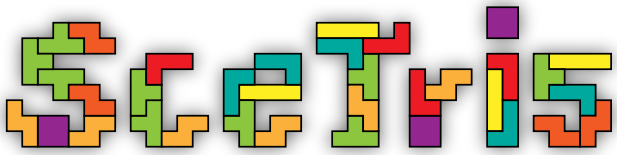
MyCourses
Summary Report



SCORE Contest 2011

David Bialik, Julian Fleischer, Hagen Mahnke,
Konrad Reiche, André Zoufahl

January 15, 2011

Freie Universität Berlin

Department of Mathematics and Computer Science
Institute of Computer Science

www.fu-berlin.de

Contents

Abstract

This document summarizes the activities of a team of undergraduate students participating in the student contest on software engineering [SCORE] 2011[1]. The carried out project was *MyCourses*[2]. The document contains descriptions of the work done during the development process, in particular about the requirement solicitation, requirements specification, design and the implementation.

1 Team

We are a team of five undergraduate students of the Freie Universität Berlin[3]. Even though we are all in the same year our knowledge regarding the creation of software and the used technologies varied widely. We were however all inexperienced in project management. Specifically none of us had been part of a software development process before. We choose the team name *Scetris*. *Scetris* is an artifical word composed of scheduling and Tetris – scheduler because of the project itself and Tetris because of the idea of arranging courses in the timetable in way to remove unneccesary space. The members of the team, their age and their most prominent task are:

David Bialik (23) is responsible for the maintenance of the technical aspects of our developing process. He created our ant-based build system and an automated installer that sets up the database and a webserver. He also kept an eye on documentation and unit testing.

Julian Fleischer (23) is an enthusiast regarding everything that has to do with programming languages and the semantics of data. As advanced studies he attended a course on XML technologies. He specified and implemented an XML-based format for the creation of object-relation models, from which custom code can automatically be generated. The data access layer, as part of the backend of our application, has emerged that way, as have large parts of the web forms.

Hagen Mahnke (24), interested in database technologies developed in pair programming

[1] `http://score-contest.org/2011/` – Homepage of SCORE
[2] `http://score-contest.org/2011/projects/Crnkovic.MyCourses.pdf` – Project description of MyCourses
[3] `http://www.fu-berlin.de` – Homepage of Freie Universität Berlin

with Konrad the initial scheduler algorithm. Further he has been the communicator for the team.

Konrad Reiche (22), whose primary interest lies in algorithms and their efficient implementation, designed and implemented together with Hagen the scheduling algorithm. It works both as a stand alone application as well as an integrated component.

André Zoufahl (22) is working part time in a company doing web development, thus he is experienced in the usage of HTML, CSS and accompanying web frameworks. One of his major interests is the visualization of data, thus he was primarily responsible for the development of our user interface.

Figure 1: From left to right: Andre Zoufahl, Konrad Reiche, Julian Fleischer, David Bialik, Hagen Mahnke

2 Development process

An important subfield of software engineering is the use of known methods and principles. The development process implies the used methods and principles. Thus this section will cover the chosen development process, its ideal procedure and how the development process actual took place including a timeline of events.

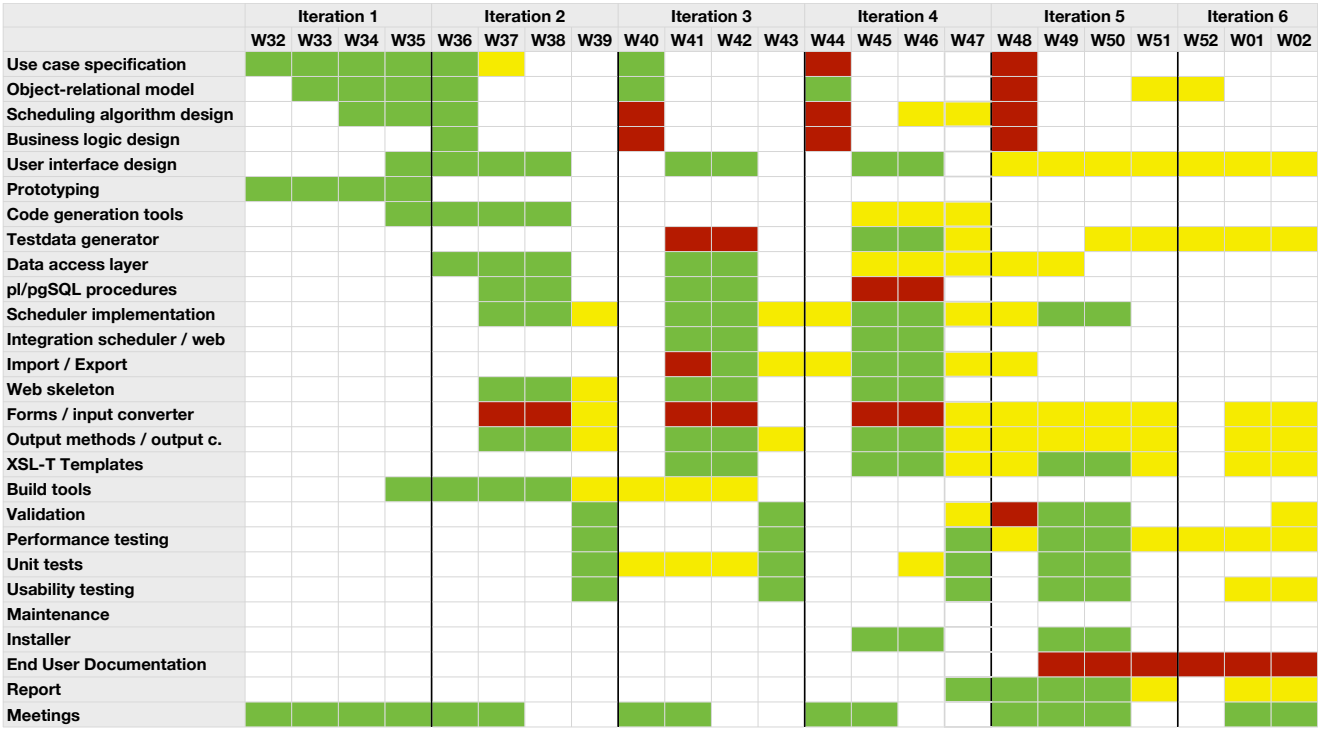
2.1 Intended development process

As mentioned above *MyCourses* is the first software project for everyone in the team. The selection of a development process was heavily based on the software engineering course we heard in the summer 2010. A major part of the project was realized in the period of lectures. This and the long distances between our homes is why we decided against an agile development process, as we did not have the opportunity to communicate and meet on a daily basis.

For the intended development process we chose the spiral model [1]. An iterative model used incrementally seemed to be promising. Dividing our project into iterations would give us a frame of orientation for the project management. The incremental working method was fitting as we were sure there will be no functioning version of the program after the first iteration. Overall the spiral model focuses on solving those problems which reduce the projects risks at best. This fact reflected our expected work habits to adressing those problems which are blocking the project accomplishment the most.

We rather chose the spiral model as the basis for our own process and assumed that adaption would become necessary. We did not, however, have a clear concept of what might require adaption and how said adaption might be realized. Further we discussed essentials which should support and promote our work-flow. Key components of our process were therefore:

- **regular meetings** for coordination
- communication via **e-mail**
- evaluation of every iteration
- **ticket system** for defining and monitoring tasks
- **wiki** for collaborative document creation



green cells: components which we workend on as planned
yellow cells: components we worked on off our plan
red cells: components we did not work on although planned

Figure 2: Our project plan, originally conception and actual process

2.2 Actual development process

How the actual development process took place can be best understand when reading the timeline in section 2.3. In retrospect to the spiral model certain variations emerged.

The beginning of every iteration is characterized by detailed requirement definitions. An intense requirement elicitation based on the *MyCourses* specification was done in the first iteration with the result of a set of written use cases. In subsequent iterations, however, the requirement refinement was not done in the beginning of each iteration, but distributed over the iterations whenever problems with the current design emerged.

Even though we implemented a minimalistic prototype of *MyCourses* in iteration 1, there was no continuous prototype development for comparation and evaluation as proposed by the spiral model. We focused on incremental development of the final product in every iteration.

In contrast other procedures in the development process went according to the spiral model. We put a lot of effort into the evaluation of different architectures and their resulting design. Further we tried to reduce the possible project risks by prioritizing the next main goals for the iteration.

In matters of organizing ourselves and use of methods we had a steep learning curve. In August and September we selected superordinated goals, but in the middle of each iteration we often lost focus of important tasks. The biggest improvement of the development process was therefore the introduction of weekly task assignments for every team member starting in October. Based on that, the tasks and their results were evaluated in every week, which lead to a purposeful work and swift discussion of emerging problems.

2.3 Timeline

A short overview of the course our project took will be given in the next paragraphs. The goals for a phase were defined after evaluation of the previous phase had been completed. Before our

project really started we got to know each other and decided which professor should be our contact person at university.

The First iteration started on June 20, comprising an initial requirement elicitation, design of our software, prototyping of the scheduler and choosing technologies. During most of this time the academic term was still running and only a fraction of our time was devoted to the project. This iteration ended on September 9 and was evaluated the day before. Much of the time spent in this iteration was still focused on gaining orientation. Still it was necessary for us to take this time and find a suitable approach to the work ahead.

The Second iteration started on September 10 and was devoted to producing the scheduling algorithm, the web interface and our ORM as well as the servlet code. Work cycles were short and meetings were held every few days, including a weekend devoted to coding. From this point on our iterations were reduced to about one month each, the focus being shifted on implementing and testing. Requirements and design were refactored whenever we felt the need to – due to new insights or problems. This iteration ended on October 4 and was evaluated the same day.

The Third iteration accordingly started on October 5, but most work was halted until October 16 due to exams and all-day courses at university. Beginning with October 26 weekly two-hour meetings were held at our university during which everyone reported achievements and problems of the previous week. At the end of each session goals for the following week were set. These weekly meetings have proven to be a good way of coordinating work as well as motivating each other, so they were maintained for the remainder of the project. The main goal and achievement of this iteration was the integration of the scheduler with the web-interface and the connection to the database. Due to this employment of parts of our software many issues were found and fixed. Accordingly we shifted our focus further towards unit-testing and started using a tool to measure code coverage, that is how much of our code is actually being executed as part of a unit-test. At the same time more functionality was added to the web-interface. This iteration ended on November 1 and was evaluated the day before.

The Fourth iteration started on November 2, with the primary focus lying on the improvement of our web interface, adding functionality and a unified design. Several improvements to the scheduler algorithm were made, most notably the introduction of a greedy algorithm. Work on the summary report was also started during this iteration and the first version – which was the basis for all further versions – was written. The iteration ended on December 5.

The Fifth iteration started on December 6 and was mainly devoted to minor improvements to the scheduler and writing the beta-version of our summary report. One improvement to our database-access-layer which made query-caching available drastically improved scheduling performance by a factor of 8. The iteration ended on December 30.

The Sixth iteration was not part of our original project plan. The project was late and so we had to add three additional weeks for last feature implementations amd quality assurance.

3 Requirements

The following three sections are an overview of the problem and the resulting requirements. The first section states the problem, while the second describes the elicitation we deduced from it, followed by the specification in the third section.

3.1 Problem statement

Organizations like universities and schools have to allocate courses to the given resources in a sensible way. As the amount of resources and courses grows large, it gets increasingly costly to do the allocation manually. Furthermore the task of course scheduling is repeated quite often and thus a lot of work is spent on it. Often the courses and resources remain consistent each term and large parts of the previous solution can be reused. Each solution to such a problem must satisfy a number of constraints.

3.2 Requirement elicitation

Most of the requirements are already outlined in the project description of *MyCourses*. Still we considered it helpful to get more insights into these requirements. To make the requirements more specific and better understood we created use cases. These use cases helped us during implementation as they provided an important frame of orientation.

As we also desired more insight into how a scheduling process might look like, we asked an employee of our institute to give us a demonstration. The demonstration was conducted on the system used at our institute. Although it did not reveal completely unknown aspects to us, it gave us a better estimation of the importance of certain aspects.

3.3 Requirement specification

The ideal scenario is a program that automatically finds an optimal solution. However the problem is NP-hard [3] and thus a computer-aided scheduling process is the focus.

The problem of course scheduling is defined by the given constraints which have to be satisfied. In order to meet the *MyCourses* specification requirements of high configurability we distinguish constraints by *hard* and *soft constraints*.

Hard constraints are constraints which must be satisfied. If these constraints are not satisfied the lecturing of courses is not possible. This includes the following constraints:

- not more than 1 course in the same room at the same time
- the course lecturer teaches no other course at the same time
- courses belonging to the same year do not overlap with each other in time in order to ensure studiability
- every constraint defined by the user with the priority of 100%, for instance preferred time, preferred room, etc.

Soft constraints are constraints which need not to be ultimately satisfied. This includes only constraints defined by the user with a priority less than 100%.

An *optimal solution* is a scheduling which asserts the satisfaction of all *hard* and *soft* constraints.

As the interests of many employees and students at the universities are affected by the result, they should be allowed to participate in the process or at least be taken into consideration. The requirements that had the greatest influence on our initial design and each subsequent change are:

- automatic and manual allocation of courses
- satisfaction of hard and softconstraints
- suited for multi-user environments
- scalable scheduling
- defining/ presenting of scheduling-related information
- user-friendly look and feel
- access restriction to different areas

4 Architecture

The following section will briefly overview the architecture of our software.

4.1 Overview

Early on we divided our software into two major components, which – seen individually – should be able to run as stand alone applications as well as as integrated components. This approach should give us the ability to work on different parts of the software more independently. Furthermore, since both components are largely decoupled, it would be easier changing one of them without breaking the other.

These two components are:

A scheduler which provides for an automatic allocation of courses to rooms and times.

A web application which is splitted into three tiers:

- The front end
- A controlling unit *(controller)*
- The business logic

Both components should work on common ground, which emerges naturally from the need for both to work on the same data. Thus they were to access the same database and we decided to create a data access layer which should be shared by both components.

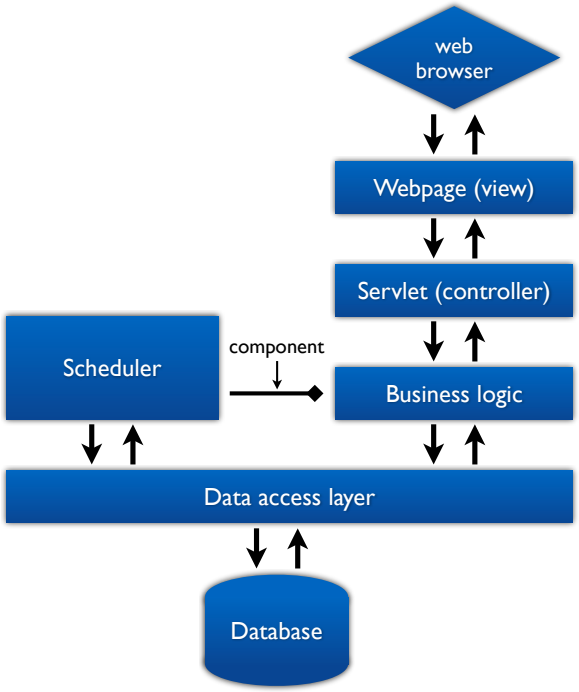


Figure 3: An overview of the components

4.2 Data access layer

In order to create a proper model of the entities we were to work with, we decided to develop an object-relational model. By that we understand a model which can easily be mapped to an object-oriented model, as well as an entity-relationship-model[4] from which the object-oriented classes as well as an entity-relationship database definition can be derived.

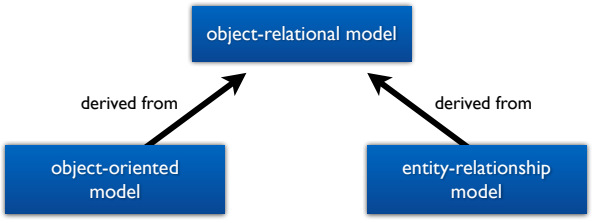


Figure 4: The object-oriented model and the entity-relationship oriented model are derived from the object-relational model. Since both the OO as well as the ER model derive from the same basic model, they are guaranteed to be in a consistent state.

[4]Not to be confused with approaches to store objects in relational databases

4.3 Scheduler

For providing a reasonably fast pre-scheduling, we wanted to use a genetic algorithm. Such algorithms roughly consist of three parts:

A fitness function which evaluates the quality of an intermediate solution.

A cross over function which intermixes two intermediate solutions in a specific manner in order to create a better solution.

A mutation function which randomly or intelligently refactors an intermediate solution.

The algorithm itself which controls the execution of the other parts, i.e. in what order the components are applied and how many intermediate solutions are created.

We decided to model these parts as exchangable components, i.e. individual objects, which could be combined in any way. Since we did not have any experience with these kind of algorithms this design should help us to better understand and scale the different parts of the algorithm.

5 Technologies

Since our design was heavily influenced by the technologies we wanted to use, we will give a brief discussion of the technologies we used and why we chose to use them.

We considered many technologies and looked at some of them intensively. Since we were building a web application we considered PHP, Ruby, Scala, pure Java and AspectJ as our main programming language. As the scheduler is a performance critical part of our application we dropped interpreted languages early. As we were all most familiar with Java and development tools for Java are widely available, we favored Java. We finally went with **AspectJ**[5], which is built on top of Java, since it offers great flexibility through so called Aspects. It is tightly integrated in our favored IDE[6], Eclipse, since it is an Eclipse project itself. We will discuss the use of Aspects within our application more in-depth later on.

As database backend we chose **PostgreSQL**[7] since it features strong adherence to the SQL standard and is the best open source implementation of *referential integrity* in relational databases we knew of. Another strong argument for PostgreSQL was, that it allows for rich use of stored procedures, i.e. business logic within the database. As a matter of fact our application thus runs with PostgreSQL only, but could be ported to another database-engine easily, since only certain parts would have to be rewritten (PostgreSQL-specific parts within the data access layer and certain Aspects).

We used the standard web technologies like **HTML**[8] and **CSS**[9] in the front end. To keep things simple we stuck to related technologies which are all part of the **XML family**, like **XSL-T**[10]. We will disucss the use of XML within our web application in the appropriate sections of this report.

5.1 Libraries & Frameworks

In order to enrich users experience, we made use of **jQuery**[11]. However, in order to keep the site accessible, we did not use it too much, i.e. the site should be fully functionial even if JavaScript was deactivated.

On the backend side we employed many libraries from the Apache Software Foundation, namely from their XML Software Stack. Amongst

[5] `http://www.eclipse.org/aspectj` – The eclipse AspectJ project
[6] Integrated Development Environment
[7] `http://www.postgresql.org/` – PostgreSQL
[8] `http://www.w3.org/html/` – W3C HTML homepage
[9] `http://www.w3.org/Style/CSS/` – W3C Cascading StyleSheets homepage
[10] `http://www.w3.org/TR/xslt` – XSL Transformations (XSLT)
[11] `http://www.jquery.org` – Homepage of jQuery
[12] `http://xalan.apache.org` – Apache Xalan XSL-T Processor
[13] `http://xmlgraphics.apache.org/fop/` – Apache XSL-FO Processor

others, these are **Apache Xalan**[12] and **Apache FOP**[13], which we used to process XSL and generate not only XHTML, but also PDF files.

To create an easy to use installer we used **IzPack**[14]. To make sure that our application works with different application servers it can optionally be bundled with Jetty[15] (an embedabble servlet container), which itself also is an Eclipse project.

5.2 Tools

As mentioned in section 2, we made use of a wiki and a ticket system. Both of them are contained in **Trac**[16], a webinterface to **Subversion**[17], which we both used for source and version control.

We used **JUnit**[18] for writing unit tests. For the automatic execution of this, as well as generating reports about line coverage and branch coverage, we used **Cobertura**[19].

To build our project, we used GNU Make initially, but later on we switched to **Apache Ant**, since it is more platform independent and features plugins which tightly integrate with AspectJ and JUnit. For testing purposes we employed **Apache Tomcat**.

Regarding the versions of mentioned tools we followed the policy to include nothing more recent than the latest stable version of the Debian GNU/Linux distribution.

5.3 Code generation toolbox

As we mentioned in section 4, we did not want to maintain separate models for our database and our application. To achieve this technically we created an XML-file which represented an abstract description of our object-relational model. The syntax and semantics were defined using XML Schema[20]. Using different XSL-T Stylesheets we automatically generated Java source files, an SQL install script, and documentation in XHTML.

As part of our build process we developed some tools for automatic generation of code. Basically the toolbox consists of several XSL-T stylesheets which can be used to create Java code, SQL scripts or XHTML documentation from a custom XML definitions file. This file contains the object-relational model and is specified using XML Schema.

The whole process of generating code is driven by Ant and part of our build process. Also the database can be easily set up using our toolchain.

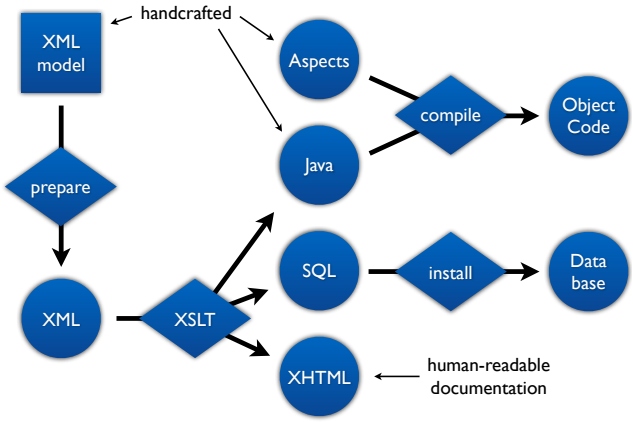


Figure 5: The process of generating code within scetris.

6 Design

In the following subsections our design decisions are briefly explained. A description of how these influenced our project and what we learned from them will be given in the section 10. The first section discusses our domain model. The second section deals with our data access layer. The third section is about our web application. The last section describes the scheduler algorithm.

[14] `http://www.izpack.org/` – Homepage of IzPack
[15] `http://www.eclipse.org/jetty/` – Jetty Web Server
[16] `http://trac.edgewall.org` – The Trac project
[17] `http://subversion.tigris.org/` – Subversion homepage
[18] `http://www.junit.org/` – JUnit Test Driven Development
[19] `http://cobertura.sourceforge.net/` – Cobertura – Code coverage analysis tool
[20] `http://www.w3.org/XML/Schema` – XML Schema

6.1 Domain model

Since universities are organized quite differently we tried to design the domain model as generic as possible. An important aspect of the application is reusability of course definitions. This requirement lead to the following design of course related information in our domain model.

Course is a well-defined set of *CourseElements* and has only a name. For example the department of physics offers the course Thermodynamics.

CourseElement is part of a *Course* with a type and a duration. The course Thermodynamics consists of two *CourseElements*, a two-hour lecture and a four-hour workshop.

CourseInstance is a realization of a course for a specific program. The course Thermodynamics is held every second academic term.

CourseElementInstance is a realization of a *CourseElement*, where several *CourseElementInstances* constitute one *CourseElement*. For instance the lecture is held on Monday from 10 to 12. The workshop on the other hand is divided into two events, that are held on Tuesday from 10 to 12 and Friday from 8 to 10.

Universities differ in their definitions of class types, features that a room may or may not have and features that a course may require. As such it appears to be infeasible to predefine these properties appropriately. Therefore our model allows universities to define the possible values of these properties. This generic modelling of properties is achieved by representing them as an attribute in a special entity. These entities are connected to the original entity indirectly through a relation that defines the connection.

Figure 6: Many-to-many relation: generic model is achieved by representing them as an attribute in a special entity.

This also applies to user management where many different roles and sets of privileges are conceivable. Privileges themselves are predefined as they are a direct result of the possible actions. Furthermore universities might want to store additional information related to a course or person. The generic properties for Features of Rooms, CourseElements and CourseElementInstances are

- CourseElement requires Feature
- CourseElementInstance requires Feature
- Room provides Feature

This means that for example a CourseElementInstance can have an arbitrary number of user defined Features. The connection is established through the relation CourseElementInstanceRequiresFeature. The connection can hold information about the minimum and maximum quantity, i.e. the number of seats a Room provides is represented in RoomProvidesFeature where the Feature has the name Seat.

- Person has Attribute
- Course has CourseAttribute

Can be used to add user-defined attributes to a Person or a Course. For example the address of a Person can be stored as an Attribute.

- Person has Role
- Role implies Privilege
- Person has Privilege

Are used by the user management to provide Privileges to groups by RolImpliesPrivilege, or directly to Persons. For example the Role secretary may have user management Privileges. Furthermore the model enables a wide range of constraints.

- CourseElementInstance prefers Room
- CourseElementInstance prefers Timeslot
- Person prefers Timeslot
- Room prefers Timeslot

6.2 Data access layer

The object-relational model we defined deals with entities and relationships. Relationships connect two entities in a certain manner. Since they can have attributes too, we decided to model them as individual objects too, i.e. they are both represented as a table in the database as well as as a class in our object-oriented domain model. These classes implement the interfaces *Entitiy* or *Relationship* which in turn extend the interface Relation.

The objects should both represent the data as well as control the communication with the database, that means, they should be *DAOs* (*data access objects*) and *DTOs* (*data transfer objects*) at the same time. To accomplish this, we designed them as *Active Records*. They feature methods to create, update and delete them. For retrieving data a special *RelationManager* was defined. It is the central access point to the database and acts as a *Factory* for the Active Records.

Since the code for the data access layer is auto generated its static structure follows a strict pattern. Each record has getters and setters to access the encapsulated data, the RelationManager has factory methods for each Entity and Relationship. Records that represent entities do also have special methods for retrieving datasets which are related via a special Relationship. One can now ask a Person-record which Privileges it has via PersonHasPrivilege-relationships, i.e. what the objects to the PersonHasPrivilege-relationship, in which that record is the subject, are. It is also possible to ask the other way around, i.e. what subjects are assigned to a given object via a certain relationship.

One key feature of the active records is, that if something (for example an attribute which represents a reference to another dataset) is requested which has not yet been retrieved from the database, it may transparently be fetched from the database. If, on the other hand, that record has already been fetched it may simply be loaded from a cache.

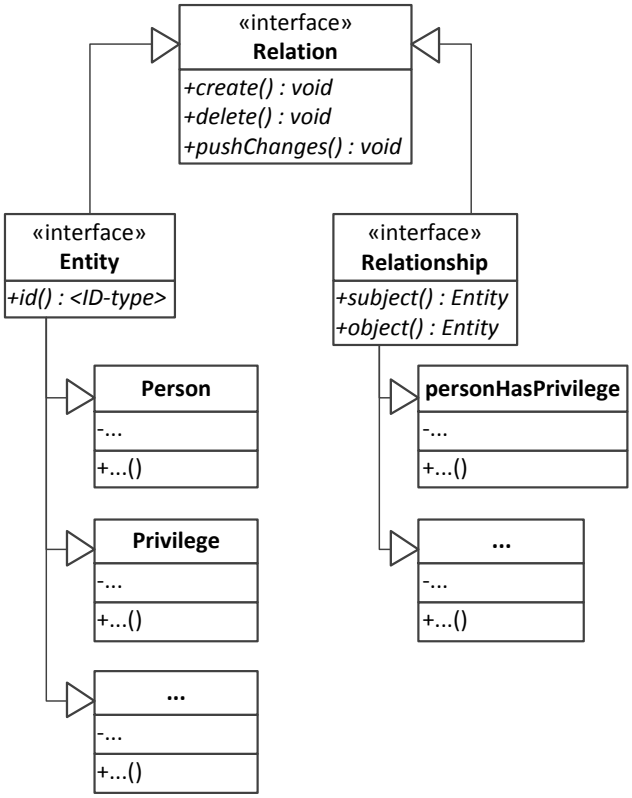


Figure 7: Overall structure of the active records

6.3 Web application

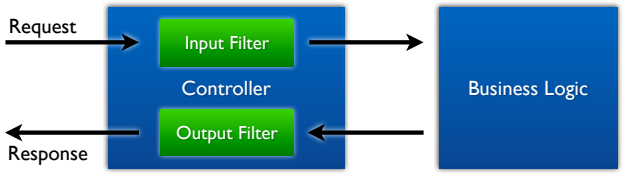


Figure 8: Data flow within our web application.

Our web application is, as stated in section 4, layered into multiple loosely coupled tiers. Communication between these tiers takes place in strictly specified ways. A request to the system is issued by the user, which directly interacts with the front end through a web browser. It is than processed by a controller, which handles input sanitizing and validating. The controller invokes the business logic which returns structured data (XML in our case). Using XSL-T that data is transforrmed into whatever format is requested and sent back to the front end by the controller.

Controller The core of our application is a Servlet according to the Java Servlet Specification. It provides the business layer with session handling and offers access to the data

access layer (which is then invoked directly from within the business layer). The main purpose of the Controller is to route data from the front end to the business layer and vice versa. Since no direct access from the front end to the business layer is possible, user privileges are also checked within the controller, so that it is not even possible to pass data to a module of which a user lacks the privilege to make use of.

Modules To achieve the decoupling of business logic from the controlling unit, we encapsulated it into Modules. These Modules are to be loaded on server startup and provided with an environment; that is, a *RelationManager* for accessing the database. Using Javas Annotation API, the Module classes are provided with information on how the controller should treat them. For example, a method may be annotated with `@Action`, which declares this method as accessible via the servlet. The Controllers task will then be, to map incoming request to that method. Modules are expected to return a `org.w3c.dom.Document`[21], which can than be processed by the controller.

OutputConverter OutputConverter are used for transforming the XML Document which is returned by the business layer into a certain format.

InputConverter In order to deal with user data, an special mechanism was designed. Data classes which represent forms are provided with declarative information via Annotations. These are to be processed by the Controller to transaprently make the data accessible to the business layer. These form classes are very similar in spirit to Active Records in the data access layer, since they are used to access data as well as transfer data.

6.4 Scheduler algorithm

The problem of course scheduling is known to be NP-hard [3]. The course scheduling problem can be approached by using search algorithms. As a matter of fact this works for simple cases only. Course scheduling, especially at universites and similiarly large facilities, goes along with complex constraints. With increasing input and more constraints an optimal solution cannot be computed within a reasonable amount of time. The widely used approach of using a genetic algorithm seemed to be promising and was therefore chosen for our project.

Genetic algorithms are a subset of the metaheuristic optimization algorithms. Metaheuristics, being part of stochastic optimization, are algorithms using randomness to some degree in order to find solutions to hard problems where the solutions are optimal or as optimal as possible. Metaheuristics are applied when little is known about what the optimal solution looks like. Heuristics are hardly useful as neccessary information is lacking. Brute-force is out of question as the space of possible solution is too large. However, when a candidate solution is given it can be rated in order to evaluate how good it is indeed [2].

In order to understand the scheduler algorithm the genetic algorithm operations, *setup*, *fitness function*, *crossover*, *mutate* and *selection*, are best conceived of as core components of the scheduler algorithm. These operations are described below.

Setup generates the initial population of candidate solutions in a random or semi-random way. A candidate solution is created when every course is allocated in a room at a given time. The *setup* process is finished when λ candidate solutions were created.

Fitness function rates the candidate solution assigning it a score. The score is mapped to the number of constraints satisfied. The more constraints are satisfied the higher the candidate solution is scored. The score ranges from 0.0 to 1.0.

Crossover creates a new candidate solution by mixing and matching parts of two given candidate solutions. How the mixing and matching is done depends to the representation of a candidate solution. As our representation is a mapping of courses to allocated rooms and times, these allocations are mixed and matched.

[21] `http://download.oracle.com/javase/6/docs/api/org/w3c/dom/Document.html` – org.w3c.dom.Document

Mutation creates a new candidate solution by taking a given candidate solution and changing a specified amount of course allocations to new, randomly chosen, course allocations.

Selection iterates the given candidate solutions and keeps only the μ best solutions. The solutions are selected, according to the score given by the *fitness function*, through dropping the rest of the candidate solutions.

It is also possible that the desired scheduling is not scheduleable at all, for instance when two courses have the hard constraint to be placed at the same room with overlapping time. In this case the scheduling is stopped and the user has to resolve the constraint conflict on their own.

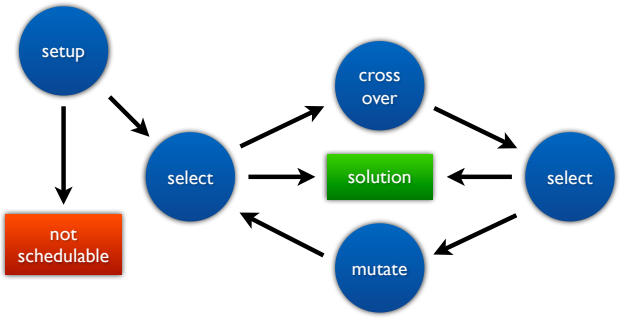


Figure 9: Routine of the scheduler algorithm. An initial population of candidate solutions is generated. Every candidate solution is rated. If non-solvable conflicts appear the scheduling has to be terminated. Otherwise the setup phase is followed by optimizing the candidate solutions through applying crossover and mutation operations until an optimum schedule is found.

7 Implementation

The Implementation section covers certain issues regarding implementation tasks. Problems and their solutions will be discussed.

7.1 Data access layer

For implementing the data access layer we defined code-templates which were simply filled with values, according to our XML schema definition – much like a serial letter. While tedious in the beginning (since the code-templates are written in XML) we quickly became used to it. Since all classes are built from the same template, it was quite easy fix bugs once and for all.

It also proved to be a quite extendable concept. For example, it took us just a few hours to implement a query cache, which increased the speed of the scheduler about 8 times. Once the API was defined, it was easy to enhance performance without putting the stability of the rest of our code base at risk.

We chose to use Java SEs native SQL-API from the java.sql-package to access the database. Thanks to the use of prepared statements we did not have any trouble with malformed user inputs, making our software virtually invulnerable against things such as SQL injections.

Use of AspectJ and PL/pgSQL One of the most compelling features of automatically generating code is, that changes made to the template are reflected in all derived classes. However, it comes at a cost. One can not manually edit that code, without either loosing the advantage of automatically adapting changes or loosing the extensions made manually. Thus we used AspectJ to *cross-cut* the auto-generated classes and so implement special functionality. Using PL/pgSQL these functions were implemented efficiently within the database, taking full advantage of PostgreSQL's rich set of server-side programming features.

An example for this is the enrollment in courses. To enroll a student into a course (actually a course instance) many requirements have to be checked (whether a user has passed required courses, whether there are enough free seats, etc.). This function was implemented in the database and included into the Java class *CourseInstancce* using an Aspect named *Enrollment*, which introduces the method *enroll(Person p)* in that class.

PostgreSQL-specific issues The most trouble we had were annoyances and inconsistencies between the promises of the Java SQL API and the implementation of it in the PostgreSQL-JDBC-Driver. For example, `java.sql.Statement.getGeneratedKeys()` returns any auto-generated keys during the last query. However, the implementation of the PostgreSQL-Driver does not support that feature and so we had to work around this short coming using a PostreSQL-specific language extension.

7.2 Controller

In order to implement the Controller and associated interfaces as described in the design section, we had to utilize Javas Reflection API. Problems we faced arose mostly from Javas Type System, especially Generics, which proved to be quite cumbersome to work with.

OutputConverter Thanks to a clean design it was easier than we thought at first to implement a multitude of different *OutputConverter*. Thus we created a whole lot of them, for XHTML, HTML5 as well as PDF and even text/plain Output.

Shared Code Since many functions were the same across many different modules in the business layer, we created common abstract classes for shared functionality.

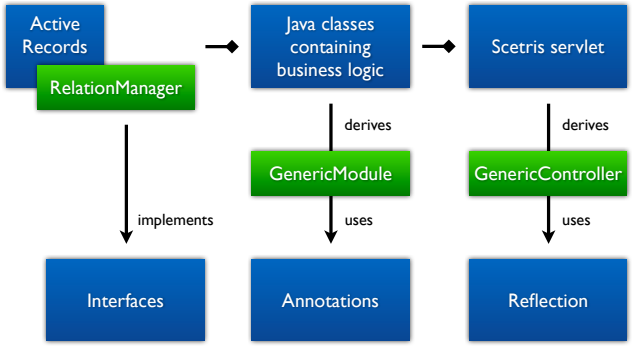


Figure 10: Composition of business logic, controlling logic and data access layer.

7.3 Front end / Templates

The front end was implemented using XSL-T templates. The main idea was to make it easier for us to build the web application. To do so we created lego-bricks – as we called them – our main form building mechanism. It simplified the work in a way, that we just handled little data into the templates and got rich, unified forms in return. As a result we did not need to bother about layout and style, since the templates always created HTML which looked similar, thus a uniform look and feel of the site.

7.4 Scheduler

The major problem of implementing a genetic algorithm is the question of how to represent the solutions. The genetic algorithm operations presented can only be applied when a fitting data model has been designed. Traditionally an array is used. The *crossover* operation applied on two candidate solutions leads to mixing these values with each other. The *mutation* operation on one candidate solution leads to randomly changing values of the array.

A generic approach to solve this problem is to encode the data model into a byte representation. But since we had chosen AspectJ respectively Java to implement the scheduler we wanted to use the concept of object-orientied programming and all its advantages. Fortunately there was a reference solution[22] implemented in C++, which we based our implementation on. The candidate solutions are represented by using a *Map*.

$$CourseElementInstance \mapsto (Room, TimeSlot)$$

Every course is mapped to a tuple defined by a room and a time slot. The time slot is the starting time slot of the course [Figure 11].

[22] `http://www.codeproject.com/KB/recipes/GaClassSchedule.aspx` – Homepage of The Code Project with example implementation of genetic algorithm

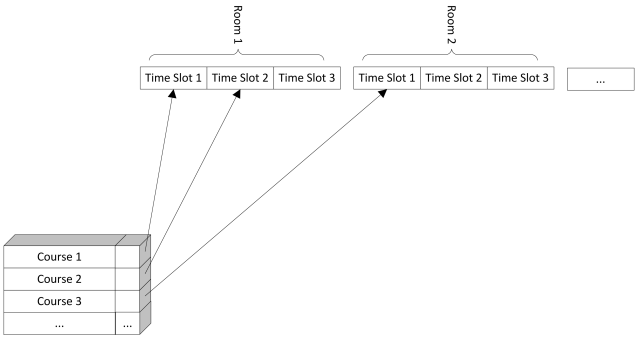


Figure 11: Every Course is mapped to a tuple defined by a room and a time slot.

As a matter of fact this is not sufficient to model the whole candidate solution. Behind this data model lies another data model modeling the whole timetable. The data model is a list of rooms with each having a further list with time slots. On every time slot there is another list with courses allocated to this position. Instead of a single course a list of courses was chosen because there is the possibility of having one course in the same room at the same time.

$$[(Room, [(TimeSlot, [CourseElementInstance])]$$

However, applying *crossover* and *mutation* only takes effect on the *Map*. An example of executing *crossover* on the designed data model is illustrated in Figure 12.

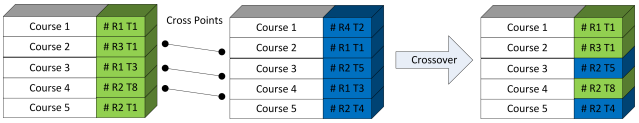


Figure 12: Crossover operation: A specified amount of cross points is chosen randomly. The courses are iterated and taken over to the offspring candidate solution. When a cross point is met the other candidate solution being crossed over is used to take over its course allocations.

8 The Application

All in one the application provides the following functionality (illustration follows)

- login/logout and get detailed information about your personal profile
- create, read, delete and modify data related to schedules
- automatic validation of all forms

In the implementation phase it turned out using a *setup* allocating the course completely random works for little input very well. With increased input, however, the results by the classical *setup* lead to quite worse results. These results are supposed to be optimized by the phase of applying *crossover* and *mutate*. As a matter of fact this happens to a certain degree but starts to converge to a certain score.

Therefore an alternative *setup* was implemented. *Greedy setup* combines the approach of a genetic algorithm using metaheuristic with the Greedy algorithm being a subset of the combinatorial algorithm.

Greedy Setup generates the initial population of candidate solutions by using a Greedy algorithm. Every course is allocated by placing it to the room which fits its constraints best, for instance the requirement for a specified amount of seats. In order to avoid choosing rooms which exceed the required constraints every room is rated and sorted. In order to avoid overlapping in space and time the courses are placed one after another in the timetable.

There is the possibility of finding an optimal solution in the phase of *Greedy setup*. Therefore the *fitness function* is applied on every generated candidate solution. If an optimal schedule has been found the algorithm terminates.

- starting, resuming of schedulings with progress indicator
- view timetables for a given room, a *CourseInstance* and every user
- import and export of course-related data
- view courseinstance data and enroll in them
- full accessibility on current browsers, even without JavaScript

Figure 13: Basic layout of our application

Figure 14: Login: input fields provide user with information on what to fill in

Figure 15: Login status is tracked on every page. Navigation provides submenus when hovering over items.

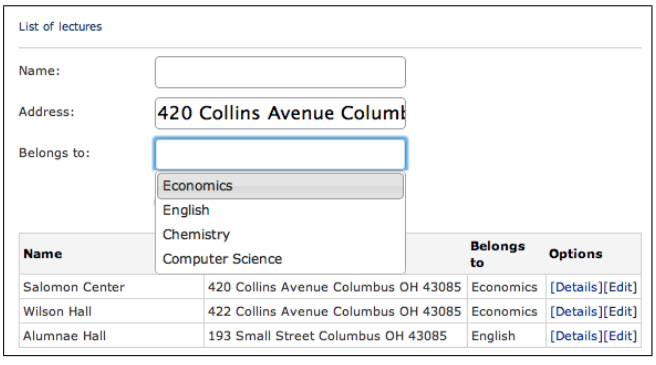


Figure 16: With JavaScript activated selections morph into textfields with automatic proposals.

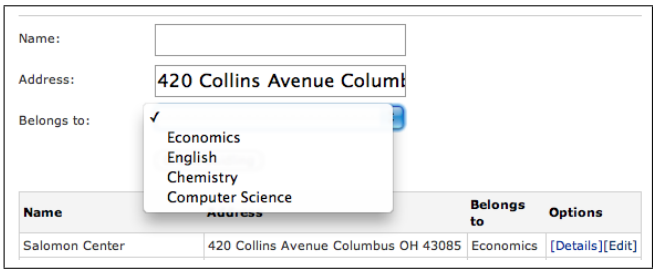


Figure 17: Without JavaScript the standard selection is used.

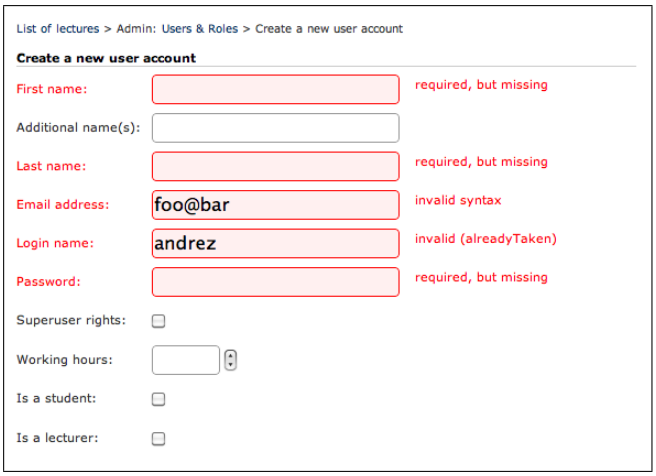


Figure 18: Form validation: no data can be send until all problems were solved.

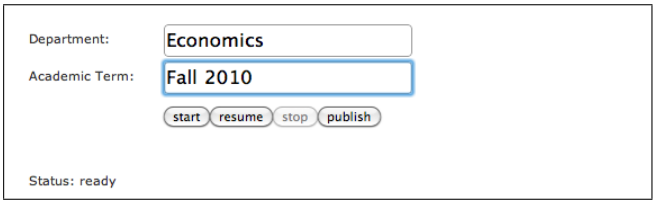


Figure 20: If the scheduler is not running, it's possible to schedule the academic term for a given department.

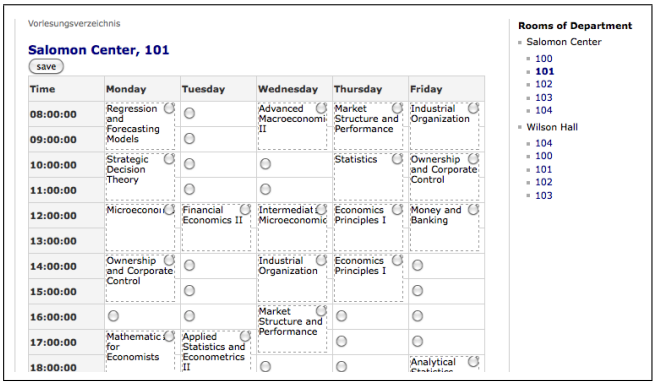


Figure 22: Manual changes: after scheduling the program manager can also swap positions of courses.

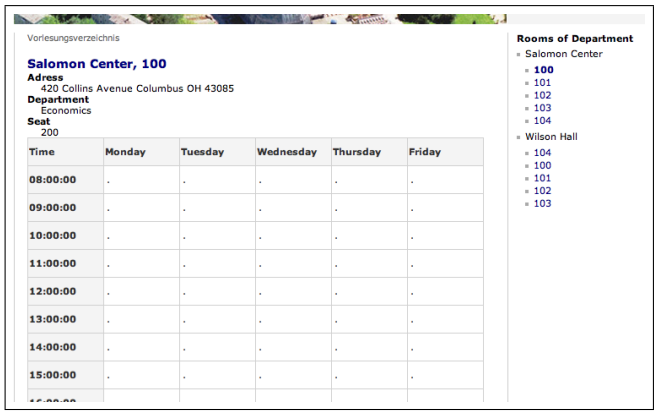


Figure 19: Before scheduling: CourseElementInstances have no time or room assigned, all rooms are free.

Figure 21: A progress bar indicates how many constraints are already solved.

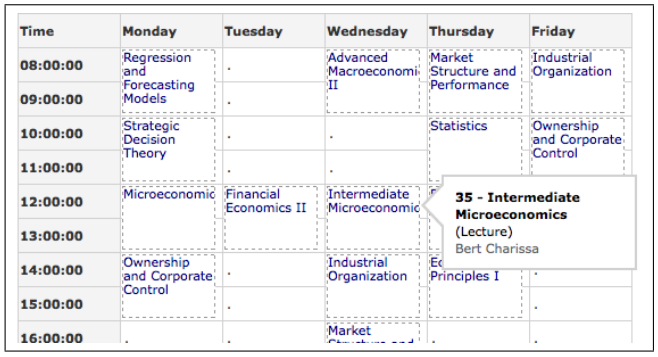
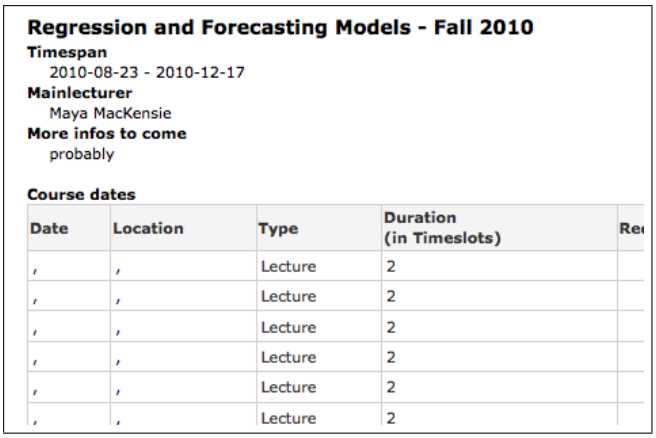


Figure 23: After scheduling: every room contains several CourseElementInstances. Hovering over an event provides the user with data about the event as well as a direct link to the related CourseInstance.



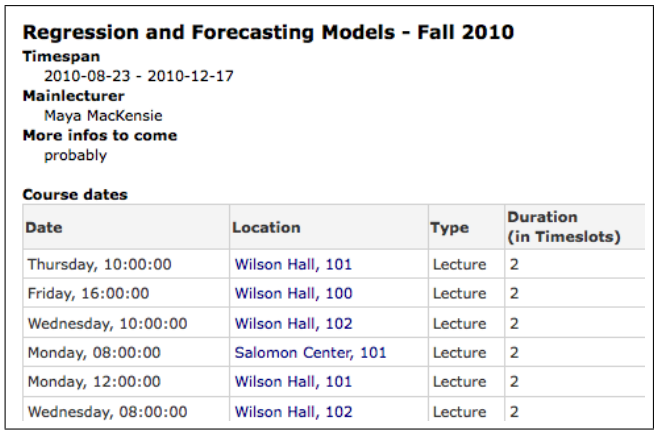
Regression and Forecasting Models - Fall 2010

Timespan
2010-08-23 - 2010-12-17
Mainlecturer
Maya MacKensie
More infos to come
probably

Course dates

Date	Location	Type	Duration (in Timeslots)	Re
,	,	Lecture	2	
,	,	Lecture	2	
,	,	Lecture	2	
,	,	Lecture	2	
,	,	Lecture	2	
,	,	Lecture	2	

Figure 24: Before publishing of the scheduling every CourseInstance may be inspected but has no dates for their CourseElementInstances.



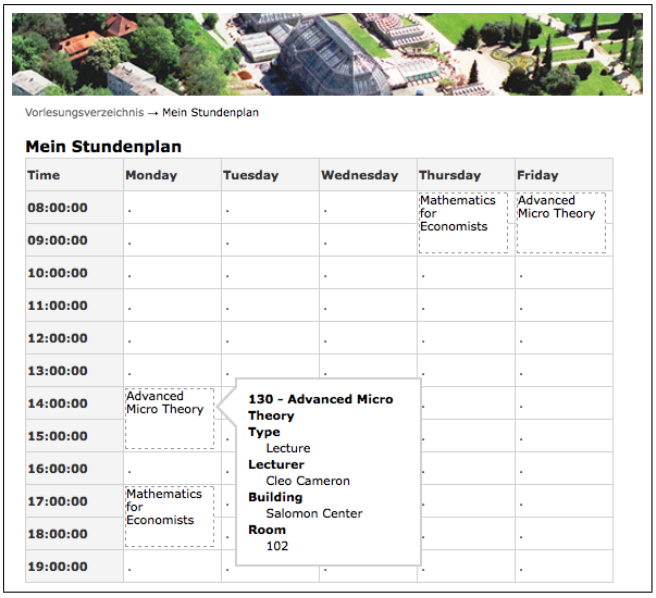
Regression and Forecasting Models - Fall 2010

Timespan
2010-08-23 - 2010-12-17
Mainlecturer
Maya MacKensie
More infos to come
probably

Course dates

Date	Location	Type	Duration (in Timeslots)
Thursday, 10:00:00	Wilson Hall, 101	Lecture	2
Friday, 16:00:00	Wilson Hall, 100	Lecture	2
Wednesday, 10:00:00	Wilson Hall, 102	Lecture	2
Monday, 08:00:00	Salomon Center, 101	Lecture	2
Monday, 12:00:00	Wilson Hall, 101	Lecture	2
Wednesday, 08:00:00	Wilson Hall, 102	Lecture	2

Figure 25: After the program manager published and freezed the scheduling it is possible to enroll on any given CourseInstance.



Vorlesungsverzeichnis → Mein Stundenplan

Mein Stundenplan

Time	Monday	Tuesday	Wednesday	Thursday	Friday
08:00:00	.	.	.	Mathematics for Economists	Advanced Micro Theory
09:00:00	.	.	.		
10:00:00	.	.	.	.	.
11:00:00	.	.	.	.	.
12:00:00	.	.	.	.	.
13:00:00	.	.	.	.	.
14:00:00	Advanced Micro Theory	.	.	.	.
15:00:00		.	.	.	.
16:00:00	.	.	.	.	.
17:00:00	Mathematics for Economists	.	.	.	.
18:00:00		.	.	.	.
19:00:00	.	.	.	.	.

130 - Advanced Micro Theory
Type
Lecture
Lecturer
Cleo Cameron
Building
Salomon Center
Room
102

Figure 26: Once the user enrolls in some CourseInstances the user can view its personal timetable for the week.

9 Verification and validation

This section will cover testing and other measurements which were taken to reassure ourselves of the correctness of our implementations. However, performance measurement is also part of verification and validation.

9.1 Unit tests using JUnit and automated testing using Cobertura

In order to test the correctness of *MyCourses* we applied JUnit. Additonally we used Cobertura to check the code and branch coverage of each unit test. Since total code or even branch coverage was a too time intense task, we decided to concentrate the unit tests on only the most important parts of *MyCourses*. This encompass unit testing for the scheduler and the complete data access layer where the tasks are most critical.

Testing the scheduler for correctness is a non-trivial task. Not only does the presence of a genetic algorithm increase the complexity, but its

random-factor also makes it hard to test conditions which have to be satisfied at all times. One approach to improve the testability of the scheduler is the introduction of a seeded random generator. Whenever randomness is applied in the scheduler, it can be reproduced by using the same seed.

In order to achieve a high guarantee of the schedulers correctness JUnit tests were implemented as fine-granular as possible. This implies unit tests for every component of the scheduler. Unfortunately, unit tests allow only little input testing. Data-driven testing is a more promising approach for testing the scheduler.

As we were in need of real world tests we wrote a test data generator. Data sets were entered from the New York University[23] Course Schedule Search[24].

9.2 Performance and Benchmarking

Performance, especially of the scheduler, can be easily tested by test runs with varying size of data input. However this gives no detailed information about the actual performance of separate components. For that reason the profiler **JProfiler** was used to inspect the Java Virtual Machine while executing the program.

This lead to a step-by-step process of picking up the slowest component display in the profiler and trying to improve its performance. In some cases this procedure gave us huge performance boosts.

A weak point of the scheduling performance was the rating of candidate solutions. When checking the constraint satisfactions many data sets have to be queried. The emerging I/O blocking slows the scheduler down. We coped with this task by introducting a query cache.

Also we provided a benchmark of our own, which runs the scheduler with different seeds and different configurations regarding the algorithm and regarding the size of the data sets. For instance the department *Economics* has the following data to be scheduled:

Courses:	43
CourseElements:	62
CourseInstances:	43
CourseElementInstances:	181

Without further constraints, but the default constraints *not more then 1 course in the same room at the same time* and *the course lecturer teaches no course at the same time*, the computation is often done in less than 15 minutes.

However, with the increase of custom hard and soft constraints the probability of finding an optimal schedule in a reasonable time decreases dramatically. To some extend the optimization done by the genetic algorithm depends on luck solving remaining conflicts.

But as the scheduler provides a fair preschedule we give the responsibility to the user solving remaining conflicts.

10 Outcomes & lessons learned

For all of us this was the first project of this dimension and duration. We took the chance to put into practice what we had learned in our software engineering course – in particular going through the whole software development cycle. We found that project oriented approaches are hard to adhere to in an educational setting. Our team could not meet as often as we wanted and our studies interferred much more with our project than we had imagined.

Thus, working on a project like this one was a very challenging task. Sometimes it was frustrating to keep working on the project, due to slow progress or heavy workload. But, it was also a very exciting experience. We could not only learn about many technologies we previously did not know, but we also learned a lot about ourselves and about team work. We experienced that we are very different people with different approaches to problems. Everyone learned to get along with the idiosyncracies of the others.

In particular we learned how to develop software collaboratively and maintain a large code base. We learned a lot about configuration man-

[23] `http://www.nyu.edu` – New York University

[24] `http://www.nyu.edu/registrar/listings/` – New York University Course Schedule Search

agement, build process automatization, web development, and unit testing, which are all topics which we heard about at our university but never got to excercise in practice. We recognized how it is not always easy to write unit tests instead of implementing features as there is the feeling of stalling progress.

Since all of this was a new domain for us, we did not manage to stick to our plan, often due to goals that were set too high to begin with. We realized how important it is to meet regularly and how hard it is to coordinate a group of just five students. The meetings had the the strongest effect on the team. While discussing problems and comparing work results we have learned to give constructive feedback to each other. It is now clear to us, that honesty and respect have a great impact on the success of a team.

However we managed to fulfill the major requirements. We are content with what we achieved, since we did have a lot of fun and the final application is a stable and well working piece of software.

References

[1] Barry W. Boehm. A spiral model of software development and enhancement. *Computer*, 21:61–72, 1988.

[2] Sean Luke. *Essentials of Metaheuristics*. Lulu, 2009.

[3] Ali M., S. Zalzala, and Peter J. Fleming. *Genetic algorithms in engineering systems*. Institution of Engineering and Technology (IET), 1999.

[4] Xia Wang, Lin Huang, and Qing Zou. The Research on Multi-Strategy Course Scheduling Algorithm. *Information Science and Engineering, International Conference on*, 0:2419–2422, 2009.