Sapphire Back-End

A Web-Based Course Grading Management System

Thomas Kriechbaumer

Sapphire Back-End

A Web-Based Course Grading Management System

Bachelor's Thesis

at

Graz University of Technology

submitted by

Thomas Kriechbaumer

Institute for Information Systems and Computer Media (IICM),
Graz University of Technology
A-8010 Graz, Austria

28 February 2014

Advisor: Ao.Univ.-Prof. Dr. Keith Andrews

Sapphire Back-End

Ein webbasiertes Verwaltungs- und Bewertungssystem für Lehrveranstaltungen

Bachelorarbeit

an der

Technischen Universität Graz

vorgelegt von

Thomas Kriechbaumer

Institut für Informationssysteme und Computer Medien (IICM),
Technische Universität Graz
A-8010 Graz

28. Februar 2014

Diese Arbeit ist in englischer Sprache verfasst.

Begutachter: Ao.Univ.-Prof. Dr. Keith Andrews

Abstract

Teaching a course for hundreds of students is a time-consuming task, considering the preparation time for the classes and exercises and the time for grading. Reviewing and evaluating each student's submissions can take up a great deal of resources and manpower. Spreadsheet-based approaches for managing and grading students, exercises, and points reach their limits of efficiency and scalability. This thesis describes the back-end of the Sapphire project, a web application designed to reduce the amount of effort and time it takes to grade hundreds of students.

Sapphire provides an intuitive interface for lecturers, tutors, and students to manage, submit, grade, and review exercise submissions. The main subsystems are an automated importer for all student registrations in a single course each term, a points overview page to present the results of each exercise to the students, giving them feedback on their performance, and a unified evaluation subsystem which allows tutors to easily review and evaluate a single submission based on a list of ratings.

This thesis focuses on the back-end of the individual subsystems, as well as the deployment and testing of the whole code base. The overall concept, the model-based relations between the data, and the implementation of Sapphire as a Ruby on Rails web application are documented and explained.

Kurzfassung

Das Abhalten von Massenlehrveranstaltungen mit mehreren hundert Studenten ist ein zeitintensives Unterfangen was die Vorbereitung für die Unterrichtseinheiten und Übungen betrifft. Kontrolle und Beurteilung der Studentenabgaben beanspruchen viel Arbeitszeit. Aktuell eingesetzte Abläufe, basierend auf computergestützter Tabellenkalkulation, für das Verwalten und Beurteilen der Studenten, Übungen und Gesamtpunkteanzahl stoßen aufgrund der limitierten Flexibilität der eingesetzten Technologie an ein Limit in Bezug auf Effizienz und Skalierbarkeit. Diese Arbeit beschreibt das Fundament des Sapphire-Projekts, einer Webapplikation zur Vereinfachung der oben beschrieben Arbeitsabläufe für das Beurteilen von mehreren hundert Studenten.

Sapphire stellt dem Benutzer eine einfache und intuitiv zu bedienende Weboberfläche zur Verfügung, welche sowohl von Lehrveranstaltungsleitern, Tutoren, als auch Studenten selbst verwendet werden kann, um Übungsabgaben zu verwalten, einzureichen oder zu beurteilen. Die Hauptkomponenten sind ein automatisierter Datenimport für Studentenregistrierungen zu einer Lehrveranstaltung und einem Semester, einer Punkteübersicht mit den erreichten Punkten pro Übung und Student, welche ihnen Rückmeldung über die erbrachte Leistung liefert, sowie eine Beurteilungskomponente welche den Tutoren erlaubt auf einfachem Wege die Studentenabgaben zu prüfen, bewerten und anhand einer Liste von Beurteilungskriterien mit Punkten zu versehen.

Diese Arbeit legt das Hauptaugenmerk auf den funktionalen Unterbau der unterschiedlichen Einzelkomponenten, sowie die Entwicklung, Spezifikation und das Testen des gesamtem Programmcodes. Das abstrakte Konzept, die modellbasierten Relationen zwischen den Daten und der Implementierung von Sapphire als Ruby on Rails Webapplikation, wird dokumentiert und beschrieben.

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Place	Date	Signature

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Ort	Datum	Unterschrift

Contents

List of Figures

List of Listings

Abbreviations

AJAX Asynchronous JavaScript and XML. 36

API Application Programming Interface. 41

BDD Behaviour-Driven Development. 25, 26

CD Compact Disc. 12

CSS Cascading Style Sheet. 13, 42

CSV Comma-Separated Value. 29, 30, 35–37

DSL Domain-Specific Language. 8, 9, 26

DVD Digital Versatile Disc. 12

ERB Embedded Ruby. 7, 9

HTML Hyper Text Markup Language. 7–9, 13, 36

HTTP Hypertext Transfer Protocol. 8, 9, 35, 36, 43

HTTPS Hypertext Transfer Protocol Secure. 33

INM Internet and New Media. 13

JS JavaScript. 13

JSON JavaScript Object Object Notation. 36

MVC Model-View-Controller. 1, 7

PORO Plain Old Ruby Object. 15, 16

RegEx regular expression. 27

RVM Ruby Version Manager. 5, 33

SQL Structured Query Language. 8, 9

SSH Secure Shell. 31, 33

STI Single-Table-Inheritance. v, 15, 17

TDD Test-Driven Development. 25

URL Uniform Resource Locator. 8

Acknowledgements

I am indebted to my colleagues at the university and friends on campus who have provided me with a constant never-ending stream of cookies, coffee and well meant advice during the whole time of implementing the Sapphire project and writing this thesis. A special acknowledgment goes to my advisor, Keith Andrews, for his essential annotations during drafting sessions for the Sapphire subsystems. I also want to thank him for his endless amount of effort in proofreading this thesis.

Special mention goes to Matthias Link for his help on the Sapphire project and especially for giving a great deal of useful tips during the implementation process. I also want to thank him for pushing the initial idea of Sapphire and kindling my new love for Ruby and the Rails web framework.

Last but not least, I want to thank all the people who have not complained about the endless monologues I gave when asking for help.

Thomas Kriechbaumer
Graz, Austria, February 2014

Credits

In particular I would like to thank the following persons for their work and contributions:

- Matthias Link for his constant effort and work on the Sapphire project. Also for the joint work on the Chapters 2, 3 and 4.
- Keith Andrews for providing a simple thesis skeleton [Andrews, 2012], which I used to derive my own LaTeX template.

Chapter 1

Introduction

This bachelor's thesis explains the architecture of Sapphire, a web-based system for course management and student grading. The Sapphire core and its underlying technology are described in Chapters 2, 3, and 4. These chapters were jointly written by the author and Matthias Link. In essence, Thomas Kriechbaumer, the author of this thesis developed the back-end of the Sapphire system, while Matthias Link developed the front-end, although of course there is necessarily a certain amount of overlap and interaction between the two parts.

Chapters 2 and 3 present the programming language Ruby and the top-level Model-View-Controller (MVC) web framework Rails. Chapter 4 introduces Sapphire, with a short analysis of existing systems and an evaluation of the currently used spreadsheet-based workflows.

The following chapters provide detailed technical insight into the implementation and the concepts of the Sapphire web application. The data structures and associations between the logical subsystems are explained in Chapter 5. User-based account registrations and student groups are discussed in Chapter 6 with additional emphasis on the *one account for one user* concept. Chapter 7 covers the engineering requirements, the creation of a specification, and the *test-first* approach as a documentation substitute.

Selected details for specific subsystems within the Sapphire application are described in Chapter 8. Finally, Chapter 9 outlines the current implementation status and gives a comprehensive list of features to be implemented in future development cycles.

For future developers, Appendix A provides an overview of the Sapphire development workflow.

Chapter 2

Ruby: The Language

During the planning phase of every software project, one of the first questions is the determination of the programming language to be used for each system under consideration. In the early days of software applications, a common choice was a compiled language like C, C++, or Java. During the last decade, scripting-based languages gained more and more momentum within the community. In the specialised case of web development, scripting languages have been used since the beginning: Perl and PHP have been a commonly used.

Due to recent enhancements and runtime speed improvements, high-level scripting languages like Python and Ruby are now used in web development for server-side application logic. This chapter describes the scripting language Ruby, its reusable software libraries, and a typical Ruby development environment with the most widely used tools and helper applications.

2.1 Language Design

Ruby is a strictly object-oriented programming language with strong dynamic, reflective, and almost no functional paradigms. The language and the standard library are not restricted to simple scripting tasks, instead deep integration into the operating system is possible, allowing the developer to use Ruby in a general purpose fashion. It must be distinguished between the software programming language described here [Thomas, Hunt, and C. Fowler, 2013], and the hardware design and notion specification developed at the Oxford University also called Ruby [Jones, 2014].

Yukihiro "Matz" Matsumoto created the initial Ruby language in the early 1990's in Japan [Smyth, 2014]. Due to its easy-to-understand syntax, garbage collection, and the strong dynamic approach with duck-typing, the Ruby language gained popularity very fast around the year 2000, also opening up to the English-speaking community. The Ruby interpreter runs on every modern platform. The language core application is written in plain C, and is therefore easily portable to any architecture. Tanaka *et al.* used this approach to create a Ruby interpreter running on an embedded system [Tanaka, Matsumoto, and Arimori, 2011].

The feature set of the language allows a simple approach to meta programming, with the creation of metaclasses and mixins while also honouring the well known inheritance of classes as described in [Cuadrado and Molina, 2007].

The syntax of Ruby provides a type-less interface to all variables and objects using the duck-typing principle. Using a dynamic-typing approach leads to the loss of explicit information and metadata. The need for static-typing systems might occur and can only be addressed in a more complex and abstract combination of various techniques. Type-safe usage might for example be advantagous during interaction with a database. Therefore, type checking systems can be

implemented [An, Chaudhuri, and Foster, 2009] to ensure correct behaviour as expected by the developer.

A special feature are so called *blocks*. A *block* describes a set of instructions and in general consists of a few lines of code. This allows the developer to easily use anonymous code execution in a different context. Variables can be passed into the *block* like arguments for a function. This feature is compareable to lambda functions [Günther and Fischer, 2010], which can also be realised with a derivative of the block syntax.

2.2 Gems

Most programming languages have the capability to serve small amounts of code to the user on demand. In most environments and communities this functionality is provided as packages contained, distributed, and installed with a package manager. Python has EasyInstall and pip projects to achieve the mentioned tasks. Node.js makes use of the npm utility. There are many different package managers, each well-tuned to the specific needs of a programming environment and language.

In the Ruby world such small, reusable, independent and easily installable software code libraries are called gems. The default gem manager, as well as the corresponding web site, is called RubyGems. It is pre-installed in every default Ruby environment and makes use of the flexibility and dynamic of the language itself. The web site offers to download each gem in the latest version, while also providing a fallback solution for older versions, if still needed by a project.

The most convenient way of managing the dependencies of a given project is by making use of a so-called `Gemfile`, which holds a list of required gems, and optionally their version information. The Bundler gem can read this list and install all missing gems or update all gems automatically. This ensures a very easy workflow for Rails applications during deployment. By invoking the bundler command, all gems are installed: in the correct version and native extensions are even compiled during the installation process.

2.3 Environment

During a development workflow, it is often important to obtain the latest version of specific gems or even an upcoming or almost deprecated version of Ruby itself. To ease this switching process, the community came up with the concept of Ruby environment management tools. These little helper tools take care of downloading, installing and switching between different versions of Ruby, changing a gemset or simply maintaining the current stable release of each library and the Ruby interpreter.

Most Unix-based operating systems, like Linux in all its different flavours, Mac OS X and FreeBSD, ship with a Ruby version which is not state-of-the-art. This creates the strong need for an environment manager, allowing developers to obtain the latest Ruby version, as well as different implementation of the Ruby language such as the standard MRI Ruby, Rubinius or JRuby.

This allows developers to work on different projects with different requirements concerning the Ruby version. The environment manager takes care of switching the Ruby version, exchanging the installed gems, and setting up matching links and binaries for the developer to use.

2.3.1 Ruby Version Manager

The Ruby Version Manager (RVM) is designed to manage multiple installations of different Ruby interpreters and the related gemset and toolchain. RVM setup can be done on a per user basis, as well as a system-wide configuration. This helper tool allows the user to define a default Ruby version to be used every time a new shell sessions is launched. It is possible to switch between different gemsets, a collection of gems of a predefined version.

RVM is designed as a collection of shell scripts loaded into an active shell session. The Ruby versions are installed into a predefined location which is added to the environment variable `PATH` for the shell to access the binaries. The installation can be tricky in a shared environment, due to different permissions in system-wide directories.

2.3.2 rbenv

The rbenv project provides a more application-centric abstraction layer for managing different Ruby versions. It is designed around the idea of using a specific environment for a specific application and therefore provides multiple configuration options and ways of persisting these for the user. Many developers choose rbenv over RVM because of the better integration in a collaborative environment for multiple users to share the same Ruby version and environment settings.

The key concept of rbenv are so-called shims, which are basically catch-all clauses for the `PATH` environment variable. A shim must be prepended to the already existing directories in the `PATH`. This allows the manager to take control over all Ruby related executables and insert them into the call hierarchy.

In addition to shims, the gemset workflow in rbenv makes use of the already existing Bundler application, allowing the user to specify the gems and optionally a specific version number of each gem. The need for loading multiple files into the developer's shell sessions is superfluous, because everything is already managed with the correctly placed shim.

Chapter 3

Rails: The Web Framework

Rails was originally part of 37signal's basecamp application, which was developed by David Heinemeier Hanson. Its core was extracted and released as open source software in 2004 [Ruby, Thomas, and Hansson, 2013]. This chapter describes the underlying concepts of Rails, mainly the Model-View-Controller (MVC) pattern, the web stack, and railties.

3.1 Model-View-Controller Pattern

The MVC pattern [M. Fowler, 2002, pp. 330] is one of the core concepts of Rails, which affects the whole structure of Rails applications. There are three main parts: Models, Views and Controllers.

- **Models:** Models contain the data of the web application, which typically are stored in a database, such as MySQL or PostgreSQL. Models themselves can be interconnected by typical relational database connectors such as 1:N mappings. A model contains business logic as well as methods needed to modify its related data.

- **Views:** Views are required to present the data stored in the models appropriately to the user. Views in Rails are written in Embedded Ruby (ERB), which basically is Hyper Text Markup Language (HTML) with the addition of ERB tags, which look like this `<%...%>`. The content of each ERB tag is evaluated by the Ruby language and the result is inserted into the HTML instead of the tag itself.

- **Controllers:** Controllers prepare the models required for a certain action and pass the data on to the view layer, where it is rendered by the rendering logic, typically a templating engine. The controller is the first part of the pipeline and connects the models with the views.

3.2 Rails Web Stack

This section describes the web stack, when using Rails. When a request is sent to a Rails application, a web server handles the request and calls the web application. The following section describes how a request is handled within the framwork.

3.2.1 Rack

Rack defines a simple interface for Ruby-based web applications to communicate with the web server. It is basically an array, where the first entry is the Hypertext Transfer Protocol (HTTP) status code, followed by a hash of HTTP headers and lastly by the HTML body. This interface is designed to be lightweight while providing an easy way to provide middleware, which will be described in the next section.

3.2.2 Middleware

A request to a web application is not processed directly by the application itself, but first passes thorugh a stack of preprocessors, called *middleware*. It is called middleware because, as the name suggests, it is situated between the server and the application. Rails' middleware stack is rather large, consisting of at least 21 individual parts, called frames, by default. It serves several purposes, such as logging the request, parsing the HTTP request, preparing sessions, and caching database queries.

3.2.3 Routing

After the request has been propagated through the middleware stack, it has to be serviced by the application. Since a Rails application usually consists of more than one controller, the request has to be associated with the corresponding one. For every Rails application a special file, written in plain Ruby, exists where all mappings between requests and controllers are stored. These routes also define the Uniform Resource Locator (URL) schema and paths under which specific resources and pages are accessible. The router is able to take parameters, such as HTTP headers, the requested path, or subdomains into account and instantiate and execute the appropriate controller as requested.

3.2.4 Serving The Request

After the controller has been instantiated, an action, which is also determined by the request, is called on the controller. This method then either constructs a response using models and views, or just redirects the user to another location (such as a login page). When the response has been created, it passes back through the middleware stack, which is realised by recursive method calls, to the web server. The web server then sends the data back to the user.

3.3 Railties

Rails itself is split into parts, which are called *railties*. A railtie provides initialisers and hooks in order to extend the framework's functionality. Every component of Rails is a railtie, which makes them easily extendable and exchangeable. The following section describes the core railties, which were heavily used throughout the Sapphire project.

3.3.1 ActiveRecord

The standard way of achieving data persistence and communication with a database is ActiveRecord. This railtie implements the ActiveRecord pattern, which originally was introduced by Martin Fowler in 2003 [M. Fowler, 2002, pp. 160]. It provides an easy-to-understand Domain-Specific Language (DSL) by not only mapping corresponding columns to method names, but also the model's associations. As a result, a developer does not have to write Structured Query

Language (SQL) queries by hand, but ActiveRecord compiles them to suitable queries for the current database software back-end.

ActiveRecord is able to handle so-called scopes. These are small parts of the DSL which are defined directly on the model. They are often used to provide mechanisms for filtering, ordering or grouping records. Combining scopes is also possible, so reusing scopes is highly encouraged and leads to very concise code.

Usually, the development process takes place at the developer's computer, while the production environment stays untouched. This is necessary to prevent any data loss on the servers. Eventually, as development of the web application progresses, the developer will change the schema of the database. Migrations are ActiveRecord's solution to this problem. These small files, which provide an incremental history of changes to the database schema, are stored in the project and are used to setup an empty database, or migrate an existing database with all its tables and content to the same schema state. As a database is migrated, ActiveRecord first checks which migrations have already been executed, and which are still pending. Even complex changes to the schema, as well as migration of data, can be tested before running them in the production environment.

ActiveRecord was designed to support the three relational SQL databases, namely MySQL, PostgreSQL, and SQLite. Due to migrations and the simple DSL, switching between databases is easy, since no SQL query has to be touched and the corresponding SQL syntax is chosen automatically. Similar to other railties, ActiveRecord is completely independent from Rails and its features can be used outside of Rails without any drawbacks.

3.3.2 ActionPack

ActionPack combines ActionController and ActionView, two railties which are tightly linked together. ActionController provides basic controller behaviour, while ActionView is responsible for the view-layer of a Rails application.

ActionController provides a basic controller, from which all application controllers are derived. It provides basic functionalities, such as request hooks, basic HTTP authentication and redirect handling. On every request, a separate ActionController is instantiated, according to the routing mechanism discussed in Section 3.2.3. An ActionController passes all of its instance variables to the view-layer, where the data is rendered accordingly. A controller calls methods on various model objects and then presents the results to the user by rendering a view template.

An ActionView is usually an ERB file. It is used to generate the appropriate HTML, which will be served to the user. It uses the instance variables, previously defined by the controller, to insert data at specific places in the HTML structure. ActionView also provides helpers for frequently used functionality, such as date formatting, translation and number formatting. The developer can create additional helper methods to simplify the ERB markup and extract all logic into Ruby code, leaving only simple render calls inside the ERB tags.

Chapter 4

Sapphire

At a university there are many courses where students have to submit exercises to the lecturer for assessment and receive a grade based on the performance in these assessments. The task of evaluating, reviewing, and grading a mass course with hundreds of students can be quite time-consuming. The sheer number of students requires a simple, yet powerful and flexible system to manage the various tasks and exercise-specific details of a given course. This chapter describes the architecture of the Sapphire system.

4.1 A New Approach: Sapphire

In Sapphire students can use a web-based submission system to hand-in exercises. After the exercises have been assessed, each student is able to review their points and deductions. This allows the student to obtain speedy feedback.

The lecturer is ultimately responsible for each student's grades. Often, teaching assistants (tutors) assist the lecturer in the grading of exercises. Sapphire uses a points deduction framework, in which a tutor generally only has to assess whether a particular criterion applies or not (binary decision) and the lecturer is responsible for setting the amount of deduction (number of points) corresponding to each criterion.

Tutors have an easy-to-use interface from which each submission of each student in a specified tutorial group is accessible and can be quickly assessed. Automated tasks and checks can also be configured, in order to reduce the workload for each tutor.

4.2 Current Workflow with Spreadsheets

In the current grading workflow, each tutor uses a spreadsheet consisting of several worksheets: one for each exercise, one overview, and an additional sheet for adding students, exported from TUGRAZonline [ZID, 2014]. The exercise worksheets are formatted as follows: The first two columns contain the ratings and corresponding point deductions, grouped into rating groups. Each rating group provides a specific number of points. Each rating indicates a criterion which a student commonly does wrong, resulting in a deduction from the points total for this rating group. The points of a rating group cannot be less than zero. The first row of an exercise worksheet consists of the students' names, sorted by surnames then forename.

At the beginning of each course, one tutor copies the spreadsheet from a previous instance of the course and clears out all evaluations and students. Then the tutor sends out this spreadsheet

to all colleagues. Each of them inserts their students into the spreadsheet and prepares the spreadsheets for the current term.

During the course, several exercises have to be done by the students and are then submitted through several different mechanisms, for example by posting to a newsgroup, uploading files to a server or simply sending an email to their tutor. The tutor has to gather and evaluate all submissions. Doing so involves multiple specific tools and grabbers, as well as automated checkers. When a tutor finds an error in a student's submission, this is noted with an "`x`" in the appropriate cell of the spreadsheet.

At the end of each course, every tutor combines all submissions, renames them to a specific naming scheme, burns them onto a Compact Disc (CD) or Digital Versatile Disc (DVD) and hands them in to the lecturer along with the final spreadsheet containing the points and grades of all students.

4.3 Key Advantages of using Sapphire

The goal of Sapphire is to improve the workflow mentioned above as a whole. Several improvements have been made. The first step of improving the grading process, was to take away the initial setup time, by introducing a simple mechanism to duplicate all exercises and their respective rating groups and ratings, which strips away hours of work cleaning up the spreadsheet and bug fixing.

Furthermore, "integrated evaluations" were introduced, where rating are displayed alongside submissions on a single page per exercise and student. By leaving out the overhead of finding the correct submissions and files and stepping by away from the classic table approach, a simpler form of grading students is provided. The risk of a tutor grading the wrong student, by accidentally opening a different submission, is eliminated. The tabular overview of ratings and points for each student is still provided as a separate overview page.

Grading on several different devices is now supported, since Sapphire is able to display all submissions appropriately, without the need of additional applications other than a modern web browser.

Another benefit of using a web-based application is its flexibility. While releasing intermediate results on a large scale was impossible while using Excel spreadsheets, Sapphire enables the course leader to publish results on a per exercise basis, while maintaining students' privacy, as it is only possible for them to view their own results.

4.4 Subsystems of Sapphire

Sapphire consists of several submodules, each serving a particular purpose. This section covers two important parts of Sapphire: the submission viewers, mainly used by tutors to view submitted files directly in Sapphire, and the points overview, which provides early feedback to the students for the active course.

4.4.1 Submission Viewers for Tutors

Submission viewers are one of the key advantages of Sapphire. They provide easy access to submitted files and allow the tutors to quickly open submissions, without having to worry about where these files are located in the file system.

A viewer is not displayed in the usual Sapphire layout in order to present the submission accurately and without distraction. Furthermore, a viewer provides information about the current

submission, such as the name and the matriculation number of the student whose submission is currently displayed. Additionally it provides an easy way of switching between the different files of a single submission, realised using a select box and JavaScript (JS). The information is shown in a transparent overlay located at the top right corner of the page, which becomes opaque when hovering over it.

The rest of this section is about the internals of two selected submission viewers: the web site viewer and the Cascading Style Sheet (CSS) viewer.

4.4.1.1 Web Site Viewer

The web site viewer is responsible for displaying web sites submitted for Exercise 4 of Internet and New Media (INM) [Andrews, 2014a] including HTML, CSS files, and images. Since the path to these resources is different in Sapphire than it is on the students' web space, we have to rewrite all link and anchor-tags in order to link them correctly to Sapphire's internal serving mechanisms.

Therefore, before displaying the submission to the tutor, the HTML files created by the students are parsed and all links to relatively linked assets, such as images and link tags, are replaced. Furthermore, the destinations of relatively linked anchor tags are altered to match those of the viewer. After all necessary links have been changed to match Sapphire's convention, the head and the body sections of the HTML file are extracted and placed separately into a specially crafted HTML template, to simplify later injection of further styles and HTML tags, such as the overlay. For this particular viewer, the overlay allows the tutor to easily see which further HTML pages were submitted by the student and to switch between them if desired.

4.4.1.2 CSS Viewer

The CSS viewer, used for Exercise 5 of INM [Andrews, 2014b], displays a web site with a predefined set of stylesheets the tutor can activate. Since the lecturer provides a specific HTML file to be styled by the students, the viewer does not alter the contents of the submitted files, but renders the contents of the stylesheets inline into the HTML grid. This both reduces the number of server requests and enables stylesheets to be made interchangeable via JavaScript.

The viewer provides the basic HTML file and renders the stylesheets inline into special containers, which by default do not affect the styling of the HTML template at all. Additionally, some simple JS is injected, which loads the first stylesheet as soon as the tutor opens the view. The overlay allows switching the stylesheets to accommodate the tutor's needs, and the JS replaces the previous stylesheet with a new one by replacing the contents of a special style tag. Hence, stylesheets do not interfere with one another, while additionally providing the possibility of quickly switching between stylesheets, without having to load the page over and over again.

4.4.2 Points Overview for Students

Evaluating submissions from over 400 students for a mass course like INM takes a great deal of time and effort, even if the workload is split up between multiple tutors, which breaks it down to around 60-80 students per tutor (for the last few years). Typically exercises are given out faster than the tutors are able to finish evaluating the previous exercise submissions. Therefore, students do not receive feedback on their already submitted work and have to complete the following exercises in good faith. This creates a certain problem because students might make the same mistake for several exercises in a row. Giving early feedback on the performance of individual students, based on actual submission evaluations, would increase the quality for upcoming exercises and reduce errors.

For each exercise, a tutor needs a certain amount of time to evaluate all the submissions and then at least one meeting with all tutors and the lecturer to discuss any edge cases, in order to provide consistent grading for all tutorial groups. These meetings are currently held on a weekly basis during the term and are mandatory for all tutors.

Using Sapphire, students can obtain incremental feedback on their performance and review their mistakes in the points overview section of Sapphire. The tutor can evaluate submissions and publish the preliminary results directly to the students. Edge cases in particular ratings can be discussed at any time using a commenting system accessible by all tutors. This speeds up the evaluation period for each exercise and allows students to prevent repeating mistakes already made in one of the previous exercises.

All students have access to their personal results in the Sapphire system. The entered data is marked as preliminary as long as there are open issues to discuss with the lecturer and other tutors. Once the lecturer signs off, the results for the particular exercise are marked as provisional and are only changed after the grading review process. The personal student-based points overview is sorted by exercise and shows the submission with all handed in files or other data (newsgroup posts, web sites). A list of mistakes is provided to the student to indicate which requirement of the exercise specification was not met. This prevents the student from making the same mistake multiple times. If a student wants additional information about certain ratings or mistakes, the tutor's contact email address is provided to enable the students to quickly send an email.

The points overview for each student is available as soon as the tutor publishes the first preliminary results for an exercise and remains available for some time after the finalisation of grades. Each student receives login information to access Sapphire and its subsystems. The personal points overview page is considered confidential to a single student and therefore login information is not shared or published to other students. The global points overview, containing only the total number of each grade is available to the lecturer and tutors. In addition, the grading scale, which indicates the points necessary to obtain a particular grade, is provided.

At the weekly meetings of the lecturer and tutors, the publishing schedule will usually be decided. This determines a fixed time slot for each tutor to finish the evaluation tasks for the current exercise and publish the preliminary results. Since each tutor works on their own, the data can be published on a per-tutorial group basis.

Chapter 5

Data Model and Representations

The Sapphire back-end is built around a relational database structure incorporating tables, foreign keys, one-to-many, and many-to-many relations between these tables. Each table represents a logical Sapphire component. Some cases also use Single-Table-Inheritance (STI) to reduce the complexity of the database schema. The production environment is meant to be run with a modern relational database management system such as PostgreSQL [PostgreSQL Global Development Group, 2014] or MariaDB [SkySQL, MariaDB Foundation, 2014].

The next section discusses the basic concept of logical models with a database representation in the persistence layer and also the use of view-only models to ease user interaction with the web stack and the front-end. Section 5.2 explains the abstraction layer regarding ratings, rating groups and exercises, including some general use cases for special types. Section 5.3 discusses the registration and authorisation subsystem and the various types of accounts in Sapphire.

5.1 Data Objects and Relations

The relational data model which is incorporated in the different Sapphire subsystems is outlined in Figure 5.1. The hierarchical layers are used in different contexts throughout the system. The most important components are:

- **Course:** Course is the representation of a course or lecture held at university. A course is typically a recurring event over a specific number of terms.
- **Term:** Term is a single event, corresponding to a course. A term consists of its own set of students, tutor, tutorial groups, and so on.
- **Exercise:** Exercise is a definable set of work each student is responsible for submitting during the duration of a given term.
- **Submission:** Submission is the actual work a student submits for evaluation and grading. A submission might be group work by multiple students and thus added to each student's credit.

The conceptual idea of ratings and rating groups is described in more detail in Section 5.2. The usage of accounts as well as the meaning of student groups is explained in Section 5.3 and in Chapter 6. All the above objects and classes are persisted into the relational database with ActiveRecord, using a very simple and lightweight attribute mapping from database columns to Ruby object attributes. The result is available to the developer as a Plain Old Ruby Object

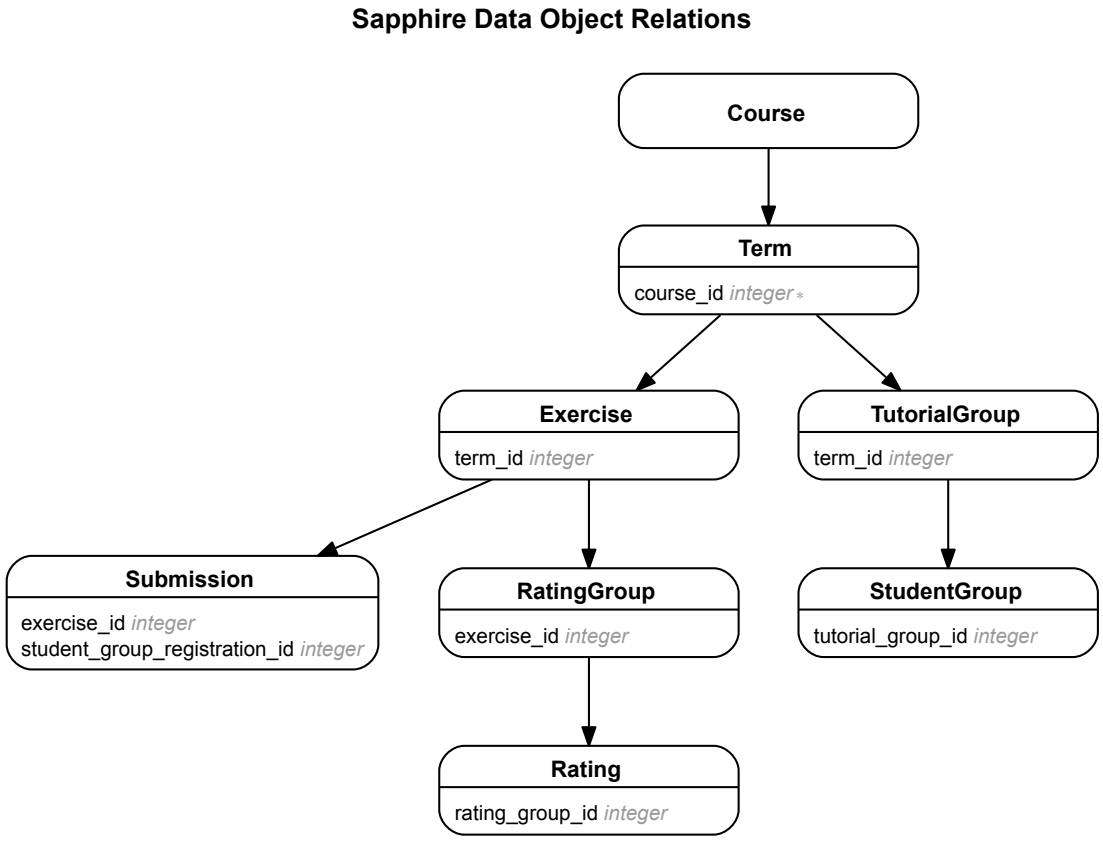

Figure 5.1: The logical hierarchy and relations of objects represented in Sapphire. Important members of each object are shown in order to indicate the relation to other classes.

(PORO), which is, as the name suggests, just a Ruby class instantiated with the requested data stored in it.

In addition to the presented objects, which are persisted in the relational database, there are a number of non-persistent objects used for user interaction. The main purpose of these objects is to ease the mapping from user input, such as parameters, checkboxes and other values, to a simple Ruby object representation. These special models, called `viewmodels`, are not associated with ActiveRecord, but instead only reside in memory during execution of a request. One example of such a `viewmodel` is the `evaluation table` view, which uses a data-aggregation algorithm and stores the requested data fragments in a PORO, without invoking ActiveRecord to establish a persistence layer for this object during runtime.

5.2 Ratings and Evaluations

One of the key subsystems in Sapphire is the evaluation subsystem. This unit allows tutors to evaluate submissions. Based on these evaluations, the final grade for each student is calculated. Ratings and evaluations always exist in pairs. Every rating is embedded within a rating group to introduce a fine-grained separation between certain areas of evaluation.

Figure 5.1 shows part of the object hierarchy for the evaluation subsystem. Every exercise

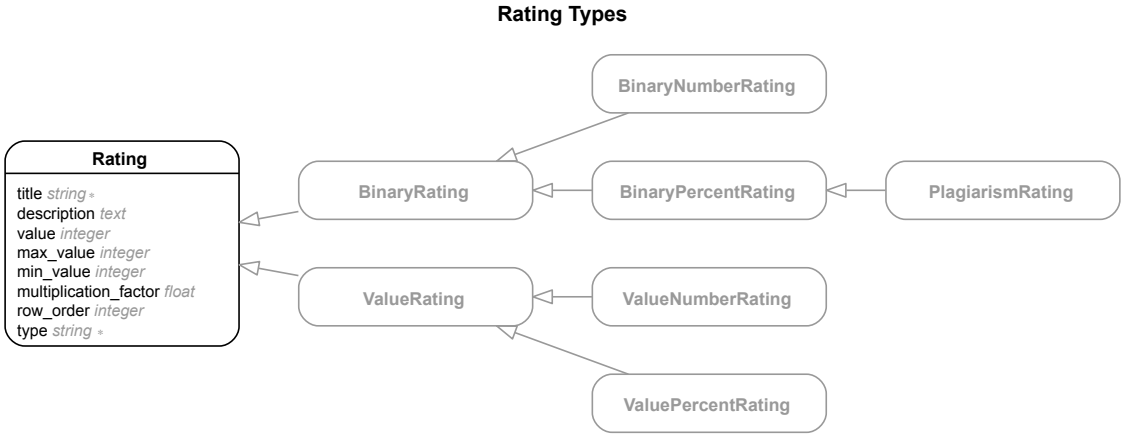

Figure 5.2: Different subtypes of ratings used to remodel the needs of most use cases migrated from the spreadsheets. All subtypes are mapped into a single database table to simplify the access and data collection during points calculations.

has multiple rating groups, each responsible for a substantial part of the exercise grading. The rating group is a collection of individual ratings of different types. Each rating can increase or decrease the points, or contain a percentage which is applied to the overall sum of points, for either the rating group, or the whole exercise.

The hierarchy of different rating types is shown in Figure 5.2. In the relational database, the rating model is represented with STI. This allows the Rails stack to use only one database table, thereby reducing the complexity of queries. The specific type of rating is saved in a special `type` column for every rating entry in the database. ActiveRecord parses this value during loading and then decides which concrete type of model has to be instantiated.

The only rating types presented to the user are the five classes seen on the right side of Figure 5.2. The `Rating` class, as well as the two intermediate classes `BinaryRating` and `ValueRating` are only used for development abstraction purposes. The five concrete rating types are:

- **BinaryNumberRating:** BinaryNumberRating offers the grader a *yes* or *no* choice. If this rating is positively evaluated the specified number of points is added to the total sum of the exercise points. Typically, the sign of the points attribute is negative and therefore the overall points are reduced by the amount of this rating.

- **BinaryPercentRating:** BinaryPercentRating offers the grader a *yes* or *no* choice, however, this rating does not add a certain number of points. The value attribute is a percentage which is applied to the total sum of points. 1.0 means that 100% of the points are given to the student, which is not a very useful value. More meaningful values are 0.5, resulting in only 50% of the total sum of points assigned to the student. In the same manner a value of 1.25 would increase the total sum of points by additional 25%.

- **ValueNumberRating:** ValueNumberRating offers a variable entry field to the grader. The application is analogous to `BinaryNumberRating`, with the additional benefit of letting the tutor choose the value. In general, the rating should be limited to certain upper and lower bounds to only allow inputs between a fixed minimum and maximum number of points. For example, a bonus of 0..5 points.

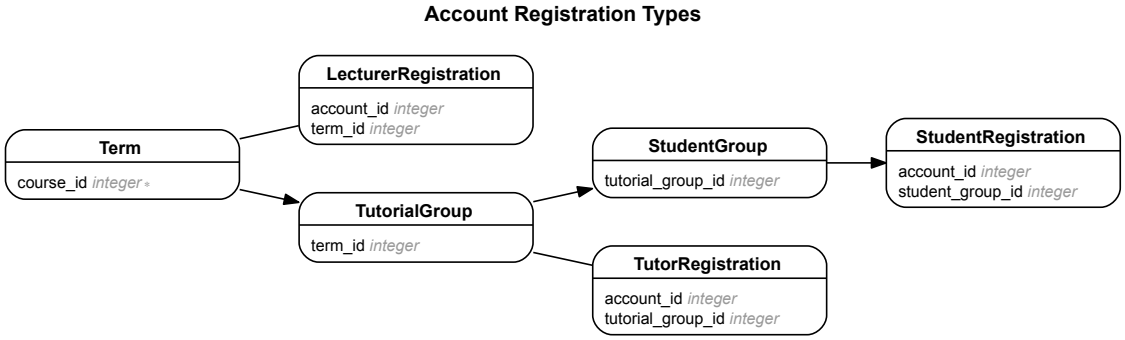

Figure 5.3: Different subtypes of account registrations used to assign roles and priviliges for different Sapphire subsystems. Each registration type is responsible for a certain user access level.

- **ValuePercentRating:** ValuePercentRating offers the grader a variable entry field. The application is analogous to `BinaryPercentRating` with additional lower and upper bounds.
- **PlagiarismRating:** PlagiarismRating is a special kind of rating due to its dependency on the `BinaryPercentRating` class. The title and value of this rating are predefined with *plagiarism* and 100%. The need for this deeper specialisation of rating is to have an indicator for use in the points overview. The overview page renders a special indicator (colour-coded or an annotation) if a submission is a case of plagiarism.

5.3 Account Usage

Sapphire uses a *one account for one person* concept to manage all user-related contextual actions. The simplification of this approach lies in the single-sign-on to the system and subsequent access to all areas which the user is privileged to use. Access control to the different subsystems is governed by a role-based authorisation principle. Every request to view, create, or manipulate data is first sent to the authentication and authorisation subsystem. The requested action is only performed if the assigned role of the current account allows it. Section 6.2 explains student account management in more detail. The current implementation and possible additional functionality is described in Section 9.2.1.

Role-based authorisation is a key concept of every modern system. In Sapphire, the list of different roles can be expanded with little effort and therefore offers the possibility to increase the level of fine-grained access control on a per-user basis. Figure 5.3 gives a high-level overview of the implemented roles, represented as registrations. Every account needs to be registered for a certain role or position in order to gain access to the functionality associated with the specified registration type and role.

The authorisation subsystem currently differentiates between three main roles. Each is represented with a registration entry in the database, merging an account and the corresponding data object from one of the upper abstraction layers:

- **Lecturer:** Lecturer is assigned for a particular term of a course. Full control over every aspect regarding the designated term is granted to this account.
- **Tutor:** Tutor is assigned to a single tutorial group. The permitted actions include evaluating of student submissions, creating new student groups, and registering students to these

groups. Exporting the evaluation results for the designated tutorial group is permitted, as well as viewing evaluation results from other tutorial groups.

- **Student:** Student is a role assigned during import of student registrations to an account. Such an account is allowed to submit exercise submissions and review associated evaluation results.

New registration types can be introduced into the Sapphire authorisation subsystem quite easily. A registration consists only of the account, a corresponding abstract model to subclass from, and a valid name to describe the new role. Due to the loosely coupled architecture, it is possible to assign multiple roles to a single account within the same context. For example, a tutor could also be a lecturer for the same course and term.

Chapter 6

Student Groups and Registration Management

The registration system of Sapphire cascades into multiple layers for each component. Each layer provides its own definition set and uses the upper layers' information to create and manipulate the registration layers below. This architecture was chosen so that one single account per person is created, regardless of the engagement a user has with the Sapphire system. Registrations are a crucial part of the role-based authorisation system described in Section 9.2.1.

6.1 Registering a Student

Most registrations are done automatically during the import from TUGRAZonline into Sapphire (see Section 8.1.3). The system first needs to find a matching account in the database. If there is none, a new account is created and pre-filled with the provided data (at least email, first name, and surname). Once an account is found there are two possibilities. Either the new account is registered for a term where no group submissions are designated, or the account is registered for a student group contained in a term.

Even when students are not working in groups, a solitary group for each student is created. Using such a solitary group, with only one student in it, allows Sapphire to utilise the same underlying logic for every term and exercises as if there were groups. The solitary group concept is explained in more detail in Section 6.5.

6.2 Managing Registrations

Student registrations specify a relation between an account (a user) and a student group. This means that the information about which term or course the student is registered for is stored in the upper layers of the registration system. A student group must be associated with exactly one tutorial group. The tutorial group itself belongs to a term which is the highest layer of registration information. Figure 6.1 shows the overall structure of the objects and data structures involved.

Since a student group belongs to a tutorial group, the Sapphire registration system needs to distinguish between active and inactive groups, because it is possible to regroup students during the term and form new groups, leaving the old groups still active for already taken exercises (past deadline) but at the same time not available for new exercises. This results in the need for

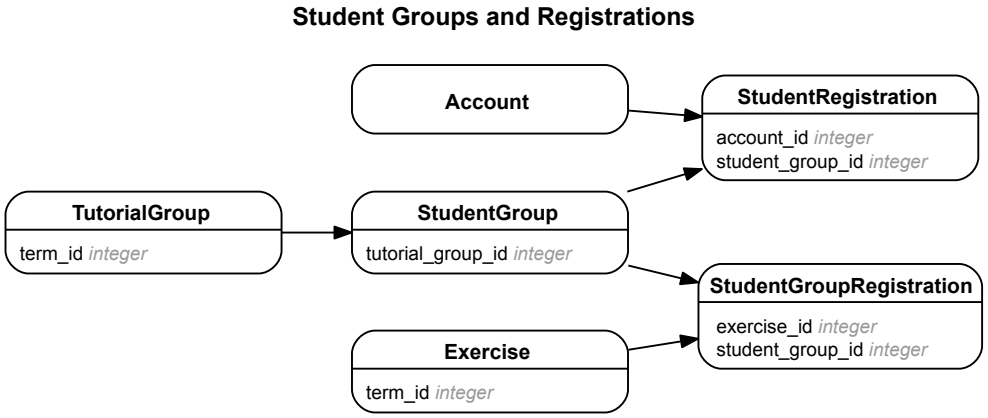

Figure 6.1: Relations between data structures concerning student groups and registrations.

student group registrations which connect a student group with an exercise, marking the group as active for this exercise and awaiting a submission.

In order not to limit the capabilities of the registration subsystem the mapping of student groups and exercises is only created once there is a submission to associate the two database entries. This allows the tutor to permit students to form new groups up until the first submission for each exercise. After that, the student is bound to a student group for that exercise. The student group is registered for the exercise and now has to create their corresponding submission. The submission can also be created by the tutor or by an automated grabber/checker plug-in, in case submissions are gathered externally to Sapphire.

6.3 Cardinality of Registrations

Since a student is able to register for multiple courses or the same course over multiple terms (for example, if they fail in a particular term), the registration subsystem must be capable of handling many-to-many relations between accounts (students) and student groups. Furthermore, it is possible that the same student becomes a tutor in the next term, or is already a tutor in a different course.

The association of all registrations to respective courses must be handled with great care to ensure a safe mapping between the account entries and all types of registration. This mechanism is built into Rails with model definitions to describe `belongs_to`, `has_many`, or `has_one` associations.

6.4 Workflow

The most convenient way of creating student registrations is by importing data from TUGRA-Zonline at the beginning of each term. This will create all accounts or update existing accounts, if they have changed (such as names or field of study) based on the unique matriculation number. Student groups are automatically defined as solitary groups, if there are no exercises which require a group submission.

Manipulation of registration data after a successful import is not recommended, due to the many different data structures which are involved. Adding a single student is currently doable, but requires extensive knowledge of the underlying registration subsystem. This is, of course,

a sub-optimal situation and an improved workflow is proposed in Section 9.2.3, the result of which would be an interface for the tutors and the lecturer. Furthermore, providing a web view to add, change, manipulate, and remove single students or groups from the registration system would improve the outlined workflow considerably.

6.5 Solitary Groups

Sapphire supports courses where students can submit their work as individuals on their own, or as a group. This submission mode can be determined by the lecturer on a per exercise basis. For example, the course *Internet and New Media* consists only of exercises in which every student submits work individually. In the course *Human-Computer Interaction*, most of the exercises are considered group work, and only the multiple choice test is taken and assessed individually.

This leads to the strong need for a distinction between group and single-person submissions. In order to reduce the complexity of this subsystem within Sapphire, the architectural choice was made to introduce the concept of solitary groups. Every submission is performed by a group. Every evaluation effects the whole group. A group consists of at least one and possibly multiple students. If a specific exercise is meant to be completed single students individually, rather than in groups, this exercise is marked accordingly in the `group_submission` property.

During the import of student and registration data, all exercises are checked for this property. If there is at least one exercise within a given term requiring individual submission, then the importer will create a `solitary group` for each student. These groups consist of a single student and are marked as `solitary` for further processing in the evaluation system. These special groups are used as a proxy object every time a single student needs to be evaluated or processed. The view layer of the Rails stack considers the `solitary` property during rendering and switches to a slightly different output if the rendering input contains such groups.

Chapter 7

Specification and Testing

For every large software project, the most important, but also most likely neglected, part is a concrete specification of the goals and the way to achieve them. In addition, the specification needs to be verified before shipping the final product to a customer. This implicitly suggests the requirement for a test suite, where all tests have to be passed prior to a new release.

A collection of automated scripts, tests, and specification verification steps can be managed in many different ways, depending on the programming environment, language, and development structure. In programming theory, there is a rough distinction between three different types of test which can be used to specify and verify the behaviour of a system. Sapphire incorporates the following types of test to ensure correct functionality at a source code level:

- **Unit tests:** Unit tests focus on a very short piece of program logic and define every possible outcome and expected faults. In most languages, every function which is exposed as a public interface is tested with a unit test to ensure the correct functionality. This kind of testing should be limited to a single software unit, therefore every module can be tested separately without the need for interaction between different classes or modules.
- **Integration tests:** Integration tests are used to ensure the interaction between multiple units is correct. Since the correctness of each individual unit is already verified by previous unit tests, an integration test only focuses on the valid interaction between two or more units.
- **Acceptance tests:** Acceptance tests are large-scale verification methods, used to evaluate the interaction of the whole system from user input down to core system libraries. This sort of test is not focused on functions, methods, or modules, but instead tests a single user scenario.

7.1 Behaviour-Driven Development

The principle of Behaviour-Driven Development (BDD) is to define the outcome of a certain interaction with a single software unit accessible by the user [North, 2014]. BDD can be classified a kind of acceptance testing, with slight variations depending on the target group (developer or management personnel).

Prior to the beginning of development of a new feature, the expected behaviour is specified in terms of a set of test cases. Then, the development process runs as long as the test cases for that feature fail and only finishes once all the test cases pass and thus fulfill the specification. Defining a test case before programming was initially introduced in the Test-Driven Development (TDD) paradigm [Beck, 2002], which BDD is closely related to.

7.2 Acceptance Tests with Cucumber

One common tool for a BDD workflow is Cucumber [Wynne and Hellesøy, 2012]. With this tool, all test cases are described in the form of a text-based feature definition. Each feature corresponds to a single software unit which should be tested. The text describing a feature is written in a business-facing manner, not only targeting the developer, but higher-up management personnel.

> "Cucumber allows you to write feature documentation in Plain Text. It means you could sit with your Client or Business Analyst to write down the features to be build on your application."
>
> (Craig R. Webster [Webster, 2014])

In the case of Ruby on Rails usage of Cucumber is quite natural, because its target language is also Ruby, and there are many gems and helper tools for Rails-based applications which can be used with Cucumber.

7.2.1 Feature Definition

A feature text is structured using multiple scenarios, each covering a certain aspect of the same software unit. This allows the programmer to define all test cases related to the same unit in the same file. The definition language is called Gherkin [Behat Project, 2014], which is a very lightweight Domain-Specific Language (DSL), especially designed to meet the requirements of easily readable behavioural test suites. An example of a feature definition in Gherkin can be seen in Listing 7.1.

In every feature file it is recommended to include a short preamble consisting of three lines, which define what its *main purpose* is, *who is going to use it* in the finished product, and what the *expected gain in functionality* of this feature should be. All of these lines are optional and can be considered just a comment, but it is highly recommended to stick to this style of introduction to allow new members of the development team to understand and use the feature definitions straight away.

After the preamble, if it exists, the scenario definitions are outlined. There can be multiple scenarios in the same file, each of which should correspond to the same feature defined in the preamble. Each scenario consists of multiple steps, one per line. Every step is later parsed with a regular expression in the step definition (see Section 7.2.2). Steps are constructed with a simple set of keywords:

- **Given:** Given steps are used to define a condition which must be fulfilled at the beginning of a scenario. This is typically used to create database entries, navigate to a certain page, or other types of preconditions.
- **When:** When steps typically describe an interaction of some kind with the application, like clicking a link or entering data into an input field.
- **Then:** Then steps express the supposed outcome of the previous steps. The result should match the user's expectations.
- **And/But** And/But steps are synonyms to improve readability through a long list of steps.

```
1 Feature: Login
2   In order use Sapphire
3   As a student
4   I have to login and gain access
5
6   Scenario: Basic Access Protection
7     Given I am on the login page
8      When I login as "somebody@example.com" with "my_secret!"
9      Then I should see "Welcome Somebody!"
```

Listing 7.1: A feature definition written in Gherkin.

```
1 When(/^I login as "(.*?)" with "(.*?)"$/) do |email, password|
2   fill_in "account_email", with: email
3   fill_in "account_password", with: password
4   click_button "Sign in"
5 end
```

Listing 7.2: Step definition in Cucumber for a typical login process.

The main objective of each scenario is to obtain a human-readable test case with a wide spectrum of possible steps and edge cases. Liberal usage of the `Given`, `When`, `Then`, `And` and `But` keywords is one of the core concepts of Cucumber.

The syntax of Gherkin specifies that each step is terminated with a line ending character. Depending on the platform, it is either a single line-feed or a combination of line-feed and carriage return. The overall structure of preamble, scenarios, and steps is created through indentation by either tabs or spaces.

In a typical testing environment, it might happen that all scenarios of a single feature share the same initial `Given` steps. In such a case, these steps can be condensed into a single `Background` section. This additional structure is run before each individual scenario in the feature file.

7.2.2 Step Definition

A step is basically a single line of text, starting with one of the described keywords. The line can contain certain placeholders. A regular expression (RegEx) is used to parse the input line and match it against the step definition database. The keyword maps to a Ruby method which is called after parsing. For each step in a scenario, there must be a corresponding step definition somewhere in the application's test library in order to find some code to execute for this step.

The called method receives the parsed placeholders as input parameters, if there are any. Listing 7.2 shows the usage of such a step definition to fill some values into a page. In this case the Ruby syntax for a *block* is used as the execution context, receiving two parameters (email and password).

Be aware that the keywords at the beginning of each step are indistinguishable from each other and therefore, it does not matter which method keyword is used in the feature file. As stated earlier, it is highly recommended to stick to the supposed meaning, in order to prevent confusion for other developers.

Chapter 8

Selected Details of the Implementation

In the following chapter two aspects of Sapphire are described in more detail. Section 8.1 covers the initial steps and mandatory preparation in order to copy data from TUGRAZonline into Sapphire. Section 8.2 describes the setup process for hosting a new Sapphire instance on a dedicated server infrastructure and how to deploy and release new features to a production environment.

8.1 Import TUGRAZOnline Data

Graz University of Technology uses an intranet software system called TURGAZonline for internal management tasks. This includes the registration of individual students to their courses each term. Since this is mandatory from an administration perspective, it creates the need for an automated workflow to transfer this registration data into the Sapphire back-end database. This includes the creation of a Sapphire account for each student and the mapping to tutorial groups.

8.1.1 Student Data Export from TUGRAZonline

There are many different export possibilities from TUGRAZonline, of which the Comma-Separated Value (CSV) option is the most portable way of exporting, considering file format, character encoding, and parsing complexity. An authorised user can trigger such an export from TUGRAZonline with various options. The user obtains a single file with the requested data in it.

8.1.2 Preparation of Comma-Seperated Values

Since TUGRAZonline is constantly being improved and upgraded, the export functionality is subject to change. A few steps are necessary to ensure that the file format is compatible, prior to importing the data into Sapphire (see Section 8.1.3).

The first step is to check that the character encoding of the CSV file is UTF-8. Secondly, the quotation of the data parts must be normalised throughout the file. This step is necessary in order not to confuse the parser with unbalanced quotation marks. This means that each cell should be quoted and all inner quotations must therefore be correctly escaped.

8.1.3 Student Data Import into Sapphire

After the steps described above are successfully accomplished, a tutor or the lecturer is able to import the data set into a specific course and term. If the term in question has not already been used for an import, this functionality is accessible via the dashboard. Otherwise, this feature can be reached through the tutorial groups overview page. The import process is divided into three stages:

Upload: A CSV file of participants can be specified. Some file format specific settings such as the column separator, quote character, or the presence of headers in the first row, can be defined.

Mapping: Mapping is used to determine the meaning of the different CSV columns according to the internal representation in Sapphire. This area makes use of the *Smart Guessing of Mapping* feature described in Section 8.1.3.3.

Import: Import is the actual creation of new database entries from the data in the CSV file mapped as above. This process can take some time. In the current implementation there is no progress indicator to give feedback. For an explanation of this inconvenience, see Section 9.1.2.

8.1.3.1 Data Representation Settings

Within the *upload area*, there is a setting to change the parsing of the group column. The user is able to choose between *tutorial group matching* and *tutorial group and student group matching*. The default for this setting is determined for the current term by looking through all exercises and checking if there is at least one exercise with enabled group submissions. An exercise with enabled group submissions selects the *tutorial group and student group matching* preference.

The regular expression responsible for splitting the tutorial group identifier from the student group identifier can be configured as required. This corresponds to the naming convention used when first entering group names in TUGRAZonline; these names appear in the exported CSV file.

8.1.3.2 Parsing Settings

The import workflow is not limited to data exports from the TUGRAZonline system. The user is provided with options and configuration choices to ease the parsing during upload. Well-formed CSV files often have a column header in the first row to give some meaning to the raw data. If the first row already contains data, then the headers on first line option is able to adjust the parsing behaviour. Additional options are directly related to the parsing of the raw data, such as the capability to change the column separator, quotation character, decimal separator, and thousands separator which are used for numbers.

8.1.3.3 Smart Guessing of Mapping

During the import workflow, the user can assign labels to columns with the appropriate label to column header. To make this process faster for the user, the Sapphire Import Module tries to guess the most likely mapping for the columns by analysing the data values for each column of the CSV file. The labels for email address, matriculation number, and tutorial group can be guessed using a regular expression for multiple rows of the same column. If the RegEx matches all of the data cells for a column, then this label is selected as the current data label. The same

technique can be used to match column headers, if there are any, with a predefined dictionary consisting of the most likely used words.

The result of this automated preselection is then saved to the current import job and presented to the user for review and possible manual correction if necessary. The user can assign a label only once during an import job. If the user re-uses an already assigned label for a different column, then the label jumps to the new column, leaving the old column unmapped and without a label.

8.2 Setup and Deployment

During the development of a new Rails application, it is useful for developers and beta-testers to try out new features on a non-productive system with real data, but with an extended logging system and access protection in case some unexpected errors occur.

Before going live, and before each major feature addon, it is recommended to create a well-defined base system to test and verify the functionality. The current installation of Sapphire consists of two almost identical servers, each configured to host a Rails application.

The first server hosts the production system of Sapphire and is therefore runs with a Rails environment set to `RAILS_ENV = production`. `Production` means that all of the running code should be well-tested and the database is always in an optimal, non-degraded state, and is fully operational.

The second server is used as a staging server before migrating code to the production environment. All the code and workflows should already be well-tested and error free, as far as the development process allows. Running the application in the staging system emulates the real conditions in the production system as far as possible and runs under the `RAILS_ENV = staging` environment. This includes partial real datasets and related configurations. Some changes are required in the staging environment, so as not to interfere with the production system, for example rerouting the automated email delivery to be contained within the staging system and not be sent out to real users.

8.2.1 Deployment with Capistrano

One of the most advanced automation and deployment tools for Rails applications is Capistrano [Capistrano Project, 2014]. This framework is written in Ruby and provides a large number of options, and a feature-rich plugin subsystem for customisation.

The core principle is the chaining of small tasks, where each task is a set of commands to be executed on the remote server, or in some case on the local system. It allows the plug-in subsystem to hook into any existing task with a `before` or `after` hook, which creates a flat tree-like call hierarchy from task to task. One task might be the checkout and extraction of the current HEAD from the version control system Git into the new directory structure. Afterwards there might be some configuration tasks to setup a new application release. Any command that can be executed in a live shell session over Secure Shell (SSH) can also be used in a Capistrano task.

As can be seen in Listing 8.1 the syntax for custom tasks is strongly related to Rake tasks. Rake is a Ruby application used for build tasks and automation like make. The example task is always run in a new deployment before the `deploy:updated` task (see line 12), which is a core Capistrano task. The executed commands simply capture the output of a single command on the remote server and print the result back into the local shell from where the deployment was started. There is a special command to change the current working directory, called the `within`

```
1  namespace :my_tasks do
2    desc "A small example of a custom Capistrano task"
3    task :print_ruby_version do
4      on roles :web do
5        within release_path do
6          ruby_version = caputure("ruby --version")
7          puts "The server is running Ruby in version: #{ruby_version}"
8        end
9      end
10   end
11
12    before 'deploy:updated', 'my_tasks:print_ruby_version'
13 end
```

Listing 8.1: An example of a Capistrano task

statement. Capistrano provides the option to run code in parallel with an additional instruction for a task.

Scalability is a very important concern for more extensive server infrastructure. In order to account for this, Capistrano is able to differentiate between application, web, and database servers. These server types are associated with matching roles. This allows the user to only run specific tasks for the subset of servers matching a certain role. One task might be to restart all the web servers after a new deployment, or to save a database snapshot of all the database servers.

Capistrano already has built-in support for multi-stage deployments as described above with one staging server and one production server. This allows the user to quickly deploy to different targets without having to exchange configuration files.

Rails applications are usually considered to be highly available, due to the flexible nature of the Capistrano workflow. With certain web server configurations, it is possible to swap out the running application version, and replace it with the newly deployed one, without shutting down or restarting the daemons. This also allows a rollback to previously deployed versions of the application, if the user needs to pull a version from the live system, by simply switching back to a still available older version of the application code. This deploy- and rollback-flow is capable of migrating a database and undoing those changes in case of a rollback.

Every Rails application needs a set of configuration files for the database connection, the email subsystem, and any other parts which could be invoked. Since different credentials are required for each part, all files with sensitive data reside in a special folder called `shared`, which is then linked to the application's root directory with a symbolic link for each release. This allows a strong set of passwords to be used, and sensitive configuration parameters can be kept within the server.

8.2.2 Preparation

The preparation of a new hosting environment in Capistrano comprises to the following steps:

Web Server: The recommended configuration for Sapphire is to use the Apache 2 web server with the Phusion Passenger module for Ruby on Rails integration. The installation is straightforward and only consists of creating a new virtual host site and configuration file for Apache which includes a few Phusion Passenger related settings for the loading paths of the gem and the Rails root path where the Sapphire application is placed. It

is recommended to include a TLS/SSL certificate and force usage of Hypertext Transfer Protocol Secure (HTTPS) to increase transport security over the network.

Ruby Environment: The runtime executables for all Ruby-related applications must be installed and initialised prior to the first deployment, since Capistrano uses the Ruby language during deployment. The RVM or rbenv Ruby environment concept is explained in more detail in Sections 2.3.1 and 2.3.2.

Deployer User Access: The Capistrano tool chain works over an SSH connection on the remote server during the execution of Capistrano tasks. In order to create an automated and easy to use system, the server should be reachable for the specific user used during deployment. SSH keys are recommended for authentication in the tool chain. In the case of Sapphire, it is also necessary to upload an SSH key to the GitHub repository to allow the deploying user access to the code repository. This deploy key provides read-only access to the repository only for the designated server which is currently fetching the files.

8.2.3 Configuration

The necessary configuration for a fresh Sapphire server (either `staging` or `production` environment) is included in the code repository. It consists of the repository path, from which the code should be checked out during deployment, the name of the deploying user, the target path of the Rails application (which must be the same as specified in the web server configuration) and some environment-specific settings for RVM or rbenv.

All of the settings described above are platform-dependent and can therefore vary between different Linux distributions and their release versions. The default included for Sapphire was tested on CentOS 6.4. Different options might be needed for other kinds of Linux derivative.

Chapter 9

Future Work

This chapter describes some of the upcoming enhancements and new features for Sapphire. Certain areas and subsystems showed some problems and lack of functionality during initial testing. The following changes will increase the efficiency and usability of Sapphire and its subsystems.

9.1 Improvements to Existing Features

9.1.1 Import Workflow

The import workflow for student registration data from TUGRAZonline, as explained in Section 8.1, is still quite inconvenient in the sense that the CSV file has to be prepared with some non-trivial steps to be compliant with the Sapphire Importer.

In the future these preparation steps could be included directly into Sapphire, which would reduce complexity for the user. The only necessary steps would be the creation of the export in TUGRAZonline, and the upload into Sapphire itself.

9.1.2 Import Progress Feedback

In the current version, Sapphire performs all import-related work within the actual HTTP request content. This leads to a long response time, which could confuse a user if the request takes longer than a few seconds to process. Due to the number of students per term, about 350 to 450, such an import request might take up to 70 or 90 seconds. This is considered disruptive behaviour for the user and should be prevented. On an additional note the system might run into technical limitations if the request processing time exceeds the HTTP connection timeout in the browser.

To improve the described shortcomings during an import, it is suggested to refactor this feature of Sapphire, in order to outsource the heavy work to a background job. This way the user could receive some kind of feedback about the current import progress. This could be in form of a progress bar, a label with the count of already processed students, or a combination of both. After a successful import job, the results and possible errors could be presented to the user. This could also be included within the progress feedback to ease the context switch for the user between the different import areas as mentioned in Section 8.1.3

Asynchronous work in a Ruby or Rails environment can be done with the sidekiq gem. This is a system daemon running at all time in the background, waiting for jobs to process. During a Rails HTTP request one can simply trigger this worker and pass some data for processing.

The processing of the HTTP request can be continued immediately, due to the fact that all necessary information regarding the new background job is saved into a Redis database, leaving no additional work for the Rails stack. The sidekiq worker then fetches a new job from the database, executes it and saves the results back into the database. The user sees a refreshed progress feedback with a simple Asynchronous JavaScript and XML (AJAX) polling loop.

9.2 Additional Features

9.2.1 Role-Based Authorisation and Feature Activation

The predefined roles of students, tutors, and lecturers within Sapphire are assigned via registrations between accounts and their roles, as described in Chapter 6. The shortcoming of this kind of abstraction is that it does not include an authorisation system which would prevent students from manipulating their own grades. For that reason a role-based authorisation system has to be implemented to restrict the user from performing certain actions and prevent illegitimate data manipulation.

Since a user must have an account in order to access Sapphire, the first step of securing the system is already in place. The registrations can therefore be utilised to allow and deny certain actions and present different views to the user, based on the permissions associated with the user's current registration depending on the term and course. This means that an account with a lecturer registration should be able to access and manipulate all data regarding the term identified by the lecturer registration. However, a student registration should provide only read-only access to that student's points and grade.

Using gems like Authority, Protector and Pundit is recommended for this kind of role-based activation of features. In certain ways the core component is a definition of actions on a model- or controller-level which then bubbles through the Rails stack to ensure the correct abilities based on the currently logged in user. This allows the developer to control the granularity of permissions in a model-object-based fashion, before targeting the view layer of the web stack.

9.2.2 Export of Evaluations and Metadata

Data manipulation is one of the core features of Sapphire. The use of PostgreSQL for the Sapphire database schema is already documented in Chapter 5. Currently, the only abstract export method is rendered through the HTML view layer of Sapphire. This approach is not suitable for a reusable, formatted and lightweight data export into a CSV file or a similar data container. A better way of data export would be a model-based solution, as well as a student-centric perception, aggregating the points for each exercise within a term and creating an overview of all students on a term-based level. This export feature might reconstruct the Sapphire database into the former spreadsheet-based format.

PostgreSQL already supports the creation of JavaScript Object Object Notation (JSON)-based file structure containing all the data. This is database functionality, available at any given time by invoking the associated commands directly in the database management system. This approach is also used for creating backups and restoring data after a fatal error to prevent data loss. Using JSON as an intermediate data representation, it is possible to export spreadsheet-based file formats like OpenXML or native Microsoft Excel formats.

A more advanced solution could be implemented to create a human-readable data container from the raw data. CSV is the recommended choice of standardised data representation and exchange format. The points overview page, containing all grades and points for each exercise regarding every student of a particular term, would be a perfect use case for any proposed

CSV export functionality. For more fine-grained integration, the evaluation table would be an additional candidate for export.

9.2.3 Online Management of Registration Data

The Sapphire core strongly encourages the all-in-one import solution described in Section 8.1. The reason this method was chosen, is that it simplified the initial development process by letting the developers focus on the more important parts needed for the initial test operation. A proper user interface for registering new students and creating new student groups will follow during the next development iteration.

Before the next system evaluation test, it is recommended to implement a user interface. This interface module should be accessible by tutors and lecturers to allow them to create and manage registration data and alter student associations within Sapphire.

Chapter 10

Concluding Remarks

In this Bachelor's thesis, I have presented the Sapphire web application emphasing the back-end functionality and the underlying data structures. Chapters 2 and 3 contain a brief introduction of the Ruby programming language and the Rails web framework, which form the foundation of the Sapphire system.

The specific implementation details of data structures, the registration system, database architecture and the realisation of student groups are outlined in Chapter 5 and 6. Selected details of the data import and the setup and deployment workflow for a new installation of Sapphire are described in Chapter 8.

This thesis concludes with suggestions for features to be implemented in the future in Chapter 9.

Appendix A

Developer Guide

This appendix describes the Sapphire development workflow. The core system of Sapphire was built around tool chains widely used for the development and deployment of Rails applications. This guide is directed to developers planning to extend or change the Sapphire core system. It assumes some knowledge of the Ruby language, the Rails conventions, and various development tools.

A.1 Version Control System

Since day one, the Sapphire project has been developed with Git as its version control system. Every commit in the master branch is intended to be stable, tested and reviewed by the main developers. New features are developed in a side branch, and are only merged if the code works and is ready for a production environment. The repository is currently hosted in a private GitHub account, which is also used for issue tracking, pull request review, and providing the back-end for the Capistrano deployment source repository.

Whitespace at the end of a line must be trimmed before committing. Every file is encoded in UTF-8, with unix-style line endings (single newline character) unless stated otherwise on a per file basis. Code style guidelines are based on the Ruby language project. Multiple empty lines should be avoided, unless to isolate parts of code blocks. Every Ruby class should have its own file in the appropriate directory hierarchy according to the Rails guidelines.

A.2 Recommended Tools

Git: Git is a version control system most effectively used from the command line with various subcommands like `git-merge`, `git-rebase`, `git-pull`, and `git-push`.

Zsh: Zsh is a shell used in a terminal. It has many additional features and improvements, such as advanced auto tab completion and support for multiple character encodings, compared to the more widely known bash.

Sublime Text: Sublime Text is a text editor available for Windows, Linux and OS X. It provides a python-based Application Programming Interface (API) for feature-rich plug-ins managed by a separate package manager. Sublime Text is very popular amongst developers in every programming language and environment, because of its very good support for language specific-features provided by the community in additional packages.

Spring: Spring is a Rails environment preloader which speeds up the most common tasks run through the `rails` command such as generating new code files from templates. It simply runs in the background as a stand-by preloaded environment and waits until the developer calls it. The execution time for a common Rails command drops from about 50 seconds down to 10 seconds.

Pow: Pow is a web server which runs on your local development computer. It can host multiple Rails applications at once by simply creating a symbolic link from your application's root directory to the web server configuration directory. A useful feature regarding multiple applications is that Pow creates a separate domain for each application, which can then be used in a browser to navigate to and use the specified Rails application.

Rake: Rake is a build utility similar to the Make command known from compiled programming languages. There are a few predefined tasks for running the test suite and asset management. New Rake tasks can be created in the Ruby language following a task-oriented notation.

Pry: Pry is an improved interactive Ruby shell. It is, like irb, written in plain Ruby and therefore has no other dependencies. It presents the code base currently running as a kind of directory tree where the user can enter a class or an object with a `cd` command, and list all currently accessible variables with `ls`, just like in an ordinary Unix shell.

Guard: Guard is a file system monitoring application which watches a predefined folder for changes of certain files and then executes some code, based on events triggered via file changes. It can be used to automatically restart the web server when some files change, or run the `bundle` command if Guard detects a change in the `Gemfile`. Guard itself is more like a framework, which is utilised by multiple other gems to provide specific event-based behaviour, for example, the `guard-pow` gem performs a Pow restart and `guard-cucumber` runs all feature tests if something changes.

Ack: Ack or `ack-grep` is a search tool similar to `grep`, but with additional defaults and features such as auto-recursive search, smart case-sensitivity and file filters based on predefined types, for example, `--tex` for `--ruby` to only consider these file types during a search. Ack is written in plain Perl and is therefore available for every modern platform with a Perl interpreter.

A.3 Important Directories and Files

For a Rails beginner, it can be somewhat overwhelming to explore a larger Rails application. There are many folders and files with different meanings in different places. It takes some time to fully internalise the conventions and default configurations of a Rails application and the Ruby language in general.

The two most important folders are `app` and `config`. Inside `app` are subdirectories containing the `models`, `views`, and `controllers` of the application. In general, these three always exist together due to the Model-View-Controller pattern of the Rails core. Helpers are small snippets of code to make the life of a developer easier. The `assets` directory contains all the JavaScript and CSS files. All configuration and initialisation files should be in the `config` directory. A common convention is that all files in the `initializers` directory are automatically loaded every time a Rails application is started. This is the reason why it can take some time for the web server and the `rails` command to start up.

The `Gemfile` and its counterpart the `Gemfile.lock` are responsible for specifying which gems must be installed and bundled. Each gem is locked into a specific version number, to prevent unintended version changes which might break the application.

Every HTTP request for a Rails application is logged in a log file in the corresponding `log` directory, which should be cleared from time to time with a provided Rake task.

All database related files are contained in the `db` directory. This includes the sqlite database used for development, as well as database schema migration files, the current database schema, and possible seeds that can be loaded using provided Rake tasks.

Bibliography

An, Jong-hoon, Avik Chaudhuri, and Jeffrey S. Foster [2009]. "Static Typing for Ruby on Rails". In: *Proc. 26th IEEE/ACM International Conference on Automated Software Engineering (ASE '09)*. (Auckland, New Zealand). IEEE Computer Society, Nov. 16, 2009, pages 590–594. doi:10.1109/ASE.2009.80. http://cs.umd.edu/~avik/projects/stror/paper.pdf (cited on page 4).

Andrews, Keith [2012]. *Writing a Thesis: Guidelines for Writing a Master's Thesis in Computer Science*. Graz University of Technology, Austria. Oct. 22, 2012. http://ftp.iicm.edu/pub/keith/thesis/ (cited on page xi).

Andrews, Keith [2014a]. *Exercise 4: Build a Web Site of Internet and New Media*. Feb. 28, 2014. http://courses.iicm.tugraz.at/inm/exercises/exer4.html (cited on page 13).

Andrews, Keith [2014b]. *Exercise 5: Three Style Sheets of Internet and New Media*. Feb. 28, 2014. http://courses.iicm.tugraz.at/inm/exercises/exer5.html (cited on page 13).

Beck, Kent [2002]. *Test-driven Development: By Example*. Kent Beck signature book. Addison-Wesley, Nov. 18, 2002. ISBN 0321146530 (cited on page 25).

Behat Project [2014]. *Gherkin*. Feb. 28, 2014. http://docs.behat.org/guides/1.gherkin.html (cited on page 26).

Capistrano Project [2014]. *Capistrano*. Feb. 28, 2014. http://capistranorb.com/ (cited on page 31).

Cuadrado, Jesús Sánchez and Jesús García Molina [2007]. "Building Domain-Specific Languages for Model-Driven Development". *IEEE Software* 24.5 (Sept. 17, 2007), pages 48–55. ISSN 0740-7459. doi:10.1109/MS.2007.135. http://modelum.es/papers/ieeesoftware2007.embedded-dsls.jsanchez.pdf (cited on page 3).

Fowler, Martin [2002]. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Nov. 2, 2002. ISBN 0321127420 (cited on pages 7, 8).

Günther, Sebastian and Marco Fischer [2010]. "Metaprogramming in Ruby: A Pattern Catalog". In: *Proc. 17th Conference on Pattern Languages of Programs*. (Reno, Nevada). PLOP '10. ACM, Oct. 16, 2010, pages 1–35. ISBN 145030107X. doi:10.1145/2493288.2493289 (cited on page 4).

Jones, Geraint [2014]. *Ruby: A Notation And Design Discipline Intended For The Development Of Regular Integrated Circuits And Similar Hardware And Software Architectures*. Feb. 28, 2014. http://www.cs.ox.ac.uk/geraint.jones/ruby/ (cited on page 3).

North, Dan [2014]. *Introducing BDD*. Feb. 28, 2014. http://dannorth.net/introducing-bdd/ (cited on page 25).

PostgreSQL Global Development Group [2014]. *PostgreSQL*. Feb. 28, 2014. http://postgresql.org/ (cited on page 15).

Ruby, Sam, David Thomas, and David Heinemeier Hansson [2013]. *Agile Web Development with Rails 4*. 4th edition. The Pragmatic Programmers. Pragmatic Bookshelf, Oct. 2013. ISBN 1937785564 (cited on page 7).

SkySQL, MariaDB Foundation [2014]. *MariaDB*. Feb. 28, 2014. http://mariadb.org/ (cited on page 15).

Smyth, Neil [2014]. *What Is Ruby?* Feb. 28, 2014. http://techotopia.com/index.php/What_is_Ruby? (cited on page 3).

Tanaka, Kazuaki, Yukihiro Matsumoto, and Hiroshi Arimori [2011]. "Embedded System Development by Lightweight Ruby". In: *Proc. 2011 International Conference on Computational Science and Its Applications (ICCSA 2011)*. (Santander, Spain). LNCS. Springer, June 20, 2011, pages 282–285. doi:10.1109/ICCSA.2011.62 (cited on page 3).

Thomas, David, Andy Hunt, and Chad Fowler [2013]. *Programming Ruby 1.9 & 2.0*. The Pragmatic Programmers. Pragmatic Bookshelf, July 4, 2013. ISBN 1937785491 (cited on page 3).

Webster, Craig R. [2014]. *Getting started with Story Driven Development for Rails with Cucumber*. Feb. 28, 2014. http://barkingiguana.com/2008/11/11/getting-started-with-story-driven-development-for-rails-with-cucumber/ (cited on page 26).

Wynne, Matt and Aslak Hellesøy [2012]. *The Cucumber Book : Behaviour-Driven Development For Testers And Developers*. The Pragmatic Programmers. Pragmatic Bookshelf, 2012. ISBN 1934356808. http://opac.inria.fr/record=b1134153 (cited on page 26).

ZID [2014]. *TUGRAZonline*. Zentraler Informatikdienst der TU Graz. Feb. 28, 2014. https://online.tugraz.at/ (cited on page 11).