Team 5
CS 374
Dr. Brent Reeves

Verification Beta

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.

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.

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.

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.

Acceptance Testing (10%)

This stage is done along with the customer (Karen). We will conduct tests on her requests to prove the system's integrity.

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.

<u>Status Reports (5%)</u>

We will further detail our work in preparing proper verification for our system in our weekly status report.