Software Design Specification

For

UWI SMART COURSE CHOICE

Version 1.0

Prepared by

Group Name: R.A.J.J

|  |  |  |
| --- | --- | --- |
| Ritesh Reddy | 620059221 | ritesh\_r\_reddy@hotmail.com |
| Ashle | 620056706 | ashlehenriques@gmail.com |
| Jordanne | <student #> | <e-mail> |
| Jeremy | 620057694 | jeremy\_df9@hotmail.com |

|  |  |
| --- | --- |
| Instructor: | Mr. Eyton Ferguson |
| Course: | COMP2140 |
| Lab Section: | *<place your lab section here>* |
| Teaching Assistant: | *Dean Jones* |
| Date: | <place the date of submission here> |

Contents

**Contents** 2

**Revisions** 3

1 **Introduction** 4

1.1 Project Overview 4

1.2 Project Scope 4

1.3 Document Preview 4

2 **Architectural Design** 5

2.1 Section Overview 5

2.2 General Constraints 5

2.3 Data Design 5

2.4 Programme Structure 5

2.5 Alternatives Considered 7

3 **Detailed Design** 8

3.1 Section Overview 8

3.2 Sequence Diagrams 8

3.3 Class Diagrams 9

3.4 Component Design Description 12

3.5 Architectural Design Consistency 18

4 **User Interface Design** 19

4.1 Section Overview 19

4.2 Interface Design Rules 19

4.3 GUI Components 19

4.4 Detailed Description 20

5 **Conclusion** 23

Revisions

| Version | Primary Author(s) | Description of Version | Date Completed |
| --- | --- | --- | --- |
| V.1.0 | Ritesh Reddy  Ashle Henriques  Jordanne Elliott  Jeremy-Dane Ferguson | Initial Version. Requirements Complete | 01/10/2013 |

# Introduction

## Project Overview

The project is designed to satisfy the requirements of an automated course advisory system for the students of the University of the West Indies. The specific requirements to be satisfied by the system are specified in the Software Requirements Specification.

## Project Scope

This project is intended to satisfy all requirements specified in Section 2 of the Software Requirements document. The major functionality of the system includes the generation of a first draft course plan for students and the validation thereafter of any course plan chosen by the user against university requirements.

## Document Preview

This document is intended for a technical readership and details the design of a system to satisfy the given requirements. The document is divided into three main sections, the Architectural Design which involves identification and organization of the subsystems, the Detailed Design which breaks down each subsystem and models the system in terms of class diagrams and finally a User Interface Design section that details the user interface design of the system.

# Architectural Design

## Section Overview

This section discusses the constraints on the system and their implications on the choice of an architectural pattern for the system.

## General Constraints

The major constraint on the system is the security of the confidential student record information that the system has access to. The architecture of the system must thus ensure that the data is safeguarded without any avenue for unauthorized access to this data.

Other constraints include the responsiveness required of the system, i.e., since the system is highly interactive, the system should be extremely responsive (less than 2 seconds) to each interaction of the user.

## Data Design

The system requires that all its operations be performed on up-to-date information and as such does not cache any of the information that it requires. Therefore, there are no database necessities for the system.

However, the system is primarily a data driven system. Therefore, there is an emphasis on the data and how the data is organized and **sorted.** Refer to section 3.4 for a detailed description of the internal data structures.

## Programme Structure

In addressing the constraints and satisfying the requirements of the system, the following subsystems were identified:

### User Interface

The User Interface subsystem is responsible for rendering all the views to the user, capturing his/her interactions and returning them to the other sub systems that it communicates with.

### Authentication

The Authentication sub system is responsible for authenticating a user with the external system and retrieving the user information and communicating that information with other sub systems.

### Validator

The Validator sub system is responsible for validating the course plan using many criterions. In the current scope, the criterions are as follows: Faculty Requirements, and Departmental (Programme) requirements.

### Course Selector

The Course Selector sub system is responsible for using a user’s record (if available), the user’s desired programme and generate a course plan for them which includes only the explicit mandatory requirements to accomplish the desired programme. It communicates with Authentication and Course & Programme Information aggregator sub system to obtain the required data.

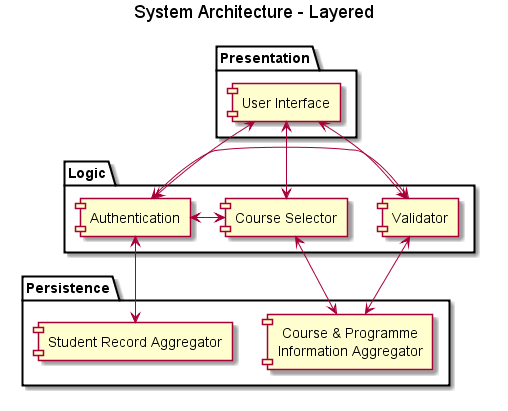
### Student Record Aggregator

This sub system enables the system to communicate with the external system in order to retrieve relevant user information.

### Course & Programme Information Aggregator

This sub system enables the system to communicate with the external system in order to retrieve information pertaining to a course and/or programme offered by the university.

The pattern used is the *Layered* pattern for communication between the subsystems and the *Model, View and Controller* pattern for the organization of the classes within the sub sytems (Refer to Diagrams in section 3).



## Alternatives Considered

Other architectural design patterns considered include the *Pipe and Filter* and *Model, View and Controller* patterns. The *Pipe and Filter* pattern was deemed unfit for the project as it is geared towards an organization which is non interactive and data only flows in one direction, i.e., the data is piped and filtered through many subsystems but never in a loop, as this would cause an infinite loop.

The *Model, View and Controller* pattern was deemed fit, however, this pattern did not entirely cater to the constraints of the system directly. However, it still provided a means to organizing the various classes and objects within the subsystems in a way as to increase the maintainability of the system.

In examining the choices for the architecture the *Layered* pattern was used primarily as it organized the subsystems in a fashion so as to address the constraints of security. The sensitive student information is never directly accessible by the user interface and the user has to go through the proper channels in order for the system to retrieve this information. Furthermore, since the layers are few in numbers, i.e., the depth of the layers, the performance constraints will also be met by use of this architectural pattern.

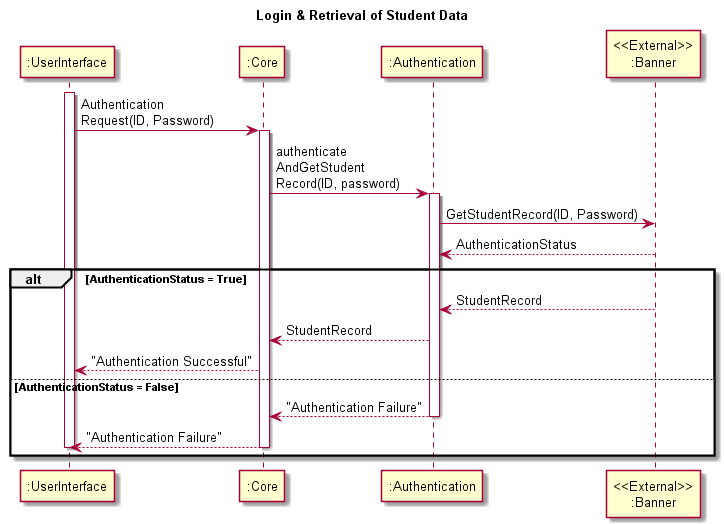
# Detailed Design

## Section Overview

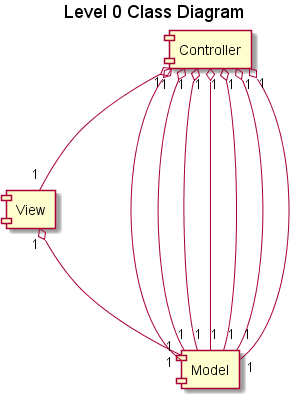
This section provides a detailed analysis and solution in the form of sequence and class diagrams. It further provides a description of each class and the pseudocode for implementing the core methods in each class. It also provides a proof of design consistency with the *Layered* architectural design pattern chosen despite the design being depicted and organized in a *Model, View and Controller* fashion.

## Sequence Diagrams

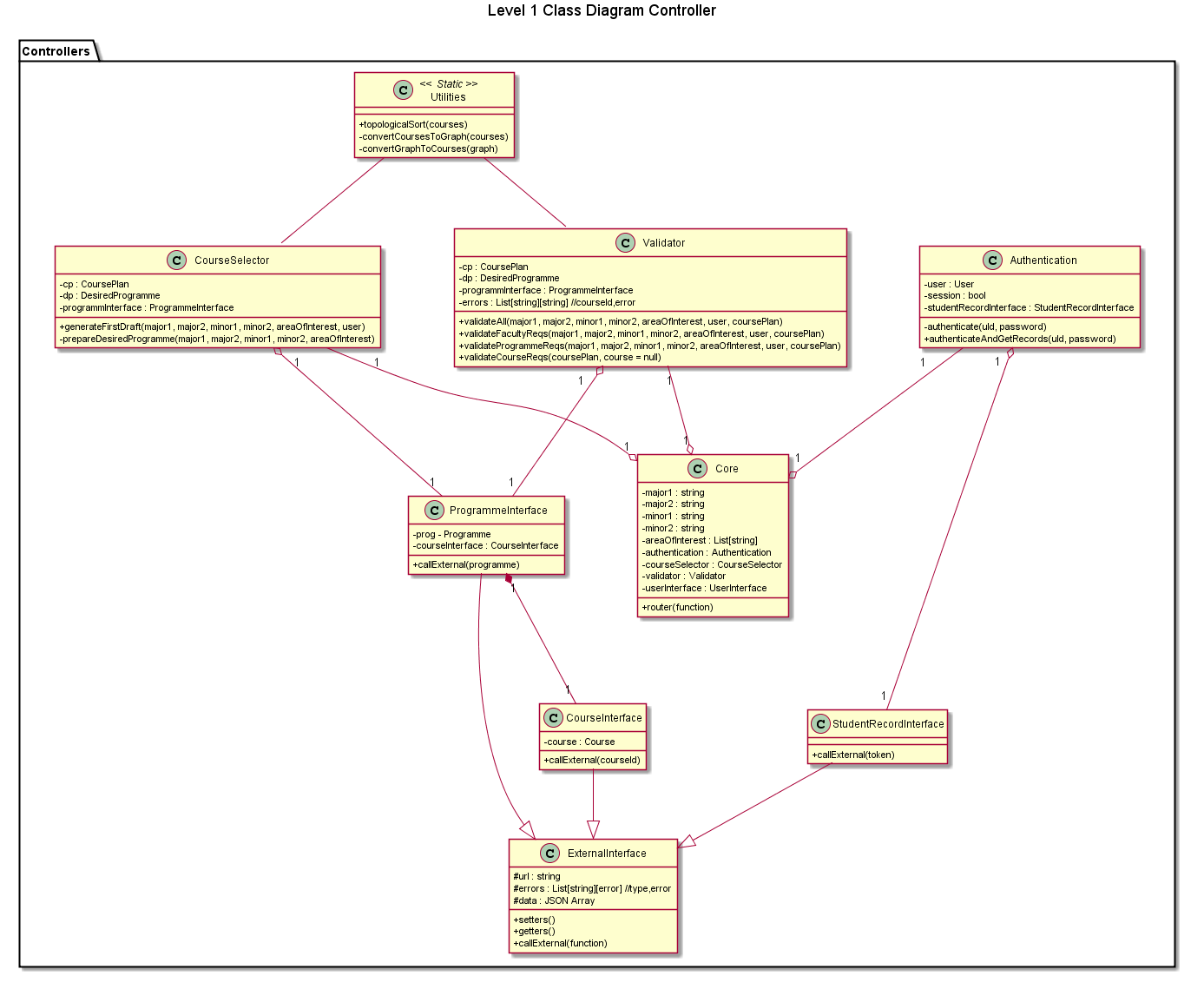
The following sequence diagrams detail the core and fundamental use cases specified in the Software Requirements Specification Document.



## Class Diagrams

The classes in the system are identified and the relationships between them are defined in the diagrams below. The classes are categorized into *Models, View and Controllers* for ease of maintenance and understanding. Due to volume of classes and relationships, the presentation of these diagrams is divided into different diagrams at different levels for ease of understanding.

## 



## 

## Component Design Description

The following details each component in the above class diagrams:

### Models

#### Course

The Course class represents the real world course element. This class provides no complex operations on the data it stores, but only provides methods to access and mutate its contents. The class is central to the system and is an aggregate of all the fields that are required to represent a course.

#### Courses

The Courses class is a wrapper class for an aggregation of a set of Courses. In the system context, many functions require a list of courses, whether as pre-requisites for another course or as requirements for a programme. This class therefore, acts as a base class for more specific implementations of course aggregates. The Courses class provides Accessors, Mutators, an addCourse and removeCourse methods. Furthermore, it provides a getTotalCredits method which returns the sum of the credits of the courses in the aggregate as follows:

begin getTotalCredits():

sum <- 0

foreach Course in Courses:

sum = sum + Course.credits

return sum

end

#### User Course Passes

The User Course Passes is a specific course aggregate that inherits from the Courses class. It represents all the courses that the user/student has already passed. It has a one-to-one relation with the User class and provides only accessor and mutator methods.

#### CoursePlan

The CoursePlan is another specific course aggregate that inherits from the Courses class. It overrides the parent class addCourse and removeCourse methods. It provides two additional methods getCreditsForSemester() and swapCourses().

begin getCreditsForSemester(semester, year):

year <- year - 1

semester <- (semester + 2\*year)

sum <- 0

foreach Course in semesters[semester]:

sum = sum + Course.credits

return sum

end

begin swapCourses(courseA, courseB):

‘Remove CourseA and add CourseB to the courseList

removeCourse(courseA)

addCourse(courseB)

end

#### Rules

The Rules class is another specific course aggregate derived from the Course Plan class. It aggregates the courses in order to represent a set of pre-requisites for a course or co-requisites. It provides no additional methods from that of its parent class.

#### Programme

The Programme class represents a single programme either a major or minor as offered by the university. It contains the name of the programme, type of programme, faculty where the programme is offered and a Courses object which dictates the requirements for that Programme. It provides access to only accessors and mutators.

#### DesiredProgramme

This class is an aggregate of Programme objects that correspond to the user’s choices. It simply provides accessor and mutator methods to each of the Programme objects.

#### User

This class models a User of the system and provides many user fields, mainly id and password. The class provides accessor and mutator methods to the fields. It is used for authentication and retrieval of User Course Pass information. Furthermore, it contains fields to represent the minimum and maximum credit limits for each semester in any given year and overall minimum and maximum credits.

### View

#### UserInterface

The UserInterface class provides all the methods required to render the different screens presented to the user. It contains fields to store the session/state in order to identify who is using the system and whether they are authenticated, a list of previous CoursePlan objects, in order to backtrack to an older course plan, the current CoursePlan object to display and a stage field to retain the stage of process that the user is currently in.

### Controllers

#### Authentication

This class provides methods to authenticate the user with the external system. The class then uses the StudentRecordInterface class to retrieve the User Course Passes from the external system.

#### External Interface

This class is the base class for all out of system communications. It provides a generic method to call a web service and parse JSON data that the service returns. It contains fields to store the URL, errors and the parsed JSON data.

#### StudentRecordInterface

This class extends the External Interface and is used to get the student records from the external system. It overrides the callExternal method and returns a User Course Passes Object.

#### CourseInterface

This class also extends the External Interface and is used to get a Course record from the external system. It overrides the callExternal method and returns a Course object.

#### ProgrammeInterface

This class also extends the External Interface and is used to get a list of requirements for a given programme from the external system. It overrides the callExternal method and returns a Programme object.

#### Core

This class is the core class that aggregates all other controllers and is analogous to a manager in a real world scenario. It provides only one method, the router. The method takes a function argument then delegates the task appropriately to a controller. In the implementation, the function can be the URL of the page visited.

#### Utilities

This is a static class and contains methods which are generic and are required by many other classes. The only method implemented in this class is a topological sort. The topological sort is a generic graph algorithm which uses a depth-first search technique on a weighted graph to order the nodes such that all the nodes point in only one direction. In this system’s context, the topological sort, given a list of courses to do, will produce the order in which to do those courses so that all pre-requisites for each course are satisfied.

#### Course Selector

This class contains the logic to generate a first draft course plan. The plan has to all explicitly stipulated requirements for the desired programmes. It should then indicate the implicitly stipulated requirements; these include optional electives defined by the programme requirements. The class uses the ProgrammeInterface (accessed as an attribute) in order to retrieve the programme requirements. It contains only one public method generateFirstDraft as defined below:

begin prepareDesiredProgramme(major1, major2, minor1, minor2, areaOfInterest)

Programme maj1 = ProgrammeInterface.callExternal(major1)

dp = new DesiredProgramme(maj)

dp.addAreaOfInterest(areaOfInterest)

if (major2 != “”)

Programme maj2 = ProgrammeInterface.callExternal(major2)

dp.addMajor2(maj2)

if(minor1 != “”)

Programme min1 = ProgrammeInterface.callExternal(minor1)

dp.addMinor1(min1)

If(minor2 != “”)

Programme min2 = ProgrammeInterface.callExternal(minor2)

dp.addMinor2(min2)

end

begin generateFirstDraft(major1, major2, minor1, minor2, areaOfInterest, user)

prepareDesiredProgramme(major1, major2, minor1, minor2, areaOfInterest)

Courses allCourses = new Courses()

allCourses.add(user.getFacultyReqs())

foreach programme in dp:

Courses deptReqs = programme.getDepartmentReqs()

allCourses.addAllCourses(deptReqs)

Courses sorted = Utilites.toplogicalSort(sorted) // Marks the semester and year for each course to determine the order of courses

cp.addAllCourses(sorted)

return cp

end

#### Validator

This class provides methods to validate a Course Plan. The validation is split into three different aspects. The methods are validateFacultyReqs, validateProgrammeReqs and validateCourseReqs. Furthermore, it provides another method which validates all of the requirements for a course. The pseudocode for the methods is as follows:

begin prepareDesiredProgramme(major1, major2, minor1, minor2, areaOfInterest)

Programme maj1 = ProgrammeInterface.callExternal(major1)

dp = new DesiredProgramme(maj)

dp.addAreaOfInterest(areaOfInterest)

if (major2 != “”)

Programme maj2 = ProgrammeInterface.callExternal(major2)

dp.addMajor2(maj2)

if(minor1 != “”)

Programme min1 = ProgrammeInterface.callExternal(minor1)

dp.addMinor1(min1)

If(minor2 != “”)

Programme min2 = ProgrammeInterface.callExternal(minor2)

dp.addMinor2(min2)

end

def validateFacultyReqs(user, coursePlan)

Courses facReqs = user.getFacultyReqs()

Courses plan = coursePlan.getCourses()

Boolean valid = true

foreach course in facReqs:

valid = False

foreach c in plan:

if course == c:

valid = true

if not valid:

return False

return True

end

def validateProgrammeReqs(major1, major2, minor1, minor2, areaOfInterst, user, coursePlan)

prepareDesiredProgramme(major1, major2, minor1, minor2, areaOfInterest)

Courses plan = coursePlan.getCourses()

Bool valid = false

foreach programme in dp:

Courses courses = programme.getCourses()

foreach course in courses:

valid = False

foreach c in plan:

if course == c:

valid = true

if not valid:

return False

return True

end

def validateCourseRequirements(coursePlan, course=null)

if course != null:

Rules prereqs = course.getPreRequisites()

semester = course.getSemesterTaken()

year = course.getYearTaken()

semester = semester + ((year-1)\*2)

semestersList = coursePlan.getSemestersList()

foreach prereq in prereqs:

boolean valid = false

for i <- 1 to semester -1:

foreach c in semesterList[i]:

if prereq == c:

valid = true

continue

if not valid:

return False

return True

else

Courses plan = coursePlan.getCourses()

foreach c in coursePlan:

if not validateCourseRequirements(coursePlan, c):

return False

return True

end

begin validateAll(major1, major2=””, minor1=””, minor2=””, user, corusePlan):

return validateFacultyRequirements(user, coursePlan) AND

validateCourseRequirements(coursePlan, course) AND

validateProgrammeRequirements(major1, major2, minor1, minor2, user, coursePlan)

end

## Architectural Design Consistency

The design is consistent with the *Layered* architecture and this is portrayed in the *Controllers* sub section of the *MVC*. The interaction logic between the different classes that make up each sub system exists solely within the *Controller* category and thus still follows the rules established by the *Layered* architecture. This can be proven by analyzing the *Controllers* class diagram in section 3.2 above.

# User Interface Design

## Section Overview

This section details the user interface design and presents mock-ups of the major screens in the systems.

## Interface Design Rules

The interface will employ many principles to increase the efficiency of the user interface.

Firstly the interface will require very little input from the user; course names and ids are given in menus. This makes the system more User friendly and prevents user error.

Secondly the interface will be easily maneuvered; navigation and page headings are present to ensure easy access to different pages within the system.

Finally the system shall be consistent in its outputs and abide by universal conventions; use of colour as well as the basic layout of the screen will portray this consistency.

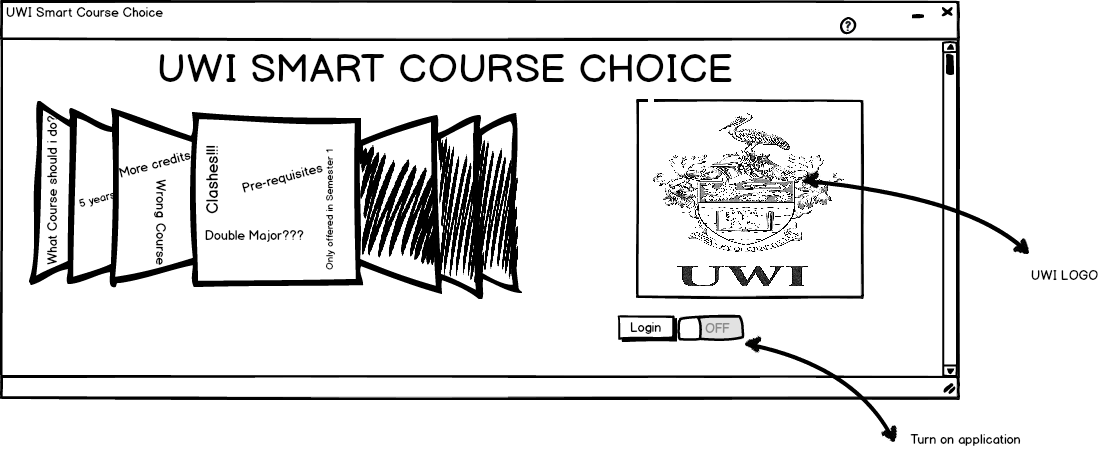
Along with the above the system interface shall comply with the basic requirements of user interfaces by being un-ambiguous, organized and interactive to optimize user experience.

## GUI Components

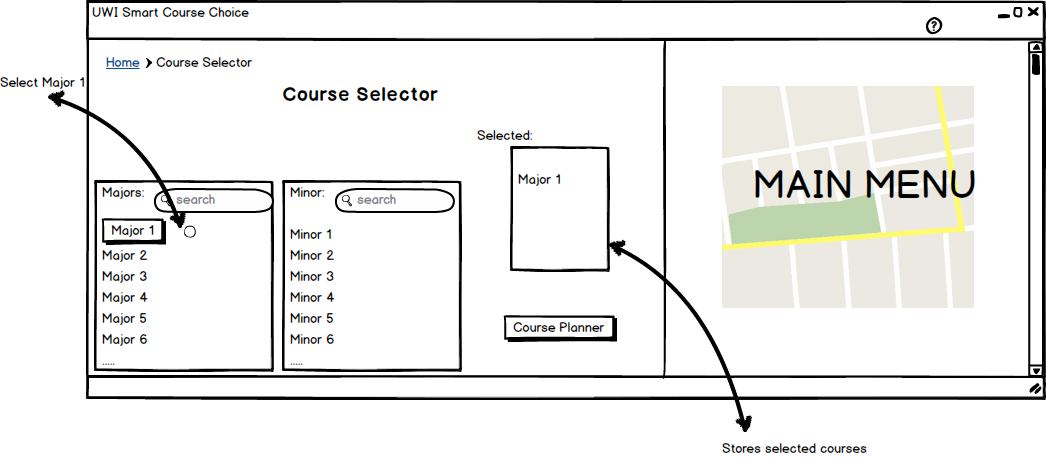
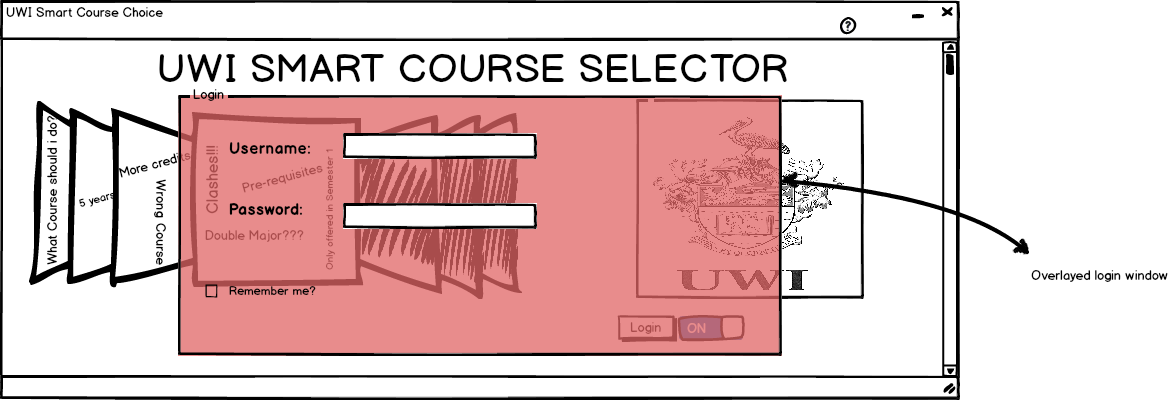
|  |  |  |
| --- | --- | --- |
| **GUI Component** | **Type** | **Use** |
| Window | Frame | The window will be used as a container to enclose the application. |
| Text Boxes | - | The dialogue boxes present an area for Username and Password to be entered. |
| Button | Push button | The Login button will trigger a new window with text boxes; pertaining to login. |
|  | Check box | The User will select their Major(s), minor(s) and the courses of interest with check boxes. |
| Menu | Pull-down | The User will be given a list of the courses based on their selection. |
| Interactive Table | Javascript enabled table | The table will be the primary interaction component of the user. It will allow the elements within to be dragged around in order to re-arrange the course plan. |
| Overlay | Overlay Divs | The overlay will be used for either to present the login form or to notify user of serious errors in the course plan. |

## Detailed Description

The following represent the mockups for the different screens of the system followed by a small description of what the screen represents:

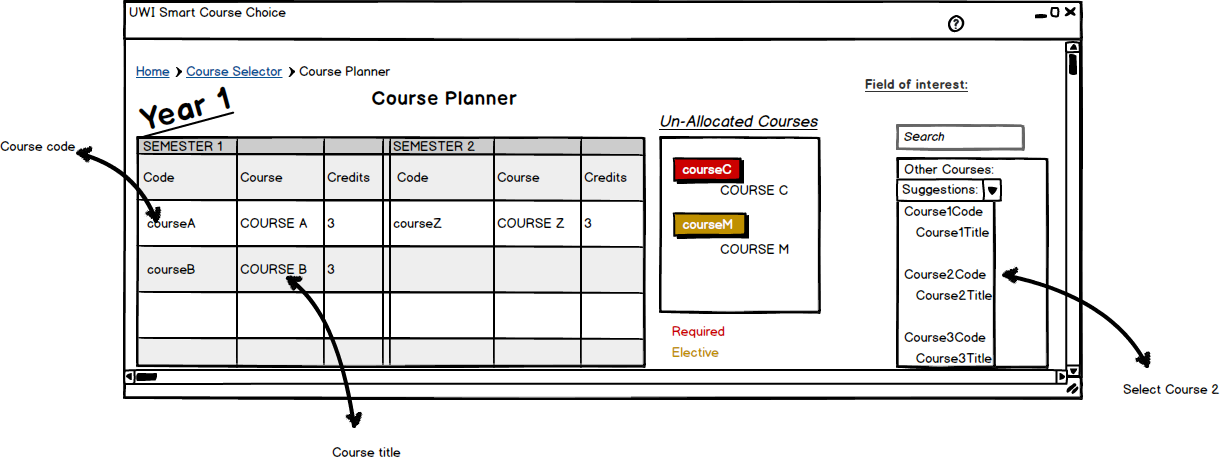


Screen 1 Welcome Screen

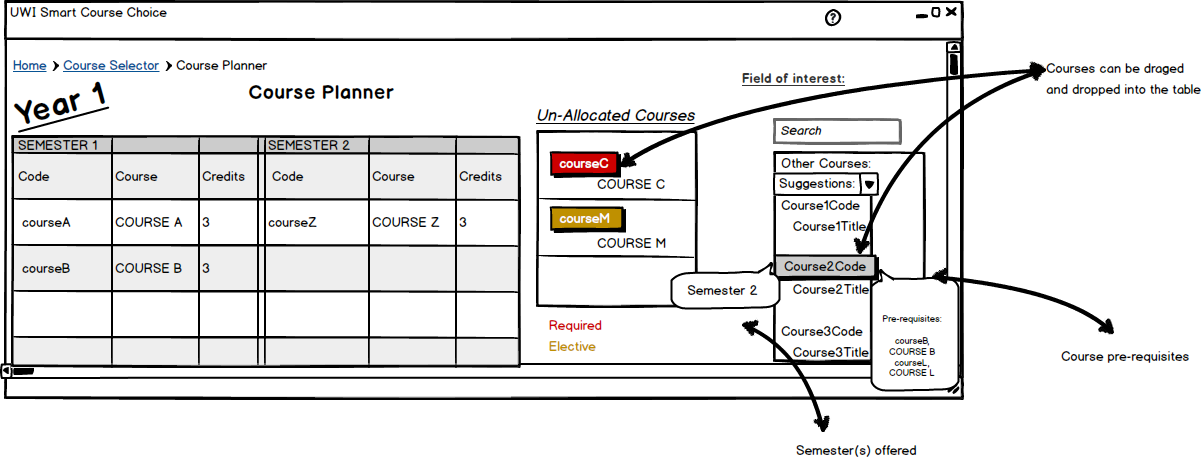


Screen 2 Login Screen

Screen 3 Programme Selection Screen - This screen shows the available courses; for major(s) or minor(s)



Screen 4 Course Plan Screen 1- This screen shows the Course Planner; a screen that allows users to rearrange their courses.



Screen 5 Course Plan Screen 2 - This screen shows the information given when a course under the field of interest is hovered over

# Conclusion

The design described in this document will be implemented using PHP as the server-side scripting language in order to realize the project. Our testing strategy includes creating unit test cases before hand and using them to test the validity of each unit and sub system of our system. This is important as a major constraint on the system is that the results are never inconsistent/ contradicting with those ascertained by human processing.