-oriented system should exhibit. Each harmony can be disturbed by a combination of different factors that are measurable via code metrics. Such a harmonic disturbance leads to a code smell which may cause trouble if it is not corrected. The harmonies can be briefly outlined as follows [20]:

- *Identity harmony* requires that an entity can justify its existence by implementing a discernible concept. Identity harmony is disturbed if an entity is unreasonably small or big or if it's bearing several unrelated responsibilities. As much as possible, classes should only use and modify their own data and distribute their complexity among their operations.
- *Collaboration harmony* ensures that entities in the system interact appropriately with each other, avoiding any extremes such as classes being used by an excessive number of other classes or classes using other classes in a majority of operations, both of which increase the coupling in a system.
- *Classification harmony* concerns itself with the placement of classes and functionality within the inheritance tree. Hierarchies shouldn't be too wide or too tall and sub-classes should always maintain a balance between inherited and new functionality. The level of abstraction should be high at the top of the hierarchy and low at the bottom.


Class Disharmonies

Brain Classes
50 occurences over all releases
2 in the latest release

God Classes
186 occurences over all releases
7 in the latest release

Data Classes
47 occurences over all releases
1 in the latest release

Refused Parent Bequest
1293 occurences over all releases
46 in the latest release

Tradition Breakers
95 occurences over all releases
3 in the latest release

Method Disharmonies [+]

Brain Methods
146 occurences over all releases
4 in the latest release

Feature Envy
1 occurences over all releases
0 in the latest release

Shotgun Surgery
777 occurences over all releases
25 in the latest release

Intensive Coupling
1171 occurences over all releases
40 in the latest release

Dispersed Coupling
472 occurences over all releases
15 in the latest release

Figure 3.3: Overview of disharmonies in the Craft Bukkit project. Clicking on an icon reveals the individual disharmonies.

SOFAS has implemented the detection strategies devised by Lanza et al. and can yield all ten different types of disharmonies they describe. In *Facets*, each disharmony has a descriptive and recognizable icon associated with it. There are different ways of exploring the disharmonies of a software project: A dedicated Code Disharmonies plugin gives an overview over all the disharmonies. The user can click on a disharmony icon at which point a list of all disharmonies of the selected type appears, together with their locations and reasons. In addition to this, the Treemap and Treebrowser plugins contain the same icons and upon hovering on an entity, any applicable icons are highlighted if the entity suffers from disharmonies. Hovering over an icon will again reveal the reasons for the disharmony. In the Treemap and Treebrowser plugins it is also possible to highlight all entities that contain disharmonies. These tools facilitate the process of finding weak points in a software project, because the user has several methods of exploring them at his disposal. He can either look for specific disharmonies by using the disharmony list or he can take a different approach, browsing entities freely by looking at their size or other metric properties while still being informed about disharmonies that appear in said entities.

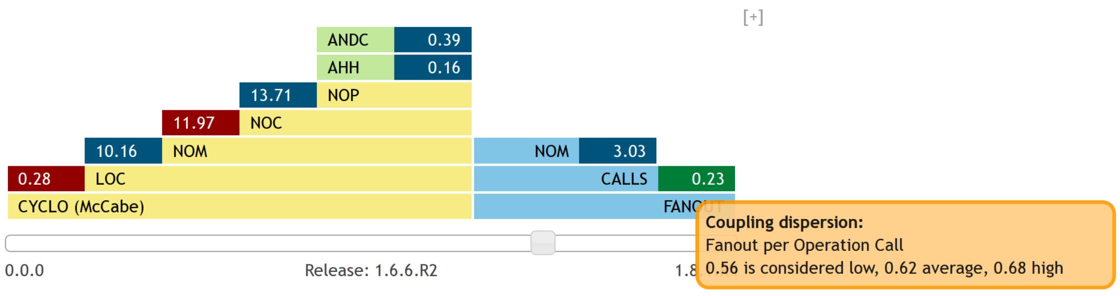
3.3 Individual Visualizations

Having explained the basic features of the web application and how some of the visualizations are accompanied by code disharmonies, we shall now look at each of the remaining visualizations.

3.3.1 Overview Pyramid

The Overview Pyramid - also developed by Lanza et al. - is a method of objectively presenting some of the most relevant properties of a project and is used to give an overall characterization of its design [20]. The pyramid consists of four basic parts:

- *Inheritance properties* - the average number of descending classes and the average hierarchy height - are displayed in the green area at the top.
- *Size and complexity metrics* - the cyclomatic complexity as well as the numbers of lines of code, methods, classes and packages - are listed in the yellow area on the left side.
- *Coupling information* - the number of operation calls and the number of called classes - is listed in the blue part on the right side.



Coupling dispersion:
Fanout per Operation Call
0.56 is considered low, 0.62 average, 0.68 high

Figure 3.4: An Overview Pyramid for one release of the Craft Bukkit project as generated by *Facets*. In this screenshot, the user is hovering over the FANOUT/CALLS ratio and is hence receiving additional context information in a tooltip.

Lanza et al. developed the Overview Pyramid after realizing that the characterization of a project by single metrics such as the number of lines of code or the number of classes is often insufficient and misleading. How the values relate to each other can actually give a much better impression. For example a project with 500 methods and 5'000 lines has quite a different shape compared to a project with 500 methods and 10'000 lines of code. Lanza et al. also note that it's important to have reference points when looking at metric data. A project written in C++ will probably have more lines of code than an equivalent Java project. They analyzed 45 Java and 37 C++ open and closed source projects of varying size and from various application domains, determining thresholds for unusually low and high ratios as well as an average for each ratio. For example the number of lines of code per method in a Java project has a lower threshold of 7 and an upper threshold of 13, with 10 being the average [20].

Figure 3.4 tells us for example, that the project in question seems to have an average shape concerning the number of lines of code per method, the number of classes per package, the hierarchical figures and the number of function calls per method. However, three of the ratios are exceptional: The project is rather complex because the CYCLO/LOC ratio is 0.28 (Upper threshold: 0.24) and each class contains an above-normal number of methods because the NOM/NOC ratio is 11.97 (Upper threshold: 10). These two values are exceedingly high. On the other hand the project exhibits a positive exceptional value as well: It is loosely coupled, because the FANOUT/CALLS ratio is 0.23 (Lower threshold: 0.56).

The Overview Pyramid not only gives a clear picture of the complexity, coupling and hierarchy of a project but it also tries to allow for an objective assesment of the measurements. The ratios are color coded to further ease the visual interpretation. In *Facets*, one Overview Pyramid is generated for each release and the user can use a slider to browse releases. Hovering over a ratio gives the user an additional explanation of what it means. As mentioned before and as with all visualizations, the user can duplicate the pyramid in order to be able to compare different releases.

3.3.2 Overview Pyramid Evolution

The Pyramid Evolution graph is a previously unproposed method of indicating the change of the Pyramid over the lifetime of the project. It takes each kind of ratio of every pyramid in the history of the project - one for each release - and plots them side by side. The vertical scale is multilinear and individual for each plotted line, but all the scales have common points where the thresholds are and at the top and bottom. This means that the purpose of this graph is on one hand to make the different ratios comparable and on the other hand to illustrate the change of each individual ratio over time. If a line crosses over from the blue to the red part, it means that it is now exceptionally high. Likewise crossing over from the blue to the green part means that the value is now exceptionally low. The red line in the given example in Figure 3.5 illustrates that the number of methods per class was exceptionally high in the beginning which was remedied early on, making it hover just over the threshold for the rest of the project lifetime. On the other hand, the coupling intensity in the project - indicated by the purple line - seems to have grown severely since the beginning. It went from the brink of being exceptionally low to clearly being exceptionally high.

It is easy to get an impression on how the project evolved, specifically if the metric ratios have gotten better or worse over time, which is why I think that this flat representation provides a valuable addition to the concept of the Overview Pyramid.

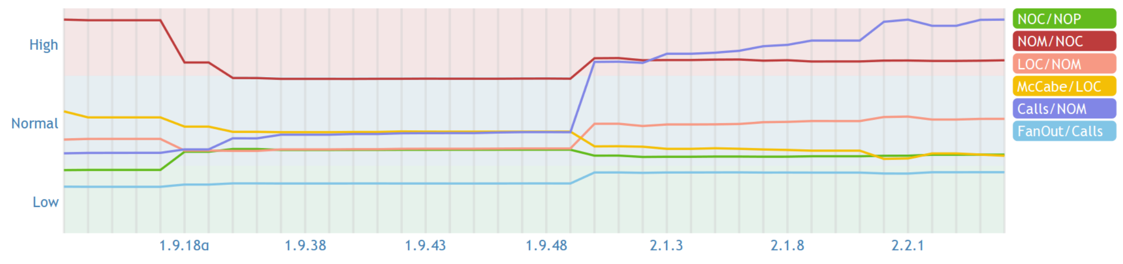
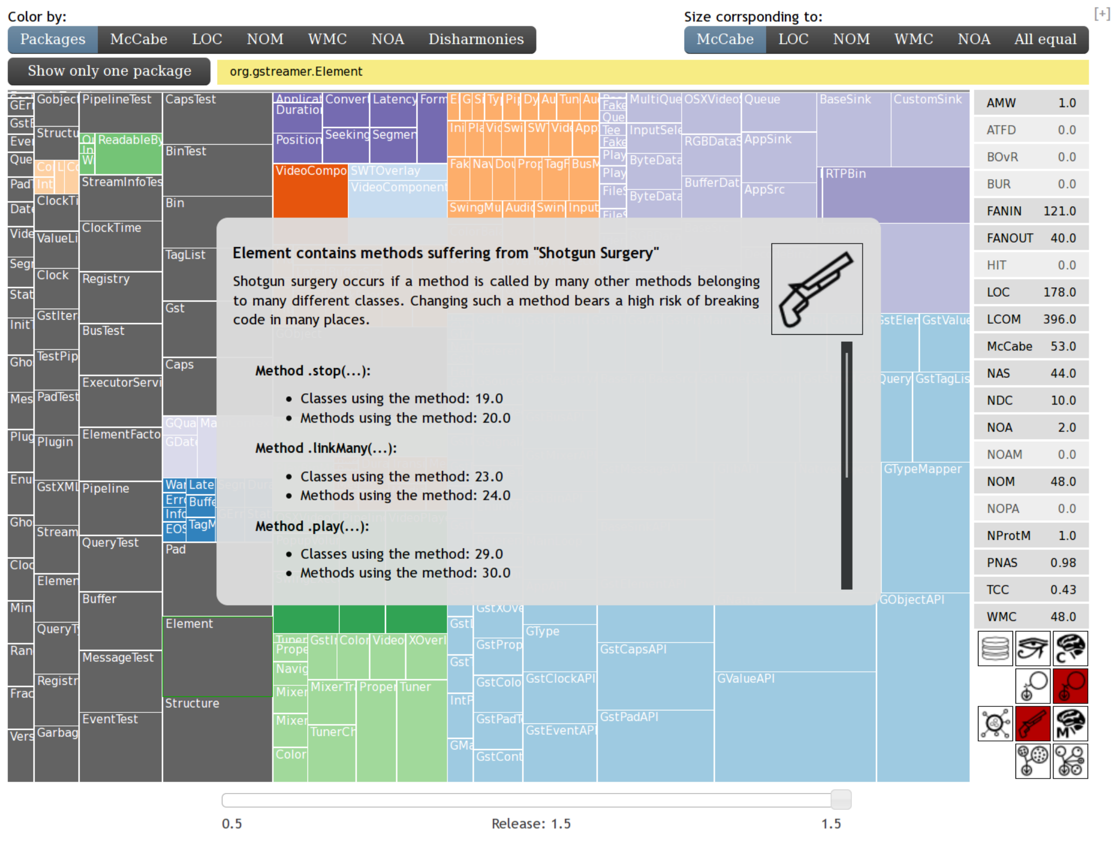


Figure 3.5: The evolution of the Overview Pyramid for Project Grizzly.

3.3.3 Treemap

The Treemap is a dense visualization of all the leaf nodes in a tree, with the size of each cell corresponding to some metric or another. The color of a cell can be used as a second dimension. Treemaps have long been popular for visualizing not only software-related figures but statistical tree-structured data in general [3].



Element contains methods suffering from "Shotgun Surgery"

Shotgun surgery occurs if a method is called by many other methods belonging to many different classes. Changing such a method bears a high risk of breaking code in many places.

Method .stop(...):

- Classes using the method: 19.0
- Methods using the method: 20.0

Method .linkMany(...):

- Classes using the method: 23.0
- Methods using the method: 24.0

Method .play(...):

- Classes using the method: 29.0
- Methods using the method: 30.0

AMW	1.0
ATFD	0.0
BOvR	0.0
BUR	0.0
FANIN	121.0
FANOUT	40.0
HIT	0.0
LOC	178.0
LCOM	396.0
McCabe	53.0
NAS	44.0
NDC	10.0
NOA	2.0
NOAM	0.0
NOM	48.0
NOPA	0.0
NProtM	1.0
PNAS	0.98
TCC	0.43
WMC	48.0

Figure 3.6: A Treemap for gstreamer-java as drawn by *Facets*. The metrics of the current entity are displayed on the right side. In this screenshot, the class *Element* has been selected. It suffers from two disharmonies. Upon hovering over the shotgun surgery icon in the lower right corner, an overlay points out the reasons for this particular disharmony.

In our case, the leaf nodes are classes in a project. The user has the choice of changing the base metric used for the cell dimensions and the cell color. Both the color and size can be chosen to correlate to the lines of code, cyclomatic complexity, number of methods, number of attributes and weighted method count. These metrics were chosen because they are usually available for the majority of entities in a project[2]. The color has two additional choices: Whether or not there are disharmonies in a class and a colorization by package or sub-package that a class belongs to. It is also possible to zoom into a single package by selecting an entity in that package and clicking on *Show only one package*. This is especially useful when viewing very large projects where the

[2]SOFAS computes different metrics depending on the type of entity. Abstract classes have a different set of calculated metrics compared to for example packages or regular classes.

Treemap becomes increasingly cluttered. The controls for size and color continue to work as expected when viewing a single package.

Hovering over a cell updates the statistics displayed on the right side of the Treemap with unavailable metrics being grayed out. If the class contains disharmonies, the corresponding disharmony icon will be shaded in red. This purely hover-based navigation allows the user to quickly glance over a number of classes, without even clicking once. Clicking on a cell focuses the class and causes the statistics and disharmony indicators to become sticky. Hovering over other cells will no longer have any effect. Now, the user can hover over a disharmony icon which causes an overlay to appear with more details on the disharmony, such as a description and the disharmony and a list of reasons which cause the disharmony. Again, minimal clicking is required, as the overlay will simply stay visible as long as the user is hovering over the disharmony icon or the overlay itself. If the cursor leaves any of these these areas, the overlay will fade out. The idea here is that the user should be able to browse the Treemap very quickly and the classical method of providing pop-ups that need to be closed manually has been avoided.

3.3.4 Treebrowser

The Treebrowser serves as a browser for two different visualizations which are drawn per-entity: metric *kiviat diagrams* and *code ownership Treemaps*. In contrast to the Treemap, which can only represent the leaf nodes of a tree, the Treebrowser also contains packages and sub-packages as selectable entities. Another difference is that the tree contains all entities that ever existed in the project. In other words, the trees of all releases have been merged into a single tree. In turn, for both kiviat diagrams and code ownership treemaps, individual sliders are provided to move through their history.

The browser itself has a number of controls that resemble the ones from the Treemap. The nodes in the tree can be colored by the same criteria as the cells of the Treemap. Additionally, the tree can be searched *as-you-type* and there are a couple of additional controls: One is used to switch between kiviat diagrams and the ownership visualization. The others are used to adjust the selected visualization itself.

Kiviat Diagrams

Kiviat diagrams[3] provide a way of visualizing changes over time in a collection of n-ary tuples. In our case, the tuples contain metrics of a given entity, there being one tuple for each release the entity existed in. For the purpose of making it easier to distinguish between releases, each release has a different color. Below the diagram there is a slider that allows the user to highlight a specific release in the diagram and to view its metrics. The user can choose between three different drawing styles for the kiviat diagram:

- *Single*: Only the selected release is displayed while all other releases are hidden from the diagram.
- *All*: Every release is drawn, the oldest at the bottom and the newest at the top. All releases except the selected release are semi-transparent, in order to allow for overlapping sections to be visible. The slider selection highlights a release by making it fully opaque.
- *Stacking* is similar to *All*, but when using the slider, all releases newer than the selected one are hidden. As such, the selected release is always on top and selecting the oldest release means hiding all other releases. Sliding from the oldest to the newest release will stack each new release on top. This makes is easy to observe how an entity changed over time.

[3]also known as spider, polar or radar graphs

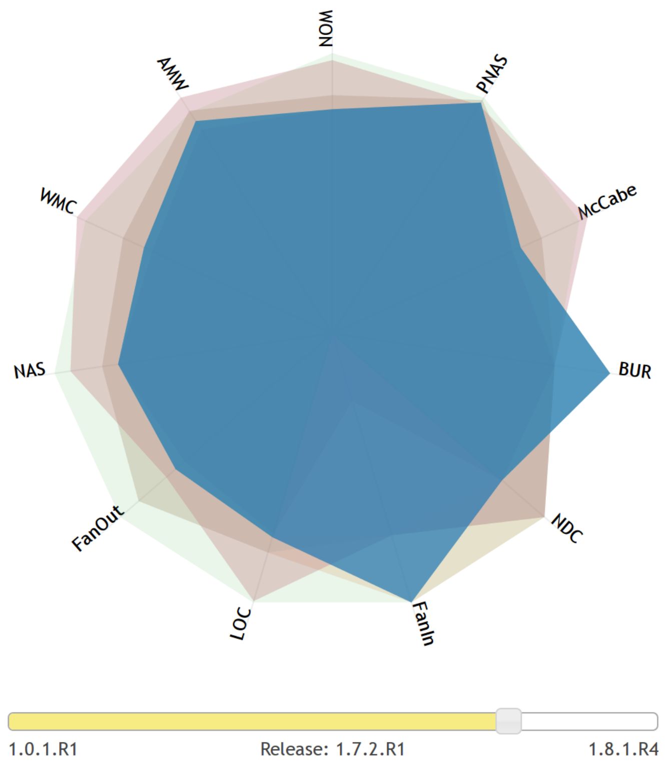


Figure 3.7: A kiviat diagram in *stacking* mode for the CraftLivingEntity class of the CraftBukkit project. Releases newer than the selected release 1.7.2.R1 are hidden and all older releases are visible below it. This is illustrated by the yellow fill of the slider. Apparently, this class has shrunk in many regards such as the number of methods, complexity and lines of code but on the other hand the base class usage ratio and the Fanin have increased. This could indicate that the developers may have moved functionality to the base class.

Using the data supplied by SOFAS for kiviat diagrams poses two minor problems: First, the values tend to be zero for a significant number of metrics. This is expected but not very interesting. Hence to keep the content of the diagrams informative, any metric that contains zero in more than two thirds of the releases is ignored and not drawn on the diagram. Secondly, SOFAS naturally does not supply the same kind of metrics for different types of entities, such as packages, classes or methods. *Facets* makes an effort to draw an entity, but if there are only two metrics left after removing metrics with too many zeroes or if the entity only ever had two metrics to begin with, the diagram will not be drawn and a message will be displayed, saying that there is not enough information available on the entity. A possible alternative would be to plot the values on a regular line chart instead.

There are however some additional problems with kiviat diagrams in general as they can be rather difficult to work with in a number of cases: If in n releases the data values are not always rising but also falling, the circular lines will start to overlap and this can cause significant confusion. Also, at least 3 metrics are needed to draw a meaningful diagram and since many entities don't have the same kind of metrics because of their nature (interfaces, abstract classes etc.), each kiviat diagram looks different. In order to curb these negative implications, *Facets* offers different ways of drawing the kiviat - stacking, all and single. Still, it's debatable if the perceived 'fanciness' of the kiviat diagram really merits the increase in complexity. Simple line graphs may not be as exciting to look at, but they do a very good job at conveying the information and they are much easier to implement.

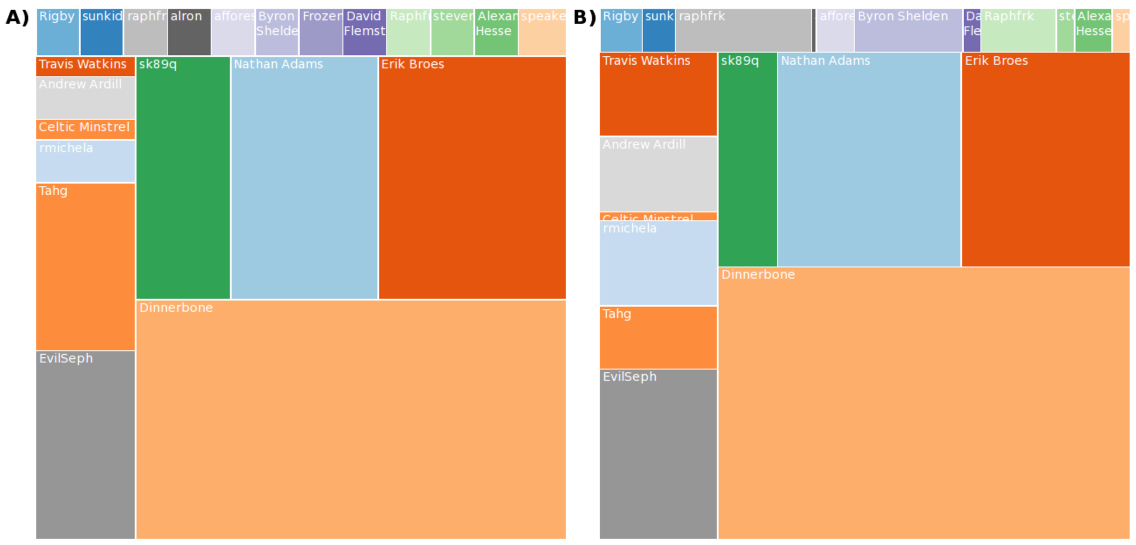


Figure 3.8: Two ownership Treemaps for the same commit. In A, the size of cells is determined by the number of commits and in B by the number of lines of code added. It becomes clear that especially for smaller contributions, the different base metrics reveal very different ownership values. For example, *Travis Watkins* (red) added more lines of code than *Tahg* (orange), but has far fewer commits.

Ownership Treemaps

D'Ambros et al. illustrate how code ownership can be visualized using fractal figures [13]. However, fractal figures suffer from aspect-ratio problems and can produce very tall or very wide rectangles, making the visualization look very imbalanced [3]. The Treemap is nothing but a fractal figure with added squarification in order to avoid these problems, which is why it has been chosen for this visualization instead of the fractal figures used by D'Ambros et al. The size of each cell in the Treemap represents the level of ownership. Committers with larger cells have stronger ownership than those with smaller cells.

Determining the code ownership of an entity is not straight-forward. SOFAS provides for each entity and each commit the number of lines added and deleted by the committer. But when deleting lines, the user may be deleting his own code or the lines of other people and SOFAS doesn't keep track of this information. This means that it's impossible to unambiguously determine how many lines of the code any specific person actually owns. As such, trying to come up with a representative indicator for class ownership percentages from the data provided is challenging. In research, it seems like the number of commits is the most widely used metric for ownership [6,13] although in some cases, more recent commits are weighted more than older commits, the reason for this being that a developer who has recently worked on an entity is more likely to remember details about it than a developer who might have created the entity a while ago [21].

To illustrate that it should not be taken for granted that this single measure is correct, a simple thought experiment can be made. Given a software project with three developers and a number

Table 3.1: Ownership percentage O and lines of code owned N for committers 1, 2 and 3 depending on different deletion and measuring scenarios.

	O1	N1	O2	N2	O3	N3	Total lines
Statistics before commit - gathered by keeping track of individual lines:							
Actual lines owned	10%	120	80%	960	10%	120	1200
Total lines added	10%	240	80%	1920	10%	240	2400
Total lines deleted	10%	120	80%	960	10%	120	1200
Commits	30%	3	50%	5	20%	2	10
Q1: Actual lines owned if P3 adds 100 lines, deleting 300 lines in four different ways:							
100 each from P1, P2 and himself	2%	20	86%	860	12%	120	1000
120 from P1, the rest from P2	0%	0	78%	780	22%	220	1000
300 from P2	12%	120	66%	660	22%	220	1000
120 from himself, 180 from P2	12%	120	78%	780	10%	100	1000
Q2: Calculating the ownership without knowing which lines got deleted:							
By lines added	10%	240	77%	1920	13%	340	2500
By lines deleted	8%	120	64%	960	28%	420	1500
By lines added and deleted	9%	360	72%	2880	19%	760	4000
By number of commits	27%	3	46%	5	27%	3	11

of commits, we can look at what happens when a new commit is made by one of the developers. Table 3.1 illustrates the thought process. In this example, we make the following preliminary assumptions: Three people have been working on the project until now and conveniently, each person has added twice as many lines to the project as they had deleted, giving us simple values to work with. Now the following questions can be raised, when a developer commits, indicated by Q1 and Q2 in the table:

1. Which lines does a committer delete? Do they only delete the code of other people, only their own, or a combination of these? If they delete foreign code, which other developers are affected? How much does the *actual* code ownership by lines of code change?
2. How do we calculate the ownership? We can only keep track of how many lines a person added or deleted and of the number of commits the person has performed. What results do we get when using different calculation methods?

When taking the number of lines of code owned as an objective measure of ownership, none of the estimates are accurate as they may significantly differ from the actual figures. For example, looking at the first deletion scenario - if P3 did in fact delete most of P1's code - only 20 lines of his code would be left. Yet calculating the ownership by the number of commits, P1 would still own 27% of the entire code.

Because of the large amount of ambiguity revolving around the exact measurement of code ownership, we have to remind ourselves of the purpose of calculating it in the first place: Usually, it's to gain knowledge on which developer may have the most expertise on an entity so that they can be contacted if problems arise with a module [21]. I ultimately take no stance and instead offer a switch to choose any of the four methods outlined above. The ownership of a module can be displayed by the number of commits, but also by the number of lines added, deleted or both. This leaves the user with deciding how to interpret the available data depending on their needs. As a rough estimate, any of the measures do a fair job, but for an accurate estimation of code ownership, further research is required.

There is one more problem that cannot be solved easily: Some committers will change their username while working on the project. In one use case, *Project Grizzly*, a person would commit as *rlubke* for several months but then change his username to *Ryan Lubke*. For a human it is clear that this must be the same person but *Facets* works with the data it gets from SOFAS and this is why these users will not be recognized as the same person but as different users.

3.3.5 Project Evolution Graphs

Two visualizations in *Facets* illustrate the history of the project ever since the very first commit. The information presented in this facet is all extracted from SOFAS' repository services, meaning that unlike in other views, the data is not presented per-release but continuously[4]. The project evolution facet gives an impression on how the project has evolved since its inception.

Lines of Code

The commits are plotted on the X axis while the Y axis represents the number of lines of code. As such, the X axis does not represent linear time. However, time stamps are provided at regular intervals to ease orientation. Hovering over the plot will show the commit information at any point on the graph, such as the date, the name of the committer, how many lines of code the project has at this point, and if the commit was followed by a release, the name of the release will be shown as well. As the plot is 910 pixels wide, a choice had to been made what to do if a project has more than 910 commits. The obvious choice is to add a scroll bar[5], however in this case a more simplistic approach has been chosen: If there are more than 910 commits, every few commits a commit will not be available for hovering such that over the entire graph, the skipped commits will be evenly distributed. This makes it possible to represent the entire history of the project within 910 pixels without sacrificing too much accuracy. Because the data used for this plot comes directly from the SOFAS version control services, the number of lines includes comments and files other than source code, such as documentation or Makefiles.

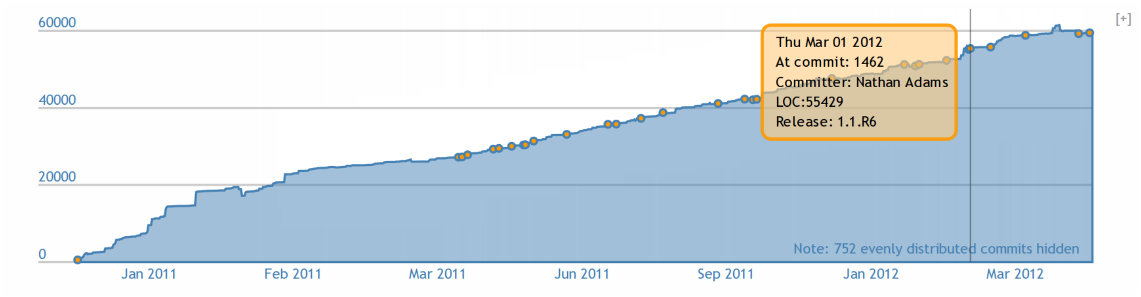
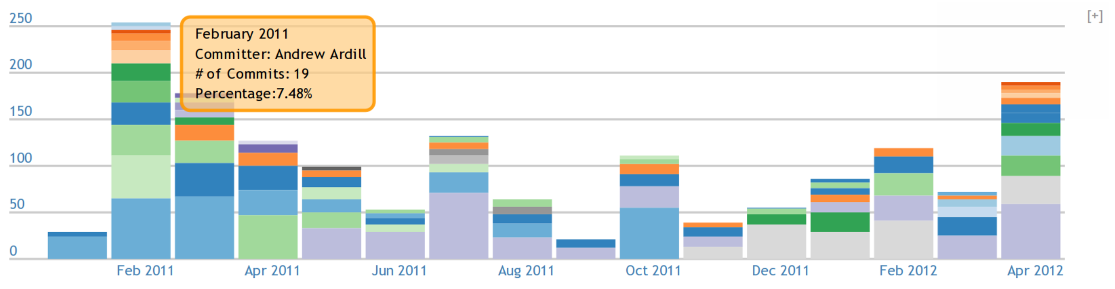


Figure 3.9: The number of lines of code from the first to the last commit. The overlay gives information on the commit currently under the cursor. It happens to be a commit that was followed by release 1.1.R6. Note that 752 equally distributed commits have been hidden to fit all commits within the 910 pixel width of the drawing.

[4]This is also true for the code ownership Treemaps in the Treebrowser facet.
[5]Ohloh has implemented this approach, for example: `http://www.ohloh.net/p/firefox/analyses/latest`

Commits per Month

Another popular piece of information when looking at the evolution of a project is how people contributed. This graph plots for each month the number of commits and divides the columns by committer. If there are many committers with very few commits - too small to draw them - they will be grouped as "Others". Hovering over the column sections reveals information on the respective committer as portrayed in the screenshot in figure 3.10. Again, the decision was made to avoid scroll bars and as such, months will shrink in width to fit on the plot. Still, more than a decade of history fits on this graph.



February 2011
Committer: Andrew Ardill
# of Commits: 19
Percentage:7.48%

Figure 3.10: The number of commits for each month since the beginning, broken up by committer. The color of each committer stays the same in each month, which allows keeping track of specific people. In this screenshot, the user is hovering over the green area of the tallest column (February 2011), which belongs to Andrew Ardill. It becomes apparent that he hasn't been contributing much after March but that he came back in December.

Chapter 4

Design & Implementation

4.1 Overview

Facets[1] consists of a number of discernible parts: A service application written in Python communicates with SOFAS, issuing analyses and querying results to be stored in a local database. Then there's the substantial amount of Javascript code necessary for visualizing the data. In-between, a very simple CherryPy web server instance acts as an interface for users to issue new analyses and view existing ones. This is an overview over the libraries and tools used:

- CherryPy is a small web framework library which supplies the bare minimum of necessary tools to create web applications written in Python. It provides a plugin system which is instantiated by the *Facets* job manager and it handles requests by the web clients.
- Jinja2 is a templating engine for Python. It is used to customize HTML and Javascript code served to the client where necessary and it is also used to generate HTML and LaTeX templates rendered to PDF when requesting a report.
- PhantomJS is an off-screen renderer that uses the WebKit engine and makes it possible to render websites or HTML documents programatically on a server. It is used to render the visualizations for the PDF reports.
- lxml is an XML parsing library for Python. When querying SOFAS via SPARQL, one of the possible response formats is XML and lxml is used to parse these responses.
- JQuery is used on the client side for basic presentational tasks and *Ajax*-style interaction with the web server.
- d3 is Javascript framework for creating visualizations using Javascript, HTML and SVG. It's a relatively young project and successor to the Prototype library.
- MongoDB, a high-performance *NoSQL* document store is used for storing the data extracted from SOFAS. Bindings for Python are available through the pymongo library.

4.2 Architecture

The core component of *Facets* is a small CherryPy application. Upon start-up, it instantiates the *JobManager* - which is written as a CherryPy plugin - and creates an instance of *pymongo.Connection*,

[1]The technical documentation for *Facets* covers some of the components in greater detail. It can be found in the appendix

which provides all the necessary tools to communicate with the database. After starting up, the application listens to client requests just like any other web server. It serves the *Facets* client application, giving users the possibility to view existing analyses or starting new ones. In case a new analysis should be started, it performs a few preliminary checks - most importantly, if the supplied repository URL contains at least two releases - and sends it off to the JobManager.

4.2.1 Analysis Work Flow

When a new analysis is issued via the web application, the JobManager makes a new entry in the database and creates a *Job* thread for it. This means that for one code repository that shall be analyzed, there is one separate thread. The work flow carried out by the Job thread can be described as follows:

1. Create a new version control analysis on SOFAS using the *Git Importer* or *SVN Importer* service and wait for it to complete. The version control services extract the entire version control history from the source code repository.

2. Supply the SOFAS *Release Famix* service with the URL of the finished version control analysis to create a new analysis and wait for it to complete. The releaseFamix service constructs a FAMIX[2] model of the Java source code for each release discovered in the repository.

3. Retrieve the list of releases from the Release Famix service, including the URL at which their FAMIX model resides.

4. For each of these releases, create two new analyses on SOFAS, one on each of the two metric analysis services and wait for all analyses to complete: The *Size and Complexity Metrics* service will calculate metrics such as the lines of code and McCabe's cyclomatic complexity. The *Object-Oriented Metrics* service will calculate a plethora of roughly 30 more metrics, such as the number of classes, packages or methods, the number of operation calls, inheritance-related information, such as base class overriding ratios or hierarchy heights and coupling-related information, such as class cohesion and access to foreign data.

5. For each pair of metric analyses that belong to a release, issue a new analysis with the *Code Disharmonies* service, which will use the detection strategies proposed by Lanza et al. to find problematic entities within the design of the project.

6. Gather all data that is relevant to the *Facets* application from the different services, synthesize it appropriately and store it in the local database. Once this is done, the analysis becomes available in the frontend.

7. In a last cleanup step, delete all analyses on SOFAS since they are not needed anymore.

4.2.2 Job Management

Because the JobManager is a plugin, it will be signaled in case that CherryPy is shutting down. This means it can finish any outstanding business and exit cleanly. Break points are scattered all throughout the analysis code so that any running analysis can stop within a few seconds if the server is signaling that it's shutting down. The analysis will be resumed at the point of interruption when the server is started anew. For the most part, functionality is safe, meaning that a step during an analysis can be executed several times without introducing inconsistencies, meaning

[2]FAMIX is a meta model for representing object-oriented software systems. [14]

that even if the job manager is killed unsafely, it will pick up the process just before it was interrupted. The owner of an analysis can delete it directly from the web interface, be it in progress or finished. At that point, any existing data will be deleted from the local database and the analyses on SOFAS will be deleted as well.

4.2.3 Analyzer and Sofas Agent

The aforementioned Job Manager controls the flow of an analysis. However the code for carrying out individual steps is contained within the *Analyzer* Class. Similarly, actual Network calls are not carried out by the Analyzer but by the *Sofas Agent*. This three layered approach separates different types of functionality in a transparent way.

4.2.4 Database

Access SOFAS directly or store a copy in a local database?

Once an analysis on a SOFAS service is complete, it is possible to retrieve the data in a number of ways. One could simply download a zip file containing the entire RDF graph as an RDF/XML file. This file could be parsed locally to retrieve the necessary data. On the other hand, it is possible to query the RDF graph directly on the server by doing a SPARQL query. This has several advantages: The data is only stored in one place and the amount of data transferred for a single piece of information is largely reduced. On top of this, the RDF graphs produced by SOFAS can be very large and parsing RDF graphs can be a taxing task for the server. The SOFAS server is already well equipped for this as it has the necessary computing power and as it is using a fast triple store. On the other hand parsing the RDF/XML files from the zip files tends to be very slow. This means that it's imaginable to create a visualization front-end that does not depend on its own database, because any information can be retrieved directly from *Facets*.

However, querying SOFAS directly also a few significant disadvantages: First and foremost, it is not easily possible to query more than one service at a time via SPARQL. This means that it's not possible to extract information from one service depending on information from another service. This means that in some cases, large amounts of data would need to be downloaded from all relevant services and then processed locally, every time the analysis is retrieved. A second problem is that while the SOFAS services can respond fairly quickly even when there are millions of triples, the time it takes to query all services quickly adds up. Lastly, a lot less data is required for the visualizations compared to what SOFAS stores on each analysis. After all, SOFAS is intended as a tool for *analyzing* software and not as a permanent database.

Table 4.1: The metric properties and physical storage needed in different contexts of three analysed projects.

Storage method	gstreamer-java	GS-Collections[3]	Cordova Android
Number of releases	7	3	29
LOC in latest release	15503	101455	8718
Classes in latest release	310	750	77
Methods in latest release	8955	13001	659
SOFAS RDF/XML (uncompr.)	355.8 MB	818.7 MB	292.4 MB
SOFAS RDF/XML (zipped)	15.9 MB	31.9 MB	12.6 MB
MongoDB documents (uncompr.)	4.5 MB	2.0 MB	1.8 MB
Transferred to web client (gzipped)	0.15 MB	0.16 MB	0.09 MB

For these reasons, *Facets* opts for using SOFAS to generate the necessary data, retrieving only the parts that are needed for the visualizations and then deleting the analyses on the individual services, freeing the resources and cleaning up after itself. Table 4.1 shows the physical amount of space needed to store the analysis data on SOFAS as compared to *Facets'* MongoDB and to how much data is transferred to the Javascript web client. The three projects used as examples are the Java bindings for gstreamer[4], GS-Collections[5] and Cordova Android[6]. The table illustrates that the the data stored in MongoDB is smaller than the compressed RDF/XML data by a factor of at least three. The data that is finally transferred to the client - including the mentioned redundancies - is gzip compressed and hundred-fold smaller than even the compressed data that can be retrieved from SOFAS in the case of downloading the entire graphs. Of course, when querying SOFAS directly for the data being visualized, not the complete set of data would need to be transferred from SOFAS to the visualization application, but still all of the data will need to remain stored on the services.

Data storage and representation

The most important requirement towards a database in case of the *Facets* application is performance. After all, this is one of the reasons why the data is stored locally. MongoDB is a high performance document store with support for indexes. Apart from its performance, MongoDB has a number of advantages. Since the documents resemble Python dictionaries[7] there is no need for an object-relational mapping. There are no table declarations in MongoDB and data is instead stored in so-called *collections* which may contain documents of arbitrary shape.

For *Facets*, one collection is used for each kind of data: the hierarchy of packages, sub-packages and classes, class metrics, commits, changesets and project-wide metrics. Two additional collections are used for internal data such as the information on submitted analyses and running jobs as well as the generated PDF reports. Indexes on exactly the right keys ensure fast loading times when querying the data. The documents stored in each collection will of course need to contain a certain set of information for the *Facets* application to work correctly.

The data that is stored is not handed to the client directly but instead, various representations can be generated using the *Representer* class. The data is stored is almost redundancy-free, while the representations transferred to the client contain some duplicated data. This is done to offload some of the calculatory strain to the server - which prepares the representations - so that the Javascript client can work more quickly. For example, the Treemap visualization requires one entity tree for each release. The Treebrowser visualization on the other hand requires all entities from all releases to be placed in a single tree. This is why both trees are generated on the server, using the same data from the database but generating different representations of it. The amount of redundancy is still rather low and it does not impact the transfer speeds significantly.

4.2.5 Extending «Facets»

The visualization code in *Facets* is decoupled from the main functionality of the site in such a way that it is very easy to write new facets. Facets are registered in the main configuration file with their representational dependencies stated. Listing 4.1 shows the default set-up. For example, the

[3]The reason why the amounts transferred to the client are similar for gstreamer-java and GS-Collections even though the amount of data stored differs significantly is that GS-Collections has few releases but many entities while gstreamer-java has several releases but fewer entities. The redundancy caused by transferring an additional entity tree for all releases is more noticable for large projects with few releases.

[4]`http://code.google.com/p/gstreamer-java/`

[5]`https://github.com/goldmansachs/gs-collections`

[6]`https://github.com/apache/incubator-cordova-android`

[7]`http://docs.python.org/tutorial/datastructures.html#dictionaries`

Overview Pyramid visualization requires the list of *releases* and the *project_metrics* representation. When a user loads an analysis, the representations are downloaded in parallel and the visualizations will be loaded automatically as soon as their required data representations are available. The facet visualizations themselves consist of two files each: One contains an HTML snippet, the other contains a function. The developer can position the visualization in a view by placing it into a simple HTML template. Again, *Facets* will automatically locate and serve the visualization as soon as it's mentioned in the configuration file.

```
[facets]
pyramid=releases,project_metrics
disharmonies=releases,disharmonies
treemap=releases,trees,disharmonies
treebrowser=releases,tree,commits,disharmonies
history_loc=commits,release_dates
history_commits=commits
history_metrics=releases,project_metrics
```

Listing 4.1: 'facets' section of the configuration file.

The HTML snippet will probably at the very least need a single container element which can be manipulated by the corresponding Javascript code. The Javascript file follows the very simple template outlined in 4.2. As such, all that is needed is a function that should be registered by saving it as a property of `facets.vis`.

```
var setup_myvis = function(container) {
   // Code reading from representations variable and
   // drawing the visualization goes here.
}
facets.vis.myvis = setup_myvis;
```

Listing 4.2: Minimal Javascript template for writing a new visualization.

This method is particularly elegant because it doesn't require any special server side handling of the content. The developer simply writes one HTML snippet and one Javascript function and registers the visualization in the configuration file. There is no necessity to modify any of the code that actually serves the analyses. It is also very easy to write new representations: The *Representer* class contains one function for each representation. Representation functions are automatically located if they are specified as required in the configuration file.

4.3 Report Generation and Printing

The user has the option to generate a PDF report for the entire project. Actually, several methods for creating a report have been explored. In the final version of *Facets*, there are three possible ways: The *Generate HTML-PDF Report* button will initiate a procedure on the server that will render a dedicated HTML template which is optimized for printing[8]. The *Download PDF Report* button does the same but uses a LaTeX template. Finally, the user can use the *Annotate & Print* functionality, which opens an additional browser window where all visualizations - including any changes done or duplicates created by the user - are present. The user can then add some annotations and print the result using regular printer or a virtual PDF printer. Another method which is not available in the final product is the server-side rendering of the client-side HTML,

[8]This functionality has been hidden from the front-end because of bugs in the third-party rendering engine (PhantomJS) used to generate the report. The bugs are outlined in section 4.3.1. Once these bugs are fixed by the developers, it is possible to re-enable the functionality by uncommenting a few lines of HTML.

which is subject to many security implications, which will be outlined shortly. We shall now look at the different methods in detail.

4.3.1 Off-Screen Rendering of a dedicated HTML template

When the user requests a PDF report through the front-end, the server will use a special HTML template dedicated for printing and fill it with content. The content of a PDF Report for a given project analysis will always be the same. The Treemap, Treebrowser and Disharmony visualizations are not present and instead, additional textual content is being generated. A list of disharmonies for the latest release will be compiled and additionally, the top 10 entities for the most important metrics will be gathered. As such, the report will for example include the entities with the most lines of code or the highest method count. The HTML is parsed and rendered to PDF using bleeding edge technology, namely by the use of so called off-screen or head-less renderers. Two of the most actively developed off-screen renderers are PhantomJS[9] and wkhtmltopdf[10]. Both of them use the Open Source WebKit[11] browser engine. Unfortunately, these solutions still suffer from significant bugs. At the time of writing, PhantomJS v1.5 does not render page breaks smoothly and causes lines to be cut in half[12]. It also ignores the important `page-break-before` and `page-break-inside` CSS directives, which in our case can be used to avoid page breaks inside visualizations[13]. Hence, the generated PDF report is not of very good quality.

As the off-screen rendering engines improve, this method may become a more viable option because after all, it is very easy to implement and it avoids a lot of duplication since one can use almost the same templates instead of writing additional export functions.

4.3.2 Annotate & Print

To complement the inflexible server-side PDF report generation and to still offer the user the option to print the interactive version of the visualizations in a useful fashion, an additional functionality is provided: As described earlier, the user can view single packages in the Treemap and focus specific cells. In the Treebrowser users can select a specific package to view its metrics and ownership history and the disharmony list is configurable as well. Coupled with the duplication functionality, this gives the user the power to focus on specific parts of the system. Clicking on *Annotate & Print*, the exact configuration as seen in the browser will be copied and modified to be better suited for printing. A few modifications are made to the body, removing some things that have to be omitted from the report (such as the menu), and additional text fields for annotation are added below each visualization. This allows the user to explain his entity choices and further comment on the report.

This solution has however one significant usability issue: Most browsers are configured by default to not print background colors of elements and they also add unappealing headers and footers to each page. It is not possible to change these settings via Javascript and the user himself has to know where to change them. If printing background colors remains disabled, the visualizations will not be printed correctly. Chrome/Chromium users will not need to worry as the browsers offers a `-webkit-print-color-adjust: exact;` CSS directive that will enable the printing of background colors. For users of other browsers, short instructions are provided on how to enable it.

[9] `http://phantomjs.org/`
[10] `http://code.google.com/p/wkhtmltopdf/`
[11] `http://www.webkit.org/`
[12] `http://code.google.com/p/phantomjs/issues/detail?id=551&sort=-id`
[13] `http://code.google.com/p/phantomjs/issues/detail?id=506`

A noteworthy implementation detail is the fact that for each visualization, the developer can add a function to the global *facets.plain* Javascript object. This function will be called on the visualization container when transforming the HTML to the printer-friendly version. This means that the developer can influence the print view of his visualization without modifying any files other than his Javascript visualization code.

4.3.3 Server-Side Rendering of Client-Side HTML

Initially, the PDF report generation was implemented differently and even more ambiciously than in the final product: The server wouldn't generate any content but instead, the actual state of the client HTML - including duplicates and selections - would be sent to the server where it would be rendered to PDF just like it is now the case with the *Annotate & Print* functionality. The entire <*body*> of the web application was duplicated on the client side and a different <*head*> was prepended to the copy. Again, modifications were made to the body to make the HTML better suited for printing - all by the use regular JQuery methods. This modified copy was then sent to the server where it was be rendered to PDF using PhantomJS. As such, the report generation was not a decoupled functionality, which is static per-project and generated on the server, but instead it would actually depend on the state of the web applications. The advantage of this approach is that the user does not need to have a virtual PDF printer in order to be able to save the modified visualizations. Not all operating systems have a virtual PDF printer installed by default.

The main issue with this approach is security: There is no sane way to prevent the user from sending any HTML content to the server for rendering. It's possible to disable Javascript in the off-screen renderer but still, there's no sandboxing and by including other directories in the HTML, the user could view server system files. The only thing preventing this would be the creation of a strict schema, however that is very difficult given the complexity of the visualizations. Another option would be to record the configuration of each visualization as it is modified and then replay the modifications upon generating the visualization on the server but again, this adds a lot of complexity and overhead. For this reason, the idea of rendering the client side HTML on the server was abandoned. Even though it actually works in practice, the security implications are too severe and unpredictable.

4.3.4 Rendering a Dedicated LaTeX Template

Since both the sever-side HTML rendering and the manual printing functionality each suffer from a few issues, it becomes clear that the media discontinuity between the web and paper is still very harsh. It's rather difficult to mend and bend the HTML-based presentation into something that still looks attractive when viewed in a PDF or on paper.

For this reason, an additional method has been developed: Similarily to the server-generated PDF report, this method produces a report that cannot be influenced by the user. However, it only renders the individual visualizations using PhantomJS, exporting them as PDF-documents. After this they are re-embedded into a LaTeX template which is rendered using pdflatex. This produces a very tidy report, since LaTeX is a superb typesetting engine and the visualizations look sharp because the PhantomJS' rasterization produces vector based shapes. The LaTeX template is also processed by Jinja2, just like the HTML templates. Because LaTeX uses curly braces extensively, the templating tokens are modified from `{% %}` to `((* *))` for block statements and from `{{ }}` to `((( )))` for inline statements. To illustrate the process, listing 4.3 contains an example snippet from the template. This snippet is responsible for generating the *Exceptional Entities* section.

```
\section{Exceptional Entities}
((* for type in extypes *))
\Needspace*{4cm}
\subsection{Top 10 entities by (((type["label"]))).}
\begin{longtable}{|p{0.10\textwidth} p{0.60\textwidth} p{0.15\textwidth}||}
   \hline \textbf{(((type["name"])))} & \textbf{Entitiy} & \textbf{Disharmonies} \\
       \hline
   \endfirsthead
   \hline \textbf{(((type["name"])))} & \textbf{Entitiy} & \textbf{Disharmonies} \\
       \hline
   \endhead
   \hline
   \endlastfoot
   ((* for d in exceptionals[type["id"]] *)) (((d["metrics"][type["id"]]|int))) & (((d["
       class"]|limitLength|escapeTex))) & (((d["disharmonies"] | countwNone))) \\
   ((* endfor *)) \hline
\end{longtable}
((* endfor *))
```

Listing 4.3: A snippet from the Jinja2-LaTeX template which generates a table of the top ten entities for each type of exceptionality.

Chapter 5

Related Work & Reflections

We have shown how *Facets* levarages the capabilities of the individual SOFAS service to create an extensive analysis of a software project. In this chapter, we shall mention a few similar approaches that try to carry the subject of software analysis and visualization to the web. Based on the experience gathered by implementing *Facets*, we will also point out some of the shortcomings of certain visualization techniques in use today. Finally, we will come back to some closing remarks on the application itself and comment on the difficulties that were encountered along the way.

5.1 Related Work

D'Ambros et al. have developed a set of tools aiming in a similar direction as the SOFAS/*Facets* combination. Churrasco[1] is a collaborative, web-based platform that can create models of software systems and record their version control history. As opposed to SOFAS, Churrasco is not only a service application used to analyze data but instead it also includes ways for users to interact with existing analyses, for example to annotate their object model. Churrasco also contains a visualization module. It can be said that Churrasco is intended as a complete tool targeted at end users while SOFAS is more of a flexible service platform, reachable via an API and to be used in any software project that might want to benefit from the analyses provided by it, like it is the case with *Facets* [10].

At the University of Auckland, Tempero et al. are exploring ways to deliver visualizations over the web using technologies and tools that already exist today. A number of smaller projects, prototypes and use cases have been developed to evaluate how well existing visualization services such as Google's Visualization API[2] or IBM's «Many Eyes»[3] can be used to visualize software systems. An example is LiveJ[4], which is a collection of visualizations created by uploading spreadsheet data to the Google Visualization Plattform. For the most part, they conclude a lack of software visualization techniques, but since the Plattforms themselves work very well, they may include some of the Visualizations in future projects [1,2].

There is also a number of implementations for single visualizations which are not embedded into any specific tool. For example, Ogawa et al. have developed an SVG visualization called «Storylines», which is able to visualize developer interaction throughout the lifetime of a project [22]. Another SVG visualization has been developed by Beyer at al.: «Storyboards» illustrates

[1]http://churrasco.inf.unisi.ch/

[2]https://developers.google.com/chart/interactive/docs/reference

[3]http://www-958.ibm.com/software/data/cognos/manyeyes/

[4]http://homepages.ecs.vuw.ac.nz/~craig/livej/

structural changes in software systems by the use of animates panels. Both of these visualizations could be included in an application like *Facets* because they are purely web based [4].

A surprisingly early solution, REportal, by Mancoridis et al. was build in 2001 as a web site where users could upload their C, C++ or Java source code to have it analyzed. The platform consisted of several reverse engineering tools that the user could access to browse and understand the project. Again, one of the main motivations for the project was the fact that users would not need to install analysis and reverse engineering software on their own computers, hence saving time and effort [7].

5.2 Web-Based Visualization Tools in Practice

On the more general topic of porting software analysis and visualization tools to the web, a number of opinions already exist. D'Ambros et al. dedicated a paper to describing the promises and perils of such an undertaking. They had previously developed a variety of desktop-based visualization tools and ported them to the web forming the previously mentioned Churrasco platform. They come to the following conclusions: Web applications are more readily available than tools that need to be installed for each user and since Churrasco allows users to annotate existing analyses, it also facilitates the collaboration between developers. The ability to gather usage statistics for the web service may be advantageous to see how developers interact. They say that through the web service, third party libraries and components can be used more easily since they don't need to be distributed to each client and are instead only used on the server instance. D'Ambros et al. also describe a number of perils which need to be evaded carefully: For example, sensible information about the analyzed software systems needs to be kept safe from unauthorized visitors to the site. The remaining issues they list are very typical for any kind of web service: There are performance penalties stemming from both the server-client architecture and the increased load on the server in case several visitors are using the platform at the same time. The web service is also a single point of failure. And of course one of the most well known problems with developing web services is the fact that different web browsers never exhibit identical behavior and as such, compatibility issue need to be handled. Finally, web technologies are still evolving rapidly and the longevity of any existing technology isn't guaranteed [11].

Many of these claims have held true during the development of *Facets*. The application should run smoothly in both gecko-based browsers such as Firefox as well as WebKit-based browsers such as Chromium or Safari. However it does not work in Internet Explorer and of course it's neither intended for nor does it work in web browsers for mobile devices. Information security is not an issue in this particular case since all the analyses are public anyway but as pointed out when describing the PDF report implementation, there are still some security constraints that cannot be avoided easily. Finally, a lot of work has gone into making *Facets* perform well, but the loading times are still noticable, not primarily because of network bottlenecks but because of the computational task that put significan strain on the browser's Javascript engine.

5.3 Working with SOFAS

Facets utilizes a large number of services offered by SOFAS. Whereas until now the services had mostly been used manually and on small and singular occasions, *Facets* uses them on a larger scale. Having had the opportunity to work extensively with SOFAS, I can say that - from a developer perspective - the service oriented architecture offers a very neat yet still flexible way of tackling the broad and complex issue of software analysis. There are some bugs left in some of the services and during the development of *Facets* feedback was frequently provided on both the ontologies as well as the data supplied by SOFAS. Numerous bug fixes and improvements were implemented.

In any case, *Facets* serves as an example of how new applications can be driven by SOFAS. By the use of its services, developers are able to concentrate on working with the data, for example to create visualizations or to create new analyses which can in turn be made a part of SOFAS instead of having to bother with the installation or even the implementation of libraries that produce the data.

5.4 Extending «Facets»

It is very easy to write new visualizations using the existing data that is stored in MongoDB. The process of writing new visualizations is completely unintrusive because no existing files - other than the configuration file - need to be modified. Not only is it possible to add a new visualization to the existing analysis page within a single Javascript file, but it is also possible to specify a function that will be run before the print view is generated.

If a new visualization should require a different representation of the data, it's very easy to just add a new function to the Representer class, again without intruding into other components of the code. Registering the new visualization with its new representation dependency is all it takes.

Regarding the synthesis of additional data, the situation is more difficult: First, adding visualizations that require new data and hence adding steps to the synthesis function means that *all* analyses need to be re-run on SOFAS. If they would not be re-run, data would be missing, since *Facets* is not intended to serve different-looking analyses depending on what data is available. Adding support for different version would have been beyond the scope of this project.

5.5 Future Work

Due to the scope of this project - it being a bachelor thesis - it was not possible to implement every feature that came to mind.

It would be a good idea to add support for non-public repositories. Right now, a repository must be publicly accessible for it to be analyzed. It would be useful to provide authentication mechanisms such as public key uploads or credentials that enable the service to access non-public repositories. In this case, the user should also be able to choose to hide his analysis from public viewing. All of these features would still be possible even without user registration by the use of access tokens.

Shorter turn-around times for new analyses of the same project are desirable: Right now, *Facets* is laid out as a tool to analyze a code base in a singular event. If a new version of a tool is released, the existing analysis will not be reused and the entire project will be analyzed anew. One of the reasons why many analysis and visualization tools are integrated into IDEs is that they can supply information continuously and the SOFAS/*Facets* combination is unable to do this. A possible solution would be to have the code repository server and analysis services work hand-in-hand: If a user pushes his commits to the repository, the analyses could be updated automatically, however this approach is divergent from the modular and decoupled approach of SOFAS and would require either a significant extension to SOFAS that enables the transmission of events or an external monitor that would rerun analyses as required.

Of course, *Facets* could always be extended with new visualizations. There are still plenty of visualizations that are worthy to be implemented, for example the Evolution Radar [12,20], simple bar and line charts for additional metrics or the previously mentioned Storylines visualization.

There are also a few other facets of software evolution which have not even been touched in this project. Code clone detection, visualized by dot plots [20] is one example.

Chapter 6

Conclusion

Software analysis tends to require preparation and can be rather time consuming. The initial effort required to even start analyzing software may pose a barrier for development teams that do not have enough personnel capacities to spend on this topic. Having an easily accessible web service and being able to analyze several aspects of a software project at the same time by simply supplying the URL of the code repository tears down this barrier.

The *Facets* application succeeds at creating a complex work flow using the services provided by SOFAS, aggregating the data produced by said services and finally visualizing the data. The web application is visually appealing and very easy to use. Browsing the visualized data of an analysis happens according to the well established *Information seeking paradigm* [19]: Overview first, zoom and filter, and then details-on-demand. The user can get an impression of the shape of his project by browsing the Overview facets. Further investigating the details of individual packages or classes is easily possible using the Treemap, where zooming into individual packages is possible, with further details available upon hovering over individual cells. Finally, the user can print or export a report. Four different approaches, each with their own upsides and downfalls, have been explored for this purpose.

For developers, the *Facets* back-end offers enough flexibility to unintrusively accommodate new visualizations in the web frontend, which will be available to all existing analyses.

Developing *Facets* has also procured a number of lessons learned regarding the still rather novel approach of providing software analysis and visualization services on the web. There exists a need for short turn-around times, which is likely one of the reasons why many analysis tools are implemented as plugins for IDEs. Another challenge is coping with the space and performance constraints given by the network and the web browser. Analyzing software repositories produces a very large amount of data, yet as little data as necessary should be transferred to the client. And even though computation resources are limited in the web browser, the demand for very fast loading times and quick browsing is higher than ever. In *Facets*, the problem is tackled by synthesizing the data from SOFAS into another database, which itself increases the complexity of the solution. Last but not least, some visualizations prove to be difficult to work with, depending on the size and diversity of the project.

Bibliography

[1] Anslow C., Marshall S., Noble J., Tempero E.: Towards End-User Web Software Visualization. In the Proceedings of Graduate Consortium at VLHCC. IEEE (2008)

[2] Anslow C., Marshall S., Noble J., Tempero E.: Web Software Visualization Via Google's Visualization API. In Proceedings of the New Zealand Computer Science Research Students Conference (NZCSRSC) (2009)

[3] Balzer M., Deussen O., Lewerentz C.: Voronoi Treemaps for the Visualization of Software Metrics. In the Proceedings of the ACM'05 Symposium on Software Visualization (SoftVis) (2005)

[4] Beyer D., Hassan A. E.: Animated Visualization of Software History using Evolution Storyboards. Proceedings of the 13th Working Conference on Reverse Engineering (WCRE'06) (2006)

[5] Belady L., Lehman M. M.: Program Evolution: Processes of Software Change. London Academic Press (1985)

[6] Bird C., Devanbu P., Gall H., Murphy B., Nagappan N.: An Analysis of the Effect of Code Ownership on Software Quality across Windows, Eclipse and Firefox. Microsoft Research TechReport MSR-TR-2010-140 (2010)

[7] Chen Y., Gansner E. R., Korn J. L., Mancoridis S., Souder T. S.: REportal: A Web-based Portal Site for Reverse Engineering. In the Proceedings of the 8th Working Conference on Reverse Engineering (2001)

[8] Comstock C., Jiang Z.: The Factors Significant to Software Development Productivity. In the proceedings of the 19th International Conference on Computer, Information, and Systems Science, and Engineering. World Academy of Science (2007)

[9] García E., Harrison R., Rodríguez D., Sicilia M. A.: Empirical Findings on Team Size and Productivity in Software Development. Journal of Systems and Software Vol. 85, Iss. 3, p. 562–570 (2012)

[10] D'Ambros M., Lanza M.: Distributed and Collaborative Software Evolution Analysis with Churrasco. In the Journal of Science of Computer Programming (SCP), Vol. 75. No. 4, p. 276 - 287 (2010)

[11] D'Ambros M., Lanza M., Lungu M., Robbes R.: Promises and Perils of Porting Software Visualization Tools to the Web. In the Proceedings of the 11th IEEE International Symposium on Web Systems Evolution (WSE) (2009)

[12] D'Ambros M., Lanza M., Lungu M., Robbes R.: The Evolution Radar: Visualizing Integrated Logical Coupling Information. In the Proceedings of the 2006 International Workshop on Mining Software Repositories (MSR '06) (2006)

[13] Demeyer S., Mens T.: Software Evolution. Springer ISBN 978-3-540-76439-7 (2008)

[14] Demeyer S., Steyaert P., Tichelaar S.: FAMIX 2.0, The FAMOOS Information Exchange Model (1999)

[15] Demeyer S., Van Rysselberghe F.: Studying Software Evolution Information by Visualizing the Change History. In the Proceedings of the 20th IEEE International Conference on Software Maintenance (2004)

[16] Ducasse S., Gîrba T., Lanza M.: Yesterday's Weather: Guiding Early Reverse Engineering Efforts by Summarizing the Evolution of Changes. In the proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM'04), p. 40—49 (2004)

[17] Fischer M., Gall H., Lanza M., Pinzger M.: Visualizing Multiple Evolution Metrics. In the Proceedings of the SoftVis '05 ACM symposium on Software Visualization (2005)

[18] Gall H. C., Ghezzi G.: SOFAS: A Lightweight Architecture for Software Analysis as a Service. Working IEEE/IFIP Conference on Software Architecture, IEEE Computer Society (2011)

[19] Keim, D. A.: Information Visualization and Visual Data Mining, IEEE Transactions on Visualization and Computer Graphics, Vol. 7, No. 1 (2002)

[20] Lanza, M., Marinescu, R.: Object-Oriented Metrics in Practice. Springer ISBN 978-3-540-24429-5 (2006)

[21] Lanza M., Hattori L., Robbes R.: Refining Code Ownership With Synchronous Changes. Empirical Software Engineering, Springer (2010)

[22] Ma K., Ogawa M.: Software Evolution Storylines. In the proceedings of the SoftVis '10 ACM symposium on Software Visualization (2010)

Chapter 7

Appendix A: Technical Documentation

7.1 Deployment

Facets is very easy to install given all dependencies have been installed. It is intended to be installed to a specific directory. For the remainder of these instructions, it will be assumed that the directory is `/opt/facets/`. This directory should be created first. If some of the Python libraries are not available through the repository of your Linux distribution, they are often available via `easy_install`, which is almost certainly available in the repositories. If not, `easy_install` can be installed manually[1].

It is recommended for both safety and transparency to create a new user for running *Facets*. It will be assumed that this user is called `facets` and that its home directory is `/opt/facets`.

The hardware requirements for *Facets* can be outlined as follows: MongoDB needs a lot of memory to perform properly. The bigger the database gets, the more ram is needed. This means that for testing and development, anything will suffice, however for production, at least 1GB of ram should be reserved for MongoDB alone, with more being needed if the data set grows. On 32-bit operating systems, MongoDB cannot store more than about 2GB of data, so a 64-bit operating system is recommended.

7.1.1 Installing Dependencies

Python

The *Facets* server application requires at least Python 2.7 and it is not compatible with Python 3. Python 2.7 is available in all popular Linux distributions and sometimes it is installed by default.

CherryPy

CherryPy is a fast and stable production-grade web development framework. It brings all the necessary tools - including the web server itself - for writing web applications in Python, yet it still has a very small footprint. Version 3.2 is the minimum requirement for *Facets*. If CherryPy 3.2 is not available through a repository, it can be downloaded as a tar file[2]. It can then be either installed on the system by running `python setup.py install` or it can be installed locally and just for *Facets* by unpacking only the `CherryPy-3.2.2/cherrypy` folder to `/opt/facets/`.

[1]`http://pypi.python.org/pypi/setuptools#cygwin-mac-os-x-linux-other`

[2]`http://download.cherrypy.org/cherrypy/3.2.2/CherryPy-3.2.2.tar.gz`

lxml

SOFAS answers SPARQL queries in XML. lxml is a light-weight XML parser for Python. At least version 2.3.3 is required. If it is not available through the repositories, it can be downloaded[3] and installed manually or via `easy_install --allow-hosts=lxml.de,*.python.org lxml` as outlined by the build instructions.

httplib2

httplib2 is used to communicate with the SOFAS services. At least httplib2 0.7 is required. If this version is not available through the repository, it can be downloaded and installed manually[4]

Jinja2

Jinja2 is the templating engine used in conjunction with CherryPy to generate the HTML, Javascript and LaTeX code being used to generate the documents served to the user. *Facets* has been tested with Jinja 2.5 but older version may work. As usual, if it is not available through the repositories, it can be installed via `easy_install Jinja2` or from the latest tarball[5].

MongoDB

MongoDB serves as the database for *Facets* where the synthesized data from SOFAS is stored. It is a high performance NoSQL document store. Installing it via the repository may directly install it as a service in which case no further configuration is required. It can also be installed following the instructions on the website[6].

Javascript Libraries

Facets utilizes JQuery[7], JQueryUI[8] and d3[9] for its interface and visualizations. The appropriate version are incorporated in the *Facets* source tree which means that no manual installation is required.

7.1.2 Installing «Facets»

Facets does not need to be built and can be grabbed directly from the source code repository. Simply change to the designated directory and clone the source tree - or if using the version from the CD, unpack the contents of `facets_application.tar.gz`. The hierarchy can be described as follows:

- `facets.py`: The main executable to run *Facets*
- `cfg`: Contains all configuration files
- `lib`: Contains *Facets'* server side libraries

[3] `http://lxml.de/files/lxml-2.3.4.tgz`
[4] `http://code.google.com/p/httplib2/wiki/Install`
[5] `http://pypi.python.org/pypi/Jinja2`
[6] `http://www.mongodb.org/display/DOCS/Quickstart+Unix`
[7] `http://jquery.com/`
[8] `http://jqueryui.com/`
[9] `http://d3js.org/`

- `static`: Contains statically served files such as Javascript, HTML and image files.
- `static/viz`: Contains all visualization Javascript files
- `template`: Contains templates used to dynamically generate Javascript, HTML and LaTeX source files.
- `misc`: Various extra files that are not needed for running *Facets* but that may prove useful.

7.1.3 Configuration

Configuring CherryPy

By default, *Facets* will run on Port 8090. If you want to run it on a different port or if you have installed it to a different directory, you will need to edit the CherryPy configuration file located at `cfg/facets_cp.ini`. The file is self-explanatory and the only parameters that need to be changed are `server.socket_port`, `log.access_file` and `log.error_file`.

Configuring *Facets*

Many run-time parameters in *Facets* can be configured. The configuration file resides at `cfg/facets.ini`. The configuration file contains the following sections:

- `sofas`: The options in this section are used to tell *Facets* where the SOFAS services are residing. The `xml_encoding` parameter should match the configuration of the services.
- `owl`: Stores the locations of the ontologies so they can be adjusted if necessary in the future.
- `auth`: The username and password used to access the services.
- `param`: Various options necessary for operating *Facets*:
  - `check_interval`: Polling frequency in seconds when waiting for analyses to finish. Each poll is an HTTP call to SOFAS to check the HEAD status of the analysis.
  - `fail_retries`: How many times a failed SOFAS analysis will be restarted before final failure is admitted.
  - `mongo_db_name`: The name of the database in MongoDB.
  - `job_verbosity`: Logging style for analysis progress. The two possibilities are `short` (regular logging) and `direct` (Print entire job document from MongoDB without formatting.
  - `phantomjs|rasterize|pdflatex`: The locations of the binaries for PhantomJS, the rasterize script and pdflatex.
- `visualizations`: Each visualization in the client application needs a certain set of data *representations*. This section contains the $visualization \rightarrow representation$ dependencies. The history of lines of code for example needs to know about the commits and the release dates, hence the configuration reads `history_loc=commits,release_dates`. This section is especially important when extending *Facets* because new visualizations need to be registered in it.

Configuring MongoDB

In many Linux distributions, MongoDB will not require any configuration. Simply make sure that it's running. It's also unnecessary to create any scaffolding (like tables or users in SQL databases) because they will be created automatically when *Facets* is running. Just remember that the database name for *Facets* to use can be configured in `cfg/facets.ini`.

7.1.4 Operation

After everything is installed and configured, it may be best to run a `chown facets:users /opt/facets` just for good measure to make sure that all files are owned by the facets user. After this, the server can either be started manually by running something like `python2 /opt/facets/facets.py` upon which the output from the application will be logged directly to the terminal. This is useful while developing and debugging.

Daemonized Operation

To run *Facets* as a system daemon, you will need to write a daemonizing script for your Distribution. A script for OpenSUSE can be found in the `misc` folder. Guides for other distributions can be found in their respective wikis. There's not much that can go wrong since it's impossible to run two instances at the same time. Running *Facets* when it's already running will abort early since it can't bind to the specified port (8090 by default).

Logging

There are three log files: `log/facets.log` contains the regular output from the *Facets* application (logging startup, shutdown and new analyses and their progress). `log/facets_access` is the regular HTTP log access file and CherryPy logs its own bus and operational information to `log/facets_cherrypy`.

7.2 Extending «Facets»

There are three components that can be extended. The first and easiest to extend is the visualization part. It is very easy to write new visualizations and incorporate them into the *Facets* analysis front-end. The second component is the *Representer* Class which reads data from the MongoDB database so that it can be used by the visualizations. There are already a handful representations that may be reused by newly written visualizations but if the need arises, new representations can be written without much effort. Finally, it may be necessary to write new steps during synthesis if the data currently stored in the database does not suffice.

7.2.1 Writing Visualizations for «Facets»

First of all, developers needs to familiarize themselves with the components involved in generating the visualizations. Chapter 4 of the thesis explains the most important aspects. A new visualization can be written in three easy steps. In the following sections we will write a trivial sample visualization to illustrate the process. Let's call it *release_loc* - it will tell the user the number of lines of code for each release.

Step 1: Write the necessary HTML and Javascript Code

Visualizations are embedded into the main analysis page as simple snippets. The HTML needs to go into the template folder, so we create a file `template/release_loc.html` that may look like listing 7.1:

```html
<h6>Lines of Code per Release</h6>
<p>This sample visualization tells us the number of lines of code for each
 release. <span class="more">Text inside this span will initially be hidden.
 A [More] button will be available for the user to click on to reveal the
 contents. Citations can be included using the sup element. Inside it, we
 place the name of the source. Facets will automatically resolve it and
 replace it with the appropriate number<sup>lanza</sup>.</span></p>
<div class="container" id="release_loc_0">
   <div class="release_loc">
      <div class="container">
         <div class="duplicator">[+]</div><div class="deleter">[-]</div>
         <div class="content"/>
      </div>
   </div>
</div>
<script type="text/javascript" src="/static/viz/release_loc.js"></script>
<script type="text/javascript">
{{wrapper}}
</script>
```

Listing 7.1: Sample visualization HTML code.

The corresponding Javascript code should go into `static/viz/release_loc.js`, which may look like listing 7.2. Note that there is just one function `setup_release_loc` into which the visualization code goes. The last line of the file just after the function is very important: It registers the function which enables the loading and duplication of visualizations.

```javascript
var setup_release_loc = function(container) {
   // The following condition is needed if you want to use the duplication
   // eature. It ensures that the original visualization cannot be deleted.
   if (container.attr("id") != "release_loc_0") {
      container.select(".deleter").style("visibility", "visible");
   }

   // You can reference any representation you want. But make sure you don’t
   // modify them! Copy the values first, if you need to modify them. Note that
   // the following lines do not copy them and you’d still be modifying the
   // originals! This is just for convenience.
   var releases = representations["releases"]
   var projectMetrics = representations["project_metrics"]

   // The container will refer to the ’container’ class element in the HTML
   // we created. From here on, do whatever is needed using d3 or JQuery.
   var vizWidth = 700;
   var maxLines = 0;
   d3.values(projectMetrics).forEach(function (a) {
           if (maxLines < a["LOC"]) { maxLines = a["LOC"] }
   });
   var widthMap = d3.scale.linear()
      .domain([0, maxLines])
      .range([0, vizWidth]);
   var content = container.select(".content")
   container.style("width", vizWidth + "px")
      .style("margin", "0 auto 0 15px").style("position", "relative")
      .style("font-size", "0.8em").style("padding", "15px");
   var rows = content.selectAll("div")
      .data(projectMetrics).enter().append("div")
      .style("background-color", "#ffdd55")
      .style("width", function (d) { return widthMap(d["LOC"]) + "px" })
   rows.attr("class", "release_row").append("span")
      .style("width", "60px").style("float", "left")
      .text(function (d, i) {return releases[i] + ":"; });
   rows.append("span")
      .text(function (d, i) {return projectMetrics[i]["LOC"] })

   // Make use of duplication feature. You could add more modifications.
   container.select(".duplicator").on("click",
         function () { duplicate("pyramid", container); });
   container.select(".deleter").on("click",
         function () { deduplicate(container); });
}

facets.vis.release_loc = setup_release_loc;
```

Listing 7.2: Sample visualization Javascript code.

Note that for efficiency reasons, the CSS declarations have been kept in one file, `static/main.css`. You can add you own definitions to that file - prepending all your definitions with the appropriate visualization name class, which in this case would be `.release_loc` referring to the div containing your visualization.

Step 2: Register the new visualization in the configuration

As mentioned earlier, all visualizations are registered in `cfg/facets.ini` together with the representations they depend on. In our example, we need the list of releases and the list of project metrics. So we add our new visualization to the respective section:

```
[visualizations]
pyramid=releases,project_metrics
disharmonies=releases,disharmonies
treemap=releases,trees,disharmonies
treebrowser=releases,tree,commits,disharmonies
history_loc=commits,release_dates
history_commits=commits
history_metrics=releases,project_metrics
release_loc=releases,project_metrics # newly added visualization
```

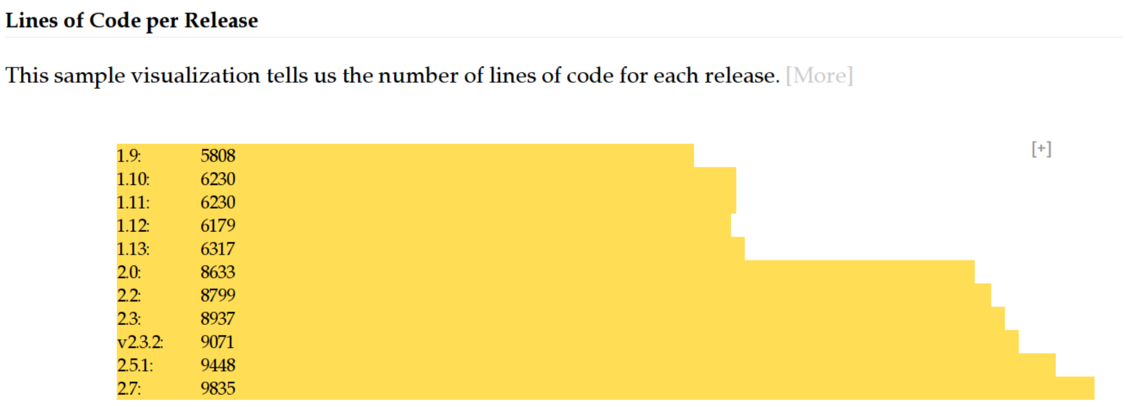
Listing 7.3: visualizations section of the configuration file with the newly registered visualization.

Step 3: Embed the visualization into a view page

A special template can be found in `template/facets.html`. It determines where the visualizations are drawn - in which category and at which point. We add it in the Overview section so that the file now looks like in listing 7.4.

```
<div class="facet" id="facet_overview" data-index="0">
   {{facets["pyramid"]}}
   {{facets["history_metrics"]}}
   {{facets["disharmonies"]}}
   {{facets["release_loc"]}} <!-- newly added visualization -->
</div>
<div class="facet" id="facet_metrics" data-index="1">
   {{facets["treemap"]}}
   {{facets["treebrowser"]}}
</div>
<div class="facet" id="facet_vc" data-index="2">
   {{facets["history_loc"]}}
   {{facets["history_commits"]}}
</div>
```

Listing 7.4: The complete facets.html template, showing how visualizations are placed in different facets.



Lines of Code per Release

This sample visualization tells us the number of lines of code for each release. [More]

[+]

```
1.9:     5808
1.10:    6230
1.11:    6230
1.12:    6179
1.13:    6317
2.0:     8633
2.2:     8799
2.3:     8937
v2.3.2:  9071
2.5.1:   9448
2.7:     9835
```

Figure 7.1: A crude sample visualization that shows the number of lines of code for each release.

7.2.2 Writing Additional Representations

Each representation is a function in the *Representer* class, which is contained in `lib/represent.py`. If an additional representation is required, simply add a function that gets the data from the database. There is no further configuration required, as long as this new representation is mentioned as a dependency to your visualization. Listing 7.2.2 shows a simple representation function that will return the number of commits. Of course this is redundant because one could count the number of commits in Javascript from the *commits* representation, which already exists, so this is just an example.

```
   def num_of_commits(self,name):
      nodes = self.db["commits"].find({"name": name})
      nodes = [node for node in nodes]
      return len(nodes)
```

7.2.3 Writing Additional Steps During Synthesis

The data which is saved to the MongoDB database is retrieved from various SOFAS services and during the synthesis step, a lot of data is pulled in at the same time. To store additional data in the database during synthesis, the `synthesize` function of the `Analyzer` class - which is contained in `lib/analyzer.py` - can be modified. Before starting, the following things should be considered and kept in mind:

1. Is the additional data really needed? Can't it be derived from the existing data? There are two places where data can be modified further: Either in a representation function or directly in the visualization code. If possible, stick to the existing data.

2. If additional data is required because of a new visualization, all existing analyses will need to be deleted and re-run because else, the data would be missing from existing analyses. Since the data will have already been deleted from SOFAS, it is necessary to resubmit every analysis.

3. MongoDB documents cannot be larger than 8 or 16MB, depending on the version. It is usually best to use a new collection for one *kind* of data and to keep the document size down.

Retrieving Data from SOFAS

Facets provides a helper function to retrieve data from SOFAS. It takes the following parameters:

- `url`: The URL to query.

- `query`: The query to use, but without any prefixes. Only the part starting from `SELECT` is needed. If a new prefix is indeed needed, it can be added to the sparql function and inserted in `facets.ini`.

- `store`: This argument takes a **function** that will be run on each row in the resulting result table. The store function takes a container - for example a list or dictionary - and a row as arguments. The store function in listing 7.5 for example takes a dictionary and then stores the contents of row v using the contents of row k as the key.

- `container`: The container that is supposed to be filled. Of course the container must be compatible with the store function. For example one can supply `{}` in this example, but one could also supply a dictionary that already contains something, for example a default key-value pair.

- `sanitize`: If this is set to True, which is the default, any URLs appearing in the results will be stripped of their path component so that only the trailing part of the URI will remain.

```python
def store_key_value(d, row):
   """In a dict, store the value from row v under the key in row k"""
   k, v = row["k"], row["v"]
   d[k] = v
```

Listing 7.5: One of many possible store functions.

Storing Data to MongoDB

Listing 7.6 shows the function that stores the metric information for all classes to MongoDB. The `packages`, `metrics`, `disharmonies` and `children` variables all contain different data which was gathered from different SOFAS services using the above method. Now, for every class in every package, the children, metrics and disharmonies are collected and stored to the `classes` collection in MongoDB. Finally, an index on the "name" key is ensured[10].

```
# store metrics, disharmonies and children for all classes
for classes in packages:
   for aClass in packages[classes]:
      if self.stopped(): return False
      doc = {"name": self.info["name"],
           "release": release,
           "class": aClass,
           "children": children.get(aClass, None),
           "metrics": metrics.get(aClass, None),
           "disharmonies": disharmonies.get(aClass, None)}
      self.db["classes"].save(doc)
self.db["classes"].ensure_index("name");
```

Listing 7.6: An example of how data can be stored in MongoDB.

If a new collection is created for the new data, note that the `delete_local` function of the *Analyzer* needs to be updated so that any existing data will be deleted when rerunning or deleting an analysis.

7.3 Miscellaneous Additional Information

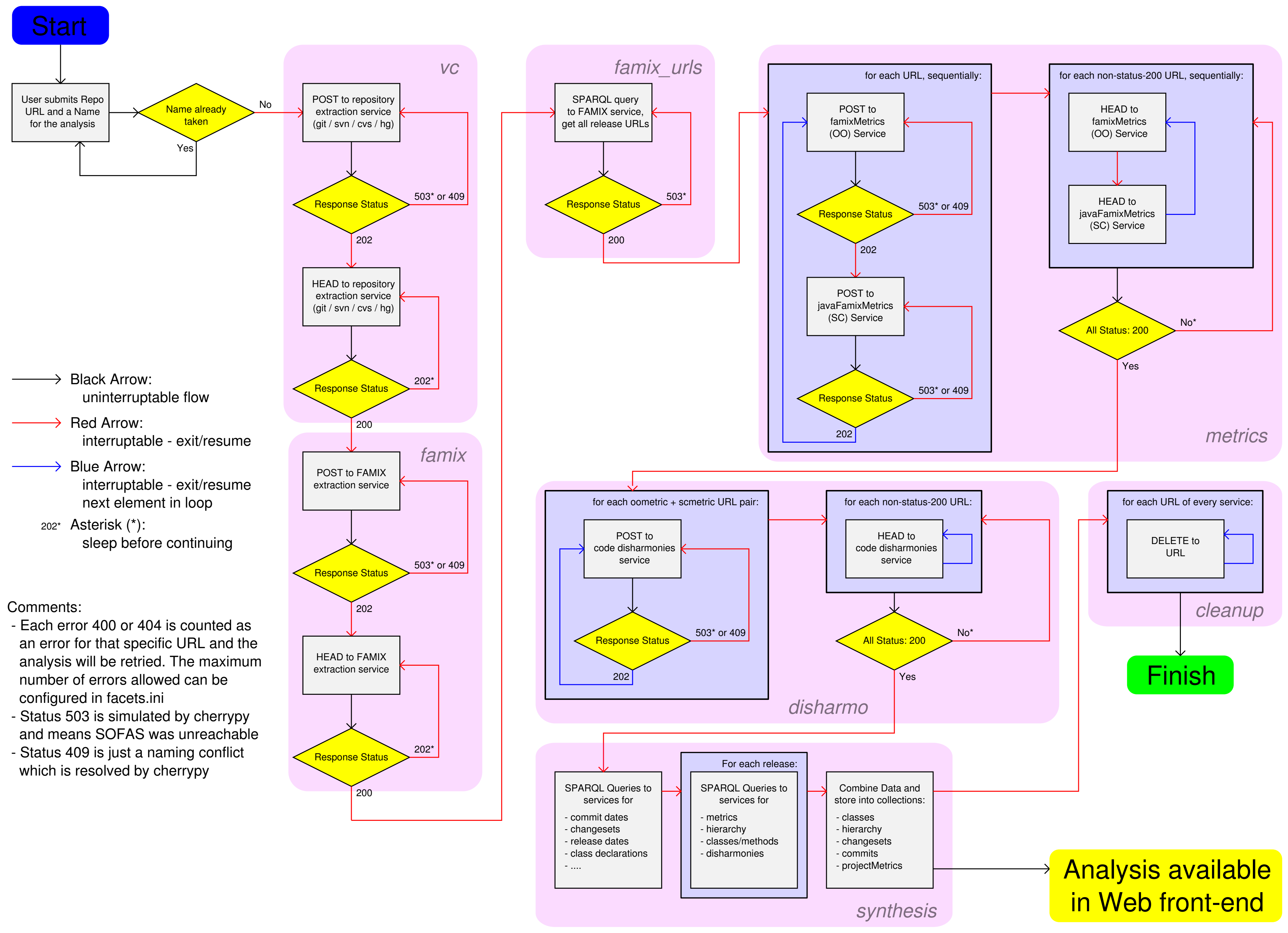
7.3.1 Database Document Templates

A description of the shape of the documents used to store data in MongoDB can be found in `misc/documentation/mongodb_documents`. The file also contains instructions on how to explore the database interactively using python2 or the mongo shell.

7.4 Work Flow Diagram

On the following page you can find a flow diagram that illustrates the whole analysis workflow from the user submitting a URL to the final analysis being presented.

[10]This means the index is created if it doesn't exist and updated if it exists


Start
User submits Repo URL and a Name for the analysis
Name already taken
Yes
No
vc
POST to repository extraction service (git / svn / cvs / hg)
Response Status
503* or 409
202
HEAD to repository extraction service (git / svn / cvs / hg)
Response Status
202*
200
famix
POST to FAMIX extraction service
Response Status
503* or 409
202
HEAD to FAMIX extraction service
Response Status
202*
200
famix_urls
SPARQL query to FAMIX service, get all release URLs
Response Status
503*
200
for each URL, sequentially:
POST to famixMetrics (OO) Service
Response Status
503* or 409
202
POST to javaFamixMetrics (SC) Service
Response Status
503* or 409
202
for each non-status-200 URL, sequentially:
HEAD to famixMetrics (OO) Service
HEAD to javaFamixMetrics (SC) Service
All Status: 200
No*
Yes
metrics
for each oometric + scmetric URL pair:
POST to code disharmonies service
Response Status
503* or 409
202
for each non-status-200 URL:
HEAD to code disharmonies service
All Status: 200
No*
Yes
disharmo
for each URL of every service:
DELETE to URL
cleanup
Finish
SPARQL Queries to services for
- commit dates
- changesets
- release dates
- class declarations
- ....
For each release:
SPARQL Queries to services for
- metrics
- hierarchy
- classes/methods
- disharmonies
Combine Data and store into collections:
- classes
- hierarchy
- changesets
- commits
- projectMetrics
synthesis
Analysis available in Web front-end
Black Arrow: uninterruptable flow
Red Arrow: interruptable - exit/resume
Blue Arrow: interruptable - exit/resume next element in loop
202* Asterisk (*): sleep before continuing

Comments:
- Each error 400 or 404 is counted as an error for that specific URL and the analysis will be retried. The maximum number of errors allowed can be configured in facets.ini
- Status 503 is simulated by cherrypy and means SOFAS was unreachable
- Status 409 is just a naming conflict which is resolved by cherrypy

Figure 7.2: Flow diagram illustrating the process of issuing analyses to SOFAS, retrieving the data, synthesizing it and cleaning up afterwards