Jeff Killeen, Jon Ice, Jonathan Nix

November 24, 2014

Project Beta

Design

Executive Summary: Karen, a degree plan specialist at ACU, was in need of an application that would ease in the process of taking students from a scheduled class and moving them into another timeslot. After she had moved a class, Karen also needed a way to view the possible conflicts students may have with this change. This application is intended to assist in the rescheduling of classes by finding the best possible timeslots for a scheduled class that has the least amount of serious conflicts.

Requirements:

Date Given	Summary
9/24/2014	User should be able to change the time (and possibly) location of a course
9/24/2014	List students who have conflicts after a class has been changed
9/24/2014	Rank the severity of each conflict by the student's classification and if the class is a required prerequisite
9/24/2014	Data that is used to display schedule must be taken from banner.
9/24/2014	Every section listed has a unique CRN and can be found through the building it is located in.

Work Breakdown Structure

Jeff Killeen: Project Lead. Has more knowledge and experience in the development of web applications and working with Ruby and HTML. Project Lead will provide direction and assistance where it is needed.

Jon David Ice: Backend developer. The business logic and database access of our application will be handled in this area with the help of Ruby and MongoDB.

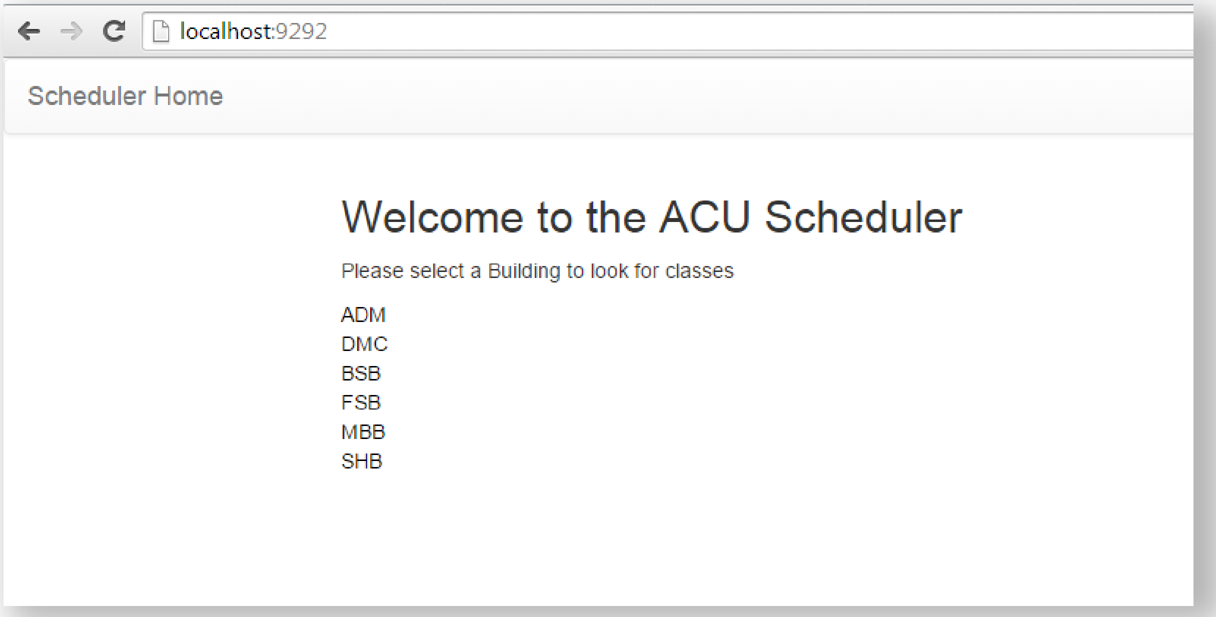
Jonathan Nix: Frontend developer. The views of the application will be handled in this area. The frontend developer will be in charge of making the webpages responsive with HTML, CSS, and Javascript with some help from other libraries such has Twitter Bootstrap.

Project scope

Detailed design by activity

Functionality of Application

1) When the user visits the home page, he/she will be shown a list of buildings to choose from. When one is selected, the application will generate the class schedule calendar for that building. The picture below is an example of what the user will see:



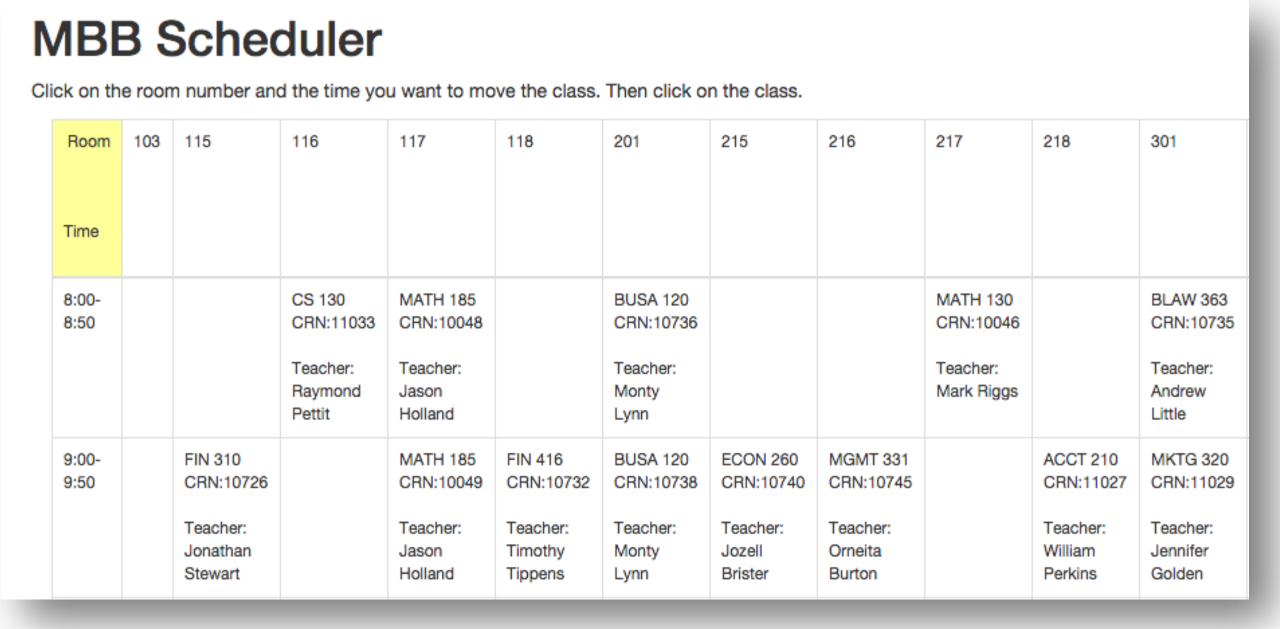
localhost:9292

Scheduler Home

Welcome to the ACU Scheduler

Please select a Building to look for classes

ADM
DMC
BSB
FSB
MBB
SHB



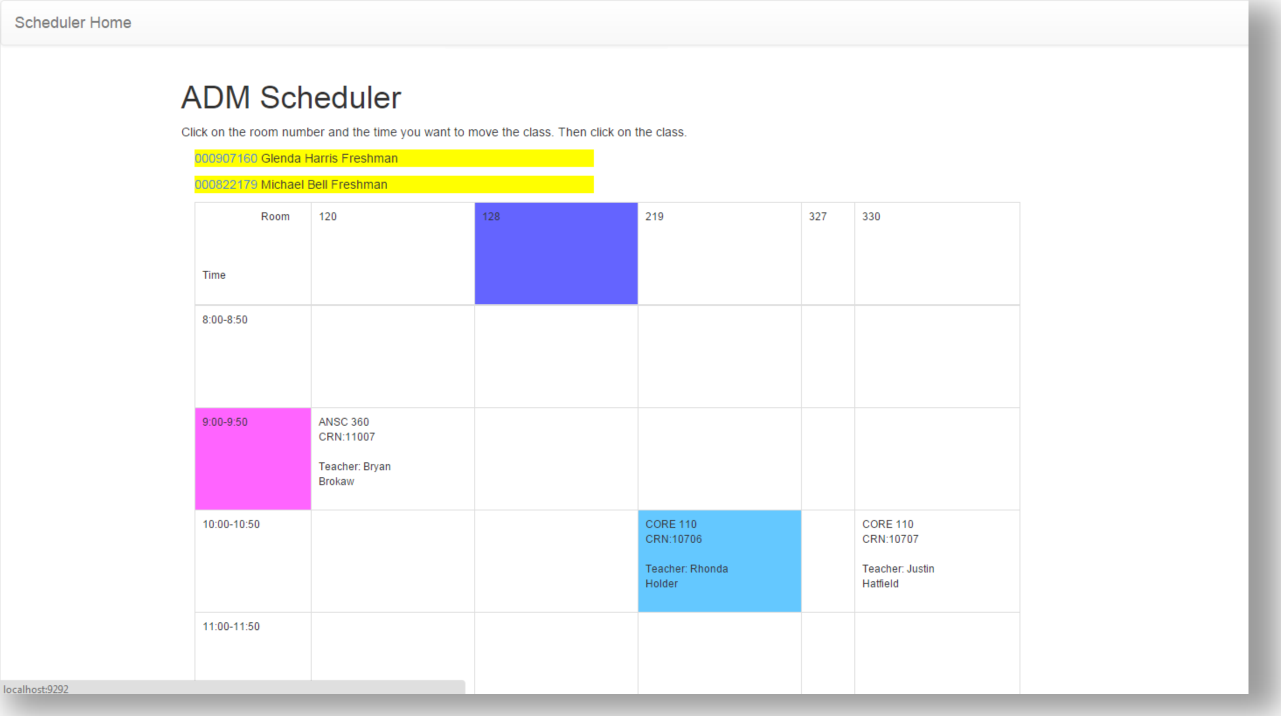
MBB Scheduler

Click on the room number and the time you want to move the class. Then click on the class.

Room Time	103	115	116	117	118	201	215	216	217	218	301
8:00-8:50			CS 130 CRN:11033 Teacher: Raymond Pettit	MATH 185 CRN:10048 Teacher: Jason Holland		BUSA 120 CRN:10736 Teacher: Monty Lynn			MATH 130 CRN:10046 Teacher: Mark Riggs		BLAW 363 CRN:10735 Teacher: Andrew Little
9:00-9:50		FIN 310 CRN:10726 Teacher: Jonathan Stewart		MATH 185 CRN:10049 Teacher: Jason Holland	FIN 416 CRN:10732 Teacher: Timothy Tippens	BUSA 120 CRN:10738 Teacher: Monty Lynn	ECON 260 CRN:10740 Teacher: Jozell Brister	MGMT 331 CRN:10745 Teacher: Orneita Burton		ACCT 210 CRN:11027 Teacher: William Perkins	MKTG 320 CRN:11029 Teacher: Jennifer Golden

2) The user will be asked which class they would like to reschedule. They will click on the building, time, and then the section they want to move represented by a CRN.

3) Once clicked, all the student conflicts will be listed at the top of the page by order of student classification.



ADM Scheduler

Click on the room number and the time you want to move the class. Then click on the class.

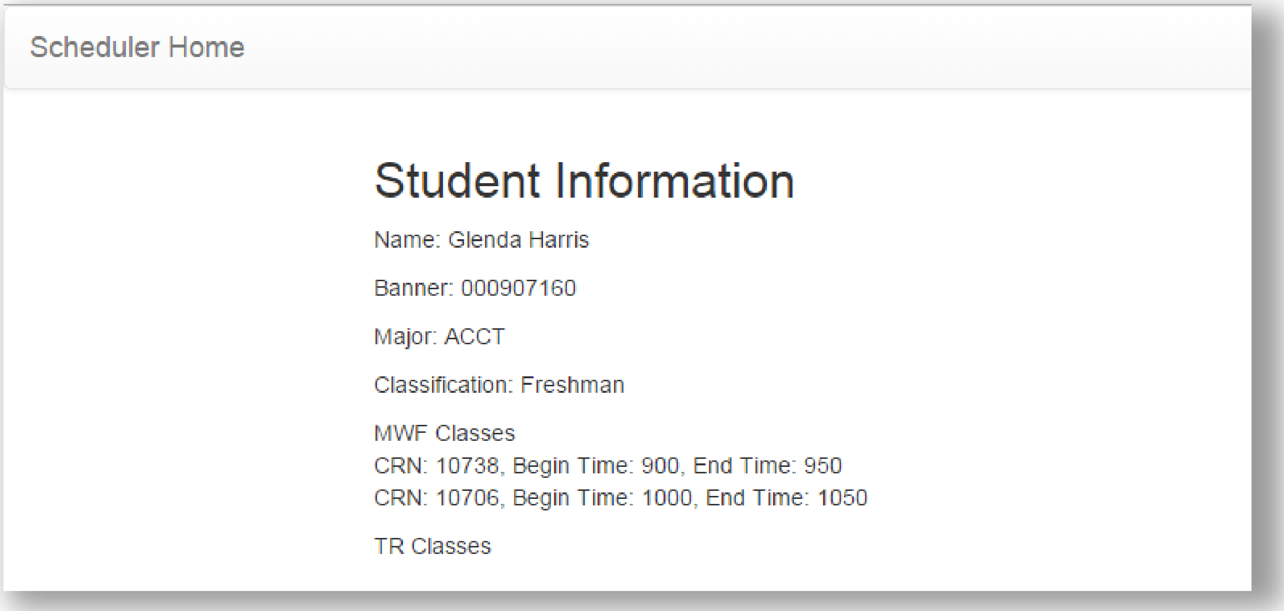
000907160 Glenda Harris Freshman

000822179 Michael Bell Freshman

Room Time	120	128	219	327	330
8:00-8:50					
9:00-9:50	ANSC 360 CRN:11007 Teacher: Bryan Brokaw				
10:00-10:50			CORE 110 CRN:10706 Teacher: Rhonda Holder		CORE 110 CRN:10707 Teacher: Justin Hatfield
11:00-11:50					

4) Once the students are listed, clicking on a banner ID will send the user to a student information page.

5) Student information page will contain information on the student and their class schedule.



Student Information

Name: Glenda Harris

Banner: 000907160

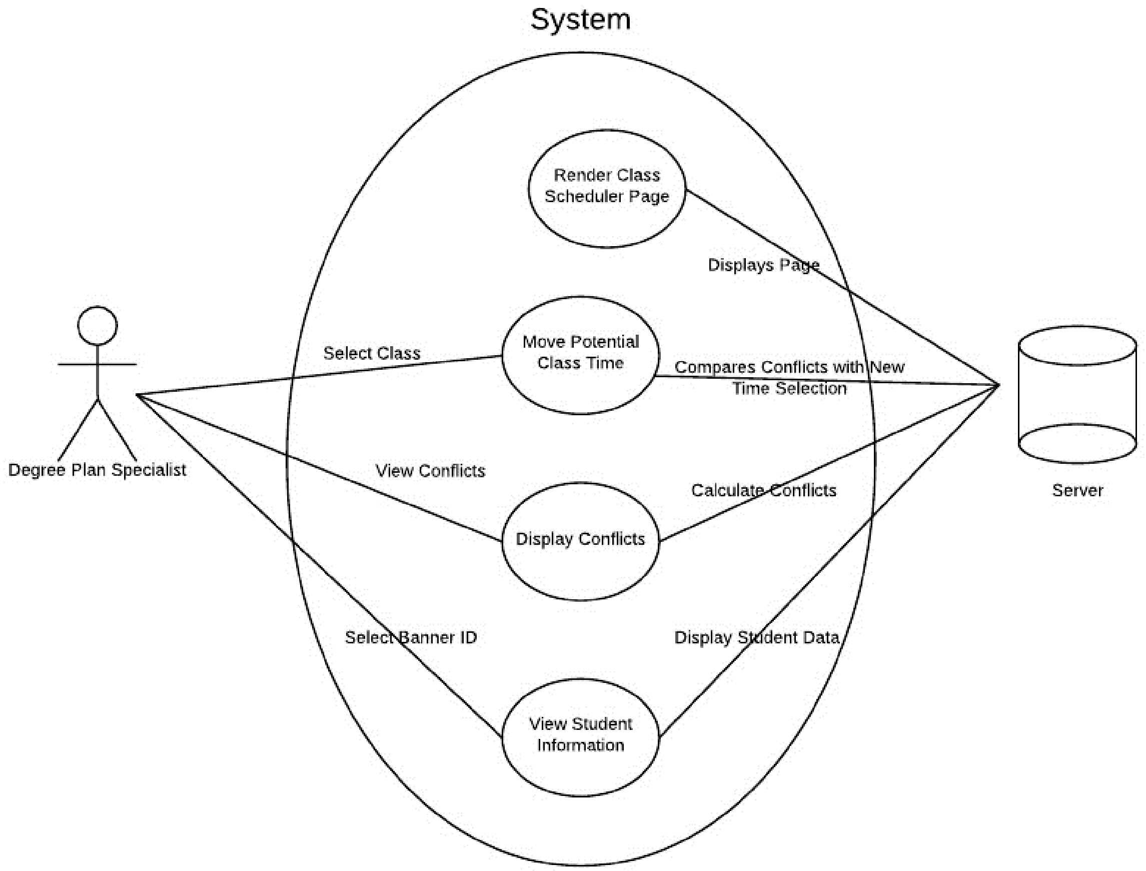
Major: ACCT

Classification: Freshman

MWF Classes
CRN: 10738, Begin Time: 900, End Time: 950
CRN: 10706, Begin Time: 1000, End Time: 1050

TR Classes

Use Case Diagram:



Architecture

In our application, we have base our implementation around the MVC design pattern. Using this design pattern will allow us to separate the logic so that there is no confusion on what each part of the application does and it will also allow the different layers of our application to communicate with one another.

Our application is initially set up using the Sinatra web framework. This means that the controller or business logic of our web application will be handled by Ruby. Component-based software engineering will be made easier for us thanks to Ruby’s assortment of gems that have been made to speed up the web development process. For the views aspect of our MVC design, we will be using embedded ruby files, or ERB for short. ERB will allow us to execute Ruby code on our previously existing HTML pages so that they will become much more dynamic.

Finally for our model, we will be making use of the Mongoid gem in Ruby, which allows us to wrap classes created in Ruby into Documents that will be used in MongoDB. The Mongoid gem lets us write code that appears to be written in Ruby that is easily translatable into MongoDB. MongoDB was chosen for the project because it is designed to be fast and scalable so that users will get minimum wait time for their queries.

Database

As mentioned above, we will be using MongoDB for the database for our web application. One helpful thing that MongoDB does is that it allows our entities to hold embedded documents. This means that we can easily see the information stored between one entity and another.

For our database, we will have multiple entities for our application. All of the entities have components that coincide with one another. The ERD listed below helps describes the relationships between our entities.

With the Mongoid gem, we are able to create classes in Ruby that can be translated into documents in Mongoid. The following classes will be created:

Teacher: A Teacher will have zero to multiple Sections. An important distinction to make is that a Teacher does not teach a Course but a Section. This will allow us to distinguish the different times a Teacher might be teaching a class.

Course: A Course will have a name, a department. A course can have many sections or no sections at all, if it is not being taught that semester. We can’t have the course many-to-many to itself, that is, have prerequisites, because the data we are provided does not contain any prerequisite information.

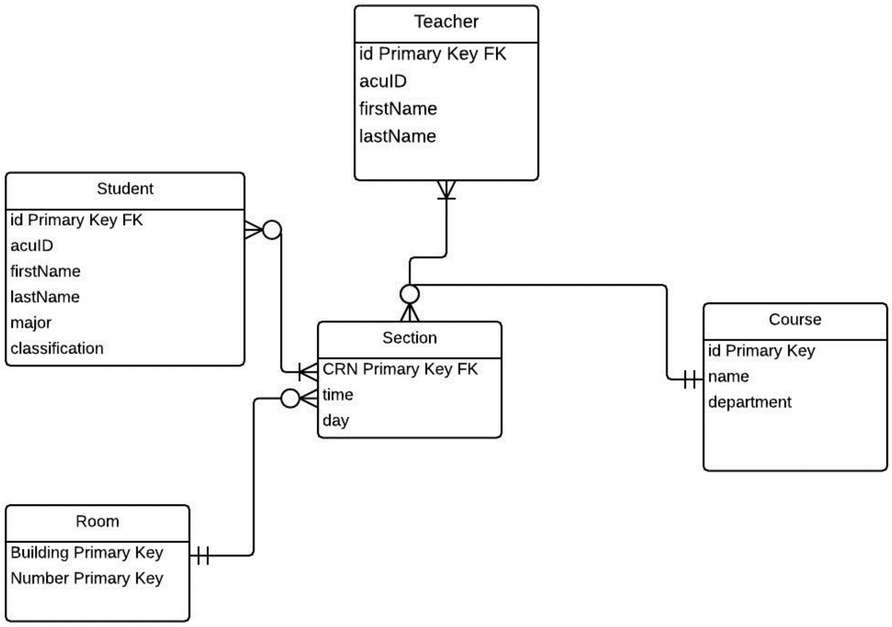
Section: A Section will have a unique CRN, with a certain date and time of the week. A section will belong to a Room.

Student: A Student will be enrolled in multiple sections. The student will need to take certain Sections that will correspond with a Course needed for graduation. The student’s classification will determine the severity of changing a course within our application.

Room: A Room will have a building attribute and a room attribute to distinguish between other rooms. A Room can be home to many Sections and a Section must belong to some room. There is no way to determine the occupancy of the room by the data provided to us so we are assuming equal room sizes, even though we know this is not always the case.

Class Scheduler

Entity Relationship Diagram



Glossary

ACU ID- Both teachers and students have an ACU ID. This describes the unique ID that is used to identify a teacher or student.

Building- Each section has a building. This describes which building the section takes place in.

Course- A course is implemented by any number of specific sections

Classification- Each student has a classification. This describes the student’s current standing at the university (e.g. Sophomore, Junior, Senior).

CRN- Each section has a CRN. This describes the unique number used to identify a section.

Day- Each section has a day. This describes which days the section is on.

Department- Each teacher has a department. This describes the department that the teacher belongs to.

First Name- Both teachers and students have a first name.

ID- Both teachers and students have ID's. This describes the unique number used to identify a teacher or student.

Last Name- Both teachers and students have a last name.

Major- Each student has a major. This describes what type of degree plan the student is currently on.

Name- Each course has a name. This describes what the content of the course is about to the user.

Room- Each section will be taught in a room

Room Number- Each section has a room number. This describes which room the section takes place in.

Section- A section is an instance of a course.

Student- A student is enrolled in any number of specific sections.

Teacher- A teacher teaches any number of specific sections.

Specification

Product Overview

This product is intended to assist users in the rescheduling of specific sections.

Requirements

Priority Definitions

1. This requirement is a “must have” for the product to function.
2. This requirement will make the product much more efficient, but is not necessary for it to function. It is still highly imperative.
3. This requirement is a luxury and is not necessarily needed. Functional Requirements

Functional Requirements

Function	Priority	Date Reviewed	Approved by
The system should provide all information available via Banner regarding the rescheduling of the section in question for the user to view.	1	8/12/14	Jonathan Nix
The system should provide the optimal solutions for rescheduling a section.	3	8/12/14	Jonathan Nix
The system should provide information in order of relevancy. The user determines relevancy and chooses which information they want to view.	2	8/12/04	Jonathan Nix

User Interface Requirements
The user interface should…
-- Provide prompts for the user to input current section information. Instructions on what input to give and what to do to run the application should be clear.
- Provide output as information in a row format. This will look be displayed in rows of data by order of relevance.

- Provide output as information in a list format. An example of this is displayed below the same existing room times' table.

Usability

The product should…

- Be easy to learn and understand. Clear instructions will be provided on each page regarding how and where to give input and how to view different output. (See User Interface)

Performance

The product should… .

- Be able provide correct feedback within an average of 5 seconds.

Manageability/Maintainability

The product should…

- Have access to Banner (currently reading from CSV that contains data from banner) and download information each night.
- Have embedded Ruby files that allows our system to be more easily maintained.

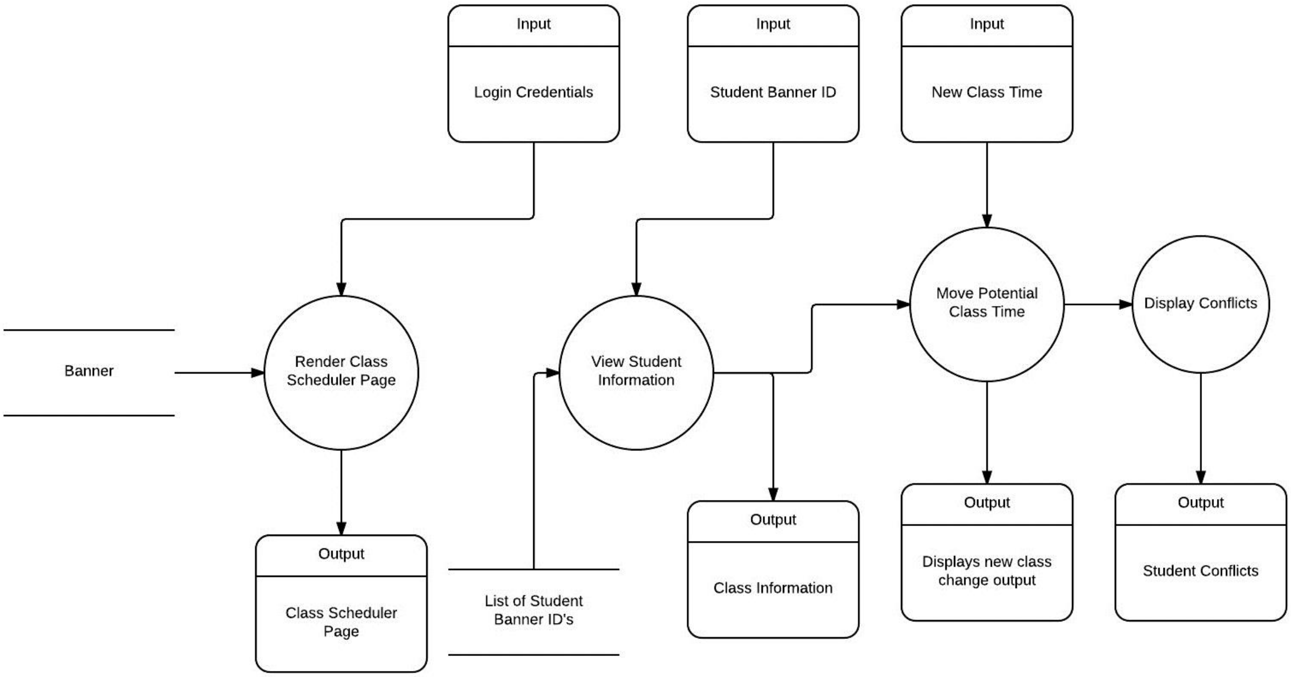
System Interface

- Our system will use MongoDB.
- Our system will use the testing framework, Cucumber.
- Our system will use Sinatra.

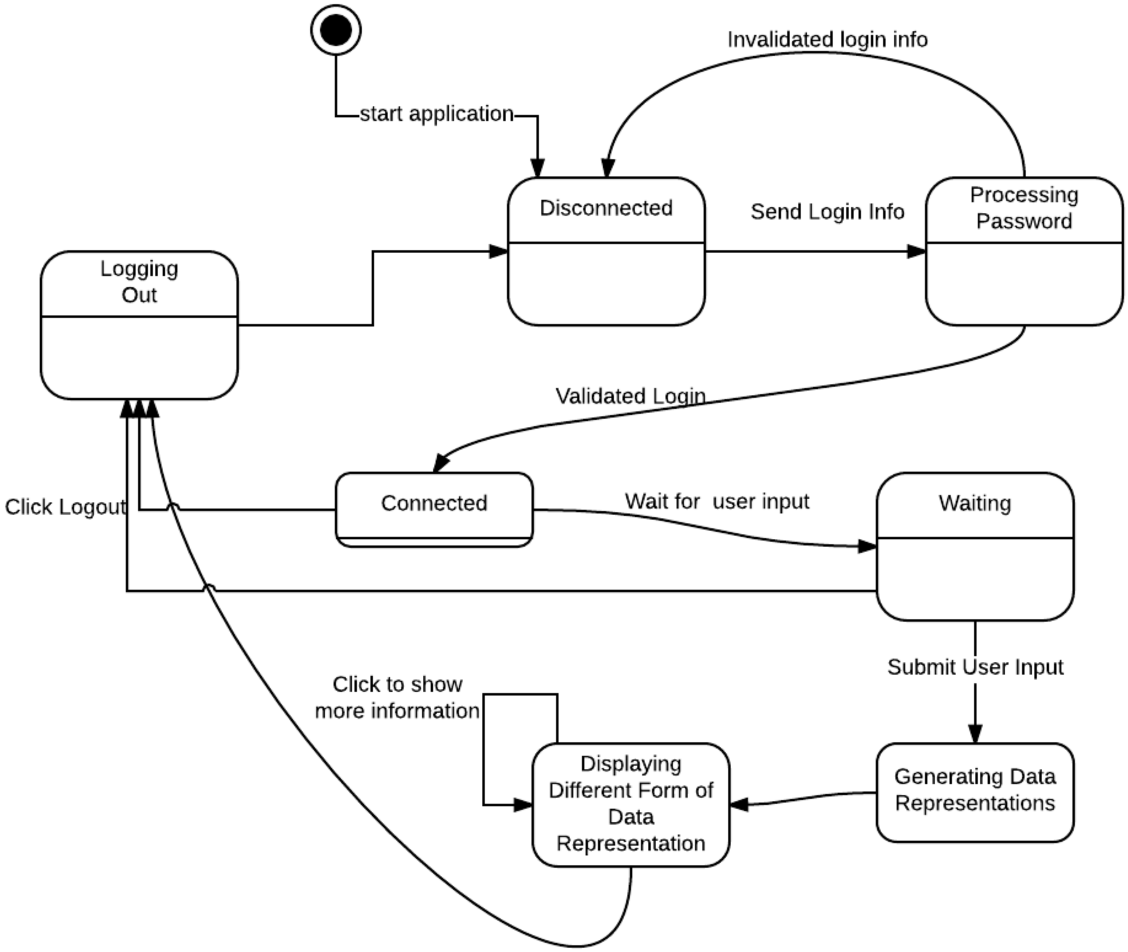
Logic and Algebraic Specifications

We felt that there was no need to describe our specifications in this way. After analyzing the different processes that our web application goes through, we saw that the addition of these descriptive specifications would only complicate our document. We felt that our previous specifications were sufficient in describing the workings of the application. The Algebraic specifications seems to be only relevant if our application heavily relied on using object oriented concepts, but we did not specify any objects oriented concepts in our design.

DFD


Input
Login Credentials
Input
Student Banner ID
Input
New Class Time
Banner
Render Class Scheduler Page
View Student Information
Move Potential Class Time
Display Conflicts
Output
Class Scheduler Page
List of Student Banner ID's
Output
Class Information
Output
Displays new class change output
Output
Student Conflicts

State Machine



Note: The logging in and out states/capabilities are not fully functional. We currently have tests for those features that are not passing.

Verification

Testing in the Small

White box testing:

Statement Coverage: We have tested every possible area that we can reach within our code to ensure that every path is covered.

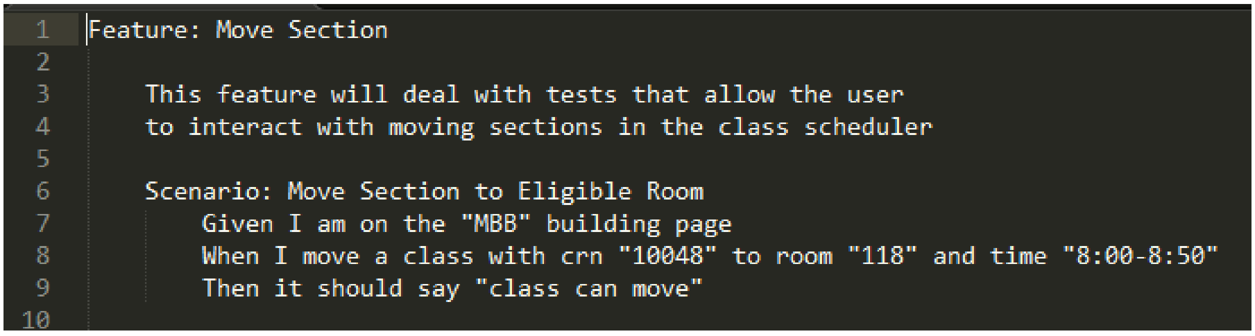
Edge Coverage: We have written tests that traverse each edge of the diagram at least once to catch as many errors as possible.

Condition Coverage: An enhancement of Edge coverage, we made sure that our tests caught even more errors. To do this, we created tests that resulted in all possible values of the constituents of compound conditionals at least once. This simply means that given a conditional such as “if A and B”, we created tests that addressed each combination of A and B where A and B can either be true or false. For our instance, that meant testing different combinations of moving a class to a different time given varying factors such as day, time, and other conceptual restrictions. We accomplish this by constructing a cucumber scenario for each possible edge based on our code.

Black box testing:

In black box testing, we checked to see if a specific part of a program did what it is intended to do. This was done through our cucumber tests in our project. By taking a test driven development approach, we were able to write tests that said how the features of our application should work and then we actually coded our features to make our tests pass. This type of testing is testing driven by logic specification.

Here is one of our tests. It is our moveSection.feature and the step definitions that go with it. This feature allows the user to move a section and lets us know if the room, time, and day are eligible.

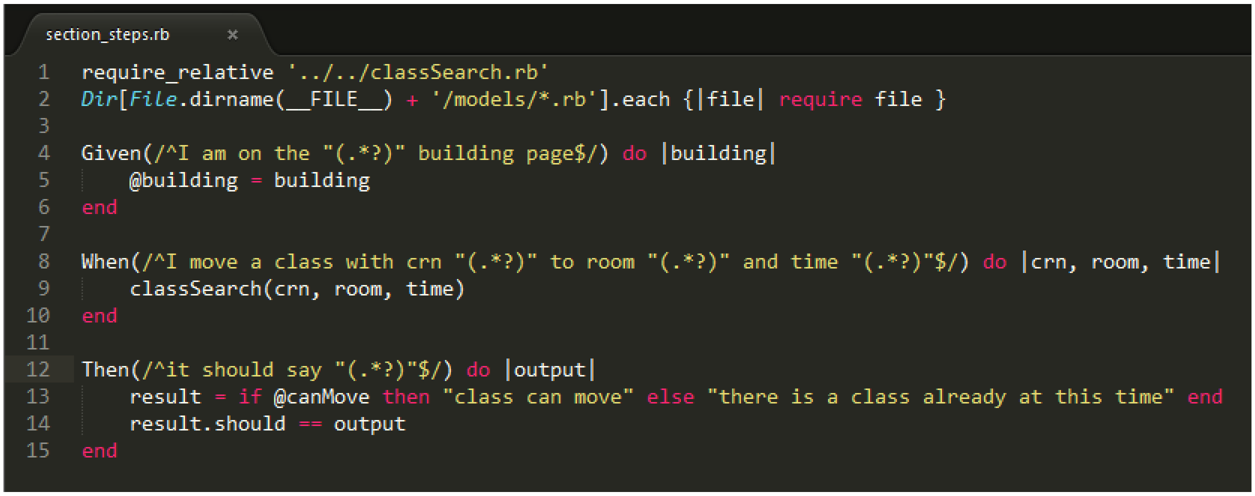


```
Feature: Move Section

    This feature will deal with tests that allow the user
    to interact with moving sections in the class scheduler

    Scenario: Move Section to Eligible Room
        Given I am on the "MBB" building page
        When I move a class with crn "10048" to room "118" and time "8:00-8:50"
        Then it should say "class can move"

```



```
require_relative '../../classSearch.rb'
Dir[File.dirname(__FILE__) + '/models/*.rb'].each {|file| require file }

Given(/^I am on the "(.*?)" building page$/) do |building|
    @building = building
end

When(/^I move a class with crn "(.*?)" to room "(.*?)" and time "(.*?)"$/) do |crn, room, time|
    classSearch(crn, room, time)
end

Then(/^it should say "(.*?)"$/) do |output|
    result = if @canMove then "class can move" else "there is a class already at this time" end
    result.should == output
end
```

More tests like these are in the project folder of our team's git hub.

Note: All of these features test backend only because we are unable to test javascript using cucumber. We are currently working towards a solution.

Testing in the Large:

There are three main aspects to our testing in the large.

Modularity Testing: Based on how we decided to implement our code, we tested each module using white and black-box techniques. We also identified which modules depended on one another to operate properly and used tests that tested the constraints of these connections.

We currently have multiple features with multiple scenarios that have tested the constraints of our system and identified whether it functions according to our stated requirements. For example we have tests that simply see that if you try to move class to a time where there is another class, the system sends back an error and does not allow it and our testing verified this system functionality.

Integration Testing: As we continued to add modules, we tested to see if the new modules integrated well with the previously tested modules.

We used a top-down approach for integration testing. This allowed us to view the application in a more abstract sense in terms of what features it had and then push down to subsets of those features.

System Testing: When we finished the prototype, we tested the entire system to ensure overall functionality was working as expected and use test data to work out bugs.

We used overload and robustness testing to figure out the limits of our application. However we did not worry about testing for regression because this is just a semester-long project that will be, for the most part, out of our hands when we are finished. If we were required to be in charge of this application and adding new features as they were requested, we have been more concerned with testing for regression.

Analysis

Code Walk-Throughs: We had several scheduled opportunities to go through our code and trace it by hand to find any logic errors that have occurred in the process of writing the code. When these errors were found they were dealt with appropriately.

Code Inspections: Similar to the code walk-throughs, we had several code inspections to look through our code with the specific intent of looking for common errors in the code. When these were found they were dealt with appropriately.

Correctness Proofs: We will not be using correctness proofs on our project and are instead focusing on testing using the Cucumber framework. We believe that this will be sufficient in providing feedback on the functionality of our system and feel that using correctness proofs would be more detrimental than helpful for the following reasons:

- would be longer and more complex than the programs it is intended to prove.
- overwhelms the designers with more details than could easily be dispensed with if they use an informal analysis.

- Even if mathematical certainty of the correctness of a program can be achieved, we cannot rely on it in an absolute way because there could be a failure in the implementation of the language (the compiler) or even in the hardware.

Symbolic Execution

Symbolic execution allows us to combine the techniques of analytical testing and experimental testing. We will not be using symbolic execution in our testing.

Model Checking

For model checking, we already have a basic finite state machine. We have used it in our testing to see if our application does fulfill each property that we have specified. If did not, we have a counterexample, or a test case that causes the application to break.

Debugging

Debugging is technically not a verification activity, but it still goes hand in hand with our other testing approaches. A failure is a behavior that does not match the product specifications. It is difficult to find certain failures in a project because the failure can be determined by multiple errors. In order to debug our project, we have a few precautions in place. Using Sinatra, we have seen if there is a certain action that breaks our system; we are immediately notified with the specific error on our web page. From there, we go through our code and see what line caused the error. If we reach a failure that greatly affects the features of our application, then we have to reevaluates the coding of those features and have our tests address that change.

Verifying Other Software Properties

Performance: For the scope of our project, we will not be using formal performance analysis technique such as worst-case runtime as it is not vital considering the intent of our application. Unless the product performs unreasonably below expected time, formal analysis is unnecessary.

Reliability: As with performance, it is unnecessary for our application to have extensive formal reliability testing that would ultimately be more extraneous than helpful and thus we will measure reliability with very basic level probability statistics such as how many times out of 10 tests does it return a correct result.

Subjective Qualities: Considering the scope and intent of our project, the measurements of subjective qualities in our application such as a program’s complexity will be left to the developers’ internal discretion. It is not necessary for us to use complex theories to validate these qualities in our project because its small scope allows the developers to gauge these using human intuition and still be able to use accurate metrics for these qualities.