Team 5

CS 374

Dr. Brent Reeves

December 2, 2014

Design Omega

Data input: Banner, desired class CRN and time from user.

Output: A list of students with class conflicts at the desired time, the conflicting classes, and possibly a hierarchy of class priorities/alternative times.

Compatibility with other systems: Not at the moment, this design deals with primarily Banner as it will be our source for student and class data for the foreseeable future. Further design for new systems may be considered at a later stage.

Likely changes: Since we are limiting ourselves to only getting input from Banner, the only change would likely be the format in which Banner presents data.

Product family: Web-based. We are going with a depth-first approach, because this program should only be accessible, at least at first, by advisors, degree plan specialists, and department chairs. These people can easily access a web-based system, and other families, such as mobile, should not be required.

Modularity: To prepare for the possibility of multiple data input possibilities, the Adapter pattern can be used to adapt incoming data to an expected format.

Mechanism vs Policy: The policies by which the mechanism we are designing adhere to are defined by the priorities in which different classifications of students have over each other. A student with a higher classification will have a higher priority in terms of class change compared to a student with a lower classification. The frequency of which a class is offered is also a concern with regards to policy. One that is infrequently offered has a higher priority over one that is commonly offered.

Model: Client-Server. This model was chosen as all data comes from Banner, making it an ideal candidate for our server. The Client is our program, which pulls data from Banner, takes ideal class time input from the user, then outputs a list of students that have conflicts.

Pattern Used: Model-View-Controller (MVC). Model is how our data is

represented within our system (MySQL Database Backend), the View is our web front-end, and our controller is our business logic that will help determine the list of conflicts, and generate a database query to access that list. The controller also determines valid class times by querying the database.

Database Design:

1. Table of students.
2. List of classes.
3. Transaction table linking students and classes. (One-To-Many)

Student Table:

- ID
- FName
- LName
- Classification
- Email

Class Table:

- CRN
- Name
- Time
- Frequency (Integer value listing priority)

Transaction Table

- Student ID
- Class ID

Frontend:

1. Login page (Powered by MySQL Database of Users and PHP scripted).
2. Once logged in, User enters CRN and chooses from a dropdown list of available class times and days.

Use Cases:

1. Enter desired class and time and get list of conflicts.
2. Get List of available sections
3. Get list of students and their banner numbers
4. Enter student banner number and get student schedule

Specifications Omega

The Class Conflict

Specifications - Z

SWITCH::= on | off

Report::= Name of the conflicts

Time: 8….18
Time > 0

Class_ID: N Cl
Class_ID >0

Student _ID: N
Student _ID > 0

Conflict: N C
Conflict > 0

TimeButtons
Input: 8...18 → SWITCH

ClassButtons
Input: 1……N → SWITCH

ReportButtons
Extput: 1…. Conlict → Report

Report:
List of the conflicts with student_ID.

Complete state space
attempt #1

TimeButtons Class
Buttons ReportButtons

Conlict !=0 →TimeButtons(NewTime) = on V Report

Buttons(NewTime) != 0

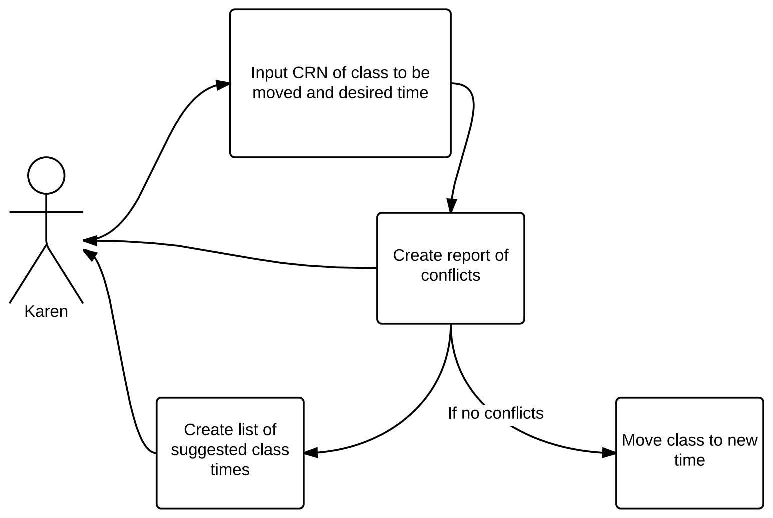
Complete state space
attempt #2
TimeButtons Class
Buttons ReportButtons

Conlict !=0
=> TimeButtons(NewTime) = on V Report
Buttons(NewTime)!=0 Conlict = 0 =>
(f:8…18 * TimeButtons(f)= off ^ ReportButtons(f)!=0)

Logic specifications
Class Conflict formula:

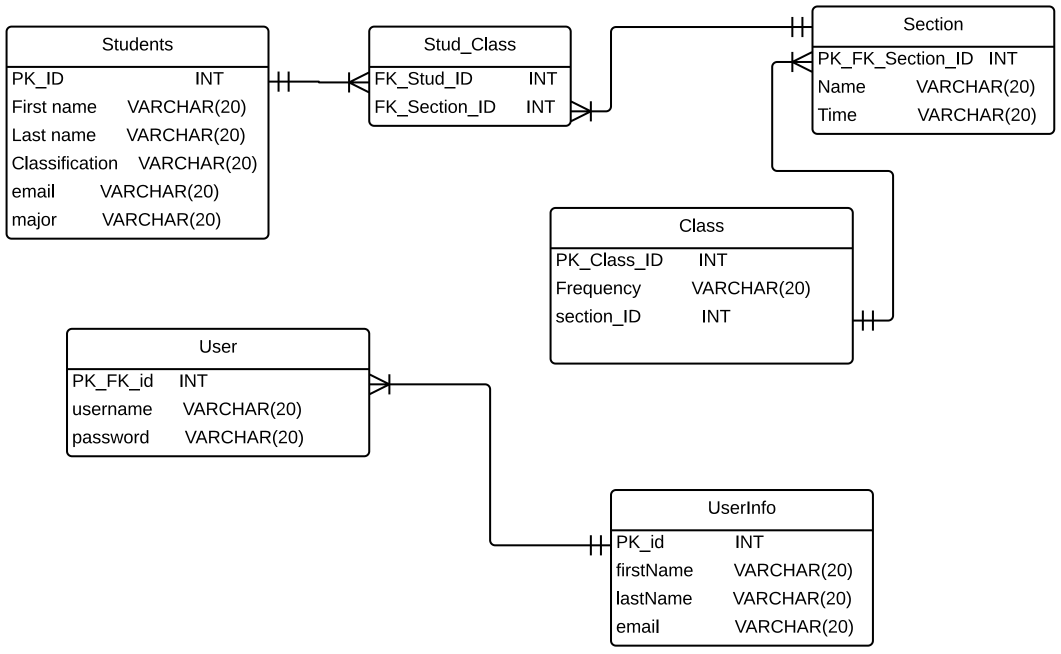
- New class time which is the class time we want replaces the old class time
  which is the class time is right now.
- Compare the new time to students schedules who are in the class.
- List all the conflicts which is the new class time conflict with students' other
  classes schedule from High priority to low priority
    a)Senior has the Highest priority then Junior then sophomore
    then freshman (class size will be considered afterwards)
- List the number of conflicts.
- Suggest a new time

UML Use Case

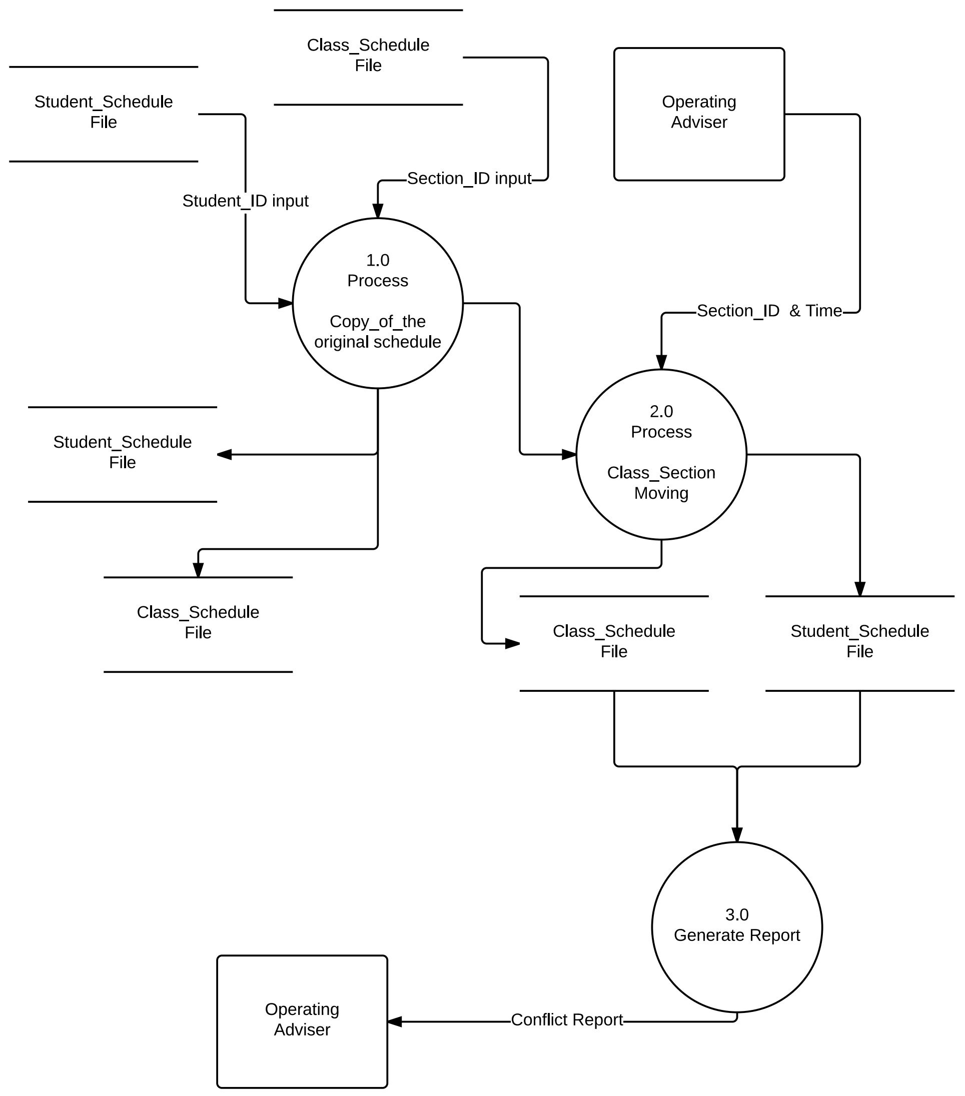

Input CRN of class to be moved and desired time
Karen
Create report of conflicts
Create list of suggested class times
If no conflicts
Move class to new time

Class Schedule

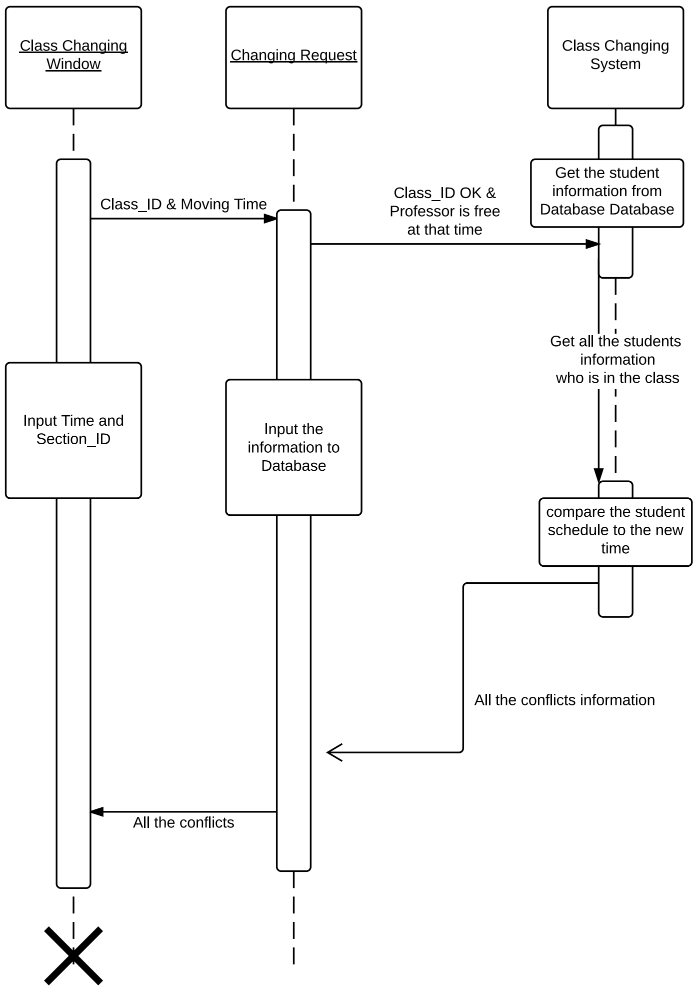
Entity Relationship Diagram



Data Flow Diagram

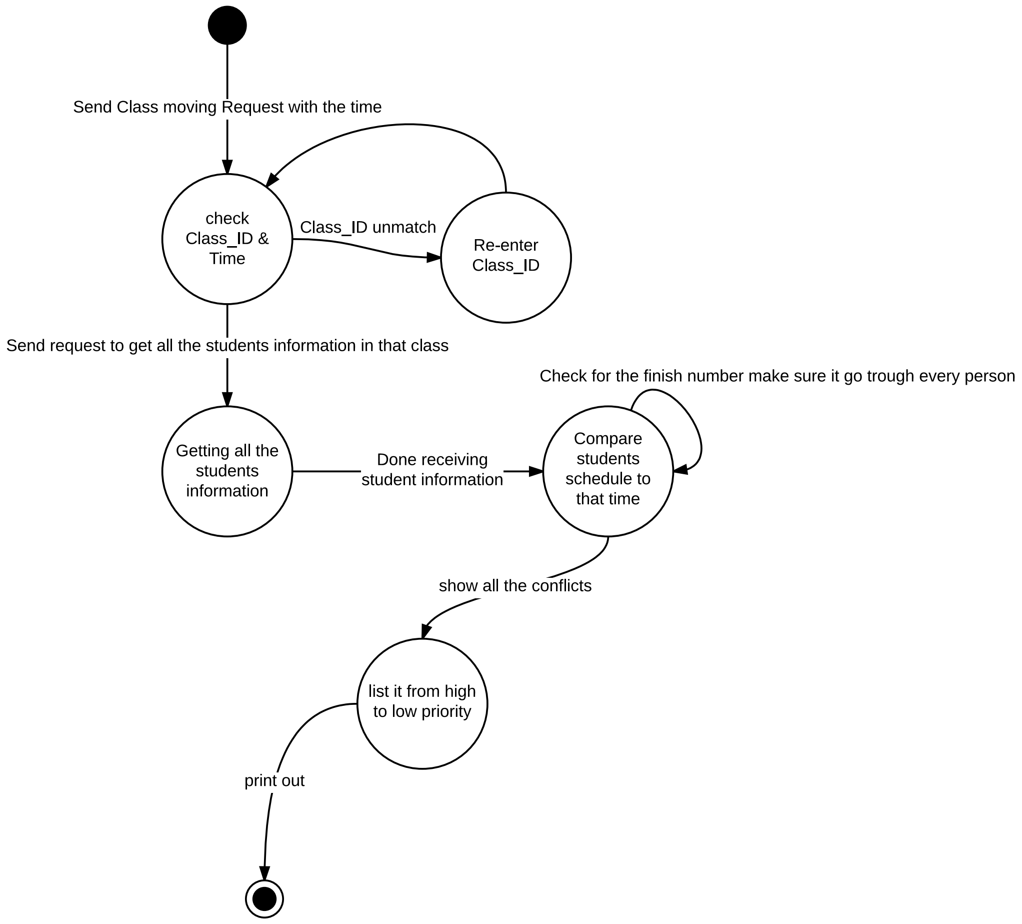


UML Sequence Diagram



Class Changing Window
Changing Request
Class Changing System
Get the student information from Database Database
Class_ID & Moving Time
Class_ID OK & Professor is free at that time
Get all the students information who is in the class
Input Time and Section_ID
Input the information to Database
compare the student schedule to the new time
All the conflicts information
All the conflicts

Finite State Machine


Send Class moving Request with the time
check Class_ID & Time
Class_ID unmatch
Re-enter Class_ID
Send request to get all the students information in that class
Getting all the students information
Done receiving student information
Compare students schedule to that time
Check for the finish number make sure it go trough every person
show all the conflicts
list it from high to low priority
print out

Verification Omega

Black Box Testing (35%)

Black Box testing, or functional testing, is based on the behavior of the program – what the program (or a piece of the program) is intended to do. Since Cucumber is the tool of choice for behavior testing in our class, Cucumbers would be prepared by us to be used to test the intended functionality of our program. Cucumber will be our main source of testing both the database side and front end side of this project. And once our tests are made, we can then make changes and additions to our existing product and we should know if our new edits affected the way our program is supposed to run.

Below are some sample Cucumbers:
Feature: Finding
Scenario: Finding conflicts.
Given the time "<input>", and the class ID "<input>"
When the system is run
Then the schedule conflicts should be "<output>"

Examples:
input	output
1:00 MWF:IT331	No conflicts
2:00 TR:CS212	Student1:IT152, Student3:BIBL101

When we input the class ID and the time we want to move the class to, the result should be that the conflicts after moving the class to the new time listed in descending order of priority are listed.

Scenario: Finding students.
Given the class ID "<input>"
When the system is run
Then all the students in the class should be "<output>".
Examples:
input	output
IT331	No students
CS212	Student1, Student3

When we input the class ID the result should be every students in the class.

Scenario: Finding schedule.
Given the time "<input>"
When the system is run
Then all the classes in that time slot should be "<output>".

Examples:

input	output
1:00 MWF	CS212
2:00 TR	None

When we input the time we want to move the class to, the result should be that every class in that time slot, if any.

Feature: Suggesting
Scenario: Finding conflicts.

Given the time "<input>" and the class ID "<input>"
When the system is run
Then the suggested schedule should be "<output>"

Examples:

input	output
1:00 MWF:IT212	3:00 TR
2:00 TR:CS352	2:00 TR

When we input the class ID and the time we want to move the class to that has previously detected conflicts, the result should be a suggested schedule. If input time and class has no conflicts, the suggested input would be the same as the input time.

Module Testing (15%)

Module testing, or unit testing, is testing a method to ensure that individual units of source code, or modules, are fit for use in the program. This allows us to detect problems earlier, as opposed to only detecting them once the modules have been integrated to the system. With this method, we will create tests to make sure that each line of our php, sql and code igniter files are being used. This way will have complete code coverage to be sure that every individual piece of our product is at least tested once before it gets put up for production.

Since module testing is a subset of White-box testing, the techniques detailed in the book to test the modules are as follows:

- Statement coverage, where we test that all parts of the code are executed, this encompasses:
  - Edge coverage: Test the flow of control of the program (generate control flow graph)
  - Condition coverage: Test the validity and correctness of conditions

- Path coverage: All paths of the control flow graph is traversed.

Integration Testing (15%)

Integration testing is ensuring that the system as a whole, and individual modules operate correctly with one another. There are many different approaches such as testing all the modules separately and then the whole system at once to find irregularities (big-bang testing), or incremental testing, when we test the integrating the modules as soon as a reasonable subset of the modules have been complete to provide some functionality.

We choose incremental testing, mainly because the number of modules we have is rather small, and that testing them as soon as some are ready seem intuitive in our situation. This allows us to localize error internally, and detect them more accurately. Integration can be done top-down or bottom-up. We choose to use a top-down approach because that is how we initially went about designing our system.

The way we are using this for our project specifically is by having test files for each individual module and controller of our PHP and SQL files. This way we can know exactly where our errors and bugs are located within our code so that we can handle the problem as quickly and efficiently as possible.

System Testing (10%)

System testing is where we finally test the entire system. After defining the expected behavior of our program under our specifications, we first test the system internally with dummy data, randomly generated, in our sandbox environment (Rackspace servers). Once that test has been passed, we will test our system in the field, with true Banner data. There are some different tests we can run on the system:

- Overload Testing: We will make sure that the transaction system will not be overloaded with requests. We will derive tests to ensure that the system will execute and respond in a reasonable amount of time.
- Robustness Testing: We will ensure that the system can handle unforeseen errors, as far as is reasonable. This includes invalid user input, power failure, and new versions of Banner data.

As of right now, we have the dummy test data working up to our expectations of how we believe our program should be running. Our next step is to test the entire system using the real banner data and hopefully it should run just the same as our test data and then we will have verified the entirety of our product is up to the project requirements.

Acceptance Testing (10%)

This stage is done along with the customer (Karen). We have created several scenarios using Cucumber to prepare for what we believe she would ask of us. We will conduct tests of her requests to prove the system’s integrity when our final product is completed and we believe it has met the requirements.

Performance Testing (5%)

This point was touched on briefly in the “Overload Testing” segment of our System Testing section. We will ensure that the system performs reasonably well, and we will do so by stress testing the system to find the worst-case performance. One of the ways we will stress test the system is by making a lot of requests to the server in an attempt to overload it, to detect the response time of our system in a worst-case scenario.

GOMS Testing (5%)

Goals, Operator, Methods and Selection are elementary actions of a user’s interaction with the system. Goals are what the user intends to accomplish, Operators are actions that are performed to achieve said goal, Methods are the sequence of Operators, and Selections describe the flow of a system. GOMS analysis will be done after beta-testing and conducting trials to analyze the human-computer interaction of our system.

Status Reports (5%)

We will further detail our work in preparing proper verification for our system in our weekly status report. Also recording the work done be each individual for that week and what our future plans are to get done the following week.