University of Passau
Faculty of Computer Science and Mathematics
Chair of Software Product Lines

Master's Thesis

Visualization and Analysis of Product-Line Evolution in Codeface

Simon Niechzial

Matrikel Nr.: 48879

August 11, 2014

First reviewer: Dr.-Ing. Sven Apel
Second reviewer: Prof. Christian Lengauer, Ph.D.
Advisors: Claus Hunsen, Olaf Leßenich

Abstract

Software Product Lines (SPLs) are the next great change in the evolution of software engineering. Growing complexity, the need for more efficiency and better quality of software-products or products that contain software urge software engineers to find new approaches to reach these goals in real-world projects.

The need to collect, aggregate, **visualize** and utilize information on the architecture and implementation of a product line during product-line evolution arises. In this thesis we are going to concentrate on the **visualize** component. We will search for ways to collect data as a basis for visualization, transform the data into interactive displays of project state and evaluate the usefulness of different representations.

Our approach in the following work consists of six steps. First we show tools for visualization of software quality metrics and decide on a framework for implementing our SPLDASHBOARD. Then we develop a domain model for software product line metrics that constitutes the basis of the following software development. Afterwards we analyse existing tools for collecting data about software product lines. This includes a summary of product line specific software quality metrics and the challenges in testing SPLs or specific products, generated from SPLs. It also involves implementation of an import system for converting data from the collection tools into structures of our model. In the following step we implement a proof of concept for visualizing the data from the domain model, using the previously selected framework. Then we validate the results with data from a selection of open-source software projects. Finally we give an outlook on possible future work in this area.

The thesis results in a working prototype with different visualization options for SPL metrics. It includes a benchmark of the data model behaviour for different sizes of projects and number of projects in the system. Also the performance of the import process is analyzed and possibilities for improvement are outlined. The visualizations are validated against our set of user experience goals and evaluated by cross checking them against other datasources such as mailinglists, changelogs, bugtrackers.

We conclude that visualization and analysis of product-line evolution in the CODEFACE framework can be done and is feasible for real-world projects. While there is room for improvement of responsivenes and extension of visualization types, valuable data can be derived from the implemented prototype.

Contents

Glossary

codeface A framework for analysis and visualization of software project metrics.

cppstats A tool for analyzing software systems written in C regarding their variability.

git GIT is a distributed revision control and source code management (SCM) system with an emphasis on speed, data integrity, and support for distributed, non-linear workflows.

JavaScript A dynamic programming language that originated from scripting in web browsers.

nginx Pronounced “engine x”, is an HTTP and reverse proxy server.

nodejs JavaScript programming framework, mainly for development of webservices.

R Programming language, mainly for statistical analysis, derive from the S language.

rjsonio A package for the R language that supports conversion to and from data in JavaScript object notation (JSON) format.

shinyserver A webserver and web framework for the R language, partly implemented in JavaScript.

Software Product Line A software product line is a set of software systems, constructed from reusable parts according to a configuration.

spldashboard A framework for display and exploration of graphical representations of software product-line metrics, developed in this thesis.

WebSocket A protocol for bi-directional communication over TCP connections between web browsers and web servers.

Acronyms

API Application Programming Interface.

CLI Command Line Interface.
CPP C Preprocessor.
CSS Cascading Style Sheets.
CSV Comma Separated Value.

DBMS Database Management System.
DDD Domain Driven Design.
DOM Document Object Model.
DSL Domain-Specific Language.

FRP Functional Reactive Programming.

GUI Graphical User Interface.

HTML Hypertext Markup Language.
HTTP Hypertext Transfer Protocol.

JSON JavaScript Object Notation.

LOC Lines of Code.

PNG Portable Network Graphics.

RAM Random Access Memory.
RDBMS Relational Database Management System.
REST Representational State Transfer.

SPL Software Product Line.
SQL Structured Query Language.

TCP	Transmission Control Protocol.
UI	User Interface.
URL	Uniform Resource Locator.
VCS	Version Control System.
XML	Extensible Markup Language.
YAML	YAML Ain't Markup Language.

List of Figures

List of Tables

Listings

1 Introduction

Software Product Lines are the next great change in the evolution of software engineering. Coming from subroutines in the 1960s, modules in the 1970s, object orientation in the 1980s and components in the 1990s. Pervasive and increasingly complex software systems as well as the need for more economic software development calls for new concepts in the organization of software creation processes.

As their counterparts from the producing industry, SPLs improve productivity, time to market, cost and quality, if applied the right way using the right tools [8].

SPLs introduce variability at large scale. This leads to new challenges in all activities related to the software development process such as testing and other quality assurance steps. See chapter 4 for some examples.

Giving software project stakeholders (such as developers, product owners, managers) the ability to track the quality of whole product lines, detect hot-spots or foresee project issues is one of the high-level goals of the project that builds the context of this thesis. Also analysis of external product lines that are to be integrated in own products can be important.

Architectural analysis captures the "health" of the features of a system over time. In [24] the authors give examples for questions that project stakeholders may ask while improving software development and maintenance processes. Questions regarding architectural analysis may be

1. Is the software modularized (architecturally alligned) according to the features it implements?
2. How are features evolving and how is evolution taking place?
3. Are features becoming more complex? In which respect (coupling, size, scattering, etc.)?

In the last question we see, that answers can be given by aggregating existing code and process metrics on a per feature basis as well as by feature specific metrics like scattering, tangling coupling and cohesion (see table 4.1). Collecting and visualizing these metrics in a usable manner is the target of this thesis.

1.1 About this Thesis

The need to collect, aggregate, visualize and utilize metrics on the architecture and implementation of a product line during product-line evolution exists. This thesis describes the efforts in developing a working prototype application that enabels users to achieve these goals. The software development concentrates on the visualize component. Tools for collecting data as a basis for visualization do exist and will be described. The target is to transform the data into interactive displays of project state and evaluate the usefulness of different representations.

Our approach in the following work consists of six steps. First we show tools for visualization of software quality metrics and decide on a framework for implementing our SPLDASHBOARD in section 3.1. Then we develop a domain model for software product line metrics that constitutes the basis of the following software development in section 3.2.1. Afterwards we analyse existing tools for collecting data about software product lines in chapter 4. This includes a summary of product line specific software quality metrics and the challenges in testing SPLs or specific products, generated from SPLs. It also involves implementation of an import system for converting data from the collection tools into structures of our model. In the following step we implement a proof of concept for visualizing the data from the domain model, using the previously selected framework. Chapter 5 builds the implementation report. Then we validate the results with data from a selection of open-source software projects in section 5.3. Finally we give an outlook on possible future work in this area in chapter 6.

1.2 Conventions

In the following we give an overview of the typographic conventions used in this text.

Linebreaks Linebreaks in sourcecode listings, that have no technical reason but have been introduced because of space limitations, are marked with a leading $\hookrightarrow$ in subsequent lines.

Products Product and project names are set in SMALL CAPITALS.

Source Code Source code chunks that appear in the text are set in a `teletype font family`.

Source Code References References to explicit lines in source code listings are written as (line) after the reference.

2 Context and Framework

2.1 Software Product Lines

The tools described and developed in this thesis cover Software Product Lines in general and feature-oriented software product lines in particular. The concept of software product lines exists since the 1960s. The basic ideas are software reuse and mass customization. The former covering the left hand side of the software production process, where software is constructed from reusable and combinable components instead of building everything from scratch for each project. The latter taking the other sides perspective, where customers can select from a large set of configuration options to make the product match their requirements.

A large scale example of a software product line is the linux kernel with a set of up to 10.000 configurable features. The linux kernel uses preprocessor techniques (the C preprocessor CPP in particular) to implement compile time feature selection. This is only one of many approaches for implementing variability and will be discussed later in this thesis.

While there are initial costs of setting up a product line approach or transitioning a set of products into a product line, at some point the benefit of reduced costs for generation of new products (based on the existing artifacts) outweighs that. So one of the promises of software product line development is reducing costs. Other benefits are made-to-order products that are of much greater use for the customer than standardized or preconfigured software. Also the overall product quality should improve, as reused parts can be tested more thoroughly and in different contexts. Assembling software products from existing parts can even reduce time to market significantly. All these benefits are not for free. Besides the mentioned initial investment in setup of a product line, development of multiple (sometimes theoretical) products amplifies complexity. The configuration space grows exponentially with the number of configuration options. This leads to new challenges regarding testing and general quality asessment as the number of possible products is orders of magnitude to large for simple “test them all” approaches (for an example calculation see the introduction in chapter 4) [2].

2.2 The PYTHIA Project

The PYTHIA project is about "Analysis Techniques and Prediction Models for Sustainable Product-Line Engineering" [3, 4].

The Applicants of the research grant proposal suggest to introduce new analysis techniques for sofware product lines based on implementation knowledge (**software metrics**, static analysis, **mining techniques**, measurement of non-functional properties and **feature-interaction analysis**). The higher level goal is to provide more precise prediction models that consider **structural** and behavioral properties of the architecture and **implementation assets**. These prediction models can then be used to foresee requirements regarding new features and feature development.

In [24] a dashboard is suggested to show aggregated views on different measures over features. These views should aid strategic decision making in software development processes. The authors describe the dashboard by a mockup that is shown in figure 2.1.

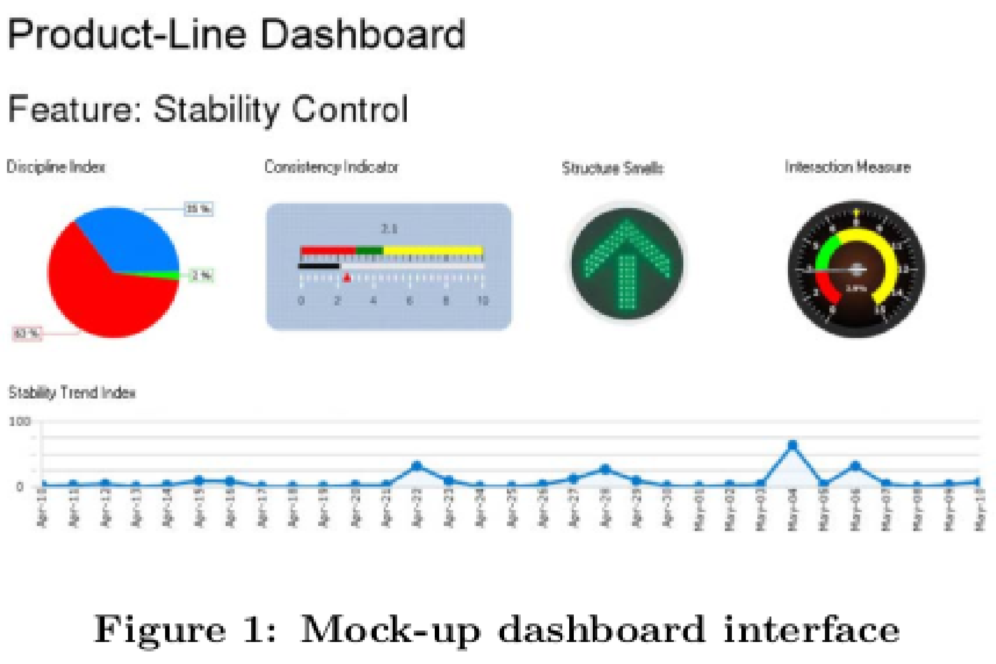


Figure 2.1: Mockup of a dashboard showing feature metrics

Such a dashboard will be developed in this thesis and will be referenced as the SPLDASHBOARD in the following text.

3 The Codeface Framework

CODEFACE[1] is a framework and interactive web frontend for the social and technical analysis of software development projects. In its current state it captures various data sources (revision control systems, bugtracking systems, mailing lists and complexity metrics). This chapter describes the codeface components and their roles as well as the extensions necessary to support analysis of SPLs.

3.1 Codeface: Existing Structures

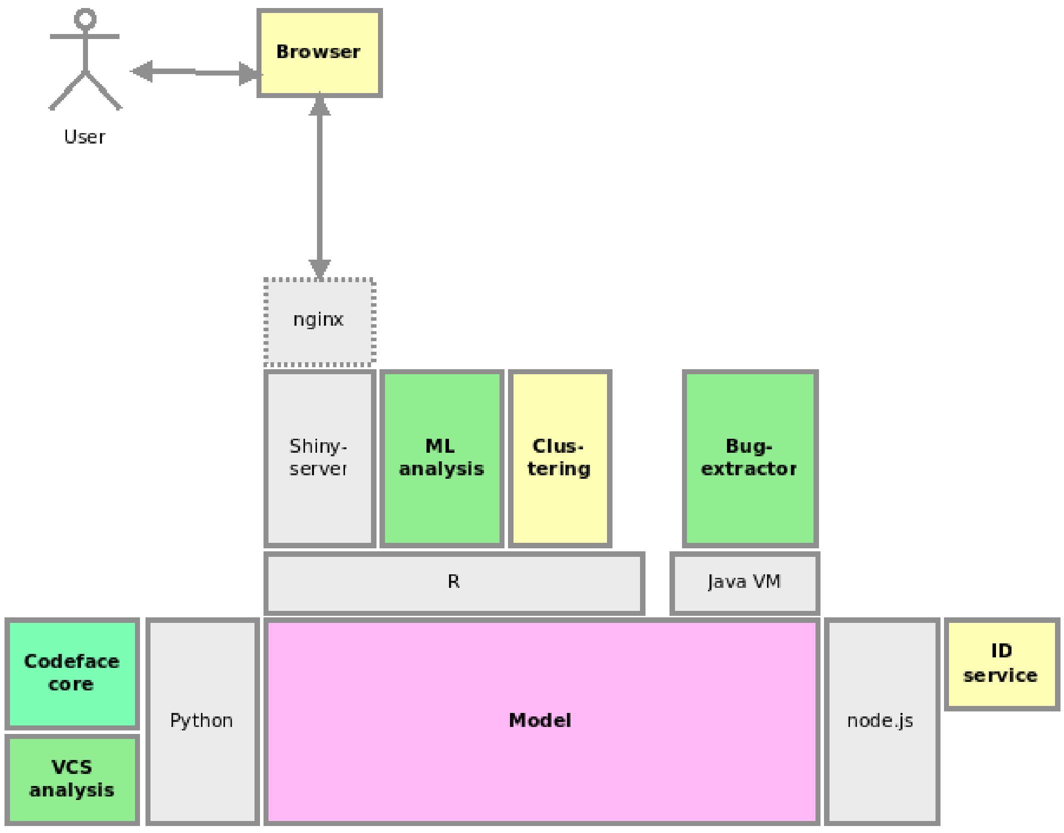


Figure 3.1: CODEFACE architecture overview

[1] `http://siemens.github.io/codeface/` (Retrieved: 08.08.2014)

Codeface is a heterogeneous system, consisting of many modules. The modules are written in different languages (R, python, Java, JavaScript) using different frameworks (shinyserver, nodejs). Communication between modules is also done by different means: REST-APIs, shared model in a Relational Database Management System (RDBMS) or config files. Codeface core consists of 56.2% R, 30.4% Python, 6% Java and 5.9% JavaScript code.

The Graphical User Interface (GUI) of codeface is realized as a web application that can be displayed in any modern browser. It uses websockets to communicate asynchronously with the server component. This avoids reloading of whole html documents. Instead only fragments are loaded and inserted or replaced in the current Document Object Model (DOM). That causes the browser to re-render only the affected parts of the website — which is a Hypertext Markup Language (HTML) document — giving the user a more desktop-application like experience.

An optional reverse-proxy-cache (nginx) caches distinct resources on the application layer (ISO/OSI model) based on their Hypertext Transfer Protocol (HTTP) headers. These resources are identified by their Uniform Resource Locator (URL) with query parameters. The cache avoids expensive recomputation (e.g. charts rendered as images) and delivers static resources, such as images, fonts and stylesheets.

The following sections discuss the distint modules which are also shown in figure 3.1.

3.1.1 MySQL Server / Model

mysql is an open-source RDBMS developed by Oracle. In codeface it serves as the repository for all analysis results, project meta-data and partly as a cache of binary graph image data.

The schema of the central SQL database is maintained in a binary format (MWB) that is produced by the mysql Workbench, a grapical schema editor for mysql databases, also maintained by Oracle[2]. For a representation of the original codeface data model see figure A.1.

The workflow for making schema changes is as follows:

1. Load the schema with mysql Workbench (version >= 6.0)
2. Make changes
3. Export by selecting `File` → `Export` → `Forward Engineer SQL CREATE script`
4. Activate the following settings to assure that code-generation does produces consistent code

[2] `http://www.mysql.com/products/workbench/` (Retrieved: 08.08.2014)

- Generate DROP statement before each CREATE statement
- Generate DROP schema
- Generate separate CREATE INDEX statements

5. Click `Next`, and choose to export table, view and routine objects
6. Click `Next`, save the generated script, and create a version control system commit with both the binary and SQL creation script changes.

A version >= 6.0 of the Workbench is required, because the code-generation procedures changed a lot between 5.x and 6.x branches and it would make change tracking through diff views difficult.

Listing 3.1 shows how to create an empty database (1) named `codeface` and assign privileges to a user of the same name (3). Listing 3.2 shows the command for importing the generated schema sql into the database (1) with the optional commands to rename the database from `codeface` to a new name by simply replacing strings (3-5).

Listing 3.1: CODEFACE database creation and permission assignment

```
1 CREATE DATABASE `codeface`;
2
3 GRANT ALL ON `codeface`.* TO 'codeface'@'localhost'
     ↪ IDENTIFIED BY 'secretpassword';
```

Listing 3.2: Importing (and optionally renaming) the CODEFACE schema

```
1 mysql -u codeface -psecretpassword < ./datamodel/codeface
     ↪ _schema.sql
2
3 cat ./datamodel/codeface_schema.sql | \
4 sed -e 's/codeface/NEW_NAME/g;' | \
5 mysql -ucodeface -pcodeface
```

3.1.2 Codeface Core

This is the user facing part of the framework, apart from the actual GUI, implemented in Python. It offers a Command Line Interface (CLI) for controlling the various import and preprocessing tasks in CODEFACE. The CLI is extensible through a command pattern. Commands can be implemented as python functions and bound to arguments that are passed to the CLI call. The framework handles parsing of configuration files in the YAML Ain't Markup Language (YAML) format that are passed as an argument to CLI invocations. The parsed configuration files can be accessed through an array in the command functions. The tasks of this module can be summarized as:

- Configuration file parsing
- CLI framework
- Job control for long running tasks

3.1.3 VCS Analysis

This module extracts various informations on a software project from a Version Control System (VCS), such as developer activity, comment discipline and collaboration. Implementations exist for GIT, but interfaces exist for extensibility. As the core, it is implemented in python.

3.1.4 shinyserver

SHINYSERVER[3] is a web framework for the R language. Its role in the whole CODEFACE structure is providing of the GUI in form of grids of widgets. These grids are referenced as topics. Their main use is display of interactive information about analyzed software projects.

SHINYSERVER uses a principle called Functional Reactive Programming (FRP) for the interaction between the view (user interface) and the controller. View-controller interaction is usually implemented by some observer design pattern and we are going to show how closely the two concepts are related. See also figures 5.2 and 5.3 for a comparison of the MCV pattern with the codeface components.

In FRP a variable is not bound to the value of the evaluated right-hand side but to the expression itself. Hence a variable does not — at a specific point in time — represent one value but a continuous stream of values (or events). So we have an object emitting a stream of events — an observable. Using this stream in another expression (by using the variables name) makes the expression a listener on the stream of events. Every time the original stream emits an event, the expression is evaluated and may cause side-effects — this matches observers listening to an observable.

Reactive Sources, Conductors and Endpoints

The SHINYSERVER Application Programming Interface (API) provides two R data-frames (`input`, `output`) with appropriate member variables.

Reactive endpoints in their simplest form (members of `output`) correspond to User Interface (UI) components defined in a specific file (ui.R). For example a `plotOutput(''distPlot'')` will be bound to `output$distPlot` (where `$` is the shorthand for data-frame member access). It is noteworthy that any named

[3] `http://shiny.rstudio.com/` (Retrieved: 08.08.2014)

UI element will appear as a DOM node in the front-end HTML code with its `id` attribute set to the UI elements name (e.g. `<div id='distPlot'...`).

Reactive sources (members of `input`) correspond to the streams of input values provided by users of the application trough the interaction with UI elements (e.g. moving a slider). For example a `sliderInput(''foo'',[...])` will be bound to `input$foo`.

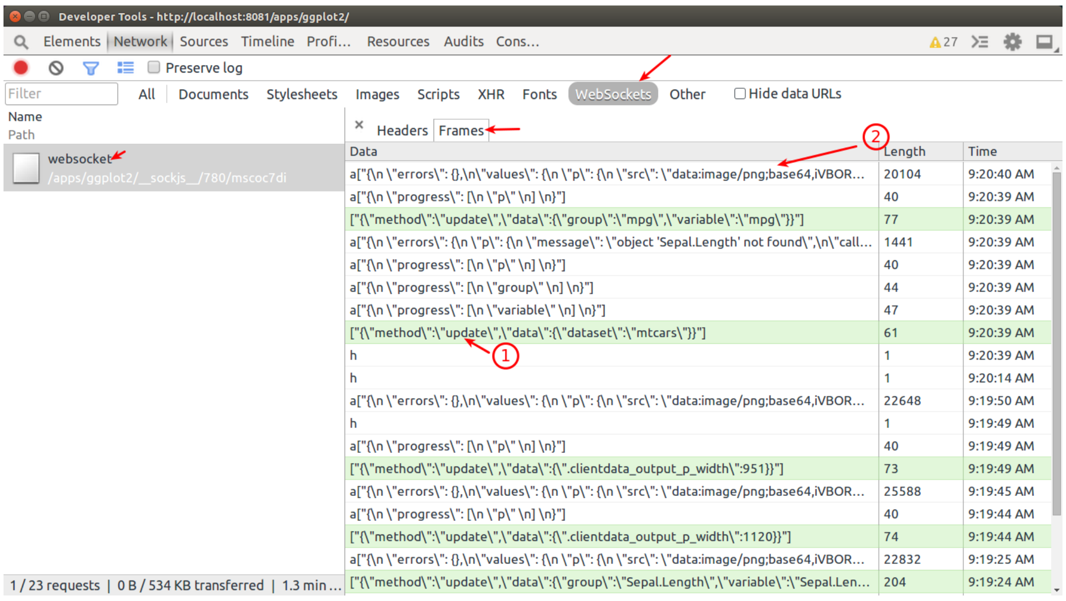
If we think of the relationship between the three reactive elements (sources, conductors, endpoints) as a graph, sources can only be parent-nodes (they are not dependent) while endpoints can only be children. What we do not have yet are inner nodes that can act as both — these are called conductors. Conductors encapsulate reactive behavior and can act as junctions, multiplexing the event stream of a source to multiple endpoints while optionally applying operations on that stream. An example for a conductor is shown in listing 3.3 (4) where it is used to multiplex the result of an expensive computation `fib()` to two endpoints. A conductor is created by passing a block of code to the `reactive()` function.

Listing 3.3: Reactive Conductor Example

```
1 fib <- function(n) ifelse(n<3, 1, fib(n-1)+fib(n-2))
2
3 shinyServer(function(input, output) {
4   currentFib         <- reactive({ fib(as.numeric(input$n)) })
5
6   output$nthValue    <- renderText({ currentFib() })
7   output$nthValueInv <- renderText({ 1 / currentFib() })
8 })
```

Binding reactive sources in R to the event handlers in the browser (JavaScript), serializing the values, transmitting them over Websockets forth and back, deserializing and updating the DOM elements in the HTML document when reactive endpoints change, is all done by the ShinyServer framework. Events and changes are communicated by a WebSocket connection [13].

The following steps describe a communication between client and server (or view and controller) that handles one interaction of the user with a GUI component and the subsequent update of the view. In figure 3.3 the corresponding GUI is shown while figure 3.2 displays a debug trace of the websocket communication.



Data	Length	Time
a["{\n \"errors\": {},\n\"values\": {\n \"p\": {\n \"src\": \"data:image/png;base64,iVBOR...	20104	9:20:40 AM
a["{\n \"progress\": [\n \"p\" \n] \n}"]	40	9:20:39 AM
["{\"method\":\"update\",\"data\":{\"group\":\"mpg\",\"variable\":\"mpg\"}}"]	77	9:20:39 AM
a["{\n \"errors\": {\n \"p\": {\n \"message\": \"object 'Sepal.Length' not found\",\n\"call...	1441	9:20:39 AM
a["{\n \"progress\": [\n \"p\" \n] \n}"]	40	9:20:39 AM
a["{\n \"progress\": [\n \"group\" \n] \n}"]	44	9:20:39 AM
a["{\n \"progress\": [\n \"variable\" \n] \n}"]	47	9:20:39 AM
["{\"method\":\"update\",\"data\":{\"dataset\":\"mtcars\"}}"]	61	9:20:39 AM
h	1	9:20:39 AM
h	1	9:20:14 AM
a["{\n \"errors\": {},\n\"values\": {\n \"p\": {\n \"src\": \"data:image/png;base64,iVBOR...	22648	9:19:50 AM
h	1	9:19:49 AM
a["{\n \"progress\": [\n \"p\" \n] \n}"]	40	9:19:49 AM
["{\"method\":\"update\",\"data\":{\".clientdata_output_p_width\":951}}"]	73	9:19:49 AM
a["{\n \"errors\": {},\n\"values\": {\n \"p\": {\n \"src\": \"data:image/png;base64,iVBOR...	25588	9:19:45 AM
a["{\n \"progress\": [\n \"p\" \n] \n}"]	40	9:19:44 AM
["{\"method\":\"update\",\"data\":{\".clientdata_output_p_width\":1120}}"]	74	9:19:44 AM
a["{\n \"errors\": {},\n\"values\": {\n \"p\": {\n \"src\": \"data:image/png;base64,iVBOR...	22832	9:19:25 AM
["{\"method\":\"update\",\"data\":{\"group\":\"Sepal.Length\",\"variable\":\"Sepal.Len...	204	9:19:24 AM

```
1 / 23 requests | 0 B / 534 KB transferred | 1.3 min ...
```

Figure 3.2: Trace of the websocket communication

1. ③ Selecting an item from a dropdown menu
2. ① Change event is handled by the SHINYSERVER JavaScript component and then sent via an existing WebSocket connection (HTTP / Transmission Control Protocol (TCP)) to the server component
3. After server-side processing the changes to reactive endpoints are communicated back ② to the client, as shown in listing 3.5 (5)
4. DOM elements are updated by the SHINYSERVER JavaScript component ④

Listing 3.4: UI component

```r
#input
sidebarPanel (
  selectInput("dataset", "Data:",
    list(iris = "iris", mtcars = "mtcars")
  ),[...]
)
```

Listing 3.5: Server Component

```r
# shiny server side code for each call
shinyServer(function(input, output, session){
  #update variable and group based on dataset
  output$variable <- renderUI({
    obj<-switch(input$dataset,
           "iris" = iris,
```

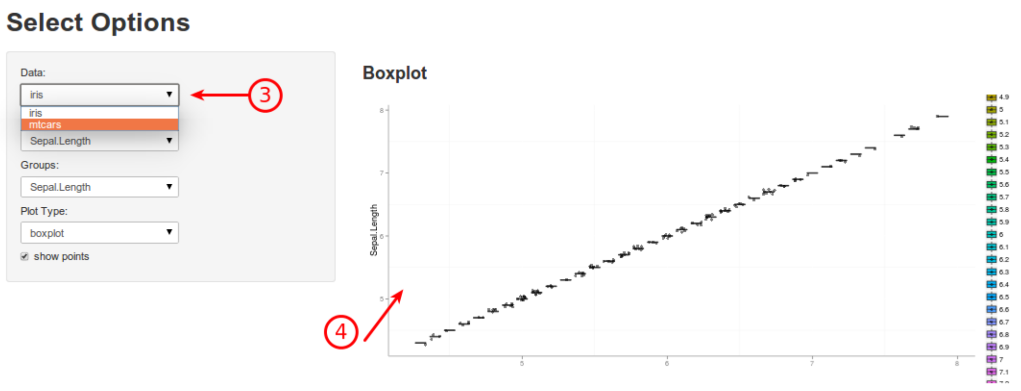


Figure 3.3: GUI interaction related to the websocket trace

```
                  "mtcars" = mtcars)
        var.opts<-namel(colnames(obj))
        selectInput("variable","Variable:", var.opts) # update UI
        })
        [...]
})
```

3.1.5 ML Analysis

This module processes posts in mailinglists given in mbox files. It extracts data about project communication and collaboration. The module is written in R and uses the ID service for mapping of persons identities.

3.1.6 Clustering

This module implements custering of developers along different categories such as activity or projects. It is implemented in R.

3.1.7 Bug Extractor

The bug extractor module fetches information from bug tracking software about the bug reports. It connects to APIs of different tools or uses screen scraping on their web interfaces. Lookup of identities is again done via the ID service.

Currently implementations for Jira[4] and Bugzilla[5] exist.

[4] `https://www.atlassian.com/software/jira` (Retrieved: 08.08.2014)
[5] `http://www.bugzilla.org/` (Retrieved: 08.08.2014)

3.1.8 ID Service

The ID service manages unique identities (persons) and offers a lookup-or-insert API to the other crawling and gathering services described before. It is implemented in JavaScript (NODEJS). If another service requests a person that is not yet present in the database, it is created and its unique ID is returned. Otherwise the ID of the existing person is returned. Lookup can be performed by e-mail address and name.

3.1.9 nginx Cache

nginx[6] is an application-layer cache for the HTTP protocol. It is not enabled by default nor is it packaged with CODEFACE but a configuration file is part of the framework. The cache makes sense for larger installations with a high number of clients, where it offloads the delivery of static resources (e.g. images, fonts, css files) from the SHINYSERVER.

3.2 Codeface: Extensions

In the following, we describe the necessary enhancements that had to be made in the codeface framework in form of an implementation report. We describe the software engineering approach that has been used for designing these changes and give reason for each change.

The central entity of the CODEFACE model is a commit. As our software product-line metrics are aggregated around file-instances, releases and projects, we need to extend the existing data model. This is described in section 3.2.2.

Some of the components of CODEFACE had to be extended to support importing of software product-line metrics that have been generated by CPPSTATS. These modifications are described in 3.2.4.

For planing and implementing the extensions, concepts from Domain Driven Design (DDD) [9] have been used. A core strategy of DDD is the ubiquitous language, it describes a common set of vocabulary used by a team building a software. The goal is to use the same consistent wording for concepts of the domain throughout the project (code, documentation, etc.). While the number of stakeholders during the development of the SPLDASHBOARD has been low, the ubiquitous language also leads to a clear and descriptive terminology for use in this thesis as well as following work or discussions. We developed and evolved this vocabulary before and during development, its documentation can be found in section 3.2.3.

When building a domain model with DDD, three concepts are used: entities, value objects and services.

[6] `http://nginx.org/` (Retrieved: 08.08.2014)

Entities Also called reference objects, they describe objects that are primarily defined by their thread of identity that runs through time and not by their attributes. An entity is distinguishable from another entity even if their attributes are the same. Although objects in languages like Java match this description, because they have built-in identity functionality in all objects (e.g. an understanding of equality as in `Object.equals()`), they should not be mixed up.

Value Objects These are objects that describe some characteristic of a thing are called value objects. In contrast to entities they have no conceptual identity. While tracking the identity of entities is essential, it is a large overhead for objects that do not need this concept. A value object can give information about an entity and should be conceptually whole according to the Whole Value Pattern[7].

Services Services are processes or transformations in the domain that are not a natural responsibility of an entity or value object. Such an operation is added as an interface to the model declared as a service. The interface must be defined in terms of the language of the domain model and the operations name must be part of the ubiquitous language. A service is stateless.

3.2.1 Domain Model

The domain model developed for the SPLDASHBOARD is shown in figure 3.4. Variables are the concrete value of a metric for a specific project, release or file instance (depending on the scope of the metric). Projects, releases and file instances are all ScopeInstances and can therefore serve as scope for a variable. Projects are identified by their ID. A release is identified by its tag (e.g. a string, naming the release that may correspond to a tag in the versioning system). A FileInstance is identified by the triple of path, release name and project ID $(path, release.name, project.id)$. Scopes represent the type of ScopeInstances and appear in Metric objects to define their level of aggregation (or scoping).

3.2.2 Data Model

As said before, the central entity of the CODEFACE model is a commit. Commits are authored by a person and are aggregated in release-ranges. The data we gather for our software product-line analysis is more coarse grained. The deepest level of detail is information per file-instance. Where file-instance stands for the state of a file in a specific release. We then have aggreated values per release and per project.

[7] `http://fit.c2.com/wiki.cgi?WholeValue` (Retrieved: 08.08.2014)

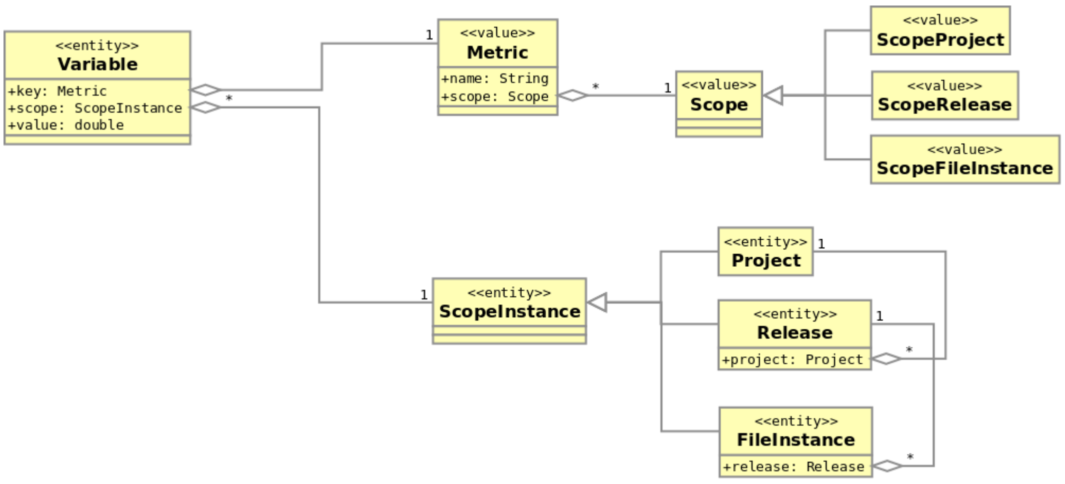


Figure 3.4: Domain model of the extensions to codeface

Files have no representation in the existing codeface model and need to be part of our extension. Releases match the release_timeline entity and projects are already represented by the project entity. Metrics, though beeing value objects, are going to have a representation in the database for easier extensibility.

In the next sections database migrations as a way of changes to the model described in a traceable and reversible manner will be explained. Also the mapping of the domain model to a relational database model and the required tools is described.

Database Schema Migrations

The concept of database migrations will be used in the following sections. Migrations are a way do describe changes to an ER model in a traceable and reversible manner. Migrations are pairs of executable SQL scripts where the first part introduces a change and the second part reverts it. If the change is destructive and cannot be reverted (e.g. a DROP of a non-empty column) this should be documented in the migration.

We used the concept of Domain Driven Design (Eric J. Evans) to create a domain-model for the process of variability analysis in software product-lines. We focused on the data collection process used by CPPSTATS. Based on the Domain Model we created a data model as an entity relationship model (ER).

The implementation of the ER model was done by graphically designing it in Oracle DbDesigner and using code generation to create SQL code compatible with the MYSQL RDBMS. This approach was required by the CODEFACE framework project.

As a documentation of our changes and as a tool for extending existing installations of CODEFACE (with an already filled database, where it is not possible to drop and reinstantiate the whole schema) we created additional migration scripts for each change.

Mapping the Domain Model to a Relational Database Schema

In this step we will derive a relational database schema from our domain model. The complete extended schema is shown in figure A.2 and the relevant entities are extracted for better visibility in figure 3.5.

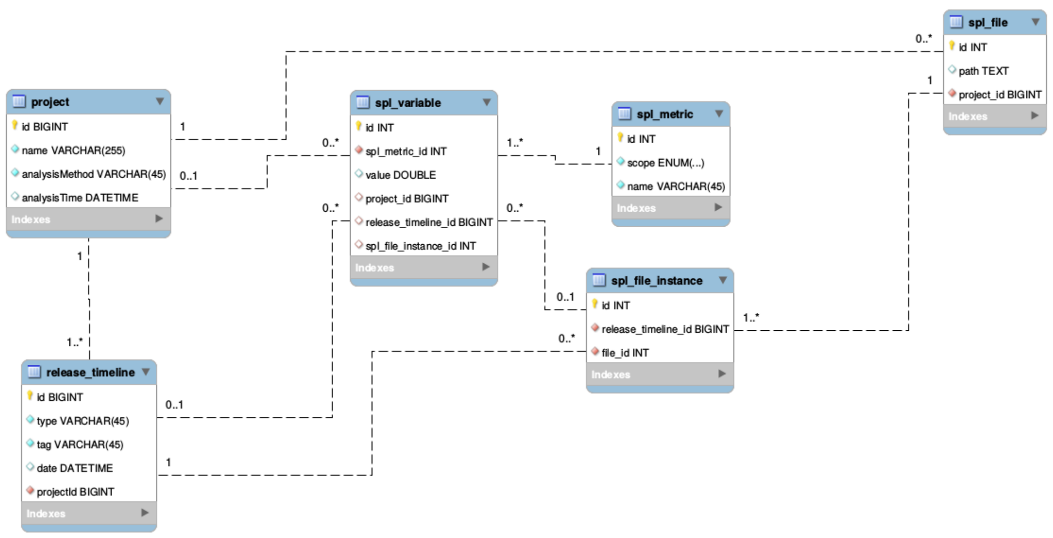


Figure 3.5: SPLDASHBOARD database schema

The type Variable and the ScopeInstance subtypes Project, Release and FileInstance are entities and as such map naturally to entities of an entity-relationship (ER) model. Project and Release actually already exist in the schema of CODEFACE. While Project has the same name there, Release maps to the release_timeline entity. Other names have been converted to match the SQL naming conventions: no camelcase but lowercase and underscores. The scope attribute of spl_variable is implicitly given throug the foreign keys on the respective ScopeInstances. If a foreign key is set, the variable is in the respective scope.

The value object Metric has been implemented as an entity because the set of possible metrics is likely to grow over time and needs to be configurable. This way, adding new metrics can be done by a simple SQL INSERT statement.

The value object Scope has been implemented as an ENUM type attribute of the metric entity. The ENUM holds values for the possible Scope types. Adding scopes would involve changing the column type of the scope attribute, whithout destroying existing data. The Scope type is also expressed as class constants in the `codeface.spl.DatabaseImporter` python class.

Unique ID generation for primary keys is implemented by the auto-increment attribute for columns, which is an extension of standard SQL that exists in MYSQL. It asserts that the column is filled with an incrementing integer value, independent of the value passed in an `INSERT` query. All above mentioned entities have a primary key with the name `id` that carries this attribute.

Example queries can be found in appendix B from listing B.2 until B.13. The full set of retrieval queries is located at `software/codeface-fork/spl/metrics_retrieval.sql` on the DVD. The concrete implementations for the dashboard widgets is located at `software/codeface-fork/codeface/R/shiny/spl/db.queries.r` on the DVD.

Benchmarking the Model

Table 3.1 shows the number of entity instances created per project per entity type. It also shows the size of the resulting database including indices and MYSQL management overhead. The query for database size retrieval is shown in listing 3.6.

The variable instances with a file scope dominate regarding entity count and database size. The number of releases multiplicates the variable instances per file, as most of the files exist through most of the releases of a project.

Listing 3.6: MYSQL database size retrieval

```
SELECT
  t.table_schema "dbname",
  SUM( t.data_length + t.index_length ) / 1024 / 1024 "
    ↪ size"
FROM information_schema.TABLES t
WHERE t.table_schema = 'codeface'
GROUP BY t.table_schema;
```

3.2.3 Ubiquitous Language

The ubiquitous language provides a set of clearly defined terms that is used throughout the whole project, its mode, the documentation and source code. The following list defines these terms for the SPLDASHBOARD project.

Project A project refers to a software project identified by its unique name. We assume knowledge about which releases exist for this project and that metrics have been gathered for this project. The assets of a project exist as files in a specific folder.

Release A release identifies the state of all assets of a project at a certain point in time. Usually a release is marked as a named tag or commit identifier in a version control system. Synonyms are revision and version.

Project	spl_variable	spl_file_instance	spl_file	Releases	Database Size (MiB)
apache	516011	39573	872	142	108
berkeleydb	130135	9922	2425	21	16.2
busybox	771164	59537	918	89	81
cherokee	488959	37928	663	139	54.9
emacs	127822	98130	863	23	16.1
freebsd	4610026	354603	30597	17	449.3
gimp	997083	76675	3174	28	100.1
gnuplot	42678	3266	279	20	8.44
libxml2	156951	11975	195	116	20.19
linux	8109018	623433	47662	22	753.15
mpsolve	1946	148	148	2	2.02
openvpn	17532	1341	334	9	5.58
parrot	56908	4364	497	16	10.67
postgresql	3573380	274685	3331	225	349.5
sendmail	107694	8241	303	51	14.91
sqlite	74718	5723	224	29	11.73
subversion	806567	61970	1559	87	83
vim	23195	1769	250	18	5.78
xfig	10897	834	200	5	2.91
xterm	56922	4316	62	74	10.61

Table 3.1: Number of entity instances created per project. Database size includes indices.

A release can be captured by exporting all files in the state identified by the release to a folder. The unique (in the project context) identifier of a release is its tag.

File Files are identified by a path relative to the project directory but do not refer to contents directly, as file contents can have many states and files do not have to exist at all points in the lifetime of a project.

File Instance The state of a file at a certain point in the lifteime of a project. A file that exists (e.g. in a release) is represented as a file instance.

Metric A metric is a standardized measurement. In this model, metrics describe characteristics of the source code of a software project, in particular the characteristics of use of variability techniques. For now we gather data about annotations that control feature selection (see 4.1).

Metrics Scope A metric can apply to different entities, these entities are called metric scopes. Valid scopes in this model are file, release and project.

Scope Instance An instance of a valid metrics scope (e.g. file, release, project) is a scope instance.

Variable The occurence of a metric on a scope instance results in a value. This triple of $(scopeInstance, metric, value)$ is called variable.

3.2.4 Framework Extensions

In figure 3.6 we show the target architecture that should be reached after the extensions are implemented. Extension points are the import process where — besides the existing data collection — data from CPPSTATS must be imported. This is described in the following sections. Also the database or more precisely its schema needs to be enhanced (see 3.2.2). The last extension point is the dashboard and its server core where the collected metrics need to be preprocessed for visualization and widgets for data display need to be developed, this is described in 5.1.

Extending the Project Import Process

The CODEFACE core already provides the components for a CLI (see 3.1.2). Extensions have been written to support reading and parsing of Comma Separated Value (CSV) files that are delivered by the analysis tools.

The path names used by CPPSTATS required some postprocessing. The tool identifies files by a absolut path to an internal XML file that looks like the following `/projects/apache/httpd-2.4.6/_cppstats/server/util_expr_parse.h.xml`. We make the assumption that the `_cppstats` directory will always be present, to extract the real files name with a regular expression: `'_cppstats/(.*)\.xml$'`.

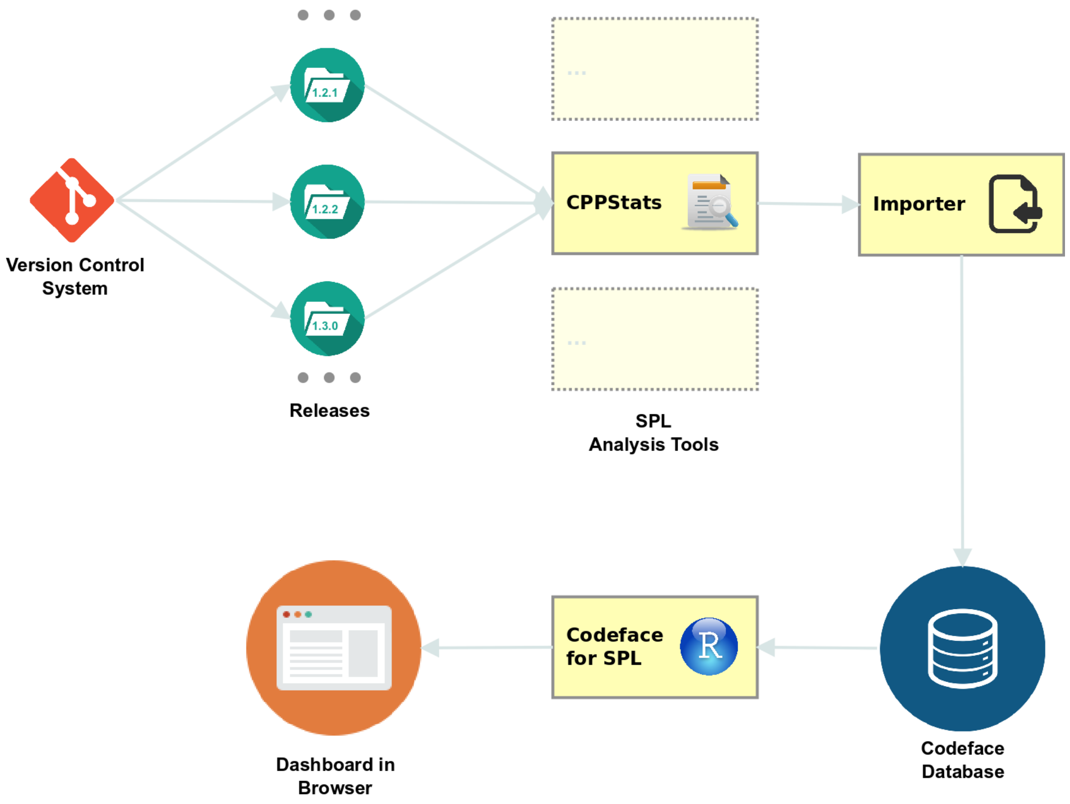


Figure 3.6: CODEFACE target architecture after extensions have been implemented

The import process can take a considerable amount of time. To improve import speed, caching has been implemented for results of get-or-create queries. These are queries that do a primary key lookup based on a set of attributes and perform an insert if no entity with that set of attributes exist. The cache uses Python dictionaries[8].

Extending the Configuration File Format

Configuration files are written in the YAML format[9] that takes concepts form programming languages (objects, lists, etc.), Extensible Markup Language (XML) and the e-mail specification. Parsers are available for all languages used in CODEFACE.

New configuration keys have been introduced for Software Product Line analysis. The `spl.cppstats` key contains a map with the following pairs:

[8] `https://docs.python.org/2/tutorial/datastructures.html#dictionaries` (Retrieved: 08.08.2014)

[9] `http://yaml.org/spec/` (Retrieved: 08.08.2014)

basePath the absolute path to the directory containing CPPSTATS output data in csv format,

revisionPattern a Python `str.format()` compatible pattern[10], for one argument of type string that builds a folder name from a given revision name. E.g. it performs a mapping from 2.4.4 → `linux-2.4.4`

perfile The name of the file containing per file metrics

perrevision The name of the file containing per revision metrics.

The complete path will be built by appending basePath, revisionPattern applied on the current revision and either perfile or perrevision.

The existing configuration key `revisions` is used to determine which revisions should be analyzed and where to find them. We provide a script that builds the revisions array given a CPPSTATS output directory. The script is located at `spl/revisionListFromCppstats.sh`. A minimum viable configuration file is shown in listing B.1.

Benchmarking the Import Process

Table 3.2 shows the import process runtime per project. Runtimes are shown with and without caching. The last column shows the improvement through caching. For an overview of the amount of data generated (in terms of generated entity instances) see table 3.1 in the benchmark subsection of 3.2.2.

The benchmark was run on a Intel Core i7-3770 4-core at 3.4 GHz with 16 GiB of RAM and an Intel 520 Series SSD. The numbers are the average values of 10 runs for each case. The benchmark control script is provided on the DVD at `/git/codeface-fork/spl/benchmark`. The command used for benchmarking is

```
/usr/bin/time -f "user:%U system:%S total:%e cpu:%P"python
↪ -m codeface.cli -l info setup -p conf/benchmark/PROJECT.conf
```
.

Open Tasks

In a later version it should be possible to run CPPSTATS directly from the import process as an option instead of just importing the CSV files. The CLI framework provides a help system. Explanations on the Software Product Line specific commands should be integrated.

[10] `https://docs.python.org/2/library/string.html#format-string-syntax` (Retrieved: 08.08.2014)

Project	without caching (s)	with caching (s)	improvement (%)	lps
apache	430.77	295.13	31.5	135.5
berkeleydb	121.17	77.72	35.9	128.9
busybox	684.58	444.24	35.1	133.9
cherokee	413.08	281.98	31.7	134.3
emacs	113.38	74.46	34.3	132.7
freebsd	10345.83	3045.41	70.6	116.5
gimp	946.34	577.41	39.0	132.9
gnuplot	34.98	25.46	27.2	130.6
libxml2	130.15	91.26	29.9	135
linux	25264.29	5558.68	78.0	112.2
mpsolve	2.09	1.59	23.9	96.9
openvpn	15.11	10.83	28.3	126.3
parrot	49.68	33.56	32.5	131.5
postgresql	2887.35	2020.95	30.0	136.3
sendmail	84.86	61.52	27.5	136.4
sqlite	61.86	43.42	29.8	133.8
subversion	668.64	456.18	31.8	136.4
vim	19.3	13.79	28.6	132.2
xfig	9.15	6.73	26.5	161.5
xterm	45.57	33.1	27.4	137.1

Table 3.2: Runtime comparison of the import process with and without caching for all projects. Also showing the CSV lines parsed per second (lps).

4 Datasources

Variable software or software product lines have general implications on measurability. As an example SQLite: Embedded, configurable Database Management System (DBMS) with 88 compile-time options. Assumed there are no dependencies between the options we get 2^{88} variants. Further assumed and an average benchmark time of 5 minutes per variant results in $2^{88} * 5/60/24/365 = 2944111585058457655296$ (two sextillion) years for each test-run [5].

Hence it is no option to build and test each product that can be generated based on a given variable implementation. There exist different approaches on tackling this complexity, for example variability aware parsing [17]. Variability can be realized in many different ways, some of the available techniques are the following.

- Runtime Variability
- Version Control or Build System Variability
- Preprocessor Variability
- Feature Oriented Programming

We will focus on preprocessor based variability, particularly C Preprocessor (CPP) annotation based variability (see 4.1). Preprocessing is not limited to the C language. Different languages with builtin preprocessors exist, such as C++, Fortran or Erlang. For other languages like C#, VisualBasic or D there are external tools available. Scala has had external support for compile time metaprogramming (preprocessing) with Kepler, that has been integrated into the official Scala Compiler since version 2.10.0. Even for Java there is support with Antenna, Munge and others.

Figure 4.1 shows the generic workflow of generating variants from reusable artifacts and a feature selection. The feature names from the feature model map to constant names in the CPP language. A feature selection is given by a set of value assignments to these constant names (or more precisely, by defining or not defining them). The generator is CPP itself, running before each compilation step.

Preprocessor usage can reduce the binary footprint of a program because it uses a tag and prune approach [14]. This can be important for systems with strict memory limitations, such as embedded systems. But benefits come at a

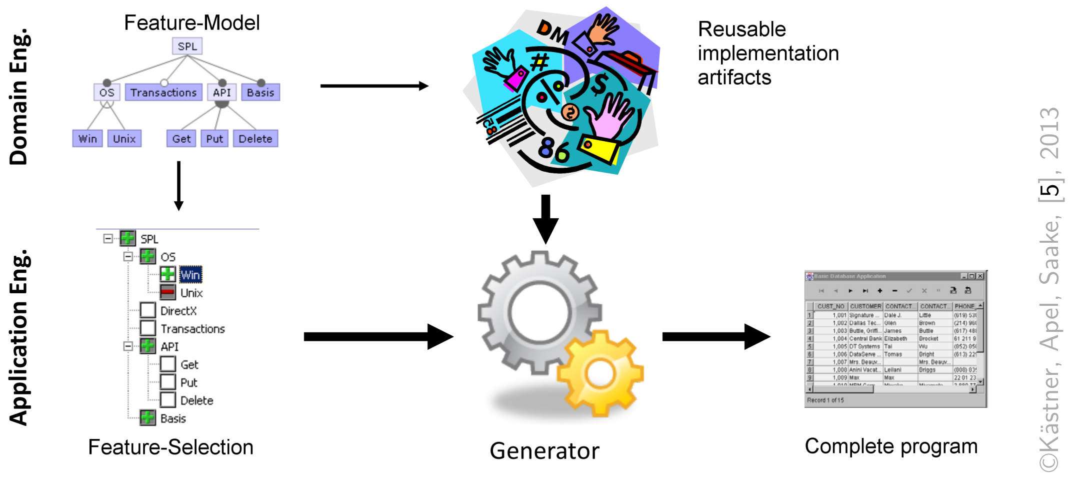


Figure 4.1: Workflow for variant generation from feature-selection and implementation artifacts

cost: readability of source code suffers. Deep nesting of annotations introduces complexity and makes reasoning about the source code a hard task. Complex annotations are error prone, because the simple string based deletions may lead to invalid syntax. Mapping of error messages to the orginial source code may be impossible. Missing tool support and undisciplined usage also lead to code that is not easily understood.

An inherent problem of the CPP apporach to variability is the scattered feature code which is a result of tagging the feature related fragments in place instead of aligning the structure of the codebase to the feature model. See the SDEG metric in section 4.1.3.

4.1 CPPStats

CPPSTATS looks at variability from the C programming languages perspective, using CPP capabilities for deriving products from a common set of C implementation artifacts. This is a global, omniscient view on variability and analyses the quantity, granularity, relationship and interaction of/between features.

CPPSTATS gathers the metrics listed in table 4.1. The metrics can be used to analyse feature-to-code mappings as described in [2, 254ff.], that is the mapping from problem-space (feature model, feature selections) into solution space (metrics SDEG, TDEG and NOFPFC). They also give information about the complexity of feature interaction (AND metric) and the types of feature boundaries (GRAN metric). The NOFC and VP metrics show the general amount of features and their appearence in the source code. LOC and LOF metrics can be used to compare the number of feature related lines with those not feature related.

A good example for the relevance of preprocessor based variability is the linux kernel. It has the largest feature model publicly known and freely available. Between 600 and 1200 developers from more than 200 companies contribute up to 10000 patches per release to the codebase. This codebase covers more than 10000 features in over 8 million lines of code [19].

4.1.1 Granularity

In [27] a feature is described as "an optional or incremental unit of functionality" [1]. This option or increment can be of different granularity. In the case of variability management with IFDEF annotations (cpp) we can identify the following levels of granularity

Global adding a structure or function

Function adding if-block or statement inside a function or a field to a structure

Block adding a code-block

Statement varying the type of a local variable

Expression changing an expression

Signature adding a parameter to a function

Additionally IFDEFs can be used in an undisciplined fashion that leads to non-classifiable or even non-detectable results. An example for such an undisciplined usage can be seen in listing 4.1 where the annotation appears at substatement level [2, 117].

Listing 4.1: Example for undisciplined annotations

```
if (!ruby_initialized) {
#ifdef DYNAMIC_RUBY
    if (ruby_enabled(TRUE))
#endif
            ruby_init();
[...]
```

In the following sections the metrics gathered by CPPSTATS will be explained in detail. An overview is shown in table 4.1 and some of the SQL queries to retrieve the metrics are shown in appendix B listings B.13 until B.2.

[1] Other definitions of features exist, such as "a client-valued function" or "a functionality structuring concept" [23].

4.1.2 File Level Metrics

Metrics gathered per file:

ND The metrics in this group ANDAVG, ANDSTDEV and NDMAX describe the nesting depth of ifdef annotations. Average and corresponding standard deviation are measured per file. The maximum nesting depth is also recorded.

GRAN* This group contains metrics with the suffixes -BL, -EL, -FL, -GL, -ML, -SL that measure granularity of ifdef statements according to the model described in 4.1.1. Additionally the -ERR suffixed metric counts undisciplined usage that could not be matched.

LOC Counts lines of code in a file.

LOF Counts lines of feature code in a file. That is code that appears inside an ifdef annotation.

NOFC Counts the number of distinct feature constants appearing in a file.

VP The count of variation points, essentially the sum of all GRAN* values. Identifies the number of locations in the code that may be affected by feature selections.

4.1.3 Release Level Metrics

Metrics gathered per release:

NOFPFC Counts the number of files per feature constant and shows how distributed a feature is through the codebase.

SDEG The metrics SDEGMEAN and SDEGSTDEV measure the scattering degree (average and according standard deviation). That is the number of occurences of a feature constant in different feature expressions.

TDEG The metrics TDEGMEAN and TDEGSTDEV measure the tangling degree (average and according standard deviation). That is the number of distinct feature constant appearing in one feature expression.

TYPE The metrics HET, HOHE and HOM in the TYPE group count the number of feature expressions belonging to a certain type. Heterogeneous expressions add distinct extensions in different places. Homogeneous expressions add code dulpicats in different places. HOHE expressions are mixed and do not match one of the categories before. The types are determined by exact string comparison of the code guarded by a feature expression.

Additionally, the following metrics can be aggregated on release level from the file level metrics: NOFC (not the sum, but a total of all appearing feature constants over all files), LOF (sum and average), GRAN and VP (average for GRAN, sum and average for VP).

4.1.4 Project Level Metrics

Project level metrics are all aggregations from the metrics above (file level and release level). Therfore they can be computed by using SQL queries. Example queries can be found in appendix B from listing B.2 until B.13. The full set of retrieval queries is located at `/git/codeface-fork/spl/metrics_retrieval.sql` on the DVD. The concrete implementations for the dashboard widgets is located at `/git/codeface-fork/codeface/R/shiny/spl/db.queries.r` on the DVD.

Field	Description	Long Description	Aggregate Functions
LOC	Lines of Code	Number of `\n`. The size of a software system.	SUM
LOF	Lines of Feature Code	# of lines of feature-code (linked to feature-expressions). Variable fraction of the codebase.	SUM + MEAN per file
PLOF	Product: LOF/LOC	Ratio of total lines of code to feature code	
NOFC	Number of Feature Constants	Configuration dimension of a SPL (variability and complexity)	SUM on unique constants per project + MEAN (avg. per file)
NOFPFC	Number of Files Per Feature Constant	Only per release	SUM, AVG (Project)
ANDAVG	Average Nesting Depth of `#ifdef`s	Average nesting depth of ifdef annotations (average per file)	see NDMAX + MEAN
ANDSTDEV	Average Nesting Depths σ	Standard deviation for ANDAVG	
NDMAX	Maximum Nesting Depth		MAX (Release, Project)
GRANGL	Granularity: Global	Adding a structure or function	
GRANFL	Granularity: Function or Type	Adding an if-block or statement inside a function or a field to a structure	
GRANBL	Granularity: Block	Adding a block	
GRANSL	Granularity: Statement	Varying the type of a local variable	
GRANEL	Granularity: Expression	Changing an expression	
GRANML	Granularity: Function Signature	Adding a parameter to a function	
GRANERR	Parse Errors	Granularity detection failed	
SDEGMEAN	Scattering Degree	Occurences of a feature constant in different feature expressions	
TDEGMEAN	Tangling Degree	Distinct feature constant appearing in one feature expression	
VP	Variation Point	# of `#ifdef` occurences, implicit through sum of SDEG*	AVG, SUM (Release, Project)

Table 4.1: cppstats Metrics

5 Visualization

This chapter is about the design, implementation and evaluation of the components that visualize metrics in the software product-line context. In the implementation report section the prototyping and planning approaches and deriving of the final components from the prototypes is described. Afterwards an overview of the software projects that have been used as a data pool for evaluation is given. The last section describes the evaluation process and its results.

5.1 Implementation Report

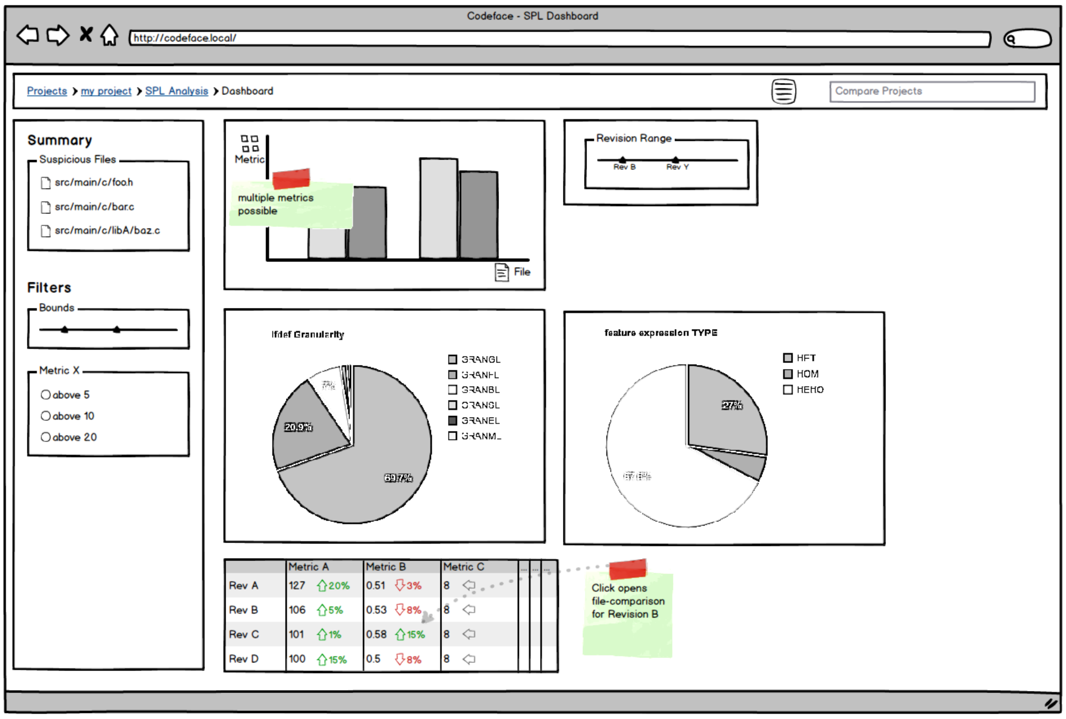
In the following sections we describe the efforts in implementing custom visualizations for software product lines in the context of the CODEFACE framework. Therefor the prototyping approach is shown, planning of user interface and user interaction is documented and the technical aspects of implementing the planned functions are discussed.

5.1.1 Prototyping

Two prototyping approaches have been used. First a mockup (see figure 5.1) has been created using the mockup drawing tool BALSAMIQ[1]. Before some sketches have been drawn together with potential users of the SPLDASHBOARD, these sketches are shown in the figures A.4.

As a second approach, a minimum example application has been built, that showcases the core functionality of the used frameworks. This application evolved during development and served as a testbed to test the feasibility of technical details like graph rendering or view-controller interaction. The example application can be found on the DVD at `/software/shiny-minimal-example`. GIT version control with disciplined comments has been used in all places so that the process is reproducible and serves as a documentation.

[1] `http://balsamiq.com/` (Retrieved: 08.08.2014)



	Metric A		Metric B		Metric C		...	...	...
Rev A	127	⇧20%	0.51	⇩3%	8	⇦			
Rev B	106	⇧5%	0.53	⇩8%	8	⇦			
Rev C	101	⇧1%	0.58	⇧15%	8	⇦			
Rev D	100	⇧15%	0.5	⇩8%	8	⇦			

Figure 5.1: SPLDASHBOARD Mockup

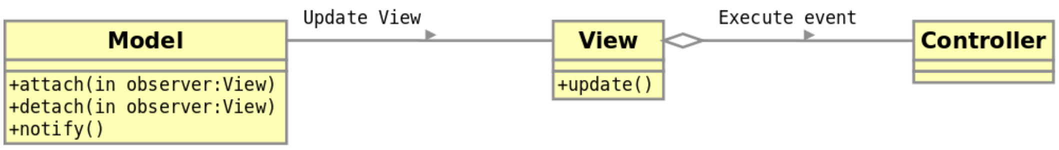


Figure 5.2: General MVC architecture

5.1.2 UI/UX Software Engineering

User Experience Design

In [1, 11-13] Stephen Anderson describes a model of user experience that can be seen as a stack of needs. It can be used as a top-down or bottom-up approach, as will be described later.

Functional The function level describes the most basic version of a product. It provides a technical solution to a problem. It is new and useful. Every new technological innovation starts at this level.

Reliable The next step in a products evolution is reliability. For example service reliability in terms of uptime or data integrity and validity.

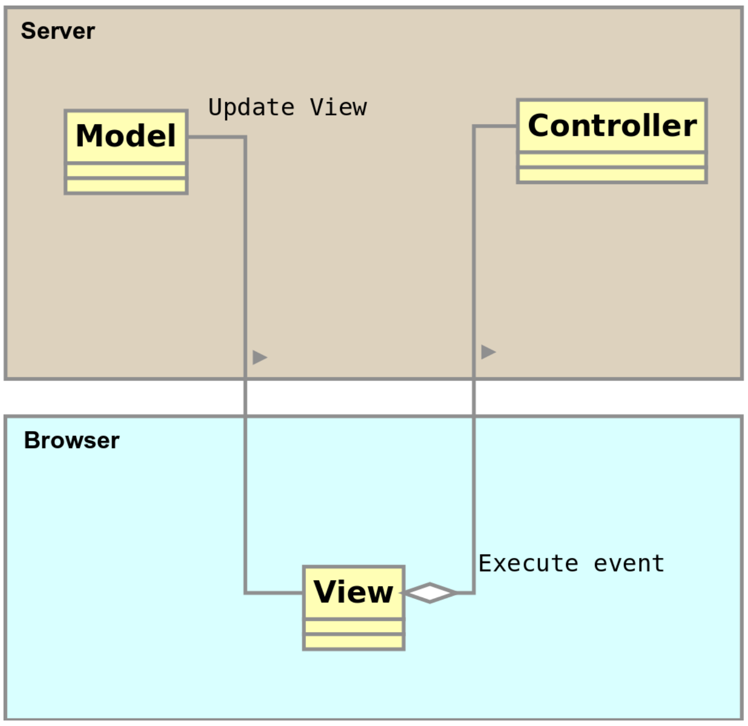


Figure 5.3: SHINYSERVER MVC architecture

Usability and Convenience In this step friction is removed and a product is made less awkward so it will eventually be perceived as usable and convenient. Usability focuses on removing known (technical) problems, while convenience tries to find more natural ways for solving a task. In this context, more natural can mean "in a technical way that acts more similar to real world concepts".

At this point, the models layer are divided. The layers already discussed are objective and can be rated based on quantifiable information. From here on the layers will cover qualitative and subjective properties.

Pleasurable and Meaningful After improving the cognitive experience, the next level focuses on affect and emotion. Making a product emotionally engaging and memorable to use is also important, even for a technical product. It will lead to a larger, faster growing user base, more feedback and therefore faster improvements (product iterations). Pleasurable products can — for example — use friendly language, aesthetics and humor to reach that goal.

Meaningful is the next and highest level of subjective/qualitative product experience. Meaning is a personal experience and so a product can not be designed to be meaningful. But this experience can be used as a starting point for product design. Swapping the described bottom-up (Functional $\Rightarrow$ Meaningful) model for a top-down approach, thinking about the user experience first and designing the product aligned to it.

User Experience Goals

The following user experience goals define what experience we want to provide to users of the SPLDASHBOARD. These goals will serve as a guideline in section 5.3 for validating the implementation.

- */UX01/* free chosing of granularity and detail
- */UX02/* "zoom in experience"
- */UX03/* explorability
- */UX04/* drill down approach
- */UX05/* intuitive click targets and paths
- */UX06/* self explaining
- */UX07/* integrated help system
- */UX08/* fast, reliable import process
- */UX09/* accessible[2] user interface

5.1.3 Extending shinyserver

SHINYSERVER is the framework used for presentation of the visualizations generated with R. It also serves as a basis for the general user interface. This framework used for visualization had to be extended to support custom user interface components and software-product line specific navigation as described in the section on user experience goals in 5.1.2. Figure 5.4 shows a schematic overview of the components explained in the following sections.

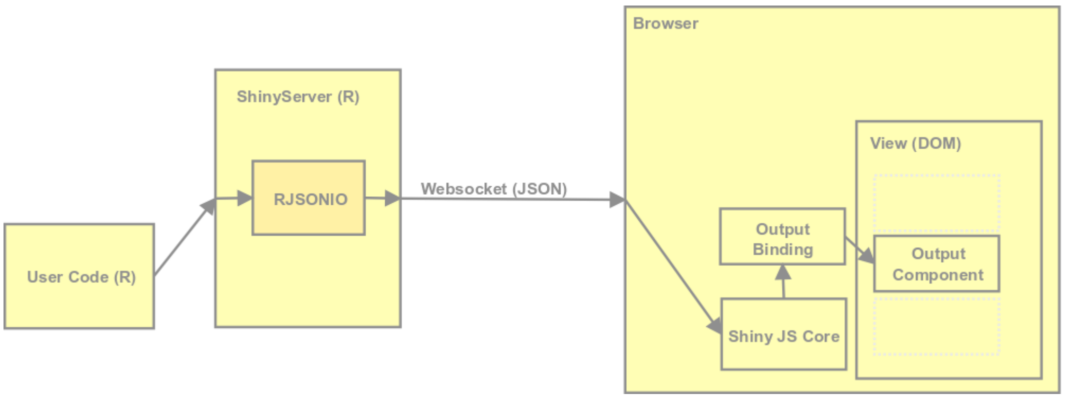


Figure 5.4: SHINYSERVER Custom Output Object architecture

[2] Accessibility in this context means the degree to which a software product is available to as many people as possible, especially considering users with disabilities.

Developing a userinterface in the context of SHINYSERVER involves techniques from client- and serverside web development. In the following we give a short introduction to the relevant methods and means.

Serialization The process of converting the state of an object into a byte stream in such a way that the byte stream can be converted back into a copy of that object is described as serialization [26]. In our case, serialization and deserialization take place in different languages on different platforms. So the process is also used for canonical representation and object conversion (between R and JavaScript).

HTML This is a markup language describing the structure of a document (website) semantically. There are some elements that control presentation but the development of the language leads to a more and more distinct separation of semantics and presentation. The listing 5.1 shows a minmal valid document according to the HTML5 specification [7].

Listing 5.1: HTML5 Minimal Example

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>title</title>
    <link rel="stylesheet" href="style.css">
    <script src="script.js"></script>
  </head>
  <body>
    <!-- page content -->
  </body>
</html>
```

CSS and CSS Selectors This separation has lead to Cascading Style Sheets (CSS). CSS specifies expressions to adress single HTML elements or collections of them, these are called selectors. Selectors can filter on the `id` and `class` attribute of HTML elements. There are also more complex operators that allow for example counting or specification of parent-child relationships. The example in listing 5.2 selects all `li` elements that are the third child of an unordered list `ul` with the `class` attribute containing the string `'addresses'` that are in turn contained in the unique `div` element with an `id` attribute value of `'address-list'`. These expressions are normally used to associate blocks of style definitions (color, font-size, etc.) with DOM elements. The described implementation will do that also, but primarily use them to address DOM elements in JavaScript code. The JQUERY library provides a Domain-Specific Language (DSL)[3] for this task that is nearly equal to CSS expressions.

Listing 5.2: CSS Selector Example

```
div#address-list ul.adresses li:nth-child(3) {...}
```

DOM The Document Object Model is a convention for representing and interacting with the objects of an HTML (XML) document in a language-independent cross-platform way [11, 312-313]. Objects in the DOM tree can be addressed and manipulated by using methods on these objects. So the DOM is an object-oriented representation of a structured document. Starting with "legacy DOM" that evolved from the development of JavaScript in NETSCAPE and JScript in INTERNET EXPLORER respectively, it was finally specified as a standard ("DOM Level 1") during the standardization of JavaScript derivatives under the ECMAScript hat. The current working draft version is "DOM Level 4". Nearly all web browsers use an internal representation of a website that is very similar to a DOM.

As we have seen before, communication between model and controller on the server-side (SHINYSERVER) and the view on the client side (web browser) is done by JavaScript Object Notation (JSON) encoded data. SHINYSERVER uses the RJSONIO library for serializing from, and deserializing to R data-structures.

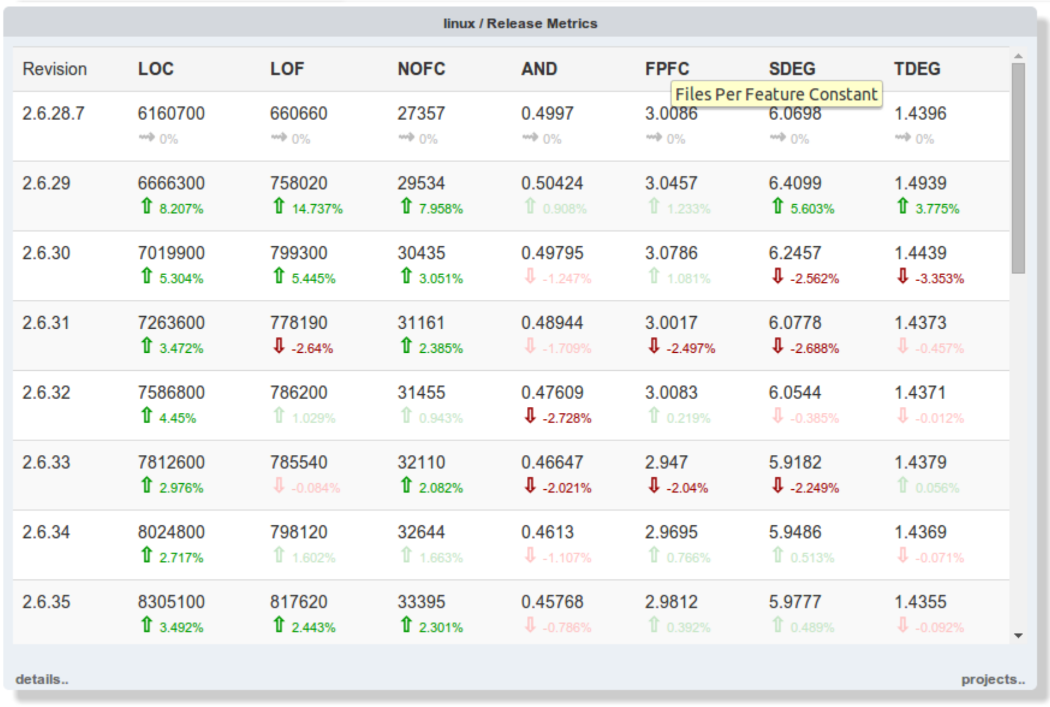
Some default data-structures are already handled by the RJSONIO default serializers. For some cases, SHINYSERVER provides helper functions. Those exist for simple tables (`renderTable()`), text output and plots (`renderPlot()`). In the special case of plots, the data is transported as binary representation of a Portable Network Graphics (PNG) format image.

Building a custom output object, like the enriched table from figure 5.5, consist mainnly of four steps.

1. Server-Side Output Functions
2. Design Output Component Markup
3. Write an Output Binding
4. Register Output Binding

Server-Side Output Function This functions task is, to translate any given data-structure to a structure that is serializable by the RJSONIO marshaller. The function will later be used to provide output for reactive endpoints. We are then able to pass in results of custom computations, serialize them as JSON and let SHINYSERVER handle the transmission to the view (web browser).

[3] `http://api.jquery.com/jQuery/` (Retrieved: 08.08.2014)



linux / Release Metrics

Revision	LOC	LOF	NOFC	AND	FPFC	SDEG	TDEG
2.6.28.7	6160700 → 0%	660660 → 0%	27357 → 0%	0.4997 → 0%	3.0086 → 0%	6.0698 → 0%	1.4396 → 0%
2.6.29	6666300 ⇧ 8.207%	758020 ⇧ 14.737%	29534 ⇧ 7.958%	0.50424 ⇧ 0.908%	3.0457 ⇧ 1.233%	6.4099 ⇧ 5.603%	1.4939 ⇧ 3.775%
2.6.30	7019900 ⇧ 5.304%	799300 ⇧ 5.445%	30435 ⇧ 3.051%	0.49795 ⇩ -1.247%	3.0786 ⇧ 1.081%	6.2457 ⇩ -2.562%	1.4439 ⇩ -3.353%
2.6.31	7263600 ⇧ 3.472%	778190 ⇩ -2.64%	31161 ⇧ 2.385%	0.48944 ⇩ -1.709%	3.0017 ⇩ -2.497%	6.0778 ⇩ -2.688%	1.4373 ⇩ -0.457%
2.6.32	7586800 ⇧ 4.45%	786200 ⇧ 1.029%	31455 ⇧ 0.943%	0.47609 ⇩ -2.728%	3.0083 ⇧ 0.219%	6.0544 ⇩ -0.385%	1.4371 ⇩ -0.012%
2.6.33	7812600 ⇧ 2.976%	785540 ⇩ -0.084%	32110 ⇧ 2.082%	0.46647 ⇩ -2.021%	2.947 ⇩ -2.04%	5.9182 ⇩ -2.249%	1.4379 ⇧ 0.056%
2.6.34	8024800 ⇧ 2.717%	798120 ⇧ 1.602%	32644 ⇧ 1.663%	0.4613 ⇩ -1.107%	2.9695 ⇧ 0.766%	5.9486 ⇧ 0.513%	1.4369 ⇩ -0.071%
2.6.35	8305100 ⇧ 3.492%	817620 ⇧ 2.443%	33395 ⇧ 2.301%	0.45768 ⇩ -0.786%	2.9812 ⇧ 0.392%	5.9777 ⇧ 0.489%	1.4355 ⇩ -0.092%

Figure 5.5: SPLDASHBOARD Project level overview of metrics per release

Output Component This describes the HTML markup used to render the component in the view. The minimum output components markup consist of a container-element, usually a `div` [7, sec. 4.4.13]. Two important attributes need to be set: the class attribute of an HTML element contains a space-separated list of class names, referencing CSS classes. The fact, that a referenced CSS class does not necessarily need to be defined in a stylesheet is used here. The classname is then used to identify a set of container-elements that will render the same type of output data. The `id` attribute contains an unique identifier, that binds one instance of an output-container-class to a reactive endpoint (see 3.1.4) [7, sec. 3.2.5.1;3.2.5.7].

Output Binding An Output Binding models the connection between reactive endpoints and output components. It uses the SHINYSERVER JavaScript core as a basis and is represented as a JavaScript object that implements an interface, allowing the following operations. An example is shown in listing 5.3.

`find(scope)` given a HTML document or element scope, finds all instances of a custom output component. The fact that all output elements share the same CSS class can be used, to find them by a CSS expression as explained above. The interface is defined by the abstract class `OutputBinding` which is a member of the global `Shiny` object defined in `shiny/inst/www/shared/shiny.js`. The `Shiny` object is bound to the global namespace `window.Shiny` so it can be used inside custom scripts.

Listing 5.3: Custom Output Binding

```
exampleOutputBinding.find = function(scope) {
  return $(scope).find('.exampleComponentClass');
};
```

The essential method is `renderValue(el, data)`. It is called each time a new value arrives that matches the output components id attribute. `el` is a reference on the DOM element, data contains the JSON data received from the server and generated by the server-side output function and RJSONIO.

The next method, required by the interface, is `getId(el)` that must return the output ID (which is the name of the reactive endpoint). The two remaining methods `renderError(el, err)` and `clearError(el)` implement error handling.

Default implementations exist for `renderError(el, err)`, `clearError(el)` and `getId(el)`. The latter returning the `data-input-id` attribute or — if it does not exist — the `id` attribute of the given DOM element. The former do a simple replacement of the current output component container content with the error message (`err.message`) and additionally set a CSS class that indicates an error state.

The last step is to register the custom output binding as shown in listing 5.4 with the SHINYSERVER JavaScript framework using the `register(binding,id ↪ )` method, where `binding` is a reference to the bindings object and `id` is a unique string identifying the binding, similar to a package and class name in Java.

Listing 5.4: Register Custom Output Binding

```
Shiny.outputBindings.register(exampleOutputBinding, "
    ↪ yourname.exampleOutputBinding");
```

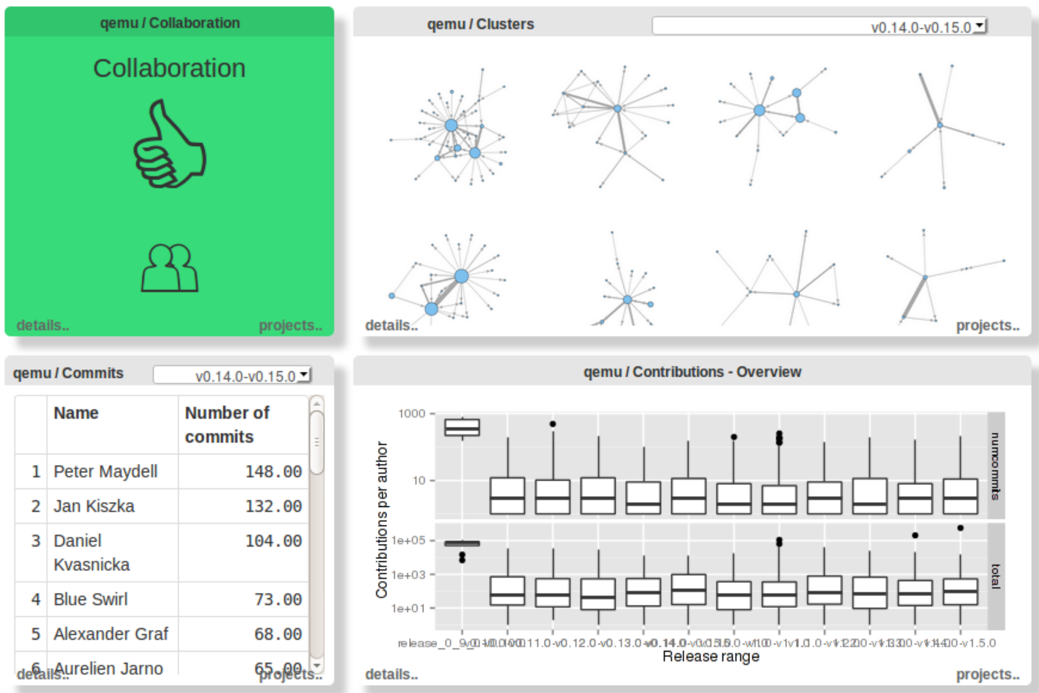
To separate the custom output bindings namespace, self executing anonymous functions as a way of scoping the modules against eachother, have been used. The technique uses a function scope as a private namespace [11, 248-250].

Finally, the GIT repository `shiny-minimal-example` on the DVD contains a sample implementation invloving all the above mentioned processes.

Creating Widgets and Dashboards

Widgets are abstractions of UI components. Widgets have a specific size that is defined in terms of height and width tuples. Both values are discrete, so the widgets can be placed on a grid. Such grids are provided by dashboards that act as a container. The initial layout of a dashboard is defined in the application, but widgets can be re-arranged through drag-and-drop by the user.

Widgets can also be deleted from or added to the layout of a dashboard. An example of a dashboard, comprising different widgets with different content is shown in figure 5.6.



	Name	Number of commits
1	Peter Maydell	148.00
2	Jan Kiszka	132.00
3	Daniel Kvasnicka	104.00
4	Blue Swirl	73.00
5	Alexander Graf	68.00
6	Aurelien Jarno	65.00

Figure 5.6: Example of a dashboard with widgets

The process of creating a widget involves four steps that are outlined in the following list.

1. Create widget file in `codeface/R/shiny/widgets/[WIDGET_NAME].r`
2. Implement an S3 class, extending `widget` [20, pp.208] from `codeface/R/shiny/widgets.r` with the minimum method set of
   a) `initWidget` - initialize data variables for widget
   b) `renderWidget` - build the reactive output of the widget, which can be anything that is representable as JSON (including an HTML fragment)
3. Call `createWidgetClass(...)` to register a template (list of information on how to create a widget instance) in `widget.list[CLASSNAME]`
4. Include the widget implementation using `source(...)` at the bottom of `codeface/R/shiny/widgets.r`

5.1.4 Implementing the Extensions

During testing of new widgets, it was usefull to see the error messages that are generated by R and the database layer. This can be achieved by starting SHINYSERVER with the environment variable `SHINY_LOG_LEVEL` set, as with this statement `export SHINY_LOG_LEVEL="TRACE"`. Even more unfiltered output with less indirection can be seen by starting the dashboard application directly in an `R` console. A script is provided at `codeface/R/shiny/apps/dashboard/debug_start.r` that can be run with the command `R --no-save < debug_start.r`.

Additional URL Parameters

Dataflow from the HTTP layer of ShinyServer through to the widget implementations is a bit nested. Parameter parsing and validation happens in `/codeface/R/shiny/apps/dashboard/server.r`. From there on, data takes the following path (file paths relative to `/codeface/R/shiny/`):

1. `shinyServer(...)` function in `apps/dashboard/server.r`
2. `widgetbase.output(...)` function in `apps/dashboard/server.r`
3. `initWidget(...)` function in `widgets.r`
4. `newWidget(...)` function in `widgets.r`

These four functions needed an exteded parameter list, to pass the `file` and `release` GET URL parameters to the widget. Therefore a list `spl` containing reactive sources `file` and `release` is created in `shinyServer(...)` and finally set as a member of the widget instance returned by `newWidget(...)`.

Release Overview Table

The release overview table has a relatively complex layout and supports interaction with the table-cells by clicking. This fact and reduction of the amount of data that flows from server to client for table updates lead to the decision to generate the table markup on the client side. The serverside processing of the widget returns just the data backing the table (arrays of column values) including the calculated differences. The template library MUSTACHE.JS[4] is used to transform the received data into HTML fragments by replacing variables in a template. These fragments are then appended to the existing DOM. The rendering process is implemented in the output binding of the custom output object (see 5.1.3).

The function for identifying and highlighting relevant changes for different metrics may be different according to the metrics properties. A mechanism has

[4] `https://github.com/janl/mustache.js/` (Retrieved: 08.08.2014)

been developed to support new highlight functions: the object `importance` in the `spl/metrics_table_binding` namespace has attributes that are functions mapping $metric\_value \rightarrow \{\texttt{'low', 'med', 'high'}\}$. The image of this function is the set of possible css classes that are dynamically set on the table cells. Style definitions for these classes have been developed to emphasize on the high values while putting low ones in the background. A pluggable highlight function registered in the `importance` object can be exchanged per metric. The table widget shown in figure 5.5 uses a generic function, with the following mapping.

$$\begin{array}{lll} |change| > 20 & \rightarrow & \texttt{'high'} \\ |change| > 2 & \rightarrow & \texttt{'med'} \\ \text{else} & \rightarrow & \texttt{'low'} \end{array}$$

Average Nesting Depth Chart

The average nesting depths average (ANDAVG, see 4.1.3) can be shown as a dot-plot for all file instances in a release [6, 21, 25]. The widget supports a filter by minimum ANDAVG value. The example shown in figure 5.7 shows files in the 1.3.29 release of the APACHE http server project, filtered by values greater than 1. Additionally the charts lines are grouped by the following scheme and colored accordingly: $v \leq 1.5 \rightarrow 1$, $v > 1.5 \rightarrow 2$, $v > 2 \rightarrow 3$.

Extension Type Chart

Pie charts are not a good way of displaying information as the eye is good at judging linear measures and bad at judging relative areas. However two widgets have been implemented as a pie chart, because they are very common in management dashboard views and have been in the prototype concepts. The extension type chart graphs the data of the TYPE metric described in 4.1.3. An example for this chart is shown in figure 5.9.

Granularity Chart

An example for a granularity chart is shown in figure 5.8. It graphs the data of the GRAN metric described in 4.1.3.

Open Tasks

As shown in the graph of the ANDAVG metric, the space can be very densely filled. This applies to all metrics with a lot of instances (mainly releases and files). Therefore interactive filtering should be implemented. Rendering the charts on the server and transmitting the binary image data has two negative effects. First, the amount of data is large, compared to the raw data that builds

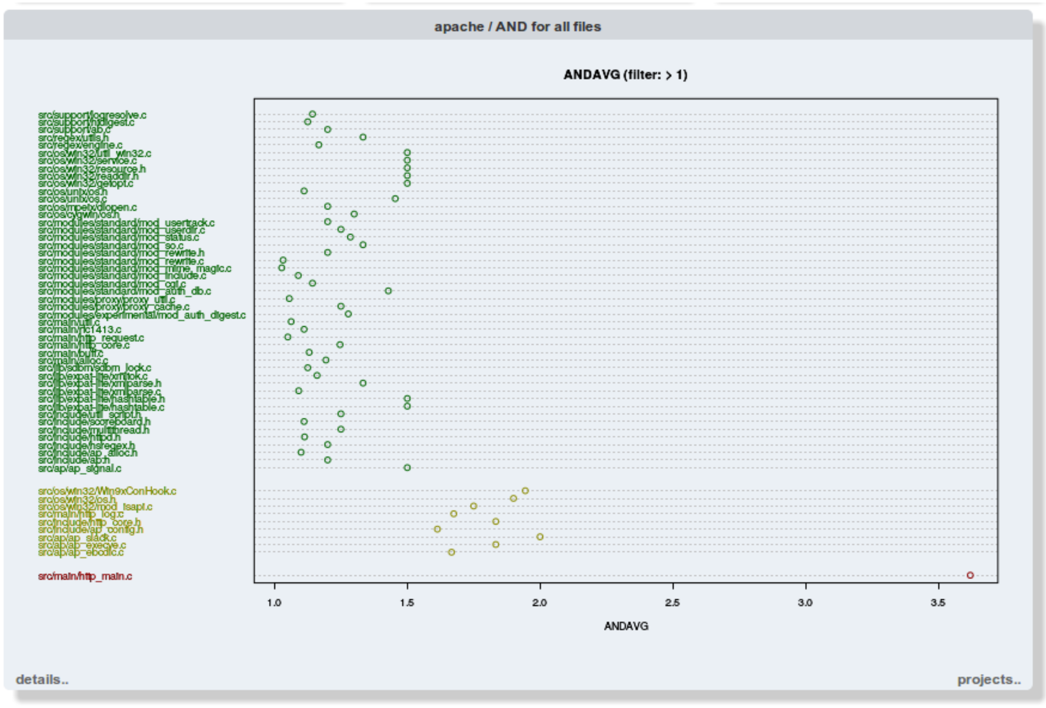


Figure 5.7: ANDAVG metric for all files in a release, filtered by a minimum value of 1 and represented as a dot-chart

the graph. Second, the image is not interactive, as is consists of static color per pixel information. Client side rendering approaches like D3JS[5] exist and should be evaluated.

5.2 Description of the Example Data Sets

The example data sets all stem from open-source projects. The set of projects has been chosen because one of the advisors of this thesis used them successfully for other evaluation purposes in the context of CPPSTATS. The metrics have been used in the state from the previous work, so that comparison of results is possible.

Table 5.1 shows statistics about the example projects such as total Lines of Code (LOC) processed for all releases, the number of lines of metrics data in CSV files and number of releases analyzed.

[5] `http://d3js.org/` (Retrieved: 08.08.2014)

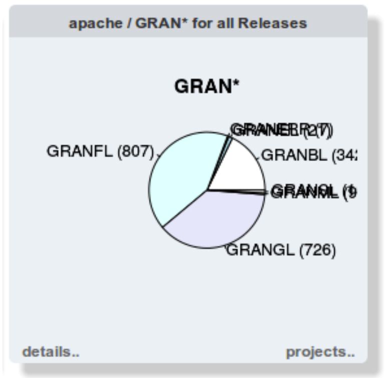


Figure 5.8: GRAN metric widget

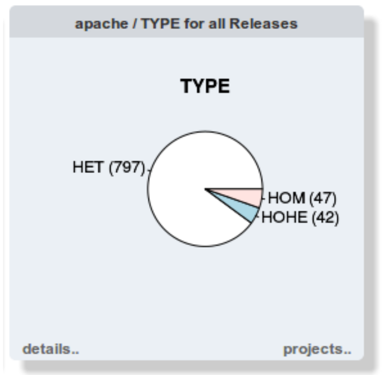


Figure 5.9: TYPE metric widget

Project	LOC (all releases)	Lines of cppStats data	releases analyzed
apache	15020247	39999	142
berkeleydb	4017259	10016	21
busybox	13918465	59484	89
cherokee	6954656	37856	139
emacs	4746643	9882	23
freebsd	108197779	354654	17
gimp	18387512	76759	28
gnuplot	1466682	3326	20
libxml2	12558102	12323	116
linux	193926300	623818	22
mpsolve	32190	154	2
openvpn	432132	1368	9
parrot	1663382	4412	16
postgresql	99314873	275360	225
sendmail	3561737	8394	51
subversion	34779108	62231	87
vim	2801816	1823	18
xfig	276840	1087	5
xterm	3992459	4538	74
sqlite		5810	29

Table 5.1: Software projects that served as test data pools.

Project	first load (s)	subsequent load (s)	Database size
apache	10.67	2.19	108
berkeleydb	4.01	2.21	16.2
emacs	4.29	1.89	16.1
gimp	14.33	2.14	100.1
postgresql	139.15	2.68	349.5

Table 5.2: Dashboard load time (for first and subsequent load) per project.

5.3 Analysis

All */UXxx/* references in the following sections refer to the user experience goals set in section 5.1.2.

5.3.1 Accessibility

Accessibility of the GUI is required in (*/UX09/*). Because of the implementation as a web application, some accessibility features that are available in web browsers can be used. Browser integrated features like text-zoom, custom color schemes and contrast modes do work without further investment.

A large part of the HTML markup of the dashboards is dynamically generated after the page has been loaded. The markup may change at any time if events arrive from the controller. Besides the markup is missing annotations that express the semantics of elements that allow interaction. Because of that, features like keyboard navigation, screen reader and braille devices fail. Statistical data rendered as graphs in images also can not be displayed by these technologies.

5.3.2 Responsiveness

The SQL queries for retrieval of the metrics (that feed data to the widgets) have been written in an expressive style, that documents their intention. As a result a lot of subqueries are used where (materialized) views would make a lot more sense. The subqueries lead to temporary table generation[6] and the poor performance on large projects that is shown in table 5.2. As queries for single widget instances are generally static, the MYSQL query cache[7] is able to absorb some of the performance problems — at least, after a dashboard has been shown once.

[6] This can be seen by prefixing the queries with the EXPLAIN keyword.

[7] `http://dev.mysql.com/doc/refman/5.1/en/query-cache.html` (Retrieved: 08.08.2014) - caches the results for exact query string matches.

Also the MYSQL server has been used with its default configuration. The performance may improve by adapting the configuration of the INNODB storage engine to the amount of available Random Access Memory (RAM).

5.3.3 Self-Explanatory

As claimed in */UX06/* the user interface should be self explanatory. In figure 5.5 we show in place explanation of short metrics names with a popover dialog. Such explanations are implemented in many places to explain abbreviations, details of variables or more precise values if the display needed truncation of decimal places. Clicking on free space of a widget shows an explanation text.

5.3.4 Fast Import Process

A fast import process is a requirement in */UX08/*. In section 3.2.4 is described and summarized in table 3.2 that the import process takes considerable amount of time. The average performance for parsing the CSV files is 131 lines per second. The largest project import, Linux, took 7 hours with the initial approach.

Caching improves import runtime by an average of 35%, but for large projects (like the Linux kernel or Freebsd) with a number of files that is a multiple of 10^4 it improves runtime by about 70%. For projects that are large considering their amount of releases, caching does not have the same effect (see postgresql for an example).

As a variable is described as a triple of $(scopeInstance, metric, value)$ and the most variables appear in the `ScopeFileInstance` scope, the lookups for file instance ids dominate. With each new release, the file instance cache is invalidated, as each instance is only valid at one point in time (release). So the product $\frac{fi}{rel}$, where fi is the file instance count and rel is the number of releases, is a good indicator for potential cache improvement. This is shown in figure 5.10.

5.3.5 Drill Down Approach

As required in */UX02/*, */UX03/* and */UX04/* the data should be explorable, giving the user the possibility to choose freely between levels of granularity of the displayed data. Peculiar values should be clearly highlighted and clickable to investigate on the cause of such values. This is realized with the three SPLDASHBOARD views that are described in the following section (and the CODEFACE projects view).

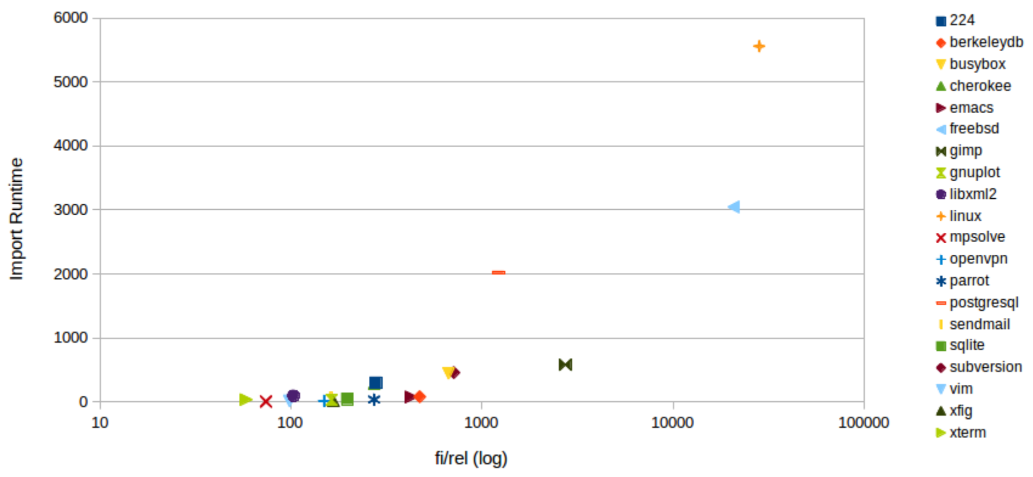


Figure 5.10: $\frac{fi}{rel}$ compared to the import runtime with caching

The Dashboards

Dashboards are grids on which widgets can be arranged (see 5.1.3). Widgets can contain hyperlinks that switch to other dashboard views. A marker badge has been implemented that clearly highlights projects with Software Product Line data available. The dashboard showing an overview of all projects did already exist in CODEFACE. In all dashboards, clicking on the grid icon in the header allows adding and removing of widgets. The example figure A.3 shows only a selection because of space limitations.

SPL Project Clicking on the Software Product Line indicator badge in the dasboard before switches to the Software Product Line project dashboard. It shows metrics aggregated per project as tables and charts. Additionally it shows the release overview table described above.

SPL Release Clicking on a line in the release overview table switches to the Software Product Line release dashboard. It displays metrics aggregated on the release level as tables and charts.

SPL File Instance Clicking on a file path that, together with the release, identifies a file instance, switches to the Software Product Line file instance dashboard. It shows metrics for the selected file instance. The gauge type widgets have their maximum set to the project level maximum of the specific metric.

6 Outlook and Future Work

As import speed is important, parallelization of the import process is an option for improvement. The CPPSTATS files (per revision and per file) have no interdependencies. Actually not even the lines of the CSV files have interdependencies, so the import process could be parallelized with little effort.

General schemes for prediction of software faults by using data-mining techniques on software metrics are questionable. But it is possible to build statistic models on a per project or per project type basis that predict post-release defects [22]. Even if the models have weaknesses regarding generalization, they exist and have practical applications [10, 15, 12, 16]. It is likely that such predictors can be built for software product line specific metrics. Data about bugs, versioning-system commits and developer communication is already aggregated in codeface and could support such analyses. With such metrics it would be possible to extend the SPLDASHBOARD with utilities that warn about upcoming problems and help to prevent them [18].

Further evaluation of the visualizations through experiments with a real user base on real projects should be done.

Retrieval, calculations and highlighting for a specific metric is currently scattered through R and JavaScript code. Packaging a metric as a module and reduce implementation to one language would make extensions easier.

Possible improvements for the SPLDASHBOARD itself have been collected during development and are listed below.

1. Adding datasoruces (parser, import) for other languages and analysis tools.
2. Adding visualizations as new widgets.
3. Integration through hyperlinks with other codeface components.
4. Run the CPPSTATS tool from within the CODEFACE CLI.

A Figures

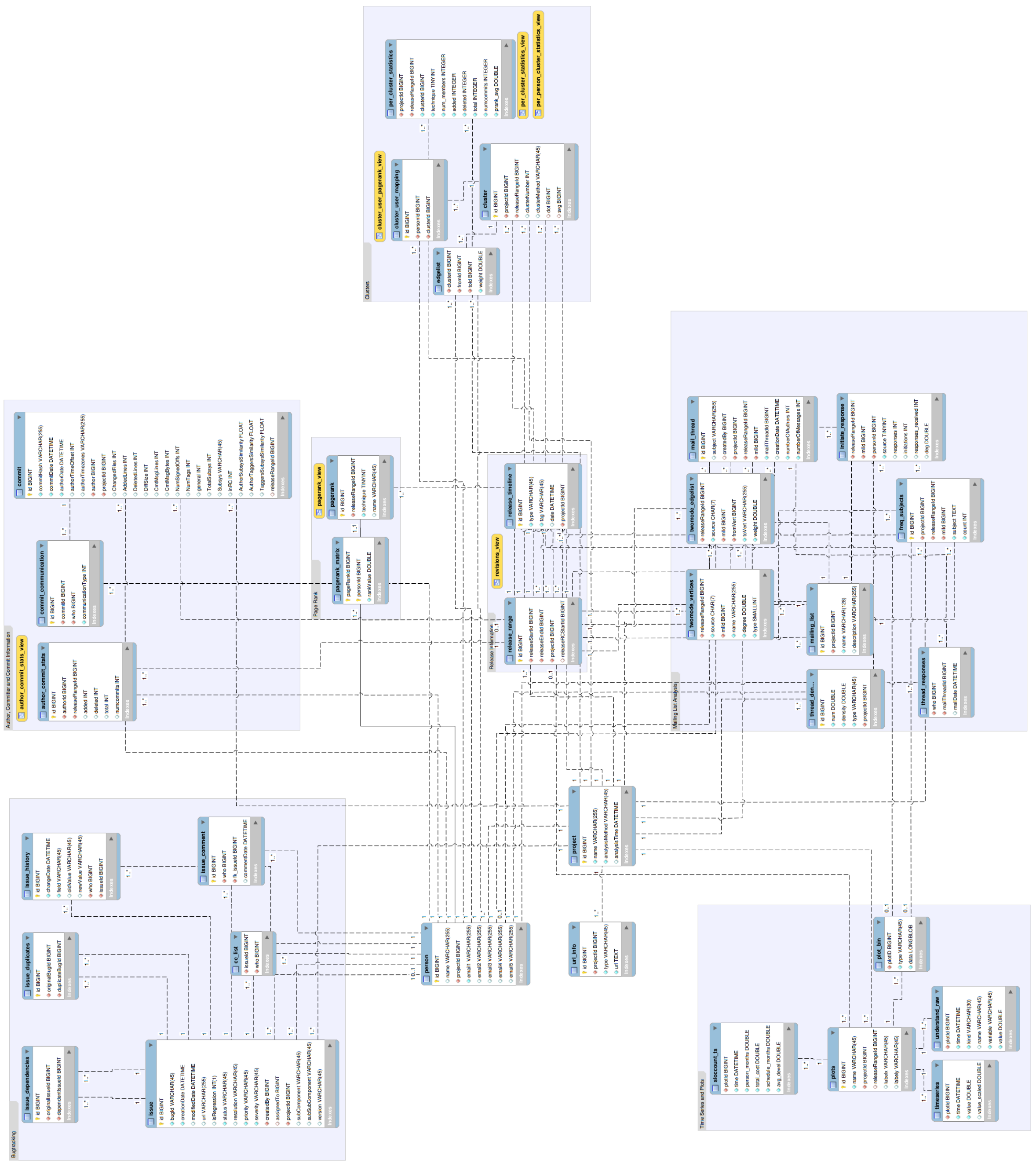


Figure A.1: CODEFACE Datamodel before modifications

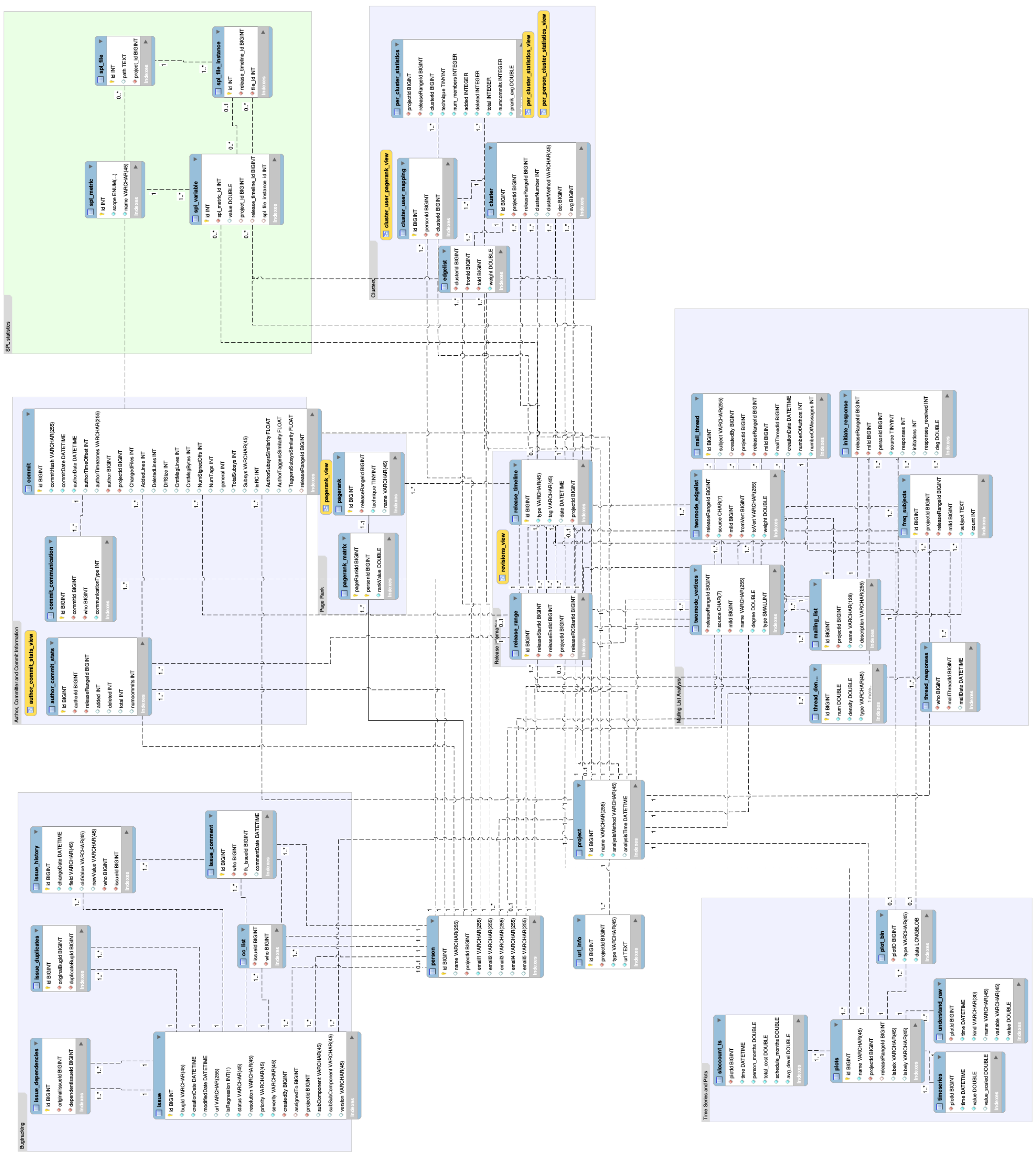


Figure A.2: CODEFACE database schema including SPLDASHBOARD modifications

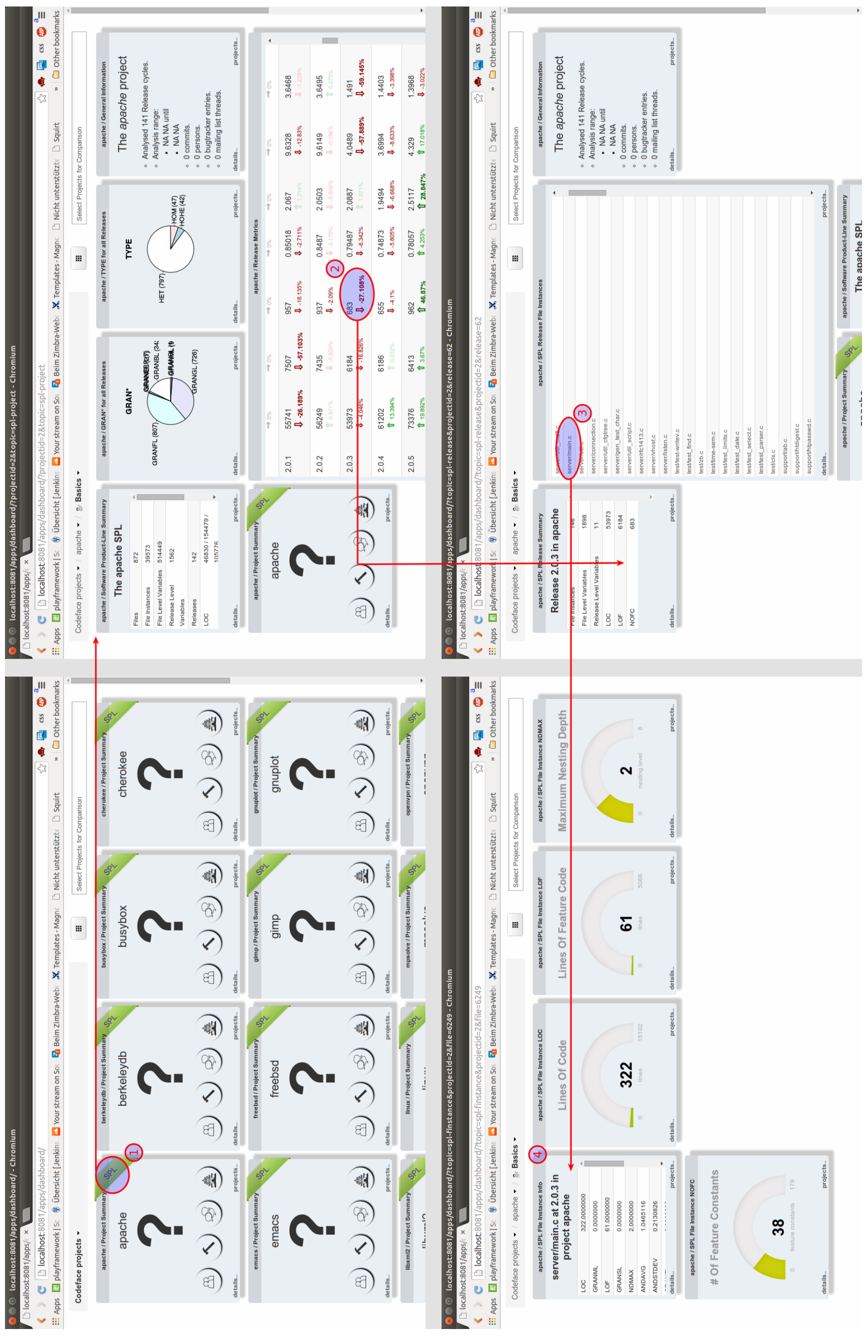
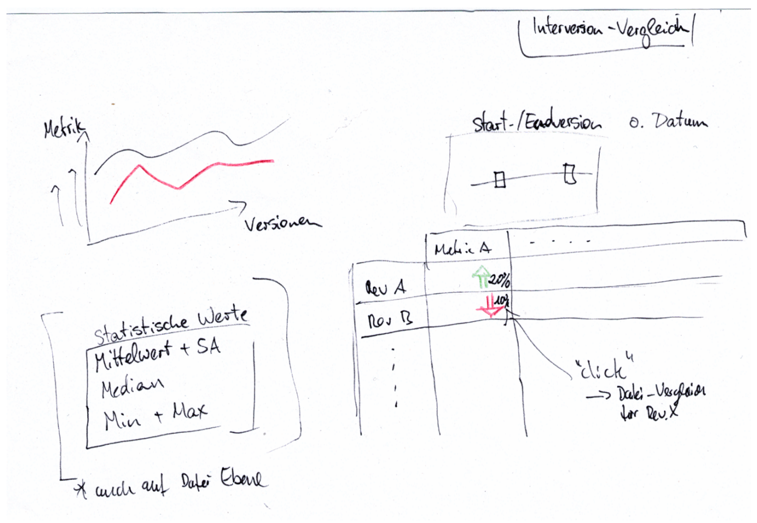
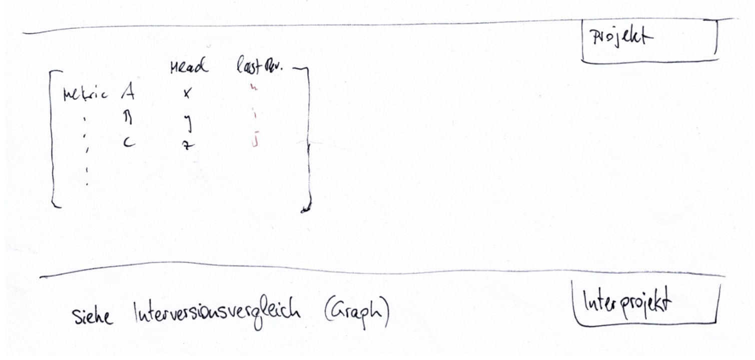


Figure A.3: Drill down approach through different dashboards in CODEFACE



(a) Interversion comparison and statistics



(b) Project metrics

Figure A.4: First sketches of the SPLDASHBOARD components

B Listings

Listing B.1: Example codeface configuration file

```
project: test
repo: test # Relative to git-dir as specified on the command line
mailinglists:
    -   name: gmane.comp.apache.test
        type: dev
        source: gmane
description: This is a test project
revisions: [ "1.3.0", "1.3.1", "1.3.2", "1.3.3" ]

tagging: proximity

bugsProjectName: test project
issueTrackerType: bugzilla
issueTrackerURL: https://issues.apache.org/bugzilla
productAsProject: true

spl:
  cppstats:
    basePath: /home/sniechzial/03_Uni/Masterarbeit/data/cppstats_analysis_data/apache
    revisionPattern: "httpd-%s"
    perfile: perfile.csv
    perrevision: merged.csv
```

Listing B.2: NOF – Number of Files

```sql
SELECT count(*) count
FROM spl_file f
JOIN project p
  ON p.id = f.project_id
WHERE p.name = 'test'
```

Listing B.3: NOF per release

```sql
SELECT rt.tag revision, count(*) count
FROM spl_file_instance fi
JOIN release_timeline rt
  ON rt.id = fi.release_timeline_id
JOIN spl_file f
  ON fi.file_id = f.id
JOIN project p
  ON p.id = f.project_id
WHERE p.name = 'test'
GROUP BY rt.tag
```

Listing B.4: LOC – Lines of Code (SUM)

```sql
SELECT rt.tag revision, SUM(v.value) LOC
FROM spl_variable v
JOIN spl_metric m
  ON v.spl_metric_id = m.id
JOIN spl_file_instance fi
  ON v.spl_file_instance_id = fi.id
JOIN release_timeline rt
  ON rt.id = fi.release_timeline_id
JOIN spl_file f
  ON fi.file_id = f.id
JOIN project p
  ON p.id = f.project_id
WHERE p.name = 'test'
AND m.name = 'LOC'
GROUP BY rt.tag
```

Listing B.5: LOC – Project Average

```sql
SELECT AVG(LOC) LOC FROM (
  SELECT rt.tag revision, SUM(v.value) LOC
  FROM spl_variable v
  JOIN spl_metric m
    ON v.spl_metric_id = m.id
  JOIN spl_file_instance fi
    ON v.spl_file_instance_id = fi.id
  JOIN release_timeline rt
    ON rt.id = fi.release_timeline_id
  JOIN spl_file f
    ON fi.file_id = f.id
  JOIN project p
    ON p.id = f.project_id
  WHERE p.name = 'test'
  AND m.name = 'LOC'
  GROUP BY rt.tag
) AS tmp
```

Listing B.6: LOF – Lines Of Feature code

```sql
SELECT rt.tag revision, SUM(v.value) LOF
FROM spl_variable v
JOIN spl_metric m
  ON v.spl_metric_id = m.id
JOIN spl_file_instance fi
  ON v.spl_file_instance_id = fi.id
JOIN release_timeline rt
  ON rt.id = fi.release_timeline_id
JOIN spl_file f
  ON fi.file_id = f.id
JOIN project p
  ON p.id = f.project_id
WHERE p.name = 'test'
AND m.name = 'LOF'
GROUP BY rt.tag
```

Listing B.7: PLOF – LOF/LOC product

```sql
SELECT rt.tag revision, (SUM(v.value) / SUM(v2.value)) PLOF
FROM spl_variable v
JOIN spl_metric m
  ON v.spl_metric_id = m.id
  AND m.name = 'LOF'
JOIN spl_file_instance fi
  ON v.spl_file_instance_id = fi.id
JOIN release_timeline rt
  ON rt.id = fi.release_timeline_id
JOIN spl_file f
  ON fi.file_id = f.id
JOIN project p
  ON p.id = f.project_id
JOIN spl_variable v2
  ON v2.spl_file_instance_id = v.spl_file_instance_id
JOIN spl_metric m2
  ON m2.id = v2.spl_metric_id
  AND m2.name = 'LOC'
WHERE p.name = 'test'
GROUP BY rt.tag
```

Listing B.8: VP – Variation Point

```sql
SELECT rt.tag revision, SUM(v.value) VP
FROM spl_variable v
JOIN spl_metric m
  ON v.spl_metric_id = m.id
JOIN spl_file_instance fi
  ON v.spl_file_instance_id = fi.id
JOIN release_timeline rt
  ON rt.id = fi.release_timeline_id
JOIN spl_file f
  ON fi.file_id = f.id
JOIN project p
  ON p.id = f.project_id
WHERE p.name = 'test'
AND m.name LIKE 'GRAN%'
GROUP BY rt.tag
```

Listing B.9: NOFC – Number Of Feature Constants

```sql
SELECT rt.tag revision, v.value NOFC
FROM spl_variable v
JOIN release_timeline rt
  ON rt.id = v.release_timeline_id
JOIN spl_metric m
  ON v.spl_metric_id = m.id
JOIN project p
  ON p.id = rt.projectId
WHERE m.name = 'NOFC'
AND p.name = 'test'
AND m.scope = 'ScopeRelease'
```

Listing B.10: SDEGMEAN – Scattering Degree (mean value)

```sql
SELECT rt.tag revision, v.value SDEGMEAN
FROM spl_variable v
JOIN release_timeline rt
  ON rt.id = v.release_timeline_id
JOIN spl_metric m
  ON v.spl_metric_id = m.id
JOIN project p
  ON p.id = rt.projectId
WHERE m.name = 'SDEGMEAN'
AND p.name = 'test'
AND m.scope = 'ScopeRelease'
```

Listing B.11: TDEGMEAN – Tangling Degree (mean value)

```sql
SELECT rt.tag revision, v.value TDEGMEAN
FROM spl_variable v
JOIN release_timeline rt
  ON rt.id = v.release_timeline_id
JOIN spl_metric m
  ON v.spl_metric_id = m.id
JOIN project p
  ON p.id = rt.projectId
WHERE m.name = 'TDEGMEAN'
AND p.name = 'test'
AND m.scope = 'ScopeRelease'
```

Listing B.12: NOFC – Number Of Feature Constants (average)

```sql
SELECT rt.tag revision, AVG(v.value) NOFC
FROM spl_variable v
JOIN spl_metric m
  ON v.spl_metric_id = m.id
JOIN spl_file_instance fi
  ON v.spl_file_instance_id = fi.id
JOIN release_timeline rt
  ON rt.id = fi.release_timeline_id
JOIN spl_file f
  ON fi.file_id = f.id
JOIN project p
  ON p.id = f.project_id
WHERE p.name = 'test'
AND m.name = 'NOFC'
AND v.value > 0 -- if NOFC is zero, no FC is existent -> sort out this data!
GROUP BY rt.tag
```

Listing B.13: NOFC – Number Of Feature Constants (std. deviation)

```sql
SELECT rt.tag revision, STDDEV_POP(v.value) NOFC
FROM spl_variable v
JOIN spl_metric m
  ON v.spl_metric_id = m.id
JOIN spl_file_instance fi
  ON v.spl_file_instance_id = fi.id
JOIN release_timeline rt
  ON rt.id = fi.release_timeline_id
JOIN spl_file f
  ON fi.file_id = f.id
JOIN project p
  ON p.id = f.project_id
WHERE p.name = 'test'
AND m.name = 'NOFC'
AND v.value > 0 -- if NOFC is zero, no FC is existent -> sort out this data!
GROUP BY rt.tag
```

C DVD Contents

Contents of the attached DVD. Folders are direct subfolders of the DVD root.

example-projects The CPPSTATS metrics data used in this thesis as CSV files.

git Git repositories include complete history. In the top level directory, exported archives of the master branch can be found.

- **codeface** The original CODEFACE project.
- **codeface-fork** The fork of CODEFACE that has been developed in this thesis.
- **shiny-minimal-example** Minimal examples and experiments referenced in this thesis.

sql Structured Query Language (SQL) dump of the extended schema together with a complete import of all sample projects.

virtual-machine A virtual machine disk image with startup instructions for the Kernel-based Virtual Machine (KVM). It includes the current master branch of codeface-fork and has already imported all example projects.

Bibliography

[1] Stephen Anderson. *Seductive Interaction Design: Creating Playful, Fun, and Effective User Experiences.* New Riders, 1 edition, 2011.

[2] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation.* Springer Berlin Heidelberg, 2013.

[3] Sven Apel, Claus Hunsen, Christian Kästner, and Olaf Leßenich. Pythia - Techniques and Prediction Models for Sustainable Product-Line Engineering. Technical report, University of Passau, 2013.

[4] Sven Apel and Christian Kästner. Analysis Techniques and Prediction Models for Sustainable Product-Line Engineering (PYTHIA). Proposal for a research grant, November 2011.

[5] Sven Apel, Christian Kästner, and Gunter Saake. Software Product Line Evolution. Lecture Slides, 2013.

[6] R. A. Becker, J. M. Chambers, and A. R. Wilks. *The new S language.* Pacific Grove, Ca.: Wadsworth & Brooks, 1988, 1988.

[7] Robin Berjon, Steve Faulkner, Edward O'Connor, Silvia Pfeiffer, Erika Doyle Navara, and Travis Leithead. HTML5 - A vocabulary and associated APIs for HTML and XHTML. Candidate recommendation, W3C, April 2014. `http://www.w3.org/TR/2014/CR-html5-20140429/`.

[8] G. Bockle, P. Clements, John D. McGregor, Dirk Muthig, and K. Schmid. Calculating ROI for software product lines. *Software, IEEE*, 21(3):23–31, May 2004.

[9] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Addison-Wesley, 2004.

[10] N.E. Fenton and M. Neil. A critique of software defect prediction models. *Software Engineering, IEEE Transactions on*, 25(5):675–689, Sep 1999.

[11] David Flanagan. *JavaScript. The Definitive Guide.* O'Reilly Media, 6th rev. edition, 2011.

[12] T. Gyimothy, R. Ferenc, and I Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *Software Engineering, IEEE Transactions on*, 31(10):897–910, Oct 2005.

[13] Harri Hämäläinen. HTML5: WebSockets. Technical report, Aalto University, Department of Media Technology, 2012.

[14] Patrick Heymans, Quentin Boucher, Andreas Classen, Arnaud Bourdoux, and Laurent Demonceau. A Code Tagging Approach to Software Product Line Development: An Application to Satellite Communication Libraries. *International Journal on Software Tools for Technology Transfer*, June 2012.

[15] T.M. Khoshgoftaar and J.C. Munson. Predicting software development errors using software complexity metrics. *Selected Areas in Communications, IEEE Journal on*, 8(2):253–261, Feb 1990.

[16] Barbara A. Kitchenham. What's up with software metrics? - A preliminary mapping study. *Journal of Systems and Software*, pages 37–51, 2010.

[17] Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. Scalable analysis of variable software. In Bertrand Meyer, Luciano Baresi, and Mira Mezini, editors, *ESEC/SIGSOFT FSE*, pages 81–91. ACM, 2013.

[18] R. Lincke, T. Gutzmann, and W. Löwe. Software Quality Prediction Models Compared. In *Quality Software (QSIC), 2010 10th International Conference on*, pages 82–91, July 2010.

[19] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wąsowski. Evolution of the Linux kernel variability model. In *Software Product Lines: Going Beyond*, pages 136–150. Springer Berlin Heidelberg, 2010.

[20] N. Matloff and N.S. Matloff. *The Art of R Programming: A Tour of Statistical Software Design.* No Starch Press, 2011.

[21] Paul Murrell. *R Graphics.* Chapman & Hall/CRC The R Series. Taylor & Francis, second edition edition, 2011.

[22] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining Metrics to Predict Component Failures. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 452–461, New York, NY, USA, 2006. ACM.

[23] Armstrong Nhlabatsi, Robin Laney, and Bashar Nuseibeh. Feature interaction as a context sharing problem. In *International Conference on Feature Interactions*.

[24] Leonardo Passos, Krzysztof Czarnecki, Sven Apel, Andrzej Wąsowski, Christian Kästner, and Jianmei Guo. Feature-oriented Software Evolution. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, VaMoS '13, pages 17:1–17:8, New York, NY, USA, 2013. ACM.

[25] R Core Team. *R: A Language and Environment for Statistical Computing.* `http://www.R-project.org/`.

[26] V. Ryan, S. Seligman, and R. Lee. Schema for Representing Java(tm) Objects in an LDAP Directory. RFC 2713 (Informational), October 1999.

[27] Pamela Zave. An experiment in feature engineering. In *Programming methodology.* Springer, 2003.

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbst verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich habe die Arbeit nicht in gleicher oder ähnlicher Form bei einer anderen Prüfungsbehörde vorgelegt.

Passau, August 2014