System and Software Architecture Description (SSAD)

Student Scheduling System

Team #06

Douglass Kinnes:	**Project Manager, Quality Focal Point, Implementation Team member**
Alexey Tregubov:	**System Architect, UML Modeler, Implementation Team member**
Mihir Daptardar:	**Operational Concept Engineer, Quality Focal Point, Implementation Team member**
Ihsan Tolga:	**Life Cycle Planner, Feasibility Analyst, Implementation Team member**
Simone Lojeck:	**IV&V, Shaper, Quality Focal Point**

5/2/2013

Version History

Date	Author	Version	Changes made	Rationale
5/2/2013	Alexey Tregubov	1.0	• Document created. • Filled in sections 1, 2.1.1 – 2.1.4	• Created initial version of the document.
10/21/2012	Alexey Tregubov	1.1	• Fixed defects 7258, 7261, 7262 • Added section 2.2	• defects are fixed • new section for draft FCP added.
10/27/2012	Alexey Tregubov	1.2	• Fixed defect 7263. • Updated artifact diagram in section 2.1.2	• defects are fixed • added new information on the diagram.
10/30/2012	Alexey Tregubov	1.3	• Fixed defects that were pointed out by TA.	• Fixed defects that were pointed out by TA.
11/14/12	Alexey Tregubov	1.4	• Fixed defects 7602 • Fixed table formatting	• Updated status of the document.
11/26/12	Alexey Tregubov	1.5	• Fixed defects: 7754-7759, 7743, 7744, 7747, 7749, 7750, 7752 • Sections 3, 4, and 5 were added.	• Fixed defects that were pointed out by TA after grading FCP. • Updates for DC package.
12/10/12	Alexey Tregubov	1.6	• Updated design class diagrams • Updated deployment diagram (browser versions included)	• ARB recommendations are included.
02/11/13	Alexey Tregubov	2.0	• Updated design class diagrams (typos fixed) • Algorithm design added.	• ARB recommendations are included.
02/20/13	Alexey Tregubov	2.1	• Pseudo code added • Pseudo code description added	• Client's request resolved. • Pseudo code description answers client's questions.
02/24/13	Alexey Tregubov	2.2	• All the hand-written formulas retyped. • Added description for each formula for each constraint. • Pseudo code description shaped.	• Client's request resolved.
03/15/13	Alexey Tregubov	2.3	• Pseudo code updated (comments added).	• Client's request resolved.
03/31/13	Alexey Tregubov	2.4	• UML diagrams updated. • Section 3 updated.	• Preparation for IOC and CCD.
04/24/13	Alexey Tregubov	2.5	• Version and file name updated. • ER diagram updated	• Preparation for code review.
05/02/13	Alexey Tregubov	3.0	• Description of each package in the project added.	• Preparation of transition set.

Table of Contents

Table of Tables

Table of Figures

1. Introduction

1.1 Purpose of the SSAD

The purpose of the SSAD is to document the results of the object-oriented analysis and design (OOA&D) of the Student Scheduling System. The SSAD is used by the builder (programmer/developer) as reference to the system architecture. The Student Scheduling System should be faithful to the architecture specified in this document. Furthermore, the SSAD is used by the maintainer and clients to help understand the structure of the system once the proposed system is delivered.

1.2 Status of the SSAD

This is a SSAD – version 3.0 for IOC #3 package (transition set). All sections are completed.

2. System Analysis

2.1 System Analysis Overview

The purpose of the Student Scheduling System is to so save students' and advisers' time spent on constructing study plan at Stevens University of Technology. The system is intended to achieve goal though automation of study plan construction. Course directors enter information in the system about the courses and degree requirements for each of three degrees (CS, SyS, IS) offered at Stevens department of Computer Science. After that students are able to enter their requirements for the study plan in the system such as desired year of graduation, preferred elective courses, and transfer credits; the system makes attempt to construct study plan satisfying these requirements. If it is not possible to find schedule satisfying the requirements the system suggests the student to relax constrains for the study plan and repeat attempt to construct the study plan.

2.1.1 System Context

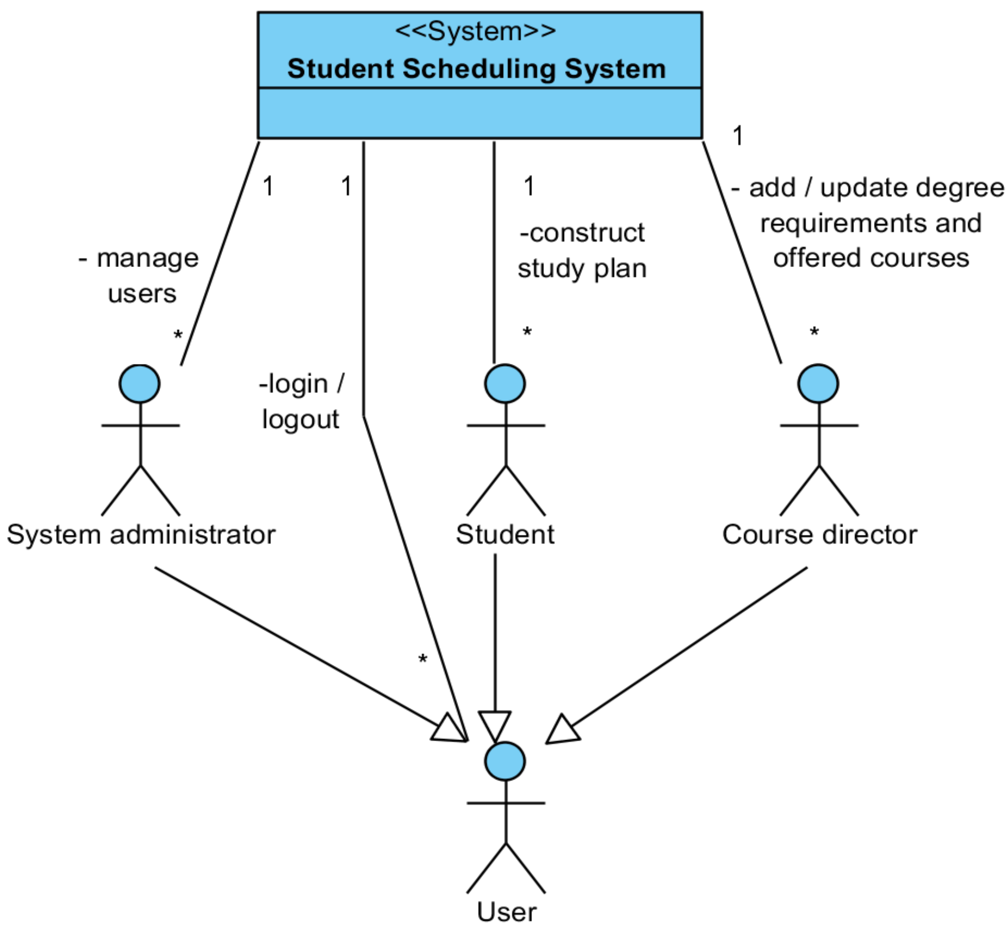

Figure 1: System Context Diagram

Table 1: Actors Summary

Actor	Description	Responsibilities
User	Any user of the system	• Log in / log out
Student	Stevens' students that use the system to construct study plan.	• Enter constraints for the study plan (year of graduation, electives, and so on) and request the system to construct the study plan.
Course director	Stevens' faculty staff who is responsible for adding and updating available courses and degree requirements.	• Add and/or update available courses and degree requirements.
System administrator	Stevens' staff who is responsible for adding, deleting, and updating user profile (accounts).	• Add, delete, and update user profile.

2.1.2 Artifacts & Information

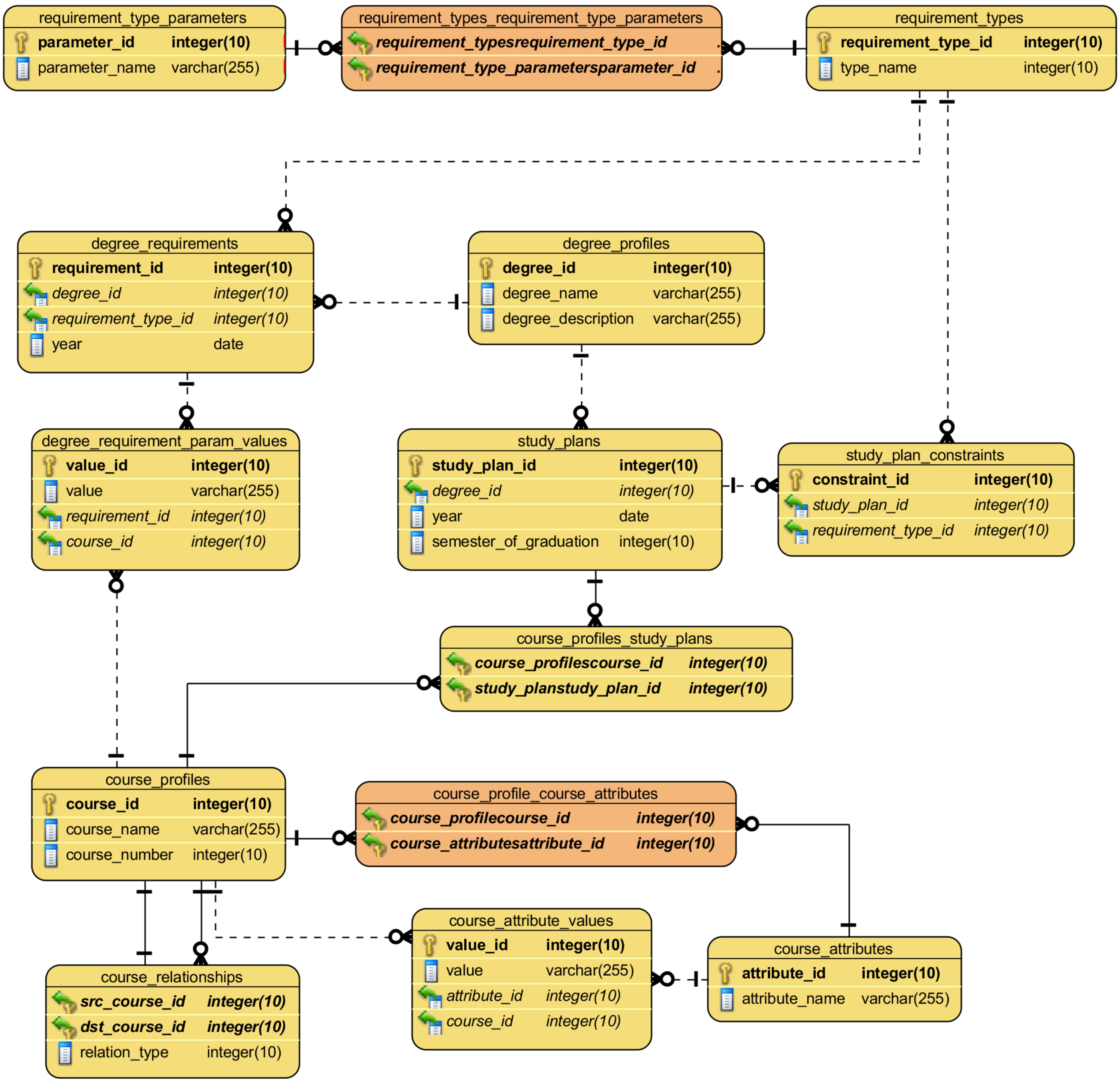

Figure 2: Artifacts and Information Diagram

In the Figure 2 the following data types were used:

- Integer – integer number
- Date – date
- Varchar – string

For this diagram default Visual Paradigm data types were used (there were no *string*, only *varchar* was available). Only Professional Edition of VP allows changing default data types.

Table 2: Artifacts and Information Summary

Artifact	Purpose
Study plan	Contains schedule of courses that satisfies student's constraints. That is a final artifact which student wants to get.
Study plan constraint	Constraints study plan construction which allow student to specify his/her desires such as preferred electives, year of graduation, etc.
Course profile	Contains description of the course and its attributes such as number of credits, level, type (CS, HUM, …). These course profiles are used to specify Study plan constraints (such as preferred electives) and Degree requirements.
Degree requirement	Represents a constraint for the study plan satisfying the requirements of a particular degree. All the Degree requirements for a particular degree comprise a set of all possible study plans solutions. Student adds his/her constrains to this set.
Degree profile	Contains general information about the degree (such as name). Student has to choose degree (one of the three CS, SyS, IS) and then specify his/her constraints for study plan.

2.1.3 Behavior

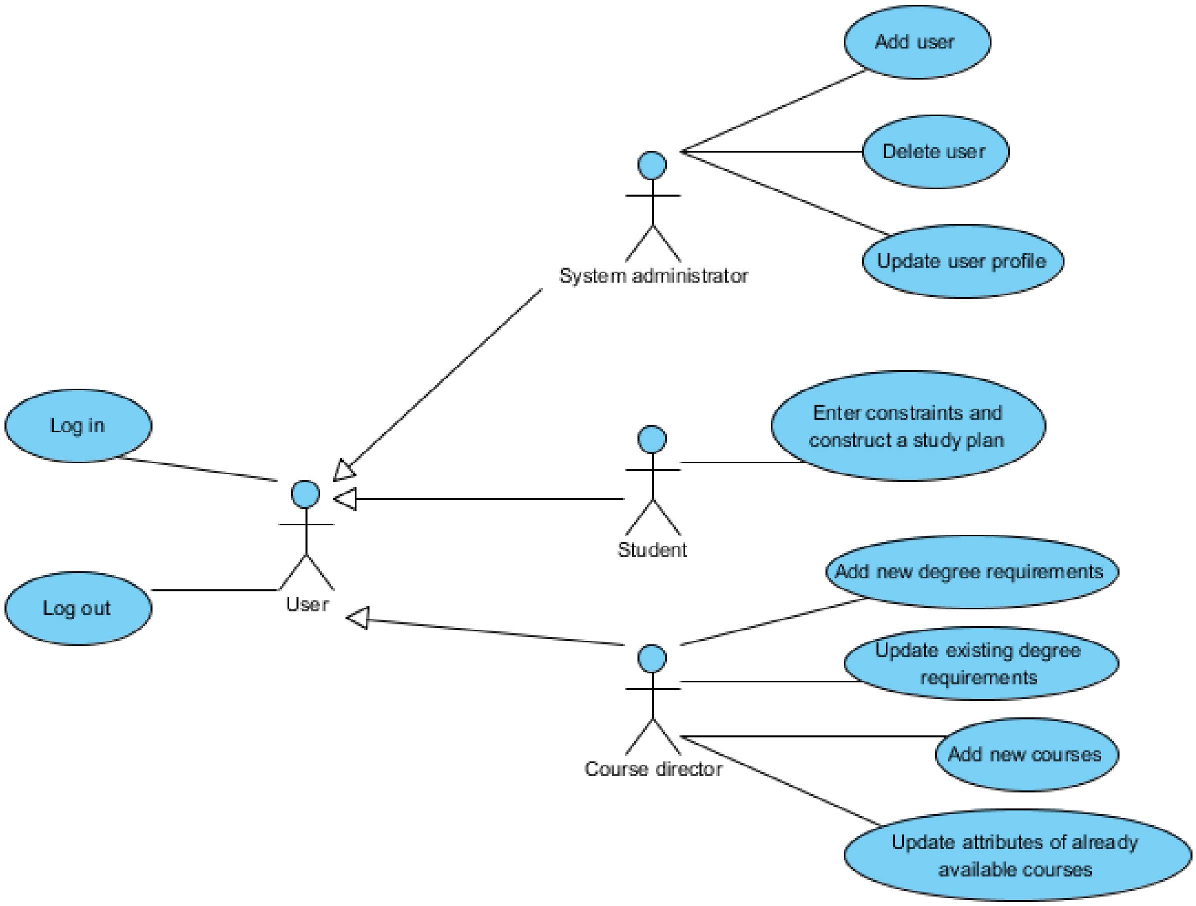

Figure 3: Process Diagram

2.1.3.1 Authentication

2.1.3.1.1 Log in

Table 3: Process Description

Identifier	UC-1: Log in
Purpose	Authorize the user to log in the system and assign the user associated system role.
Requirements	WC_1533: System must provide user privileges according his/her role (student/administrator).
Development Risks	None
Pre-conditions	Log in page is opened.
Post-conditions	Authorized users get access to the system, and initial page will be shown. Unauthorized users will be denied.

Table 4: Typical Course of Action

Seq#	Actor's Action	System's Response
1	Enters a username and password on the login page.	
2	Clicks the "Sign in" button.	
3		System retrieves information associated with given username (role, password hash) form DB and checks password (password hash). System ensures that credentials are correct (they are correct), and then it shows initial page according to the user's role.

Table 5: Exceptional Course of Action

Seq#	Actor's Action	System's Response
1	Enters a username and password on the login page.	
2	Clicks the "Sign in" button.	
3		System retrieves information associated with given username (role, password hash) form DB and checks password (password hash).System ensures that credentials are not correct. Then system shows the following message on the same page: "Username or password is wrong. Please try again."
4		System shows log in page again.

There is no Alternate Courses of Action. Blank fields are treated as wrong username or password.

2.1.3.1.2 Log out

Table 6: Process Description

Identifier	UC-2: Log out
Purpose	Authorize the user to log in the system and assign the user associated system role.
Requirements	WC_1533: System must provide user privileges according his/her role (student/course director/system administrator).
Development Risks	None.
Pre-conditions	User logged in the system. User can be on any page (except login page, because user is already in the system); “Log out” link is available on every page.
Post-conditions	User’s session is closed, and log in page will be shown.

Table 7: Typical Course of Action

Seq#	Actor’s Action	System’s Response
1	Clicks the “Sign out” button.	
2		System shows message that session was successfully closed, and then it shows log in page again.

There is no Exceptional or Alternate Courses of Action. Blank fields are treated as wrong username or password.

2.1.3.2 User management

2.1.3.2.1 Add user

Table 8: Process Description

Identifier	UC-3: Adding new user.
Purpose	Give the administrator ability to add new user in the system.
Requirements	WC_1533: System must provide user privileges according his/her role (student/course director/system administrator). User is on the initial page.
Development Risks	None.
Pre-conditions	User successfully entered the system as a system administrator. User is on initial page.
Post-conditions	One user profile is added to the system.

Table 9: Typical Course of Action

Seq#	Actor's Action	System's Response
1	Clicks the button "Add new user"	
2		System shows empty user profile.
3	Enters user's name, login, role, and password.	
4	Clicks the button "Add"	
5		System checks correctness of the data (it checks repeating logins, password constraints). Data is correct. System adds new user and shows confirmation that user was added successfully.

Table 10: Exceptional Course of Action

Seq#	Actor's Action	System's Response
1-4	Actions and responses are the same as in typical scenario.	
5		System checks correctness of the data (it checks repeating logins, password constraints). Data is incorrect (for example, login already exists). System shows appropriate message (eg. "Login 'user' already exits. Please use another login."). System shows the same page.

There is no Alternate Course of Action.

2.1.3.2.2 Delete user

Table 11: Process Description

Identifier	UC-4: Deleting user.
Purpose	Give the administrator ability to delete existing user from the system.
Requirements	WC_1533: System must provide user privileges according his/her role (student/course director/system administrator). User is on the initial page.
Development Risks	None.
Pre-conditions	User successfully entered the system as a system administrator. System contains user for deletion. Edit user profile page is opened.
Post-conditions	One user profile is deleted from the system.

Table 12: Typical Course of Action

Seq#	Actor's Action	System's Response
1	Presses the button "Delete user"	
2		System requests confirmation for deletion the user.
3	Confirms deletion	
4		System deletes the user and shows confirmation that user was deleted successfully. System redirects user to the initial page.

Table 13: Exceptional Course of Action

Seq#	Actor's Action	System's Response
1-3	Actions and responses are the same as in typical scenario.	
4		The user cannot be deleted (DB is not accessible or user was not found). System shows appropriate message (ex. "DB is accessible" or "User was not found. Probably it was already deleted")

There is no Alternate Courses of Action that have special processing.

2.1.3.2.3 Update user profile

Table 14: Process Description

Identifier	UC-5: Updating user profile.
Purpose	Give the administrator ability to update existing user from the system (reset password, change name, and so on).
Requirements	WC_1533: System must provide user privileges according his/her role (student/course director/system administrator). User is on the initial page.
Development Risks	None.
Pre-conditions	User successfully entered the system as a system administrator. System contains user for updating. . Edit user profile page is opened.
Post-conditions	One user profile is updated.

Table 15: Typical Course of Action

Seq#	Actor's Action	System's Response
1	Updates fields (for example, name or new password) and press the button "Save"	
2		System checks correctness of the data (repeating login, password constraints). Data is correct. System updates user profile and confirmation that user was added successfully.

Table 16: Exceptional Course of Action

Seq#	Actor's Action	System's Response
1	Actions and responses are the same as in typical scenario.	
2		System checks correctness of the data (repeating login, password constraints). Data it is incorrect (for example, login already exists). System shows appropriate message ("Login 'user' already exists. Use another logint") on the same page.

There is no Alternate Course of Action.

2.1.3.3 Study plan construction

2.1.3.3.1 Entering student defined constraints and study plan construction

Table 17: Process Description

Identifier	UC-6: Entering student defined constraints and study plan construction.
Purpose	Give the student a tool to specify his/her desires (degree, year of graduation, elective courses, and so on) and construct a study plan, which satisfies these constrains.
Requirements	WC_1533: System must provide user privileges according his/her role (student/administrator). WC_1354: Student requests that system concstructs schedule. WC_1512: System must be able to construct a study plan based upon the inputs from the student within the degree requirements maintained by the advisers. WC_1355: System will return issue resolution information (list of issues with the student's inputs and/or alternate plan and suggestions), if a solution could not be determined.
Development Risks	Construction of the study plan may be algorithmically difficult. User interface for specifying all the possible constrains is difficult to implement. It may require extra time.
Pre-conditions	User successfully entered the system as a student. User is on the Degree program selection page.
Post-conditions	Student got a study plan or notification that it is impossible to construct it.

Table 18: Typical Course of Action

Seq#	Actor's Action	System's Response
1	Chooses degree and year (it is not a year of graduation; system will use degree requirements for this year)	
2		System shows list of requirements such as mandatory courses, electives that student may choose.
3	Enters desired semester of graduation, coursed for which s/he has credits, desired electives, and presses the button "Construct study plan"	
4		System makes an attempt to construct study plan. Study plan was found. The system shows the study plan.

Table 19: Alternate Course of Action

Seq#	Actor's Action	System's Response
1-3	Actions and responses are the same as in typical scenario.	
4		System makes an attempt to construct study plan. It is impossible to construct study plan. The system shows a message "Unfortunately it is impossible to construct a study plan satisfying all the constraints. Try to relax constraints (for example, let the system choose elective courses) and repeat search again".
5		Then the system shows the page with the study plan constraints.

There is no Exceptional Courses of Action.

2.1.3.4 Entering and modification of the degree requirements

2.1.3.4.1 Add new requirement

Table 20: Process Description

Identifier	UC-7: Adding new requirement
Purpose	Give the course director a tool to specify new degree requirement.
Requirements	WC_1350: Administrator can input degree requirements for each year of entry and degree combinations. WC_1329: As an administrator I must be able to enter degree requirements as complex as those that have been in effect, for the CS, IS, and CyS undergrad degrees and are listed, on the Stevens CS dept. website as well as additional clarifying details provided by the client to the team.
Development Risks	Construction of the study plan may be algorithmically difficult. User interface for specifying all the possible constrains is difficult to implement. It may require extra time.
Pre-conditions	User successfully entered the system as a course director. User is on Add new requirement page.
Post-conditions	Updated degree requirements.

Table 21: Typical Course of Action

Seq#	Actor's Action	System's Response
1	Chooses requirement type from the available list of types	
2		System shows parameters which must be specified for the requirement. (For example, if type is "Select *n* courses from the list" we need to specify *n* and the list of the courses. If it is a composition of other requirements then user need to choose other requirements)
3	Specifies necessary parameters and press the button "Add"	
4		System checks correctness of the data. Data is correct. The system updates requirement and shows confirmation that new requirement was added successfully.

Table 22: Exceptional Course of Action

Seq#	Actor's Action	System's Response
1-3	Actions and responses are the same as in typical scenario.	
4		System checks correctness of the data. Data it is incorrect (for example, not all fields are field in). The system shows appropriate message on the same page.

There is no Alternate Courses of Action.

2.1.3.4.2 Update requirement

Table 23: Process Description

Identifier	UC-8: Updating the requirement
Purpose	Give the course director a tool to change existing degree requirement.
Requirements	WC_1350: Administrator can input degree requirements for each year of entry and degree combinations. WC_1329: As an administrator I must be able to enter degree requirements as complex as those that have been in effect, for the CS, IS, and CyS undergrad degrees and are listed, on the Stevens CS dept. website as well as additional clarifying details provided by the client to the team.
Development Risks	Construction of the study plan may be algorithmically difficult. User interface for specifying all the possible constrains is difficult to implement. It may require extra time.
Pre-conditions	User successfully entered the system as a course director. User is on Edit requirement page.
Post-conditions	Updated degree requirements.

Table 24: Typical Course of Action

Seq#	Actor's Action	System's Response
1	Updates requirement's parameters and presses the button "Save"	
2		System checks correctness of the data. Data is correct; system updates user profile and shows confirmation that the requirement was updated successfully.

Table 25: Exceptional Course of Action

Seq#	Actor's Action	System's Response
1	Actions and responses are the same as in typical scenario.	
2		System checks correctness of the data. Data is incorrect (for example, not all fields are field in) system shows appropriate message on the same page.

There is no Alternate Courses of Action.

2.1.3.4.3 Add new course

Table 26: Process Description

Identifier	UC-9: Adding new course in the system.
Purpose	Give the course director a tool to change existing degree requirement.
Requirements	WC_1349: Administrator can input information about individual courses.
Development Risks	None.
Pre-conditions	User successfully entered the system as a course director. User is on Add new course page.
Post-conditions	Updated list of the available courses.

Table 27: Typical Course of Action

Seq#	Actor's Action	System's Response
1	Enters course attributes such as name, number of credits, type and presses the button "Save"	
2		System checks correctness of the data. Data is correct; system updates user profile and shows confirmation that the requirement was added successfully.

Table 28: Exceptional Course of Action

Seq#	Actor's Action	System's Response
1	Actions and responses are the same as in typical scenario.	
2		System checks correctness of the data. Data is incorrect (for example, not all fields are field in); system shows appropriate message on the same page.

There is no Alternate Courses of Action.

2.1.3.4.4 Update the course

Table 29: Process Description

Identifier	UC-10: Updating the course attributes in the system.
Purpose	Give the course director a tool to change existing degree requirement.
Requirements	WC_1349: Administrator can input information about individual courses.
Development Risks	None.
Pre-conditions	User successfully entered the system as a course director. User is on Edit course page.
Post-conditions	Updated list of the available courses.

Table 30: Typical Course of Action

Seq#	Actor's Action	System's Response
1	Updates course attributes such as name, number of credits, type and presses the button "Save"	
2		System checks correctness of the data. Data is correct; system updates user profile and shows confirmation that the requirement was updated successfully.

Table 31: Exceptional Course of Action

Seq#	Actor's Action	System's Response
1	Actions and responses are the same as in typical scenario.	
2		System checks correctness of the data. Data is incorrect (for example, not all fields are field in); system shows appropriate message on the same page.

There is no Alternate Courses of Action.

2.1.4 Modes of Operation

The Student Scheduling System operates only in one mode.

2.2 System Analysis Rationale

The most counter-intuitive aspect of the system is algorithm for finding a satisfactory study plan. Preliminary analysis of the domain shows that finding a satisfactory study plan in Student Scheduling System is typical combinatorial optimization problem, which is an NP-hard problem. That is why it is really important to be sure that the system is able to find a solution in a reasonable time.

Another interesting aspect of the system is a mathematical model of the study plan, degree requirements and student's desires. As it was mention before Stevens degree requirements are combine a wide range of different requirements of high complexity. For this reason we need to spent additional time on identifying and modeling these requirements.

The process of this analysis includes the following steps:

1. Identification of degree requirement types (informal description)
2. Identification of possible student defined constraints (informal description)
3. Building mathematical model for each constrain in steps 1,2 (formal mathematical model)
4. Developing test case which covers most of the constraints (description of degree requirements for one degree and description of student desires for study plan).
5. Refining UI prototype for the test case.
6. Developing application for finding study plan.

These steps allow getting feedback from the client as soon as possible and mitigating risks associated with constraint solving complexity.

3. Technology-Independent Model

Since we need to satisfy technology-dependent requirements defined by client stakeholders (otherwise they will not be able to maintain the system), this section is partially skipped.

3.1 Algorithm design

The purpose of this section is to describe algorithm for study plan construction. Scheduling problem that system has to solve is formally known as combinatorial optimization problem. It is NP-hard.

There two main strategies to solve this problem:

- Constraint Programming – backtracking. In worst case it may have brute force efficiency. However, this approach can be significantly improved by using heuristics.
- Integer Linear Programming – is a mathematical optimization or feasibility program in which all of the variables are restricted to be integers. It is NP-hard as well; however, it is possible to try to apply Linear Programming (without integer constraint) and round solution to integers. This approach can work faster, but there is no guarantee that result would be found.

3.1.1 Mathematical model of the solution and constraints

This section defines formal mathematical model of study plan, constraints, and solution.

Study plan is a matrix S, where every row is a semester, and every column is a course. Matrix S contains all the courses that can be taken. Each variable $s_{i,j}$ in S defines if course *j* (*j* – is a column number in the matrix S) is taken in semester *i* (*i* – is a row number in the matrix S). If course *j* is taken in semester 1, than $s_{i,j}$=1, else $s_{i,j}$=0. The very first row (*i*=0) of the matrix S defines all the courses were taken by student in the past (it also could be AP credit).

	CS115	CS135	CS185	CS220	CS435	CS510	CS315	CS320	SSW533	SSW564	HUM315	HUM420	MAT240	PHL102
Past (or AP credits)	$s_{0,1}$=0	0	1	0	0	0	0	0	0	0	0	0	0	0
Fall12	$s_{1,1}$=1	1	0	0	0	0	0	0	0	0	1	0	0	0
Spring13	0	0	0	1	0	0	0	0	0	0	0	0	0	0
Fall13	0	0	0	0	1	0	0	0	0	0	0	0	0	1
Spring14	0	0	0	0	0	1	0	1	0	0	0	0	0	0
Fall14	0	0	0	0	0	0	0	0	1	0	0	1	0	0
Spring15	0	0	0	0	0	0	1	0	0	0	0	0	0	0
Fall16	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Spring17	0	0	0	0	0	0	0	0	0	1	0	0	0	$s_{n,k}$=0

Figure 4: Example of Study plan (matrix S)

Courses and semester in the matrix S are enumerated: semesters from 0 to n, courses from 1 to k.

Constraint is any kind of limitation on values in matrix S. All the constraints can be represented as a set of equations/inequalities and their boolean combinations.

The following list contains all the types of constrains that can be elicited from degree requirements, student's desires, and subject domain limitations in general:

- Representation constraint (every $s_{i,j}$=1 or $s_{i,j}$=0 in the matrix S):
  $\forall_{i \in [0;n]\, j \in [1;k]}((s_{i,j} = 1) \vee (s_{i,j} = 0))$.
- General constraints:
  - Some courses are not available in certain semesters. This means that we should consider course unavailability: if course *j* is not available in semester *i* then $s_{i,j}$=0.
  - Every course is taken only once or not taken at all:
    $\forall_{j \in [1;k]}(\sum_{i=1}^{n} s_{i,j} \le 1)$.
  - Every semester student can take only limited number of courses (for example, more than 1 and less than 5 courses per semester):
    $\forall_{i \in [1;n]}(\sum_{j=1}^{k} s_{i,j} \le a_i)$ where a_i is the maximum number of courses that can be taken by student in semester *i*;
    $\forall_{i \in [1;n]}(\sum_{j=1}^{k} s_{i,j} \ge b_i)$ where b_i is the minimum number of courses that can be taken by student in semester *i*.
    This constraint is not applicable for courses taken in the past.
  - Every semester student can take only limited number of credits (for example, more than 12 and less than 24 credits per semester):
    $\forall_{i \in [1;n]}(\sum_{j=1}^{k} (credit_j \times s_{i,j}) \le c_i)$ where c_i is the maximum number of credits that can be taken by student in semester i, and $credit_j$ – is a number of credits that student can get by taking course *j*;
    $\forall_{i \in [1;n]}(\sum_{j=1}^{k} (credit_j \times s_{i,j}) \ge d_i)$ where d_i is the minimum number of credits that can be taken by student in semester i, and $credit_j$ – is a number of credits that student can get by taking course *j*.
    This constraint is not applicable for courses taken in the past.
- Course requisites:
  - Prerequisites:
    - Simple case: *course c1 is a prerequisite of course c2* (*c1* and *c2* are the order numbers of the courses):
      $\forall_{i \in [1;n]}(\sum_{l=0}^{i-1} s_{l,c1} \ge s_{i,c2})$. In the matrix S it looks like this:

	1	…	c1	…	c2	…
0	$s_{0,1}$	…	$s_{0,c1}$	…	$s_{0,c2}$	…
1	$s_{1,1}$	…	$s_{1,c1}$	…	$s_{1,c2}$	…
…	…	…	…	…	…	…
i-1	$s_{i-1,1}$	…	$s_{i-1,c1}$	…	$s_{i-1,c2}$	…
i	$s_{i,1}$	…	$s_{i,c1}$	…	$s_{i,c2}$	…
…	…	…	…	…	…	…

    - Complex case (Boolean combination): course *c1* and *c2* are prerequisites of course *c3* (*c1*, *c2*, and *c2* are the order numbers of the courses). This can be represented as the following combination of constraints:
      (c1 is a prerequisite of c3) and (c2 is a prerequisite of c3).
      As we see this Boolean combination is constructed out of simple constraints such as "*c1 is a prerequisite of c3*". So the final set of inequalities for this complex case will look like:
      $\forall_{i\in[1;n]}(\sum_{l=0}^{i-1} s_{l,c1} \ge s_{i,c3})$ *and* $\forall_{i\in[1;n]}(\sum_{l=0}^{i-1} s_{l,c2} \ge s_{i,c3})$
  - Corequisites:
    - Simple case: *course c1 is a corequisite of course c2* (*c1* and *c2* are the order numbers of the courses):
      $\forall_{i\in[1;n]}(\sum_{l=0}^{i-1} s_{l,c1} \ge s_{i,c2})$ In the matrix S it looks like this:

	1	…	c1	…	c2	…
0	$s_{0,1}$	…	$s_{0,c1}$	…	$s_{0,c2}$	…
1	$s_{1,1}$	…	$s_{1,c1}$	≥	$s_{1,c2}$	…
…	…	…	…	…	…	…
i	$s_{i,1}$	…	$s_{i,c1}$	≥	$s_{i,c2}$	…
…	…	…	…	…	…	…

      If course *c2* has been taken before then there is no need to add any constraint.

    - Complex case (Boolean combination): course *c1* and *c2* are corequisites of course *c3* (*c1*, *c2*, and *c2* are the order numbers of the courses). This can be represented as the following combination of constraints:
      (c1 is a corequisite of c3) and (c2 is a corequisite of c3).
      As we see this Boolean combination is constructed out of simple constraints such as "*c1 is a corequisite of c3*". So the final set of inequalities for this complex case will look like:
      $\forall_{i\in[1;n]}(s_{i,c1} \ge s_{i,c3})$ *and* $\forall_{i\in[1;n]}(s_{i,c2} \ge s_{i,c3})$.
- Degree requirement constraints:
  - Simple requirements (for example "student has to take 1 course from CS235, CS240, CS250"): *N* courses from {*c1, …, ck*}

$\sum_{i=0}^{n} \sum_{j \in \{c1,\ldots,ck\}} s_{i,j} \geq N$, where *N* is a number of courses that student has to take from the list according to the requirement, and *c1, …, ck* are the order numbers of courses from that list.
In the matrix S it may look like this:

	1	…	c1	…	c2	…
0	$s_{0,1}$	…	$s_{0,c1}$	…	$s_{0,c2}$	…
1	$s_{1,1}$	…	$s_{1,c1}$	…	$s_{1,c2}$	…
…	…	…	…	…	…	…
n	$s_{n,1}$	…	$s_{n,c1}$	…	$s_{n,c2}$	…

$\geq N$

- Boolean combinations of the other requirements: requirement1 and requirement2, where requirement1 is N_1 courses from { *c1, …, ck* } and requirement1 is N_2 courses from { *d1, …, dm* }:

  $\sum_{i=0}^{n} \sum_{j \in \{c1,\ldots,ck\}} s_{i,j} \geq N_1 \textit{ and } \sum_{i=0}^{n} \sum_{j \in \{d1,\ldots,dm\}} s_{i,j} \geq N_2$

- Student desires:
  - Already taken courses or AP credits:
    - *c1* has been taken:

      $s_{0,c1} = 1 \textit{ and } \forall_{i \in [1;n]} (s_{i,c1} = 0)$

      In the matrix S it looks like this:

	1	…	c1	…
0	$s_{0,1}$	…	1	…
1	$s_{1,1}$	…	0	…
…	…	…	…	…
n	$s_{n,1}$	…	0	…

    - *c1* has never been taken:

      $s_{0,c1} = 0$

      In the matrix S it looks like this:

	1	…	c1	…
0	$s_{0,1}$	…	0	…
1	$s_{1,1}$	…	?	…
…	…	…	…	…
n	$s_{n,1}$	…	?	…

  - Complete desire for requirement: "Course *X* must be taken in semester *Y* to satisfy requirement *Z*":

    $s_{Y,X} = 1 \textit{ and } \forall_{i \in [1;Y-1] \cup [Y+1;n]} (s_{i,X} = 0)$

    In the matrix S it looks like this:

	1	…	*X*	…
0	$s_{0,1}$	…	0	…
1	$s_{1,1}$	…	0	…
…	…	…	…	…
Y	…	…	1	…
…	…	…	…	…
n	$s_{n,1}$	…	0	…

Course *X* must be excluded from all degree requirements other than *Z*.

- Incomplete desire for requirement: "Course *X* must be taken to satisfy requirement *Z*":

  $s_{0,X} = 0 \;\; and \; \sum_{i=1}^{n} s_{i,X} = 1$

  In the matrix S it looks like this:

	1	…	*X*	…
0	$s_{0,1}$	…	0	…
1	$s_{1,1}$	…	$s_{1,X}$	…
…	…	…	…	…
i	…	…	$s_{i,X}$	…
…	…	…	…	…
n	$s_{n,1}$	…	$s_{n,X}$	…

  Course *X* must be excluded from all degree requirements other than *Z*.

Each cell $s_{i,j}$ in matrix S can be considered as a variable that needs to be found.

NOTE: terms variable and cell are equivalent and can be used interchangeably.

Solution is a combination of variables values (values in the cells) that satisfies all the constraints applicable to study plan (matrix S).

Some variables in matrix S can be determined explicitly from constraints, for example, student's desires and course unavailability; some variables cannot be determined explicitly.

Decision variable is a variable which value cannot be explicitly determined from constraints.

Decision variables are used in solution search by appropriate algorithms such as backtracking (Constraint programming) or Simplex method (Integer linear programming).

3.1.2 Constraint programming approach

In general any Backtracking algorithm incrementally attempts to extend a partial solution that specifies values for some of the decision variables, toward a complete assignment, by repeatedly choosing a value for another variable consistent with the values in the current partial solution.

Choco Solver documentation says: "A key ingredient of any constraint approach is a clever search strategy. In backtracking or branch-and-bound approaches, the search is organized as an enumeration tree, where each node corresponds to a subspace of the search, and each child node is a subdivision of its father node's space. The tree is progressively constructed by applying a series of branching strategies that determine how to subdivise space at each node and in which order to explore the created child nodes. Branching strategies play the role of achieving intermediate goals in logic programming" (2012, Choco Solver documentation, p.33).

Standard backtracking approach in constraint programming develops the enumeration tree in a Depth-First Search (DFS) manner:

1. evaluate a node: run propagation
2. if a failure occurs or if the search space cannot be separated then backtrack : evaluate the next pending node
3. otherwise branch: divide the search space and evaluate the first child node.

More detailed algorithm for study plan construction described below. The following pseudo code uses the following notation:

- $s_{i,j}$ `<- 0` variable is assigned to value
- `DecVars-{`$s_{i,j}$`}` set subtraction
- `{`$s_{i,j}$`/1}` variable $s_{i,j}$ assigned to value 1 in the set

Input data:

- `S` empty matrix S.
- `Constraints` set of constraints (see section 3.1.1 for definition)
  Variable `Constraints` contains only the following types of constraints:
  - Course requisites
    - Prerequisites.
    - Corequisites.
  - Degree requirement constraints:
    - Simple requirements: "student has to take 3 courses from … "
    - Boolean combinations of the other requirements.
  - Student desires:
    - Inomplete desire for requirement: "Course X must be taken to satisfy requirement Z"

  General constraints are verified during consistency check (procedures `consistent_1` and `consistent_2`).

```
procedure construct_schedule(S, Constraints)
      S <- construct_rows_and_columns(S) // each row in the matrix S
            would be associated with corresponding semester and each
            column would be associated with a course.
      initialize courses that were taken in the past (first row)
      initialize every s_i,j according to courses unavailability
      initialize s_i,j according to student's complete desire for
            requirements
      initialize all s_i,j that are not covered by degree requirement
            constraints or by course pre/corequisites. // this simply
            means that courses which cannot be used to satisfy any
            requirements must be removed from search.
      Solution <- every s_i,j that was initialized // partial solution
      DecVars <- every s_i,j that was not initialized // all decision
            variables
      Constraints <- reorder_constraints(Constraints)
      DecVar <- reorder_vars(DecVar) // reorder decision variables
                                       according to requirements and
                                       chronological order of the
                                       semesters. This defines order of
                                       variables for fronttracking.
      R <- backtrack(DecVars, Solution, Constraints)
      return R
end construct_schedule

procedure backtrack(DecVars, Solution, Constraints) // this procedure
                           performs fronttracking when it recursively
                           calls itself, and it backtracks when it
                           returns Fail result.

      if DecVars = {} then return Solution // no undefined decision
                                              variables (cells) left,
                                              all constraints satisfied.

      pick next s_i,j from DecVars  // selecting next decision variable
                                       for front tracking.
      if constraint that could be satisfied by s_i,j isn't satisfied then
            s_i,j <- 1   // we assign 1 if current constraint isn't
                                                      satisfied yet.
            if consistent_1(Solution+{s_i,j/1}, Constraints) then
                  R <-backtrack(DecVars-{s_i,j}, Solution+{s_i,j/1},
                        Constraints)
                  if R not fail then return R
            end if
            s_i,j <- 0
            if consistent_0(Solution+{s_i,j/0}, Constraints) then
                  R <-backtrack(DecVars-{s_i,j}, Solution+{s_i,j/0},
                        Constraints)
                  if R not fail then return R
            end if
      else
            s_i,j <- 0 // we assign 0 if current constraint was satisfied.
```

```
          if consistent_0(Solution+{si,j/0}, Constraints) then
               R <-backtrack(DecVars-{si,j}, Solution+{si,j/0},
                    Constraints)
               if R not fail then return R
          end if
          si,j <- 1
          if consistent_1(Solution+{si,j/1}, Constraints) then
               R <-backtrack(DecVars-{si,j}, Solution+{si,j/1},
                    Constraints)
               if R not fail then return R
          end if
     end if
     return fail // backtrack to previous variable
end backtrack

     // procedures consistent_1 and consistent_2 check constraints
     // satisfaction.

procedure consistent_1(Solution+{si,j/1}, Constraints)
     R <- check_row(Solution+{si,j/1}, Constraints) // check number of
               courses and credits per semester
     if R fail then return fail
     R <- check_col(Solution+{si,j/1}, Constraints) // check if each
               course is taken once
     if R fail then return fail
     R <- check_prerequisites(Solution+{si,j/1}, Constraints) // check
               if all prerequisites are satisfied
     if R fail then return fail
     R <- check_corequisites(Solution+{si,j/1}, Constraints) // check
               if all corequisites are satisfied
     if R fail then return fail
     for each C in Constraints do
          if C is not satisfied by Solution+{si,j/1} then return fail
     end for
     return true
end consistent_1

procedure consistent_0(Solution+{si,j/0}, Constraints)
     R <- check_row(Solution+{si,j/0}, Constraints) // check number of
               courses and credits per semester
     if R fail then return fail
     R <- check_prerequisites(Solution+{si,j/1}, Constraints) // check
               if all prerequisites are satisfied
     if R fail then return fail
     R <- check_corequisites(Solution+{si,j/1}, Constraints) // check
               if all corequisites are satisfied
     if R fail then return fail
     for each C in Constraints do
          if C is not satisfied by Solution+{si,j/0} then return fail
     end for
     return true
end consistent_0
```

```
procedure construct_rows_and_columns(S)
     put semesters in chronological order as row headers in S

     // column construction:
     put all courses as column headers in S // from now column headers
                                                and courses are synonyms.
     group courses according their belonging to simple requirements,
          courses that cannot be used to satisfy any requirement must
          be put in the end.
     divide table into sections according to course groups constructed
          in the previous step.
     for each course in each section do
          if course has prerequisites then move all courses that
               constitute prerequisite and located after that course
               to current section and place them before that course.
          if course has corequisites then move all courses that
               constitute corequisite and located after that course
               to current section and place them after that course.
     end for
     return S
end construct_rows_and_columns(S)

procedure reorder_vars(DecVar)
     // This defines order of variables for fronttracking. This
          defines the order in which they each variable will be
          assigned to value in the backtrack() procedure.
     array reorderedDecVars;
     int counter <- 0;
     for each section in S do
          for each row in S do
               for each column in S do
                    if s_i,j is a decision variable then
                         put s_i,j in reorderedDecVars;
                         reorderedDecVars[counter] <- s_i,j
                         counter <- counter + 1
                    end if
               end for
          end for
     end for
     return reorderedDecVars
end reorder_vars(DecVar)
```

NOTE: Every time we check constraint satisfiability we imply that some of the constraints can be checked with partial assignment of it variables too. For example, number of courses per semester can be checked before all variables in row are assigned to values.

Branching strategy (front tracking) defines what is the next branch in the search space should be explored. In terms of “courses” it means what course and in what semester it should be taken. In other words, this strategy tells what is the next decision variable $s_{i,j}$ (cell in the matrix S) that needs to be populated with 1 or 0.

Branching strategy in the proposed algorithm is defined in the following procedures:

- `construct_rows_and_columns(S)` – allocates courses from simple requirements, so that corresponding columns in the matrix S were close to each other (see Figure 5).
- `reorder_vars(DecVar)` – enumerates all decision variables from left to right and from top to down within each simple (degree) requirement (see Figure 5).

Figure 5 shows the suggested order of the decision variables. Every cell in the table represents a variable. Cells highlighted with grey represent decision variables, which will be used by backtracking algorithm. Arrows show order in which variables are selected by branching strategy, which we also agreed to call front-tracking algorithm. In other words, arrows show how decision variables would be enumerated by procedure `reorder_vars(DecVar)`.

According to front-tracking algorithm decision variables are arranged according to

- simple requirements they belong to (such as “1 from CS115, CS135, CS185”),
- semesters: variable selected from left to right and from top to down of the table.

Figure 5 shows final state of the variables after algorithm work.

	(2 from …) OR (1 from …)													
	1 from:			3 from:			2 from:				1 from:			
	CS115	CS135	CS185	CS220	CS435	CS510	CS315	CS320	SSW533	SSW564	HUM315	MAT215	MAT240	PHL102
Past (or AP credits)	0	1	0	0	0	0	0	0	0	0	0	0	0	0
Fall12	1	0	0	0	1	0	0	0	1	0	1	0	0	0
Spring13	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Fall13	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Spring14	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Fall14	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Spring15	0	0	0	0	0	0	1	0	0	0	0	0	0	0
Fall16	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Spring17	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 5: Order of the decision variables (gray cells). Arrows show fronttracking rout. Everything that is not grey is excluded from search.

3.1.2.1 Example

Input data:

- Courses: a,b,c,d,e,f,g,h,i,j,k,l,p,q,r.
- Semesters: s1 – fall, s2 – spring, s3 – fall, s4 – spring,
- Course availability:
    - fall: a,c,e,f,h,i,k,l,q.
    - spring: b,d,g,j,p,r.
- Requisites:
    - (b) is a prerequpisite of c
    - (q AND j) are prerequisites of k
    - (l AND f) are prerequisites of d

  - (h) is a corequisite of g
  - (p OR q) is a corequisite of i
- General constraints:
  - No more than 3 courses per semester and upto 4courses in first semester.
- Degree requirements:
  - 2 courses from {a,b,c}
  - 1 course from {d,e,f}
  - (2 courses from {g,h,i}) AND (3 courses from {j,k,l})
- Student desires:
  - AP credits: c,h
  - Course d must be taken in s2
  - Course i must be taken
- Initial state of matrix S (study plan): S = null

The following shows how matrix S is modified by every step of the algorithm:

```
procedure construct_rows_and_columns(S)
    put semesters in chronological order as row headers in S
```

AP credits
s1
s2
s3
s4

```
    put all courses as column headers in S
```

	a	b	d	c	e	f	g	h	i	j	k	l	p	q	r
AP credits															
s1															
s2															
s3															
s4															

```
    group courses according their belonging to simple requirements,
        courses that cannot be used to satisfy any requirement must
        be put in the end.
```

							AND								
	2 from			1 from			2 from			3 from			rest		
	a	b	c	d	e	f	g	h	i	j	k	l	p	q	r
AP credits															
s1															
s2															
s3															
s4															

```
divide table into sections according to course groups constructed
    in the previous step.
```

	AND														
	2 from			1 from			2 from			3 from			rest		
	a	b	c	d	e	f	g	h	i	j	k	l	p	q	r
AP credits															
s1															
s2															
s3															
s4															

```
for each course in each section do
    if course has prerequisites then move all courses that
        constitute prerequisite to current section and place
        them before that course.
    if course has corequisites then move all courses that
        constitute corequisite to current section and place
        them after that course.
end for
```

	a	*b*	**c**	*f*	*l*	**d**	e	**g**	*h*	**i**	*p*	*q*	*j*	**k**	r
AP credits															
s1															
s2															
s3															
s4															

- b comes before c;
- f, l come before d;
- q, j come before K
- h comes after g;
- p, q come after i.

Table that shown above will be returned by `construct_rows_and_columns(S)`

```
procedure construct_schedule(S, Constraints)
    S <- construct_rows_and_columns(S)
```

	a	*b*	**c**	*f*	*l*	**d**	e	**g**	*h*	**i**	*p*	*q*	*j*	**k**	r
AP credits															
s1															
s2															
s3															
s4															

initialize *courses that were taken in the past (first row)*

	a	*b*	**c**	*f*	*l*	**d**	e	**g**	*h*	**i**	*p*	*q*	*j*	**k**	r
AP credits	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0
s1			0						0						
s2			0						0						
s3			0						0						
s4			0						0						

initialize every $s_{i,j}$ according to courses unavailability

	a	*b*	**c**	*f*	*l*	**d**	e	**g**	*h*	**i**	*p*	*q*	*j*	**k**	r
AP credits	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0
s1		0	0			0		0	0		0		0		0
s2	0		0	0	0		0		0	0		0		0	
s3		0	0			0		0	0		0		0		0
s4	0		0	0	0		0		0	0		0		0	

initialize $s_{i,j}$ according to student's complete desire for requirements

	a	*b*	**c**	*f*	*l*	**d**	e	**g**	*h*	**i**	*p*	*q*	*j*	**k**	r
AP credits	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0
s1		0	0			0		0	0		0		0		0
s2	0		0	0	0	**1**	0		0	0		0		0	
s3		0	0			0		0	0		0		0		0
s4	0		0	0	0	0	0		0	0		0		0	

initialize all $s_{i,j}$ that are not covered by degree requirement constraints or by course pre/corequisites. // this simply means that courses which cannot be used to satisfy any requirements must be removed from search.

	a	*b*	**c**	*f*	*l*	**d**	e	**g**	*h*	**i**	*p*	*q*	*j*	**k**	r
AP credits	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0
s1		0	0			0		0	0		0		0		*0*
s2	0		0	0	0	**1**	0		0	0		0		0	*0*
s3		0	0			0		0	0		0		0		*0*
s4	0		0	0	0	0	0		0	0		0		0	*0*

DecVars <- every $s_{i,j}$ that was not initialized *// all decision variables*

	a	*b*	**c**	*f*	*l*	**d**	e	**g**	*h*	**i**	*p*	*q*	*j*	**k**	r
AP credits	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0
s1	?	0	0	?	?	0	?	0	0	?	0	?	0	?	*0*
s2	0	?	0	0	0	**1**	0	?	0	0	?	0	?	0	*0*
s3	?	0	0	?	?	0	?	0	0	?	0	?	0	?	*0*
s4	0	?	0	0	0	0	0	?	0	0	?	0	?	0	*0*

All gray cells constitute set of decision variables – `DecVars`.

`DecVar <- reorder_vars(DecVar)`

	a	*b*	**c**	*f*	*l*	**d**	e	**g**	*h*	**i**	*p*	*q*	*j*	**k**	r
AP credits	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0
s1	?[1]	0	0	?[5]	?[6]	0	?[7]	0	0	?[11]	0	?[12]	0	?[19]	*0*
s2	0	?[2]	0	0	0	**1**	0	?[13]	0	0	?[14]	0	?[20]	0	*0*
s3	?[3]	0	0	?[8]	?[9]	0	?[10]	0	0	?[15]	0	?[16]	0	?[21]	*0*
s4	0	?[4]	0	0	0	0	0	?[17]	0	0	?[18]	0	?[22]	0	*0*

Red numbers show the order in which variables will be assigned to values. Table above represents state of the matrix S before the very first step of backtracking procedure.

After the first step (1st recursive iteration) of backtracking (fronttracking) S will look like:

	a	*b*	**c**	*f*	*l*	**d**	e	**g**	*h*	**i**	*p*	*q*	*j*	**k**	r
AP credits	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0
s1	1[1]	0	0	?[5]	?[6]	0	?[7]	0	0	?[11]	0	?[12]	0	?[19]	*0*
s2	0	?[2]	0	0	0	**1**	0	?[13]	0	0	?[14]	0	?[20]	0	*0*
s3	?[3]	0	0	?[8]	?[9]	0	?[10]	0	0	?[15]	0	?[16]	0	?[21]	*0*
s4	0	?[4]	0	0	0	0	0	?[17]	0	0	?[18]	0	?[22]	0	*0*

After the second step (2nd iteration) of backtracking (fronttracking) S will look like:

	a	*b*	**c**	*f*	*l*	**d**	e	**g**	*h*	**i**	*p*	*q*	*j*	**k**	r
AP credits	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0
s1	1[1]	0	0	?[5]	?[6]	0	?[7]	0	0	?[11]	0	?[12]	0	?[19]	*0*
s2	0	0[2]	0	0	0	**1**	0	?[13]	0	0	?[14]	0	?[20]	0	*0*
s3	?[3]	0	0	?[8]	?[9]	0	?[10]	0	0	?[15]	0	?[16]	0	?[21]	*0*
s4	0	?[4]	0	0	0	0	0	?[17]	0	0	?[18]	0	?[22]	0	*0*

After the 5 step (5th iteration) of backtracking (fronttracking) S will look like:

	a	*b*	**c**	*f*	*l*	**d**	e	**g**	*h*	**i**	*p*	*q*	*j*	**k**	r
AP credits	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0
s1	1[1]	0	0	1[5]	?[6]	0	?[7]	0	0	?[11]	0	?[12]	0	?[19]	*0*
s2	0	0[2]	0	0	0	**1**	0	?[13]	0	0	?[14]	0	?[20]	0	*0*
s3	0[3]	0	0	?[8]	?[9]	0	?[10]	0	0	?[15]	0	?[16]	0	?[21]	*0*
s4	0	0[4]	0	0	0	0	0	?[17]	0	0	?[18]	0	?[22]	0	*0*

Step 12nd: backtracking procedure will try to put 1 in the cell 12; this will cause violation of the maximum number of courses per semester (in the first semester it's 4 courses). Then backtracking procedure will try to put 0 in the cell 12; that will cause ***i*** correquisite violation (***q*** is a coreq of ***i***). This means we need to backtrack. In this case backtracking procedure returns FAIL in 12th iteration. Backtracking will be continued until we reach variable 11 (11th recursion level), where course ***i*** was assigned to one.

	a	*b*	**c**	*f*	*l*	**d**	e	**g**	*h*	**i**	*p*	*q*	*j*	**k**	r
AP credits	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0
s1	1[1]	0	0	1[5]	1[6]	0	0[7]	0	0	1[11]	0	X[12]	0	?[19]	*0*
s2	0	0[2]	0	0	0	**1**	0	?[13]	0	0	?[14]	0	?[20]	0	*0*
s3	0[3]	0	0	0[8]	0[9]	0	0[10]	0	0	?[15]	0	?[16]	0	?[21]	*0*
s4	0	0[4]	0	0	0	0	0	?[17]	0	0	?[18]	0	?[22]	0	*0*

After backtracking to 11th iteration backtracking procedure will put 0 to cell 11. On the 12th iteration table will look like this:

	a	*b*	**c**	*f*	*l*	**d**	e	**g**	*h*	**i**	*p*	*q*	*j*	**k**	r
AP credits	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0
s1	1[1]	0	0	1[5]	1[6]	0	0[7]	0	0	0[11]	0	1[12]	0	?[19]	*0*
s2	0	0[2]	0	0	0	**1**	0	?[13]	0	0	?[14]	0	?[20]	0	*0*
s3	0[3]	0	0	0[8]	0[9]	0	0[10]	0	0	?[15]	0	?[16]	0	?[21]	*0*
s4	0	0[4]	0	0	0	0	0	?[17]	0	0	?[18]	0	?[22]	0	*0*

Final state of the matrix S (solution found):

	a	*b*	**c**	*f*	*l*	**d**	e	**g**	*h*	**i**	*p*	*q*	*j*	**k**	r
AP credits	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0
s1	1[1]	0	0	1[5]	1[6]	0	0[7]	0	0	0[11]	0	1[12]	0	0[19]	*0*
s2	0	0[2]	0	0	0	**1**	0	1[13]	0	0	0[14]	0	1[20]	0	*0*
s3	0[3]	0	0	0[8]	0[9]	0	0[10]	0	0	0[15]	0	0[16]	0	1[21]	*0*
s4	0	0[4]	0	0	0	0	0	0[17]	0	0	0[18]	0	0[22]	0	*0*

3.1.3 Integer Linear Programming approach

Linear programming relaxation is an alternative approach. This is another way for solving optimization problems. This relaxation technique transforms an NP-hard optimization problem (integer programming) into a related problem that is solvable in polynomial time (linear programming)[1].

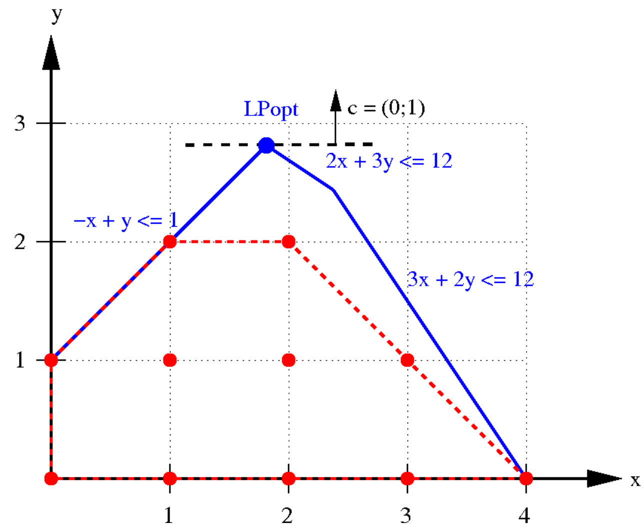

Figure 6: A (general) integer program and its LP-relaxation[1]

The linear programming relaxation of a 0-1 integer program is the problem that arises by replacing the constraint that each variable must be 0 or 1 by a weaker constraint, that each variable belong to the interval [0,1].

This algorithm works faster than backtracking algorithm for input data that has no solution at all. This can be used for problem feasibly check (if linear solution exists or not).

This approach now considered as a potential alternative for study plan construction algorithm. This is our second choice algorithm.

[1] Taken from the Wikipedia: http://en.wikipedia.org/wiki/Linear_programming_relaxation

4. Technology-Specific System Design

Student scheduling system is a typical web application that uses three-tier architecture pattern as well as model-view-control architecture style. Detailed description of each tier and component structure is provide in the following section.

4.1 Design Overview

4.1.1 System Structure

The following diagram (Figure 7: Hardware Component Class Diagram) shows the Hardware structure. The web/application /DBMS server will be connected with client workstations through the Internet Network.

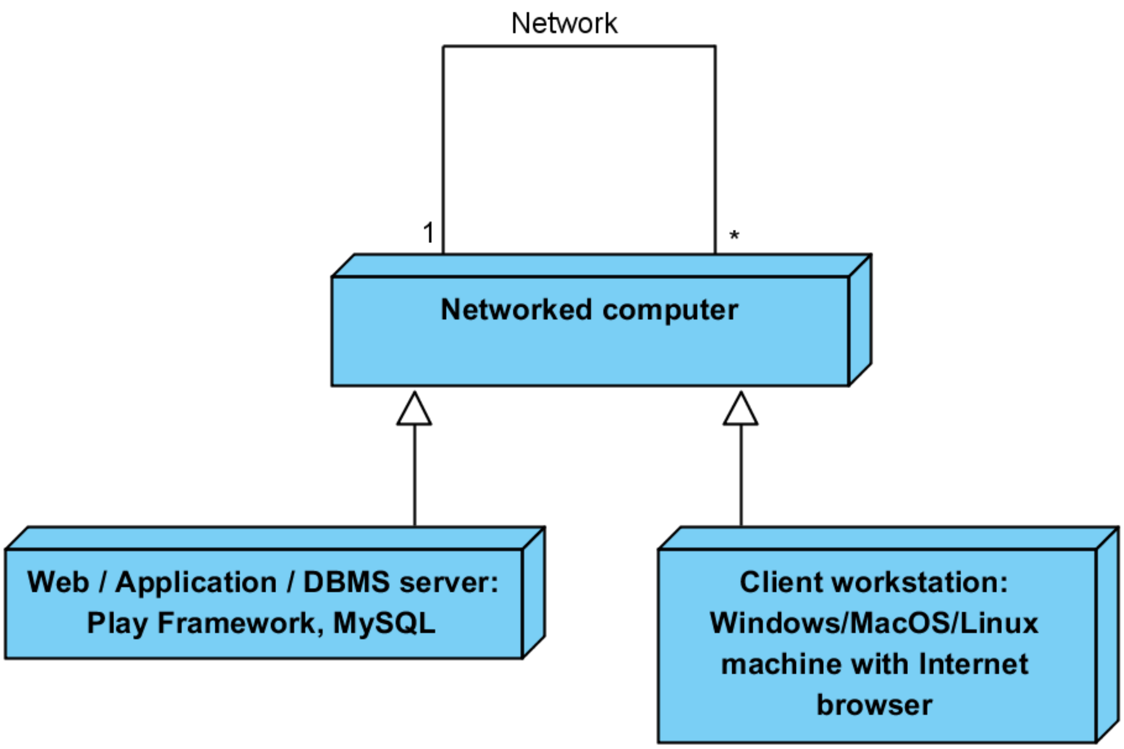

Figure 7: Hardware Component Class Diagram

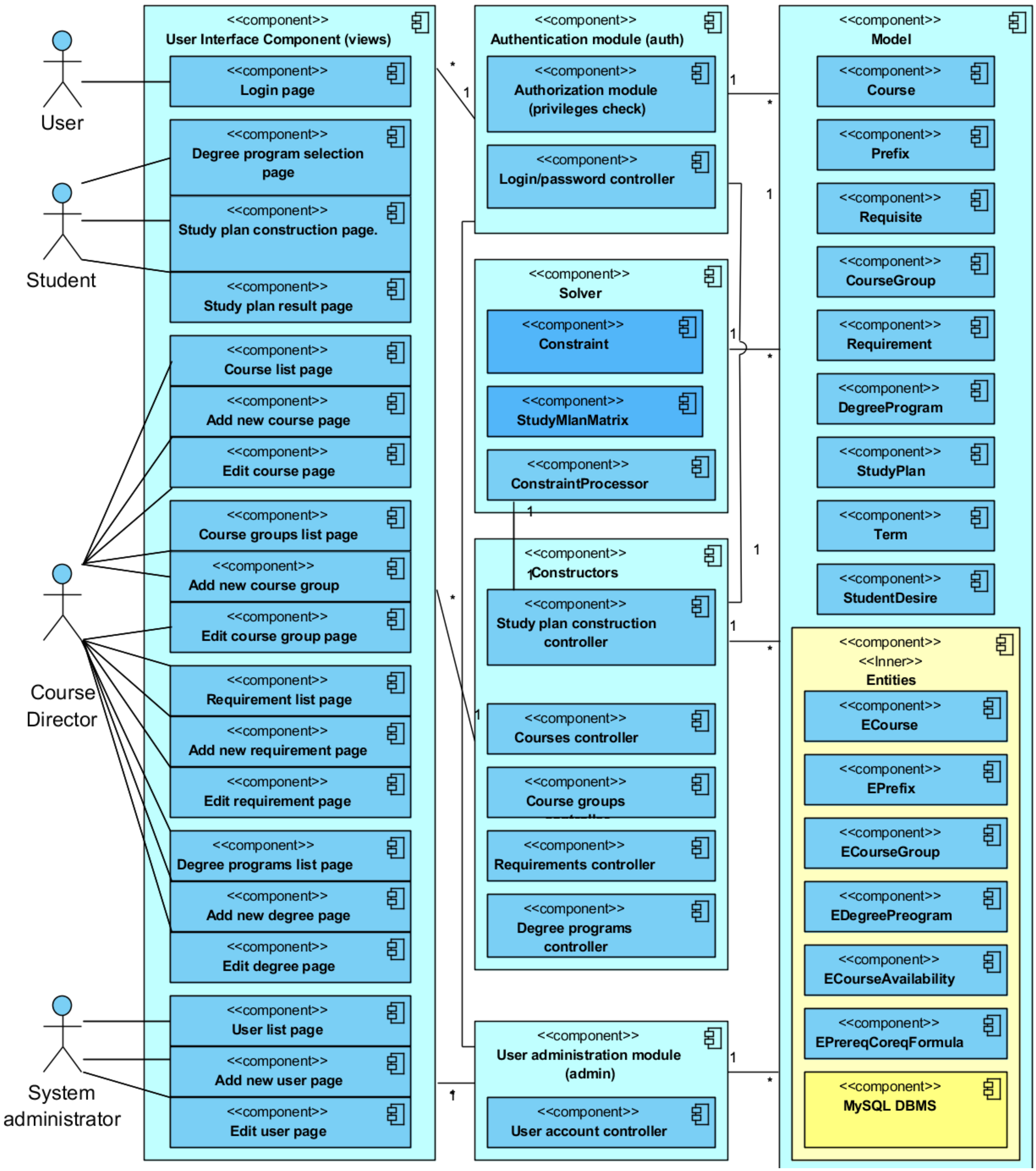

Figure 8: Software Component Diagram

The diagram above (Figure 8: Software Component Diagram) represents package structure of the project. Description of each package shown in the following table:

Table 32: Package description

Package name	Description
controllers.admin	Contains controllers responsible for user account management. For example, it has a controller that updates admins' or users' passwords. In the future all account management related controllers (new user creation, user role change) should be implemented in this package.
controllers.auth	Contains classes responsible for user authentication and user authorization check. For example, AuthCheckSecurity - it checks if user is authenticaticated in the system. This package also has LoginController, which is responsible for login page, login and logout actions.
controllers.constructors	Contains controllers responsible for creation/modification of the following business objects: • Courses • Course groups • Requirements • Degree programs • Study plans All the controllers use business objects from model package to perform their functions. Direct usage classes from model.entities is strictly prohibited. That would violate tree-tier architecture pattern.
controllers.solver	Contains classes responsible for constraint solver implementation. For example, ConstraintProcessor which has method solve(). Another important class is StudyPlanMatrix. It contains all the constraints and intermediate state of the partial solution during backtracking.
controllers.util	Contains utility classes used in various controllers. For example, Converter – responsible for parsing Boolean formulas and converting them in to appropriate business object trees.
model	Contains classes for business objects such as • Course • Course group • Requirement • Degree program • PreRequisite • CoRequisite • Study plan • Term

model.entities	Contains classes from Data Access Layer. Each class there is Data Access Object entity, which is mapped to DB entity. Every class in this package must have name starting with capital E (E stands for entity). For example: • ECourse • ECourseGroup • ERequirement • EDegreeProgram • EPrefix These classes should have only mapping to DB entities. No business logic is implemented here.
views	This package contains html templates. Each page is a view. These pages are not JavaScript intensive. JavaScript is used only for user input validation and for sending ajax delete/put http requests. These requests are sent via jQuery Ajax request. JavaScript functions are located in /public/javascripts folder.
views.forms	This package contains classes responsible for html forms. They represent html form data. This package is used by controllers package.

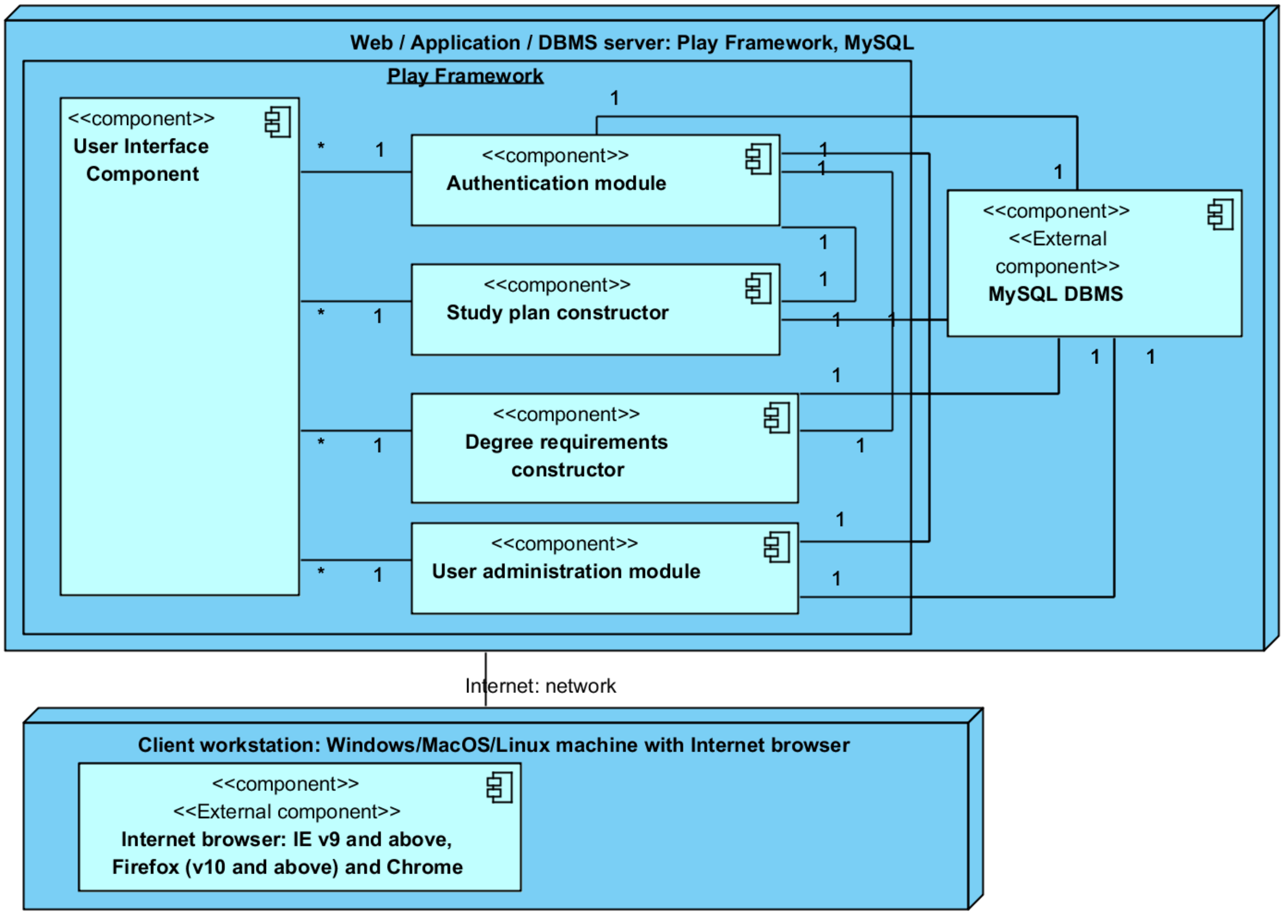

Figure 9: Deployment Diagram

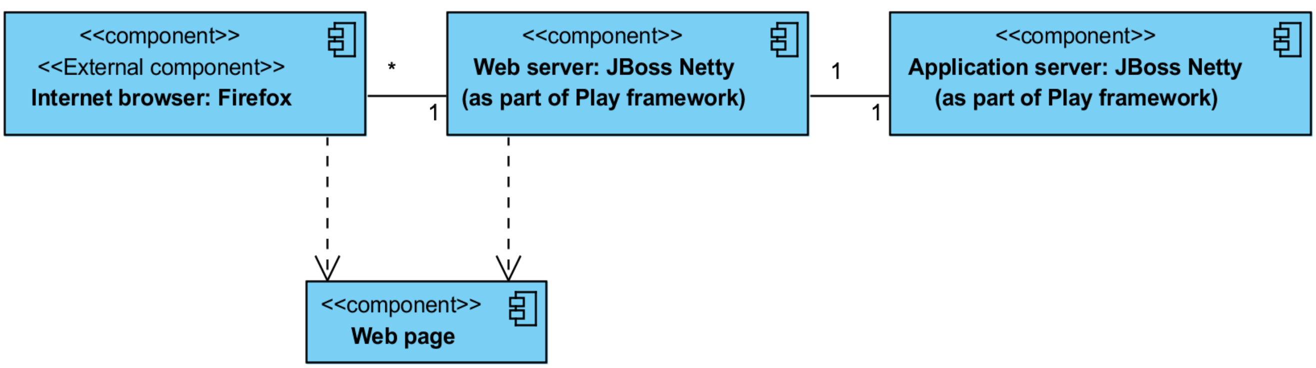

Figure 10: Supporting Software Component Class Diagram

Table 33: Hardware Component Description

Hardware Component	Description
Web/Application/DBMS Server	This component accepts connections from all users. Web and Application server are provided by Play framework. It is Jboss Netty server. Since data base is comparatively small and does not require high-performance hardware it was decided to host with Web and Application server.
Client workstation	This is a user computer. It must have a web browser (Mozilla Firefox) and it must have Internet network.

Table 34: Software Component Description

Software Component	Description
User interface component	This component contains all the webpages of the system.
Study plan constructor	This component contains controller for study plan construction webpage and classes that convert requirements into formal representation for constraint solver.
Degree requirements constructor	This component is responsible for data base content management. It contains controllers that allow performing CRUD operations over courses, course groups, requirements, and degree programs.
Authentication module	This module is responsible for privilege checking and login/password checking.
User management module	This module contains controllers that perform CRUD operations over users of the system.
MySQL DB server	Data base that stores all entities of the system. This is data layer of the system.

Table 35: Supporting Software Component Description

Support Software Component	Description
Internet browser (Firefox)	An Internet browser connected to the Internet. It displays system's webpages.
Web server (Jboss Netty)	The server component that routes all network traffic and requests between external systems and the application server.
Application server (Jboss Netty)	The server component where the Student Scheduling System is hosted. All the logical computations are done on this component.
Web pages	The actual web pages created by the Student Scheduling System.

4.1.2 Design Classes

The design classes describe the relationships among the boundary, control, and entity classes of the Student Scheduling System.

4.1.2.1 Interface Class Diagram

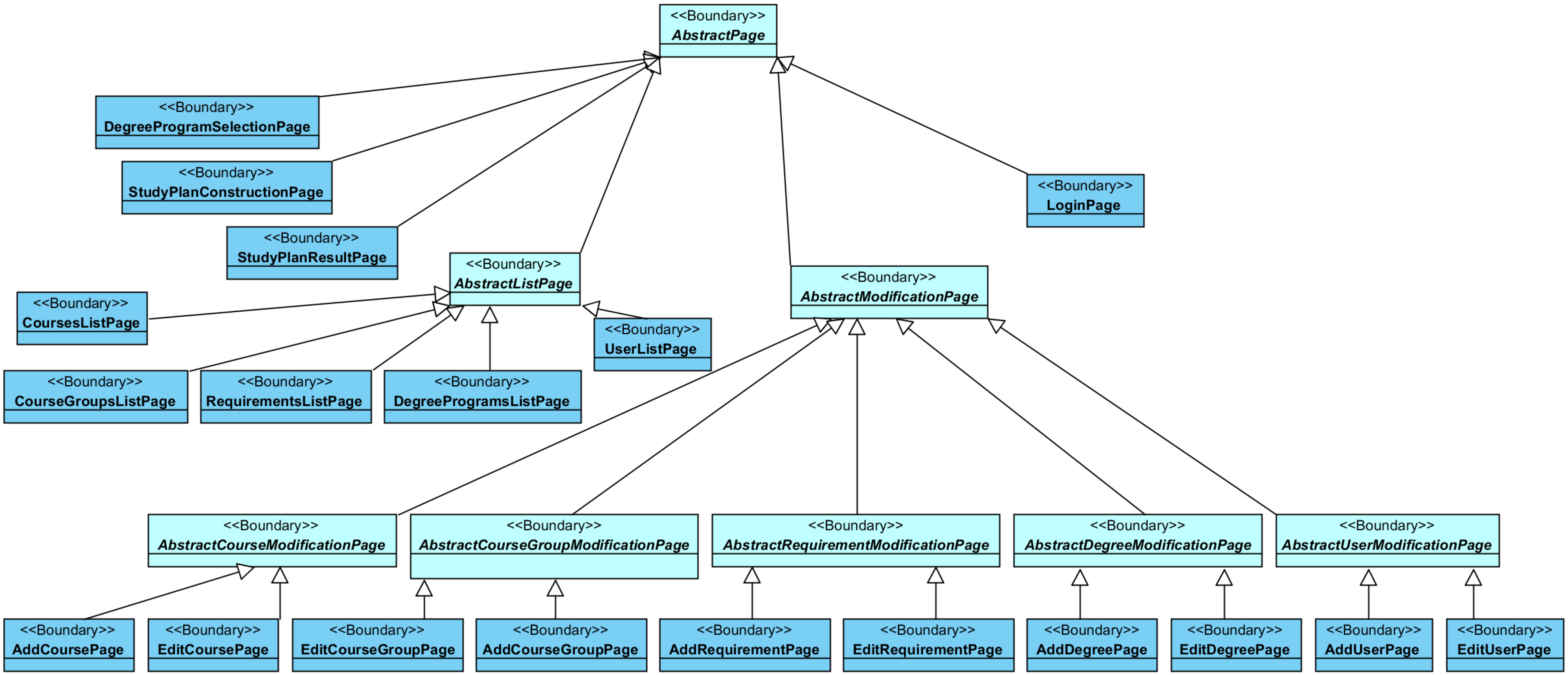

Figure 11: Interface Class Diagram

Table 36: Interface Classes Description

Class	Type	Description
AbstractPage	Boundary	This is a page template that defines general page layout.
AbstractListPage	Boundary	This is a page template that defines general page layout for any page that contains list of items such as course list or degree programs list.
AbstractModificationPage	Boundary	This is a page template that defines general page layout for any modification page such as course or degree modification.
AbstractCourseModificationPage	Boundary	This is a page template that defines general page layout for add/update course pages.
AbstractDegreeModificationPage	Boundary	This is a page template that defines general page layout for add/update degree pages.
AbstractRequirementModificationPage	Boundary	This is a page template that defines general page layout for add/update requirement pages.
AbstractUserModificationPage	Boundary	This is a page template that defines general page layout for add/update user pages.
AbstractCourseGroupModificationPage	Boundary	This is a page template that defines general page layout for add/update course group pages.
AddCourseGroupPage	Boundary	This is a page template that defines general page layout for add/update degree pages.
AddCoursePage	Boundary	On this webpage course director can add new course and specify its parameters such as name, number, prefix, prereqs and coreqs.
AddDegreePage	Boundary	On this webpage course director can add new degree program and specify its requirements.
AddRequirementPage	Boundary	On this webpage course director can add new requirement and specify its parameters such as name and type of the requirement (simple or combination).
AddUserPage	Boundary	On this webpage administrator can add new user.
CourseGroupsListPage	Boundary	On this webpage course director can see all course groups created in the system.

CoursesListPage	Boundary	On this webpage course director can see all courses created in the system.
DegreeProgramSelectionPage	Boundary	This webpage allows student to choose degree program for study plan.
DegreeProgramsListPage	Boundary	On this webpage course director can see all degree programs created in the system.
EditCourseGroupPage	Boundary	On this webpage course director can update or delete existing course group.
EditCoursePage	Boundary	On this webpage course director can update or delete existing course.
EditDegreePage	Boundary	On this webpage course director can update or delete existing degree.
EditRequirementPage	Boundary	On this webpage course director can update or delete existing requirement.
EditUserPage	Boundary	On this webpage administrator can change user’s password and privileges.
LoginPage	Boundary	This page asks user to enter password and login.
RequirementsListPage	Boundary	On this webpage course director can see all requirements created in the system.
StudyPlanConstructionPage	Boundary	This webpage allows student to specify number of semesters in study plan, all preferred courses, courses that were already taken, and other constraints.
StudyPlanResultPage	Boundary	This webpage shows student study plan if it was found.
UserListPage	Boundary	On this webpage administrator can see all users of the system.

4.1.2.2 Study plan construction class diagram

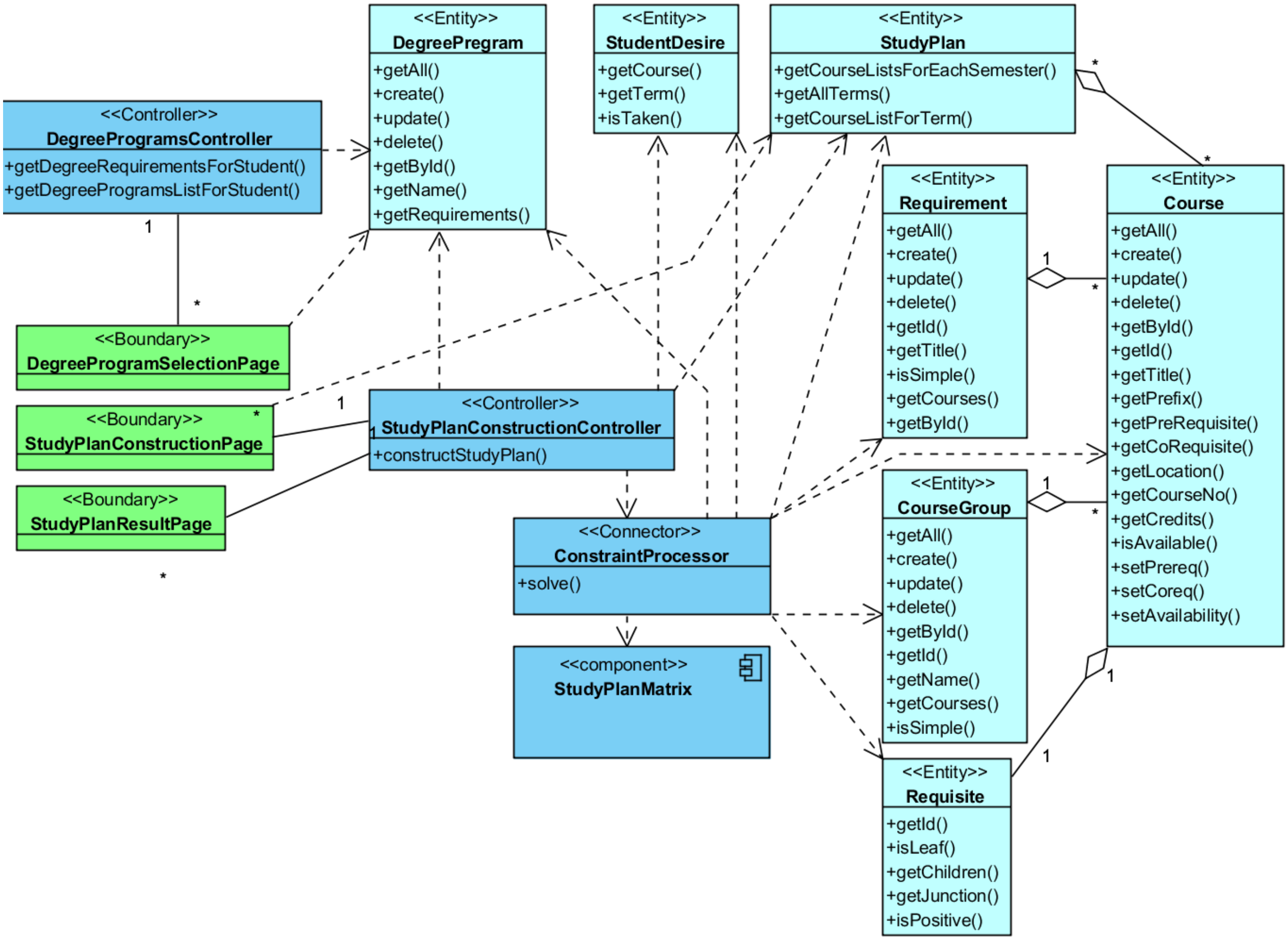

Figure 12: Study plan construction Class Diagram.

Table 37: Design Class Description

Class	Type	Description
DegreeProgramSelectionPage	Boundary	This webpage allows student to choose degree program for study plan.
StudyPlanConstructionPage	Boundary	This webpage allows student to specify number of semesters in study plan, all preferred courses, courses that were already taken, and other constraints.
StudyPlanResultPage	Boundary	This webpage shows student study plan if it was found.
DegreeProgramsController	Controller	This class implements business logic of all CRUD operations over degrees in the systems. It also allows getting all requirements for a particular degree.
StudyPlanConstructionController	Controller	This controller combines degree requirements and student's desires into a set of requirements for constraint solver. It uses To do this it uses ConstraintProcessor.
DegreeProgram	Entity	General information about degree program.
Requirement	Entity	Requirement item contains name, type (simple/composition) and additional information which depends on type.
Course	Entity	Course object, contains all course information.
CourseGroup	Entity	Group of Courses.
Requisite	Entity	Prerequisite or correquisite object.
StudyPlan	Entity	Class that describes solution.
StudentDesire	Entity	Class that contains student desires, such as - Courses student has taken - Desire to take a particular course - Desire to take a particular course in a particular semester.
ConstraintProcessor	Connector	This component performs requirement conversion for the constraint solver.

4.1.2.3 Course management class diagram

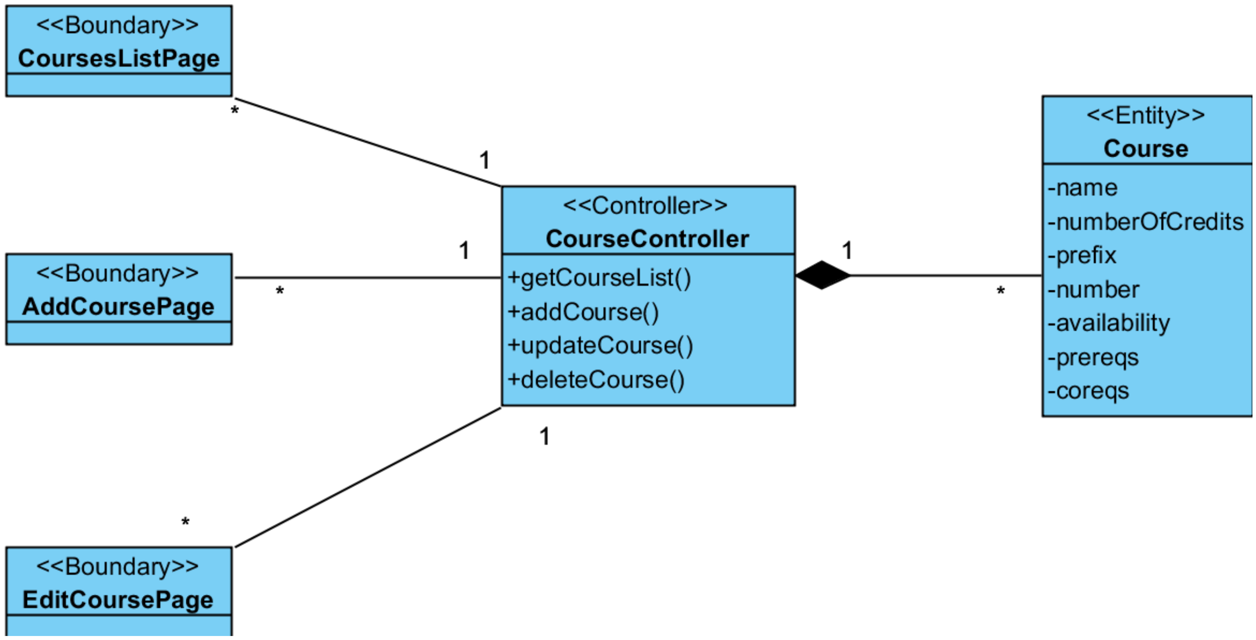

Figure 13: Course management Class Diagram

Table 38: Design Class Description

Class	Type	Description
AddCoursePage	Boundary	On this webpage course director can add new course and specify its parameters such as name, number, prefix, prereqs and coreqs.
EditCoursePage	Boundary	On this webpage course director can update or delete existing course.
CoursesListPage	Boundary	On this webpage course director can see all courses created in the system.
CourseController	Controller	This component implements all CRUD operations over courses.
Course	Entity	Course information: name, number, prefix, coreqs, prereqs.

4.1.2.4 Course group management class diagram

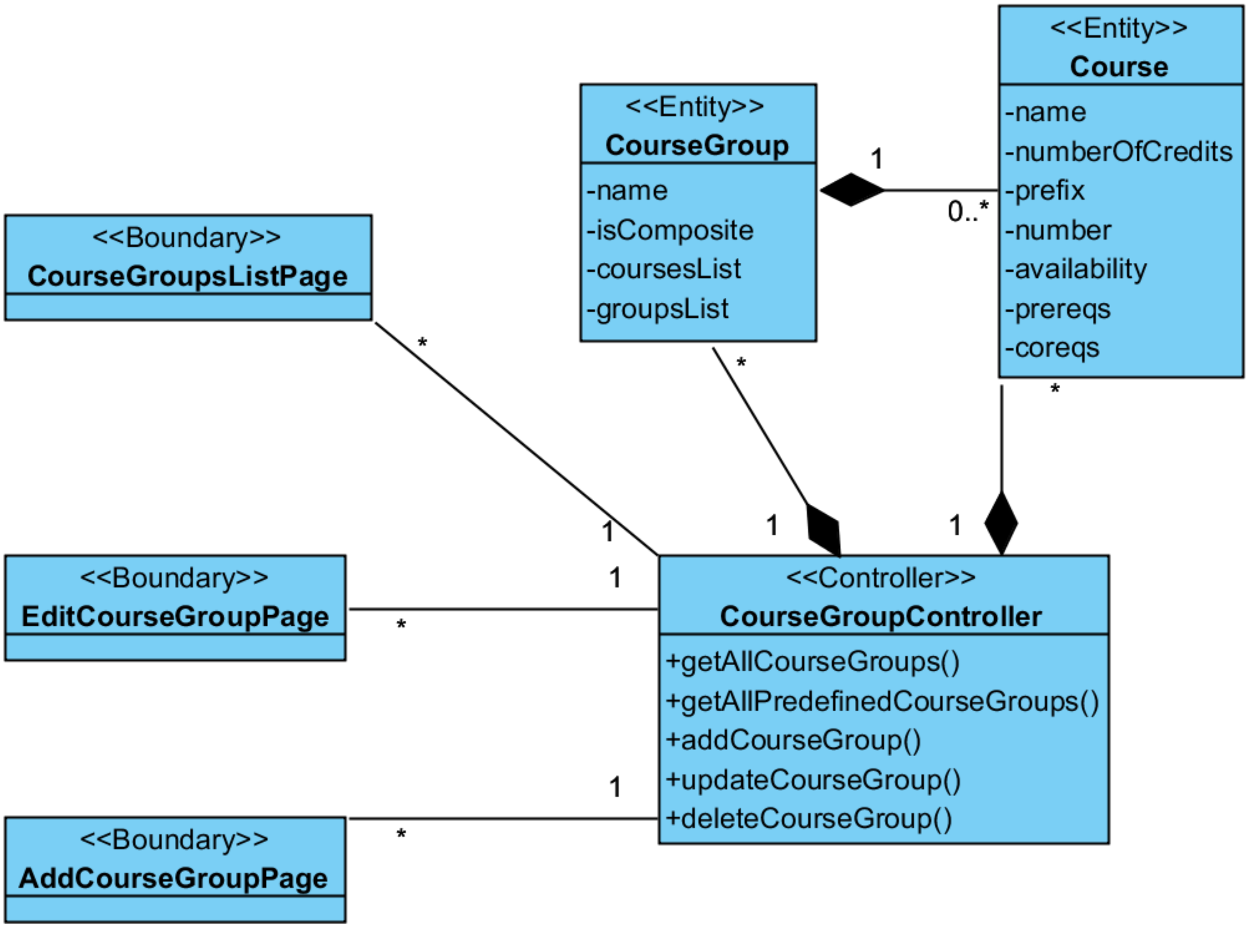

Figure 14: Course group management Class Diagram

Table 39: Design Class Description

Class	Type	Description
AddCourseGroupPage	Boundary	This is a page template that defines general page layout for add/update degree pages.
CourseGroupsListPage	Boundary	On this webpage course director can see all course groups created in the system.
EditCourseGroupPage	Boundary	On this webpage course director can update or delete existing course group.
CourseGroupController	Controller	This component implements all CRUD operations over course groups.
CourseGroup	Entity	This entity represents a set of courses. Each group can be either a simple group (direct enumeration of courses) or a Boolean composition of other groups.
Course	Entity	Course information: name, number, prefix, coreqs, prereqs.

4.1.2.5 Requirements management diagram

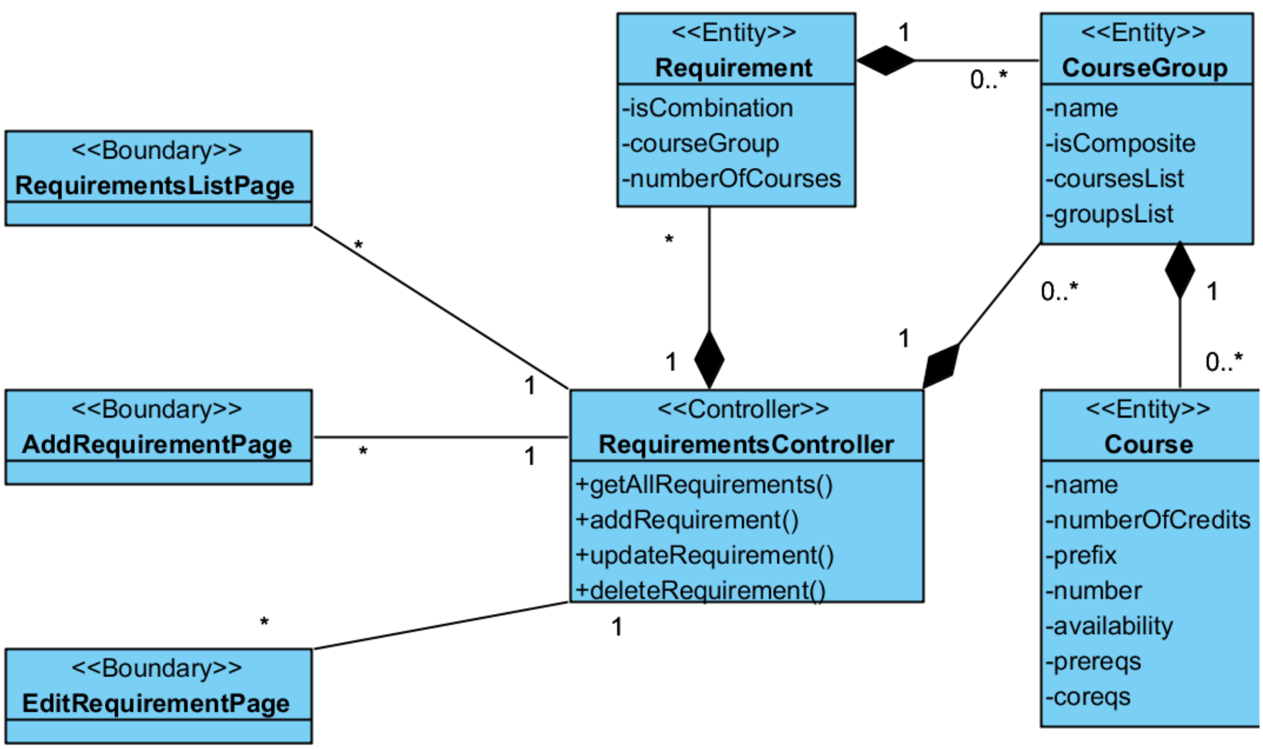

Figure 15: Requirements management Class Diagram

Table 40: Design Class Description

Class	Type	Description
AddRequirementPage	Boundary	On this webpage course director can add new requirement and specify its parameters such as name and type of the requirement (simple or combination).
EditRequirementPage	Boundary	On this webpage course director can update or delete existing requirement.
RequirementsListPage	Boundary	On this webpage course director can see all requirements created in the system.
RequirementsController	Controller	This component implements all CRUD operations over requirements.
Requirement	Entity	Requirement item contains name, type (simple/composition) and additional information which depends on type.
CourseGroup	Entity	This entity represents a set of courses. Each group can be either a simple group (direct enumeration of courses) or a Boolean composition of other groups.
Course	Entity	Course information: name, number, prefix, coreqs, prereqs.

4.1.2.6 Degree management class diagram

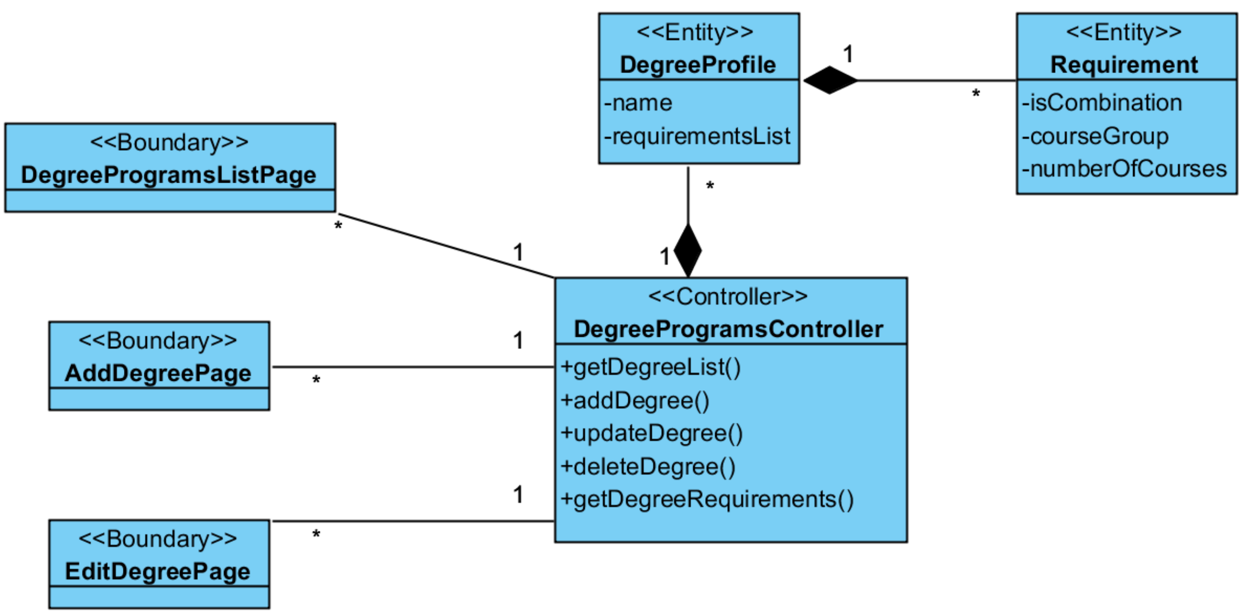

Figure 16: Degree management Class Diagram

Table 41: Design Class Description

Class	Type	Description
AddDegreePage	Boundary	On this webpage course director can add new degree program and specify its requirements.
EditDegreePage	Boundary	On this webpage course director can update or delete existing degree.
DegreeProgramsListPage	Boundary	On this webpage course director can see all degree programs created in the system.
DegreeProgramsController	Controller	This component implements all CRUD operations over requirements. It also update degree's requirements.
DegreeProfile	Entity	General information about degree program.
Requirement	Entity	Requirement item contains name, type (simple/composition) and additional information which depends on type.

4.1.3 Process Realization

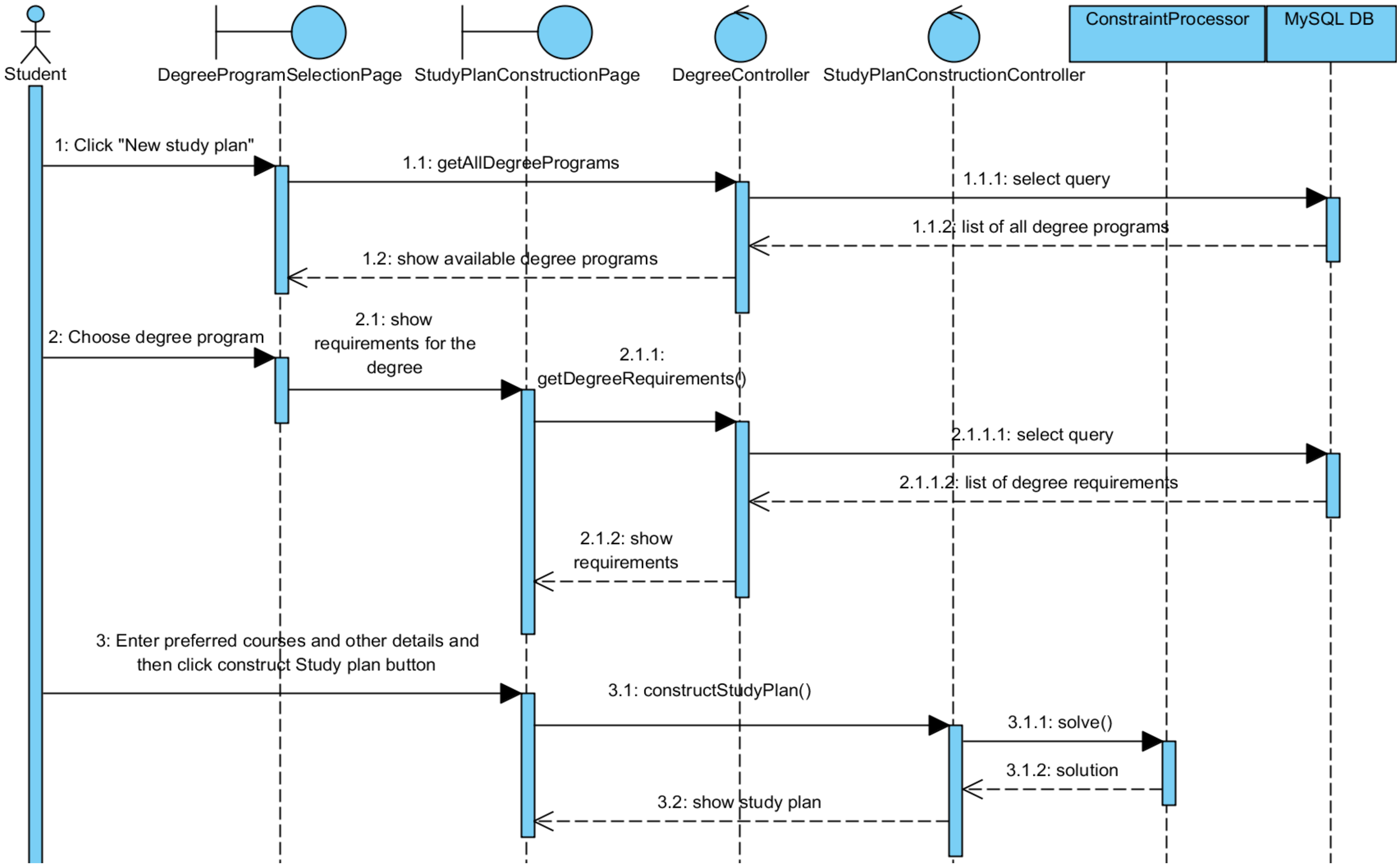

Figure 17: Request Study Plan Sequence Diagram

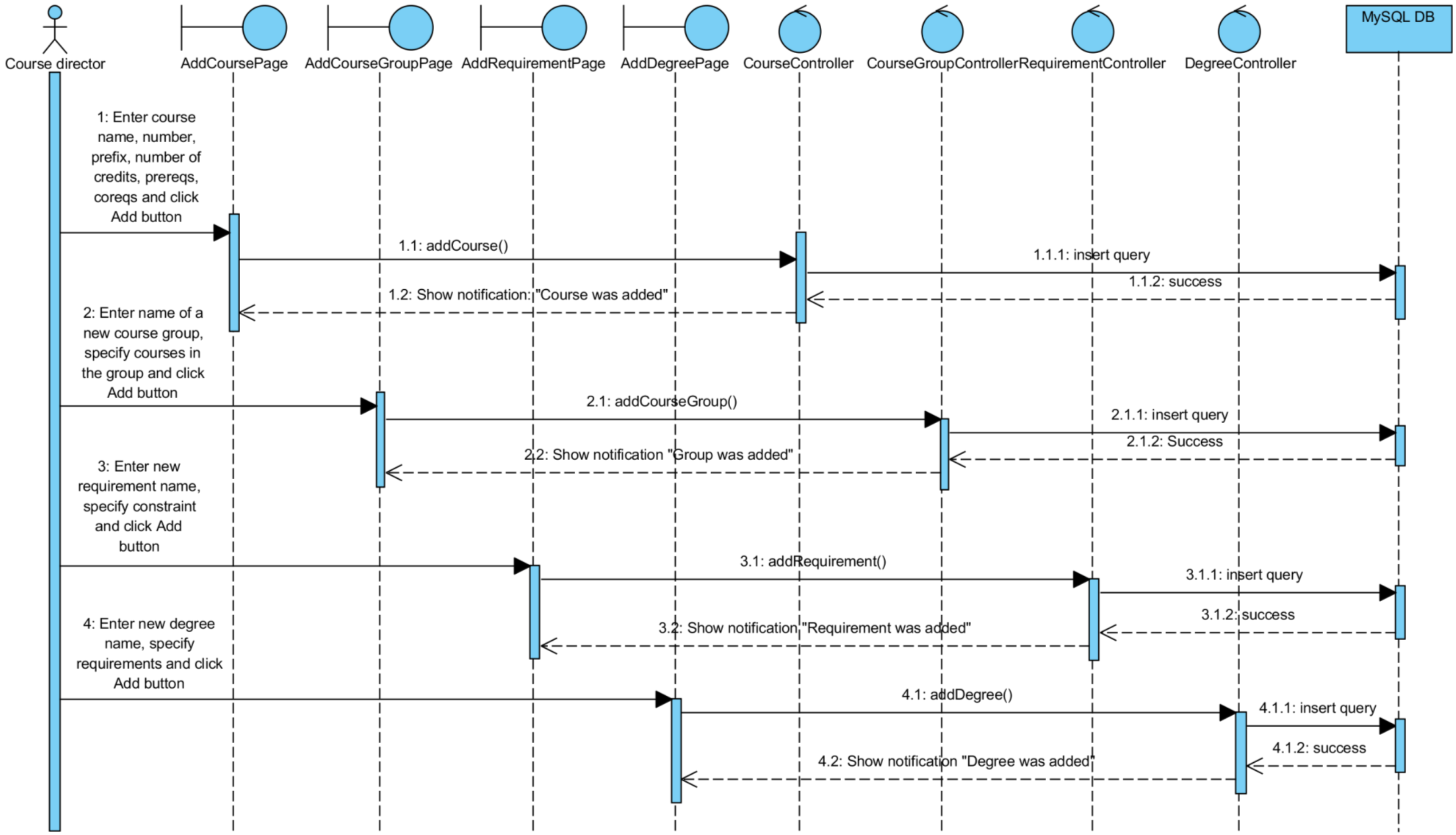

Figure 18: Add New Degree Sequence Diagram

4.1.4 Data base design

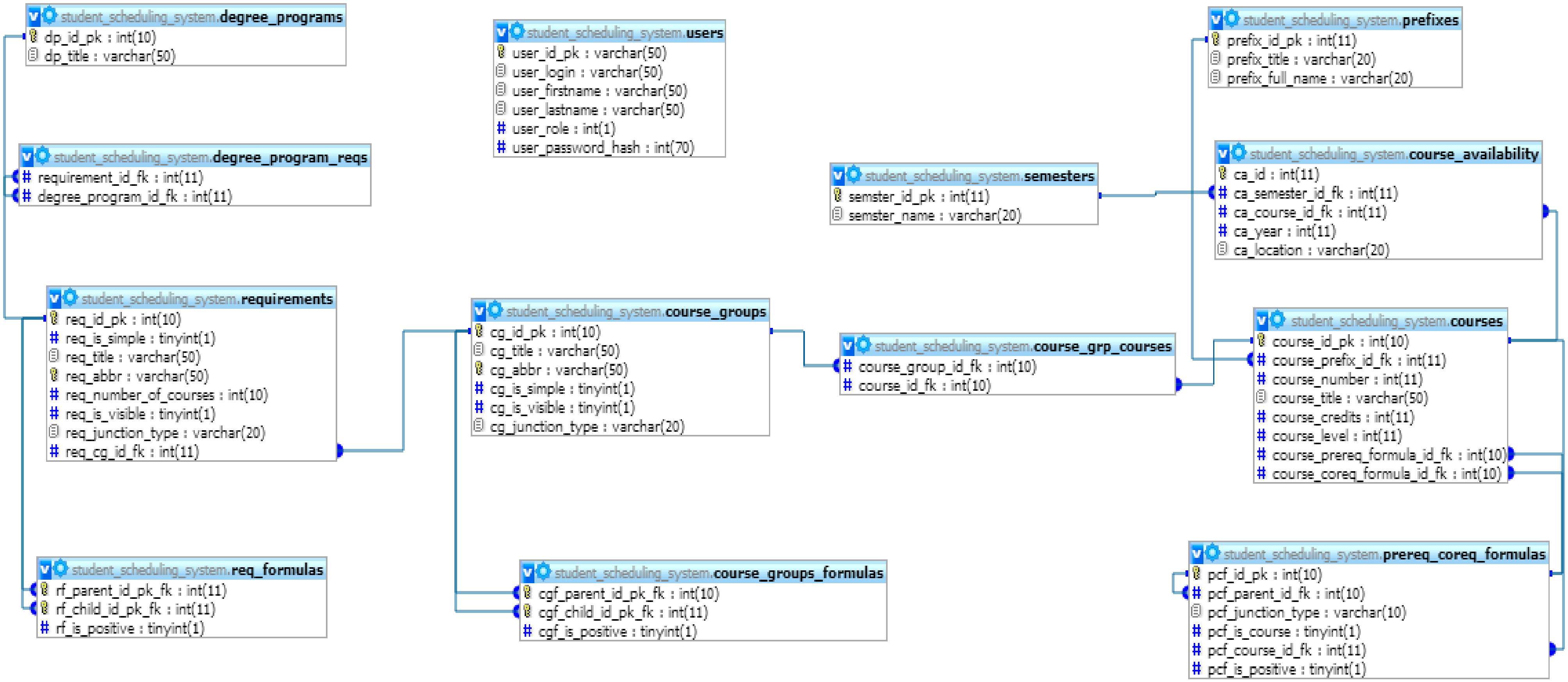

Figure 19: Data base design diagram

Figure 19 shows part of the data base schema that is responsible for storing all domain entities such as courses, course groups, requirements, and degree programs. This schema uses Oracle naming convention (http://www.oracle-base.com/articles/misc/naming-conventions.php).

4.2 Design Rationale

Student Scheduling System will be used by Stevens University students and staff. It must be accessible via the Internet for all users. Therefore, Student Scheduling System is designed as a web application.

It will have centralized data base for storing degree requirements and available courses. For study plan construction it will use external constraint solver (external library). To make system more flexible and open to future changes three-tier architecture pattern was chosen (detailed description of the potential benefits can be found in Table 42: Architectural Styles, Patterns, and Frameworks). This pattern allows us to separate three layers of the system:

- User interface (web pages)
- Business logic (controls, object model, data access objects layer)
- Data layer (data base).

In order to make application simpler and more service oriented we chose REST architecture style for interaction with the system. Play framework for java was chosen as one of the most convenient frameworks for web application development that supports REST style. Another reason to choose Play framework is that it already has secure module, which enable us to set up basic authentication and authorization management to our application.

For persistence layer we chose MySQL DBMS because it one of the most wide-used open source DBMS. Moreover, it is easy to integrate it with Play framework; they are absolutely compatible.

5. Architectural Styles, Patterns and Frameworks

Table 42: Architectural Styles, Patterns, and Frameworks

Name	Description	Benefits, Costs, and Limitations
Three-tier architecture pattern	Three-tier architecture allows to separate three layers of the system: • User interface • Business logic • Data layer (date base) In Student Scheduling System user interface is a set of webpages; business logic – all the controllers that perform calculation or data manipulation. Business logic is also a communication layer between user interface and data layer.	Separation of concerns which enables to achieve better maintainability and scalability. Changes of components on each layer do not affect other layer if interface of the components remains the same. That will allow making algorithm optimization in the future. It also enables us to use another constraint solver without changes in user interface or data base.
Client-server style	This style is used for communication between layers (three tiers).	This is the only way to build a web application.
Play framework for Java.	Play is an open source web application framework for Java. It implement model–view–controller (MVC) architectural pattern (to be more precise it is a presentation abstraction control (PAC) pattern).	Play is based on a lightweight, stateless, web-friendly architecture and features predictable and minimal resource consumption (CPU, memory, threads) (according to official website http://www.playframework.org/). Benefits: • a clean, RESTful framework, • a persistence layer based on JPA • Scala for the template engine. • Integrated unit testing: JUnit
REST architecture style	Representational State Transfer (REST) is a style of software architecture for distributed systems such as the World Wide Web.	This style is employed by Play framework.