Project Document

Team 2 - CS374 - Fall 2014

In this document, we describe the specifications for our Schedule Conflict Calculator application.

Table of Contents

1 Requirements

Our Schedule Conflict Calculator Web application will function as an extension to other academic administration technologies. Given a set of inputs related to moving a course from one time block to another, the application determines the number of student scheduling conflicts that would arise in some number of move scenarios. The application handles a number of different move scenarios, allowing the user to decide which scenario to test.

2 Use Cases and Scenarios

The user is allowed multiple scenarios for calculating schedule conflicts. For example, sometimes the user has a single *section-change description* in mind and would like to see scheduling conflicts. In other instances, a user might know that they need to move a section to another time and/or room, yet would like to know which room out of a set of possible candidates has the least number of conflicts.

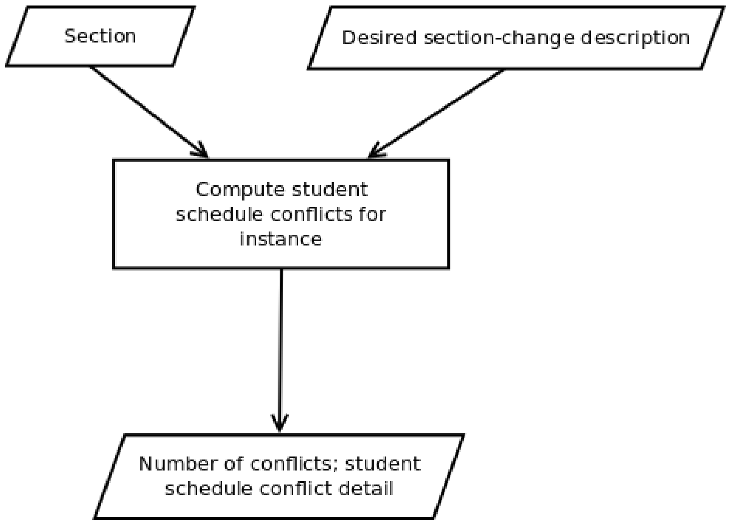
Section
Desired section-change description
Compute student schedule conflicts for instance
Number of conflicts; student schedule conflict detail

Figure 1: Describes a simple scenario for finding conflicts given a single section-change description

In the simple case, we consider a single section-change description supplied with a desired input section. Output will consist of the number of conflicts as well as a detailed look at individual conflicts. If another section is already *associated* with the section-change description, the user is informed of the conflict but is still provided the normal conflict output as if there weren't a course currently associated with the description. Other input parameters might be needed to provide a more customized operation tailored to meet user needs.

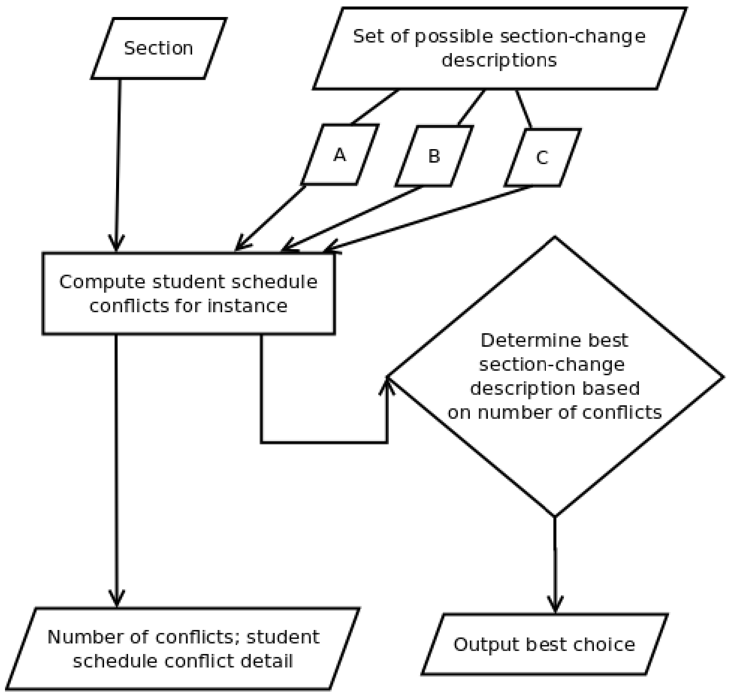
Section
Set of possible section-change descriptions
A
B
C
Compute student schedule conflicts for instance
Determine best section-change description based on number of conflicts
Number of conflicts; student schedule conflict detail
Output best choice

Figure 2: Describes a more complex scenario for finding conflicts given multiple section-change descriptions

Other scenarios are needed for more complex operations. A user might wish to provide a set of section-change descriptions to apply to the input section in order to find the best change description. This set of descriptions may encompass multiple rooms and buildings and also take into consideration moving courses already associated with section-change descriptions. This introduces added complexity.

To deal with this complexity, we introduce the terms *locked-section* and *floating-section* (see glossary). In the proposed scenario (Figure 2), a set of section-change descriptions is provided into which the input section may be moved. In a simple case, the scenario given in Figure 1 can be applied to each description. The description(s) with the fewest conflicts are then presented as output. However, sometimes more than one section can be moved within the sections already associated with each section-change description. Each one of these sections is either marked as a *locked-section* or a *floating-section.* A locked-section is one that cannot be moved. This automatically disqualifies the description to which the locked-section is associated (conflicts are still computed as in the Figure 1 scenario). A floating-section may be moved within the set of section-change descriptions.

3 Data Flow

This section describes the interaction of various modules within our application. We map the flow of data through the different modules of the application, demonstrating how it is used and transformed.

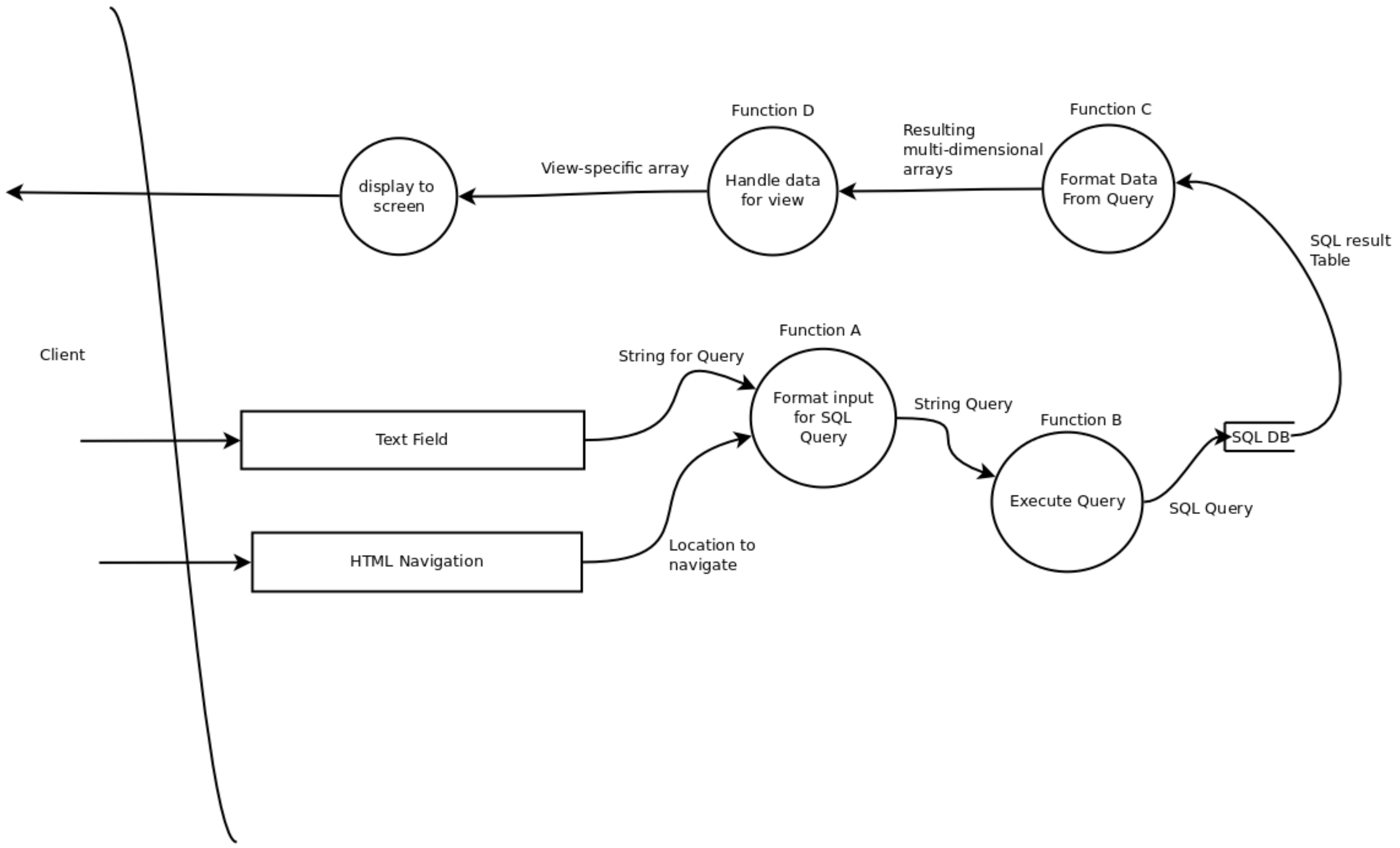

Figure 3: Data flow diagram for main application operation

In each proposed scenario, a user provides input through an HTML form which is passed to a data handler. The data handler uses an SQL database connection to issue a query and retrieve the data needed to execute the task. The handler is responsible for producing the desired result. Different handlers will require different amounts and/or types of data and thus will make different queries. Handlers will be implemented using a class hierarchy in PHP. Each derived class will implement a different handler-kind which executes a distinct scenario.

Though each handler performs a different operation, it still functions according to the same interface. Figure 3 graphically models data as it flows from the client, is transformed and goes back to the client as output. Figures 4-6 provide in more detail the functions presented in Figure 3.

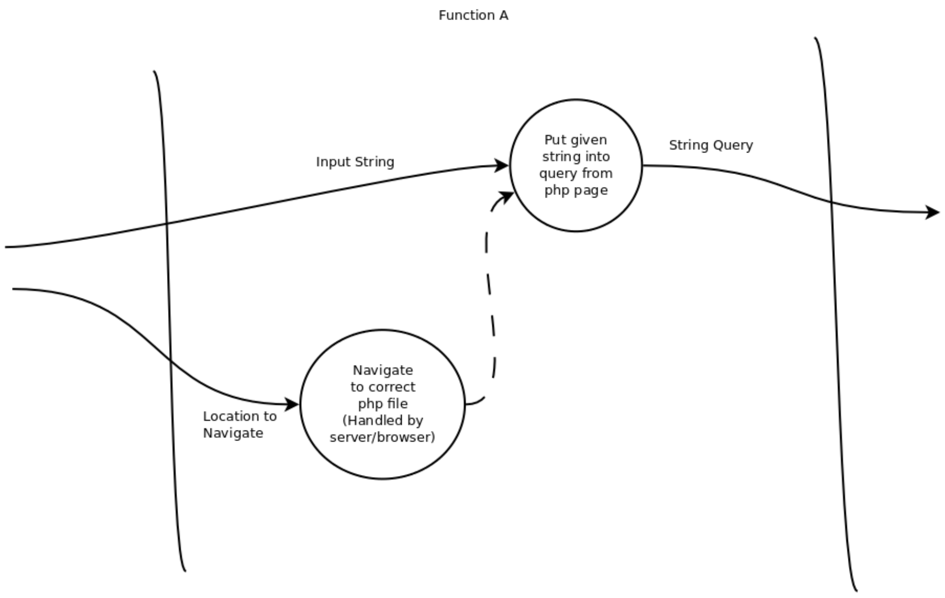
Function A
Input String
Put given string into query from php page
String Query
Navigate to correct php file (Handled by server/browser)
Location to Navigate

Figure 4: Details function A in Figure 3

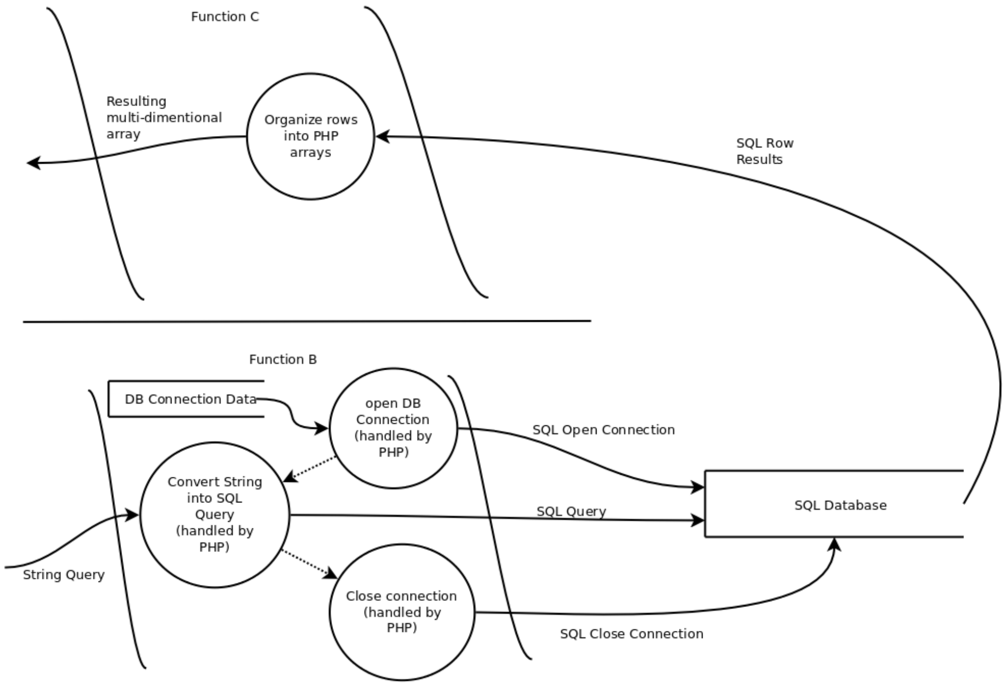
Function C
Resulting multi-dimentional array
Organize rows into PHP arrays
SQL Row Results
Function B
DB Connection Data
open DB Connection (handled by PHP)
SQL Open Connection
Convert String into SQL Query (handled by PHP)
SQL Query
SQL Database
String Query
Close connection (handled by PHP)
SQL Close Connection

Figure 5: Details function B and C in Figure 3

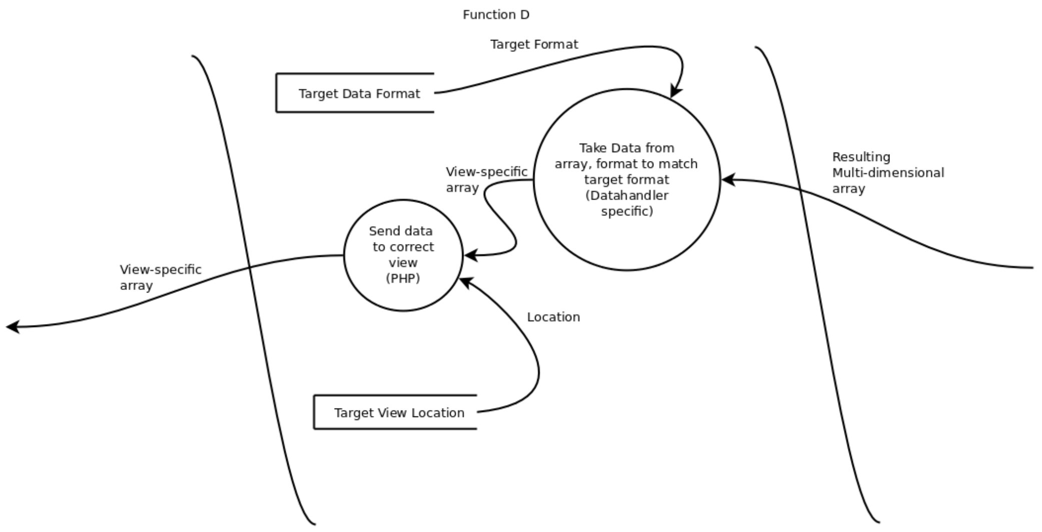
Function D
Target Format
Target Data Format
Take Data from array, format to match target format (Datahandler specific)
Resulting Multi-dimensional array
View-specific array
Send data to correct view (PHP)
View-specific array
Location
Target View Location

Figure 6: Details function D in Figure 3

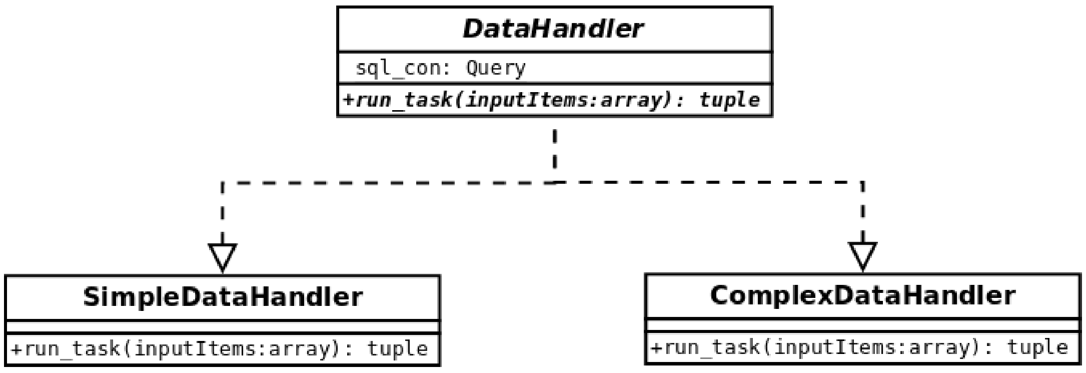
DataHandler
sql_con: Query
+run_task(inputItems:array): tuple
SimpleDataHandler
+run_task(inputItems:array): tuple
ComplexDataHandler
+run_task(inputItems:array): tuple

Figure 7: Simple UML concept for data-handler class-hierarchy

4 Design

Here is the layout of the directories in the www folder of the system:

aboutus.html
conflicts.html
index.html
schedule-browser.html
section-browser.html
student-search.html
style.css
banner/conflicts.php
banner/course.php
banner/section.php
banner/show-section.php
banner/student.php
banner/student-schedule.php
banner/student-search.php
php-bin/libhandler.php
php-bin/libquery.php

The php-bin directory holds the class declarations, and the banner directory holds the implementation of those libraries.

The following diagram is our Database diagram

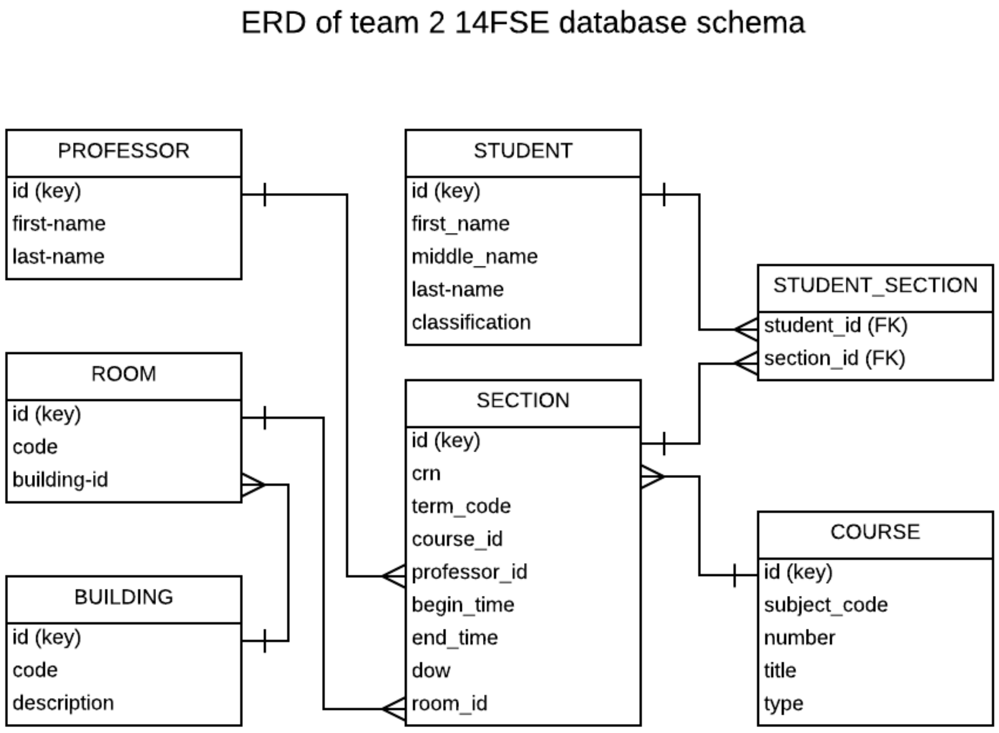

Figure 8: ERD showing the relational database

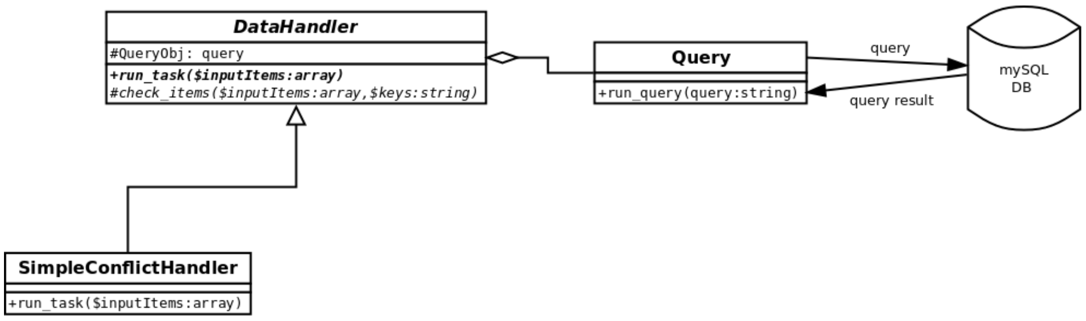

Figure 9: UML Class Diagram showing our object design

5 Verification

The verification of our project will include the following procedures:

Black-box testing
Module Testing
System Testing
Acceptance Testing
Performance Testing
Status Reports

These are the ways in which we will implement the aforementioned procedures.

Black Box Testing

Our team used the testing framework *Cucumber* for testing some site-related functionality. *Cucumber* can check for expected outputs when given an input or set of inputs. Using *Cucumber* will allow comprehensive, automated testing of our files.

Module Testing

We have manually tested module inputs and test the output, using the actual classes/functionality of our final product

System Testing

This project's entire system has been manually tested, where input from a person was testing against the expected output.

Acceptance Testing

Our team is presenting the product to the client, for the client to decide if it meets the criteria set out for it.

Performance Testing

The system has been tested at production level, and should only be operated at the capacity it is currently running, as old semester data must be removed as new semester data is added.

Also, status reports are made weekly to record the work that is done each week and how the team is performing.

6 Glossary

association The set of courses whose location matches and time duration overlaps a section-change description

building A collection of rooms. Each building has a set of floors.

course A general description of a class of sections. Each course has a string (for the course type, such as CS or IT), and a number (such as 101), which can be either a string or an integer.

data-handler An object that executes a scenario using data obtained from the database.

floating-section A section that may be potentially moved from either its current time, location or both. This does not mean it can be moved from one instructor to another, nor can the section number change.

freshman A student of Year 1. It is preferable to move or change freshman classes rather than upperclassman classes, because there is more flexibility.

junior A student of Year 3. Less flexible than Freshmen and Sophomores, which means they should be changed after Freshmen and Sophomores, but unlike Seniors, it is not critical that they not be moved.

locked-section A section that cannot be moved from its current time and location

professor An instructor of a course; there is one professor per section

office-hours A time block associated with a professor; a potential conflict item to consider. If an instantiation of a course falls within time allotted to office hours, there is a conflict; this conflict is of less importance than conflicting course instantiations.

room The meeting place of a section. Rooms are assigned to a Building. Rooms cannot be reassigned, since they do not physically move.

scenario A section change operation (e.g. moving a section from one time to another)

section An instantiation of a course. The number of sections starts at 01 and increases based on the number of classes that are to be taught for a given course such as CS101: CS101 01, CS101 02, etc.

section-change description A tuple that specifies a desired change to a section's time and location. Non-room changes have a hierarchy; change in a section's day implies change in time. A change in professor implies change in day and change in time.

semester code A six-digit sequence for the semester in an academic year represented by yyyynn. An academic year starts on the Fall semester of the current year; Fall 2014 is the start of the 2015 academic year. For nn, 10 is the fall semester, 20 is the spring semester and 30 is the summer semester. A conflict only occurs on office hours or course instantiations with the same semester code. examples: For Fall 2014 courses, the semester code is 201510 The Spring 2015 semester code is 201520 The Summer 2015 semester code is 201530

senior A student of Year 4; due to the priority on graduation, seniors should not be moved unless as an absolute last resort. Seniors who are in the second sememster of their year absolutely cannot be moved.

sophomore A student of Year 2; not as flexible as Freshmen, but more movable than Juniors and Seniors.

student A participant in a section. Each student has a unique banner ID. This banner ID is the key used. Each student also has a Major, which determines appropriate courses that they can take. Each student also has a Year: Freshman/1, Sophomore/2, Junior/3, and Senior/4. Seniors cannot be moved, due to the importance of graduation. Freshmen take the lowest priority; they should be moved before other classes.