Software Test Plan Documentation

For

Automated Intelligent Advisement Interface

Version 1.0

12/05/2016

Michael Harootoonyan

Megan Meyers

Sina Eradat

Jonathan Carrasco

Izzy Hasson

Brandon Garcia

Andrea Lee

Prepared for

COMP 380 Section 15900—Introduction to Software Engineering

Instructor: R. Lingard

Fall 2016

# Test Plan ID

CSUN\_ENR\_AI\_TP\_001

# Introduction

The document is the System Test Plan for the proposed AI2. AI2 is dynamic application for California State University, Northridge (CSUN) students that facilitates graduation by lowering the chance of human error. It does so by automating error-checking and making class-selection suggestions to ensure students do not inadvertently push back their graduation date. The application will be integrated with CSUN’s myNorthridge Portal and, by extension, the college’s SOLAR system. In addition, this application will be available via World Wide Web, and so may be reached at any time of day and from any location with access to the Internet.

AI2 services students with a myNorthridge Portal account, and possesses two interfaces. The first is for Admins, which may be accessed by college staff (for instance, for the purposes of fixing errors). The second is for Users, which can be accessed by any student. Because a myNorthridge Portal account is required, guests do not have access to this application.

This document will go over the unit, integration, and system testing used for the application. The test plan only covers the intended items for creating a basic prototype, with the assumption that additional testing will be done if approved.

* Test Items
* Available Courses
* Required Courses
* Schedule
* Generate
* Export
* Modify
* Major
* Compare Transcript to Major
* Choose Optimal Major (ensuring least amount of new classes needed as possible)
* Change Major

# References

This section specifies the references used in the creation of this document.

* Software Requirements Specification (SRS); Ver. 1.0
* Use-cases
* Software Design Document (SDD); Ver. 1.0
* High-level design
* Low-level design
* Prototypes
* Coding Guidelines provided by Professor Robert Lingard via Moodle

# Features to be Tested

This section specifies the features that will be tested as part of the test plan.

1. User Interface
   1. Planner
      1. Generate Planner
      2. Edit Planner
      3. Sort Planner
      4. Compare Transcript to Planner
   2. Schedule
      1. Generate Schedule
      2. Check Prerequisites Have Been Fulfilled
      3. Calculate Difficulty Rating

# Features Not to be Tested

This section specifies the features that will not be tested as part of the test plan, although they will be tested for at a later date.

1. Admin Interface
2. User Interface
   1. Change Major
      1. Optimize Change

# Test Approach

This section provides an overview to the approach used in the testing process.

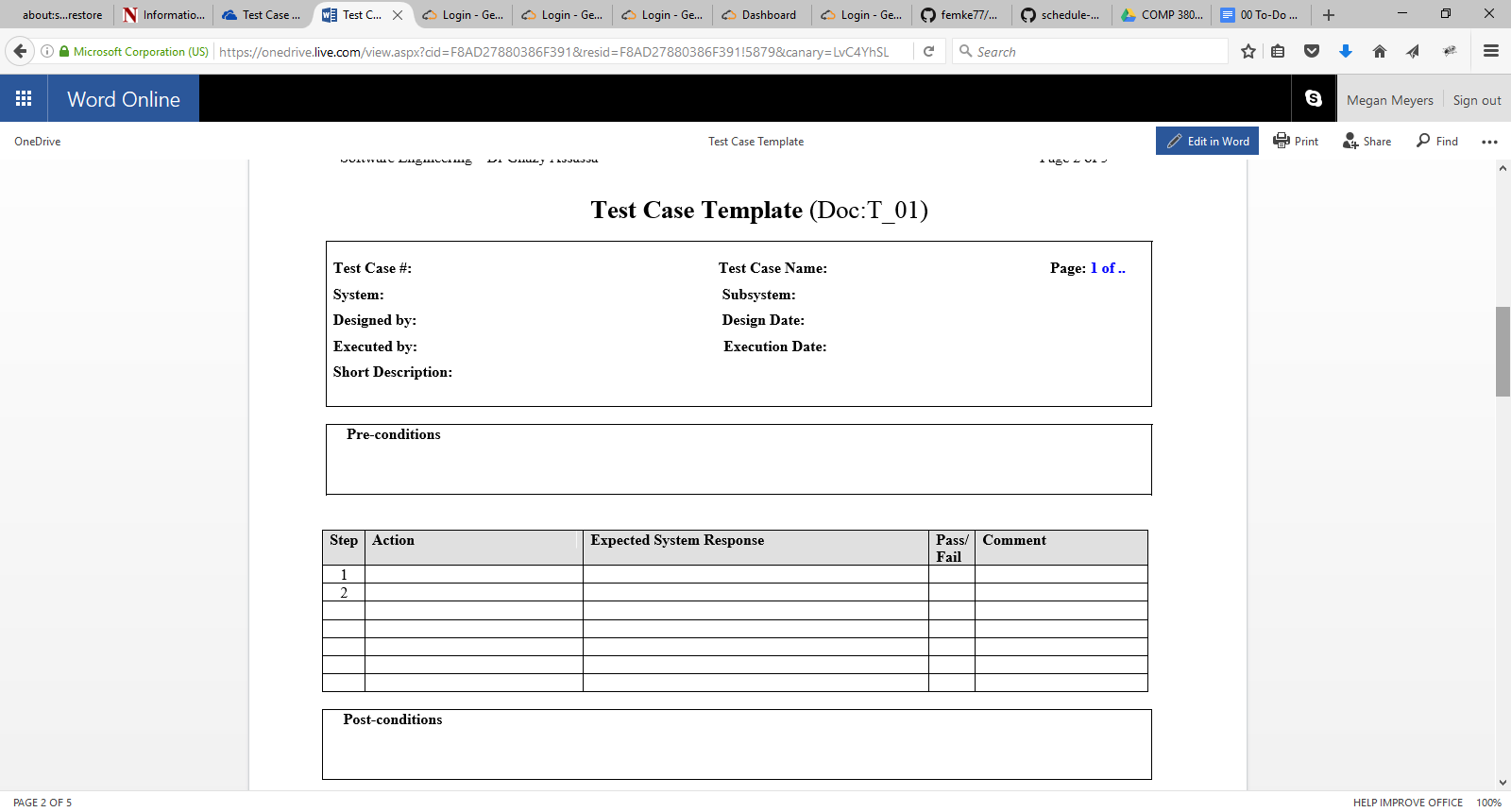
## Test Processes

The overall flow of the testing process shall be as follows:

1. Identify the requirements for testing and derive all test cases from the SRS and SDD.
2. Identify which tests will be used for unit, integration, and system testing, and which tests will best suit each module.
3. Review and verify all test cases and test data for both accuracy and adequacy.
4. Identify each test’s expected outcome.
5. Document the test data expected results on the approved test case template.
6. Run tests.
7. Document test case, data, and configuration for each test on the approved test case template.
8. Unsuccessful testing shall result in immediate correction of the defect or the return of unit/component to the development team.
9. All test documentation shall be submitted to the development team upon completion of tests.

## General Approach to Testing

The objective of testing is to verify and validate the functionality and requirements of AI2. **Figure 1** shows the test case template to be used for each test case.



**Figure 1. Test Case Template**

Unit testing will be performed during the development phase as the code is written by the programmer(s). Testing will be done as the units are created and developed. No tools requiring payment will be used due to budgetary constraints; instead, test code will be manually created by the individual using freely available Integrated Development Environments (IDE). Code-based testing techniques will be employed including code reading, inspection, and walkthrough. All units will be reviewed by a second party within the development team. Testing will be manual and include both complete statement and branch coverage. Databases accessed during testing will be known, controlled databases created by and belonging to the development team for test purposes only. Testing will be considered complete when each unit’s output is verified for correctness including error conditions.

Integration tests will be done by both unit and system testers. The unit test suite will be used as a regression during the integration of units and components. Integration test cases will be done bottom-up and will focus on the functional behavior as outlined in each use case found in the SRS and be directly developed from the SDD. Testing will be done manually and include both positive and negative tests and boundary conditions. Testing will be complete when each functional requirement use case is verified and validated to be functioning as per the requirements specifications.

System testing will include graphical user interface testing, regression testing, reliability testing, ad hoc testing, and usability testing. The tests will be done within the context of the SRS to ensure that all functional and non-functional requirement specifications are met and the system is functioning correctly. Black-box testing will be performed by users both inside and outside the development team. The actual databases this system will be accessing when live will now be incorporated into the testing. System testing will be complete when all functional validation testing is 100% successful, the GUI performs to specifications, all high-priority errors revealed in system testing are corrected, and medium and/or low-priority system errors are deemed within acceptable limits of risk by developers and test managers.

# Entry Criteria

This section species the requirements necessary for the team to begin testing.

* AI2 architectural design must be completed and approved by the Group 6 development team
* Developers must complete assigned components of architectural design, including a behavioral diagram and source code
* Source code must be completed and submitted for unit testing by another member of the Group 6 development team
* Test plan and the test cases must be reviewed and approved by the Group 6 development team
* Test case template and coding guidelines must be prepared and approved by the Group 6 development team
* Test environment must be prepared
* Test data should be available for all test cases
* Integration testing must be completed as a team
* AI2 functionality requirements should be cleared and approved by the Group 6 development team

# Exit Criteria

This section species the requirements necessary for the team to consider testing complete.

* All the test cases have been executed and passed, including regression testing, reliability testing, ad hoc testing, and usability testing
* AI2 functionality should be cleared and approved by the Group 6 development team
* All defects are fixed or closed
* Alpha testing period ends
* Deadline of December 5, 2016 has been reached

# Suspension Criteria

This section species the conditions under which testing will be placed on pause.

* AI2 architectural design is not completed and approved by the Group 6 development team
* AI2 functionality requirements are not cleared and approved by the Group 6 development team
* AI2 functionality requirements change substantially, and existing test cases or sample data no longer apply
* Test case(s) fail to pass Group 6 development team’s approval stage
* Integration testing fails to pass Group 6 development team’s approval stage

# Roles & Responsibilities

This section specifies the roles and responsibilities every member of the team holds.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sno** | **Name** | **Role** | **Responsibilities** | **Comments** |
| 1 | Michael Harootoonyan | Test Lead | Test planning, guidance, monitor and control  Test case documentation; defect reporting, tracking, and correcting, for Section class and Check Prerequisites methods. |  |
| 2 | Jonathan Carrasco | Senior Tester | Test data collection, generating test scenarios  Test case documentation; test execution; defect reporting, tracking, and correcting, for Major class |  |
| 3 | Megan Meyers | Tester | Test case documentation; test execution; defect reporting, tracking, and correcting, for Schedule class and Generate Schedule method |  |
| 4 | Sina Eradat | Tester | Test case documentation; test execution; defect reporting, tracking, and correcting, for Student class and Compare Transcript method |  |
| 5 | Brandon Garcia | Tester | Test case documentation; test execution; defect reporting, tracking, and correcting, for UI |  |
| 6 | Andrea Lee | Tester | Test case documentation; test execution; defect reporting, tracking, and correcting, for Planner class and Calculate Difficulty Rating method |  |
| 7 | Izzy Hasson | Tester | Test case documentation; test execution; Test execution; defect reporting, tracking, and correcting, for Course class and Generate Planner method |  |

# Schedule

This section species the schedule followed to complete testing, starting from 11/07/16 and ending at 12/03/16.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sno** | **Task** | **Days** | **Duration** | **Comments** |
| 1 | Understanding requirements and coding guidelines | 2 | 11/07/16 to 11/08/16 |  |
| 2 | Review meeting | 1 | 11/09/16 |  |
| 3 | Generate test scenarios  Create test activity diagram and code  Test data collection  Verify test environment | 12 | 11/10/16 to 11/21/16 |  |
| 4 | Create unit test cases | 1 | 11/22/16 |  |
| 5 | Create integration test cases | 5 | 11/23/16 to 11/27/16 |  |
| 6 | Regression testing | 1 | 11/28/16 |  |
| 7 | Create system test cases | 2 | 11/29/16 to 11/30/16 |  |
| 8 | Final regression testing | 2 | 12/01/16 to 12/02/16 |  |
| 9 | Evaluate exit criteria,  Collect all artifacts,  Test summary report | 1 | 12/03/16 |  |

# Training

This section species the testing each member of the team must go through in order to properly follow the test plan.

Training will encompass basic use of:

* Git and GitHub, including uploading, pushing, pulling, and merging to the main branch.
* Java Swing, including how to create a basic user interface
* General debugging test tools used in all industry standards, such as Integrated Development Environments (IDE). These include data management, ability to set breakpoints, ability to read information from a file, ability to analyze variables within a system, and basic Java exceptions and errors.

# Risks and Mitigations

This section species the possible risks to the plan, and how these risks can be mitigated.

|  |  |  |
| --- | --- | --- |
| **Risk** | **Probability** | **Mitigation** |
| Delay in testing due to design or development tasks running over schedule. | High | Test team can begin test design and preparation on any available units or components.  Schedule has a small buffer included. |
| Insufficient number of personnel needed for integration and system testing. | Med | Schedule can be adjusted and some extra workload can be distributed. May need to recruit more staff. |
| Defects found late in cycle. | Med | Defect management team in place to promptly address these issues. |
| Staff turnover. | Low | Testers will work in teams or pairs so that all test cycles are fully understood by more than one person and no down-time for training will be necessary.  Schedule can be adjusted. |
| Third party or independent testers or services are unavailable. | Low | Good communication with any third parties will allow for adjustments to workload and schedule for any unavailability. |
| Hardware used for testing can fail. | Low | Upload files to the cloud as they’re completed to ensure work is not tied to a specific hardware. |

# Test Environment/Lab

This section species the environment used in testing.

Hardware:

* Windows Operating System
* MacOs Operating System

Programming Language:

* Java

Software:

* Eclipse Integrated Development Environment
* NetBeans Integrated Development Environment

Tools:

* Javadoc
* GitHub

# Test Deliverables

This section specifies the deliverables to be prepared by the plan’s completion.

Test Plan

// RTM? (RTM is a matrix created at the start of project as it forms the basis of scope of testing. It is used to ensure that all the requirements are covered by mapping them to their associated test cases.)

Test Case Documents

Test Data

Test Summary Report

# Approvals

This section specifies the various stages of approval during the testing plan. The plan cannot continue until approval has been received.

|  |  |  |  |
| --- | --- | --- | --- |
| **Sno** | **Task** | **Author (Role)** | **Date & Signature** |
| 1 | Test Plan Documentation | Michael Harootoonyan (Test Lead) |  |
| 2 | Review | Andrea Lee (Senior Tester) |  |
| 3 | Approval | Brendan Garcia (Project Manager) |  |

# Coding Guidelines

The sample coding guidelines provided by Professor Robert Lingard have been used, with some simplifications. A copy has been provided below.

## Code Documentation

Use meaningful names for variables, constants, and functions. Names should be descriptive, but avoid unnecessarily long identifiers.

Each module should have a set of prologue comments preceding the actual code. The prologue should consist of: name of the module, its purpose (a brief description of the module), the author's name, and the creation date.

/\*

Name of Module: <enter name here with "sample" calling sequence>

Purpose: <enter a brief description here>

Author: <name of author>

Date: <original creation date>

\*/

The code should contain sufficient internal comments to make the code understandable. Comments should provide something extra and not just paraphrase the code. Descriptive comments should:

* Describe blocks of code, rather than commenting every line.
* Use blank lines or indentation so that comments can be readily distinguished from code.
* Be correct: an incorrect or misleading comment is worse than no comment at all.

## Data Declaration

All data items and data structures should be declared at the beginning of the module (or block) in which they are used.

## Statement Construction

Construct statements according to the following guidelines:

* Do not put multiple statements on a single line
* Use indentation to show program structure (and use it consistently)
* Avoid the use of complicated conditional tests
* Avoid heavy nesting of loops or conditions
* Use parentheses to clarity logical or arithmetic expressions
* Use spacing (e.g. blank lines) to clarify code
* Use only standard language features (if a standard exists)

# Test Cases

All test cases were run multiple times until they passed. Success criteria was based on whether or not the program provided the expected system response.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test Case #:** | 1 |  | **Test Case Name:** | Input semester and year |
| **System:** | AI^2 |  | **Subsystem:** | Planner |
| **Designed by:** | Michael Harootoonyan |  | **Design Date:** | 11/14/2016 |
| **Executed by:** | Michael Harootoonyan |  | **Execution Date:** | 11/14/2016 |
| **Short Description:** | Test the Planner's validation service | |  |  |
|  |  |  |  |  |
| **Pre-conditions** |  |  |  |  |
| The school only offers Spring, Summer, Fall, and Winter semesters | | |  |  |
| The system displays the main menu | |  |  |  |
|  |  |  |  |  |
| **Step** | **Action** | **Expected System Response** | **Pass/Fail** | **Comment** |
| 1 | Click the 'Choose Term' button | The system displays a message asking the user to enter the semester | Pass |  |
| 2 | Enter 'Spring' | The system displays a message asking the user to enter the year | Pass |  |
| 3 | Enter '2016' | The system displays the term and asks if the user to confirm | Pass |  |
| 4 | Click 'Confirm' | The system displays the courses for the term, as well as total difficulty rating and max units | Pass |  |
| 5 | **Check post-condition 1** |  | Pass |  |
| 6 | Repeat steps 1 and 2 using the 'Autumn' semester | The system displays an error and asks the user to try again | Pass |  |
| 7 | Enter a correct semester (e.g. 'Summer') | The system proceeds from 2 as usual | Pass |  |
| 8 | **Check post-condition 2** |  | Pass |  |
| 9 | Repeat steps 1, 2, and 4, and click 'Cancel' | The system exits and returns to the main myNorthridge Portal | Pass |  |
| 9 | **Check post-condition 3** |  | Pass |  |
|  |  |  |  |  |
| **Post-conditions** |  |  |  |  |
| 1 | The user has accessed their planner for a specific term | |  |  |
| 2 | The user has corrected faulty entries | |  |  |
| 3 | The user has returned to the main myNorthridge Portal | |  |  |

Defects Uncovered & Fixes Made

* The system initially assumed it would be correct and accepted any term. To fix it, entries were checked to ensure “Spring,” “Summer,” “Fall,” or “Winter” had been entered.
* Although entries were restricted, the term could technically be correct but have different casings (e.g. Spring, SPRING, spring, sPring) which would incorrectly be dubbed false. To fix it, all entries were automatically adjusted to be upper case.
* Incorrect entries would cause the program to stop, which was technically correct but not something we wanted the program to do. To fix it, the code was adjusted so that it would loop back to step 2 every time an incorrect entry was input.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test Case #:** | 2 |  | **Test Case Name:** | Udpate student planner |
| **System:** | AI^2 |  | **Subsystem:** | Student |
| **Designed by:** | Team 6 |  | **Design Date:** | 11/19/2016 |
| **Executed by:** | Michael Harootoonyan |  | **Execution Date:** | 11/20/2016 |
| **Short Description:** | Test that the Student class properly stores updated information. | |  |  |
|  |  |  |  |  |
| **Pre-conditions** |  |  |  |  |
| A student object exists and is open | |  |  |  |
| A planner unit has just been modified | |  |  |  |
|  |  |  |  |  |
| **Step** | **Action** | **Expected System Response** | **Pass/Fail** | **Comment** |
| 1 | Confirm changes to planner unit | The updated planner unit is saved to the planner, and the planner is sent to the student | Pass |  |
| 2 | Display student planner | The system outputs the student's planner, whose results reflect the changes saved in the step prior | Pass |  |
| 3 | **Check post-condition 1** |  | Pass |  |
|  |  |  |  |  |
| **Post-conditions** |  |  |  |  |
| 1 | The student container class contains the latest version of the planner | |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test Case #:** | 3 |  | **Test Case Name:** | Sort planner units |
| **System:** | AI^2 |  | **Subsystem:** | Planner |
| **Designed by:** | Michael Harootoonyan |  | **Design Date:** | 11/19/2016 |
| **Executed by:** | Michael Harootoonyan |  | **Execution Date:** | 11/20/2016 |
| **Short Description:** | Test that the orderChronologically function orders units correctly | |  |  |
|  |  |  |  |  |
| **Pre-conditions** |  |  |  |  |
| A planner class exists | |  |  |  |
| The tester can modify the code directly (e.g. using NetBeans without requiring an interface to access the code) | | | | |
|  |  |  |  |  |
| **Step** | **Action** | **Expected System Response** | **Pass/Fail** | **Comment** |
| 1 | Create planner unit for 'Spring' '2015' | The planner unit is added to the planner | Pass |  |
| 2 | Create planner unit for 'Fall' '2015' | The planner unit is added to the planner | Pass |  |
| 3 | Create planner unit for 'Spring' '2014' | The planner unit is added to the planner | Pass |  |
| 4 | Call orderChronologically | The planner organizes all its units to be listed from earliest to latest | Pass | Final order should be Spring 2014, Spring 2015, and Fall 2015 |
| 5 | **Check post-condition 1** |  | Pass |  |
|  |  |  |  |  |
| **Post-conditions** |  |  |  |  |
| 1 | The planner's units are stored chronologically, so that pulling the first unit is guaranteed to be the earliest. | | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test Case #:** | 4 |  | **Test Case Name:** | Topological sort |
| **System:** | AI^2 |  | **Subsystem:** | AI^2 |
| **Designed by:** | Izzy Hasson |  | **Design Date:** | 11/20/2016 |
| **Executed by:** | Izzy Hasson |  | **Execution Date:** | 11/20/2016 |
| **Short Description:** | Test the topological sort, which lists courses by number of prerequisites | |  |  |
|  |  |  |  |  |
| **Pre-conditions** |  |  |  |  |
| Sample list of courses available, stored as an ArrayList | | |  |  |
| Sample course table with prerequisite information available | | |  |  |
| The tester can modify the code directly (e.g. using NetBeans without requiring an interface to access the code) | | | | |
|  |  |  |  |  |
| **Step** | **Action** | **Expected System Response** | **Pass/Fail** | **Comment** |
| 1 | Call topologicalSort(ArrayList<Course> list, CourseTable gCT) | System sorts the courses according to prerequisites and returns a new sorted ArrayList | Pass |  |
| 2 | **Check post-condition 1** |  | Pass |  |
|  |  |  |  |  |
| **Post-conditions** |  |  |  |  |
| 1 | The sample courses have been sorted and returned as an ArrayList | |  |  |

Defects Uncovered & Fixes Made

* The course ordering came out wrong during the first run. Initially, the code counted the number of courses in the graph that listed the course in question as a prerequisite (e.g. if it was at COMP122, it would count how many courses required COMP122 as a prerequisite). To fix it, the code was changed to count the number of prerequisites the course had left in the graph instead.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test Case #:** | 5.1 |  | **Test Case Name:** | Get courses |
| **System:** | AI^2 |  | **Subsystem:** | Schedule |
| **Designed by:** | Andrea Lee |  | **Design Date:** | 11/20/2016 |
| **Executed by:** | Andrea Lee |  | **Execution Date:** | 11/25/2016 |
| **Short Description:** | Test that the schedule correctly pulls courses from the earliest planner unit available for a student | | | |
|  |  |  |  |  |
| **Pre-conditions** |  |  |  |  |
| Planner contains at least one planner unit | |  |  |  |
| Planner unit contains at least one course | |  |  |  |
| The tester can modify the code directly (e.g. using NetBeans without requiring an interface to access the code) | | | | |
|  |  |  |  |  |
| **Step** | **Action** | **Expected System Response** | **Pass/Fail** | **Comment** |
| 1 | Run getCourses() | The system accesses the plannerUnit as the head of the planner linked list. All courses contained in plannerUnit are moved to a schedule list. The plannerUnit is removed from the planner. | Pass |  |
| 2 | Display planner units | All planner units are shown. The (original) first unit no longer displays. | Pass |  |
| 3 | Display schedule's courses | The schedule lists the courses that were initially in the first PlannerUnit | Pass |  |
| 4 | **Check post-condition 1** |  | Pass |  |
|  |  |  |  |  |
| **Post-conditions** |  |  |  |  |
| 1 | The schedule contains the courses listed in the planner's first unit; the first unit is removed from the planner. | | | |

Defects Uncovered & Fixes Made

* Initially, the planner still possessed all planner units even after the schedule pulled one out. However, after a term passed, it would no longer be relevant to the planner, as all future planner calculations (such as checking prerequisites) would be based on the student’s transcript rather the planner unit—especially in instances where a student was unable to follow the courses in the planner unit, whether because they chose to ignore it or no sections were available. To fix it, the method was adjusted to delete the planner unit after its courses were pulled.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test Case #:** | 5.2 |  | **Test Case Name:** | Assign sections |
| **System:** | AI^2 |  | **Subsystem:** | Schedule |
| **Designed by:** | Andrea Lee |  | **Design Date:** | 11/20/2016 |
| **Executed by:** | Andrea Lee |  | **Execution Date:** | 11/25/2016 |
| **Short Description:** | Test that the courses stored in the schedule object is matched to available sections | | |  |
|  |  |  |  |  |
| **Pre-conditions** |  |  |  |  |
| The Schedule object contains at least one course | |  |  |  |
| A list of section objects matching the courses listed are available | | |  |  |
| The tester can modify the code directly (e.g. using NetBeans without requiring an interface to access the code) | | | | |
|  |  |  |  |  |
| **Step** | **Action** | **Expected System Response** | **Pass/Fail** | **Comment** |
| 1 | Call courseToSection(ArrayList<Section> section, ArrayList<Course> course), where section = sample section data, and the course list contains COMP380, COMP500, COMP450, and COMP600 | The system creates a list of all sections available that match the courses in the schedule’s course list, matching the semester and year stored in the Schedule object | Pass | Sample section data includes sections for all courses. |
| 2 | **Check post-condition 1** |  | Pass |  |
| 3 | Call courseToSection(ArrayList<Section> section, ArrayList<Course> course) | Same as step 1. In addition, the system outputs a notification stating that a course is not being offered this semester | Pass | Same section data as above, except all sections for COMP600 has been removed. |
| 4 | **Check post-condition 2** |  | Pass |  |
|  |  |  |  |  |
| **Post-conditions** |  |  |  |  |
| 1 | The system creates a list of all available sections for a particular course in a particular term | | |  |
| 2 | Same as 1, and the system notifies the user that a course is not being offered in that particular term | | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test Case #:** | 5.3 |  | **Test Case Name:** | Collides |
| **System:** | AI^2 |  | **Subsystem:** | Schedule |
| **Designed by:** | Megan Meyers |  | **Design Date:** | 11/20/2016 |
| **Executed by:** | Megan Meyers |  | **Execution Date:** | 11/25/2016 |
| **Short Description:** | Test that the boolean returns true if two sections have overlapping days and times | | |  |
|  |  |  |  |  |
| **Pre-conditions** |  |  |  |  |
| There are at least two sample sections to compare | |  |  |  |
| The tester can modify the code directly (e.g. using NetBeans without requiring an interface to access the code) | | | | |
|  |  |  |  |  |
| **Step** | **Action** | **Expected System Response** | **Pass/Fail** | **Comment** |
| 1 | Call collides(Section e), where the section 15522 calls the function and e = 15920 | System returns true | Pass | 15522 is TuTh starting at 8:30; 15920 is Fr |
| 2 | **Check post-condition 1** |  | Pass |  |
| 3 | Call collides(Section e), where the section 15522 calls the function and e = 15523 | System returns false | Pass | Both sections are TuTh starting at 8:30 |
| 4 | **Check post-condition 2** |  | Pass |  |
|  |  |  |  |  |
| **Post-conditions** |  |  |  |  |
| 1 | System returns true, as there is no overlap | |  |  |
| 2 | System returns false, as there is overlap | |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test Case #:** | 5.4 |  | **Test Case Name:** | No Friday Filter |
| **System:** | AI^2 |  | **Subsystem:** | Schedule |
| **Designed by:** | Megan Meyers |  | **Design Date:** | 11/20/2016 |
| **Executed by:** | Megan Meyers |  | **Execution Date:** | 11/25/2016 |
| **Short Description:** | Filters out all sections that meet on Friday | |  |  |
|  |  |  |  |  |
| **Pre-conditions** |  |  |  |  |
| There is a list of available sections | |  |  |  |
| User has checked the box for “Filter by day – Friday” | | |  |  |
|  |  |  |  |  |
| **Step** | **Action** | **Expected System Response** | **Pass/Fail** | **Comment** |
| 1 | Call noFridayFilter(ArrayList<Section> list) | System iterates over section list and remove all objects with Friday class meetings | Pass |  |
| 2 | **Check post-condition 1** |  | Pass |  |
| 3 | Call noFridayFilter(ArrayList<Section> list) | Same as step 1. In addition, the user is notified that no sections remain | Pass | Same sample data as 1, except all sections other than Friday ones were omitted |
| 4 | **Check post-condition 2** |  | Pass |  |
|  |  |  |  |  |
| **Post-conditions** |  |  |  |  |
| 1 | System returns list with no Friday sections | |  |  |
| 2 | System empties list and notifies the user that no sections remain | |  |  |

Defects Uncovered & Fixes Made

* Initially, the warning notification (stating that no sections remained for a course) would display every time a section object was filtered out, or removed. Because of this, it meant the warning displayed even if one or more sections in a course still remained. To fix it, a second hashmap was created which took in all sections != Friday. Using !hashmap.containsKey(), the program was corrected so that only one notification displayed.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test Case #:** | 6.1 |  | **Test Case Name:** | Check passed courses |
| **System:** | AI^2 |  | **Subsystem:** | Schedule |
| **Designed by:** | Sina Eradat |  | **Design Date:** | 12/1/2016 |
| **Executed by:** | Sina Eradat |  | **Execution Date:** | 12/1/2016 |
| **Short Description:** | Tests that a function returns true if a student has taken and passed a course | | |  |
|  |  |  |  |  |
| **Pre-conditions** |  |  |  |  |
| A student object exists and is open | |  |  |  |
| The student object possesses an up-to-date transcript list containing course objects to be evaluated | | | |  |
|  |  |  |  |  |
| **Step** | **Action** | **Expected System Response** | **Pass/Fail** | **Comment** |
| 1 | Call coursesPassed(Course i), where i = COMP182 | System returns true when course is found in the student's transcript, and the student has passed | Pass | Sample student is a Computer Science major who has taken COMP182, failed COMP282, and has not taken MATH450. |
|  | **Check postcondition 1** |  | Pass |  |
| 2 | Call coursesPassed(Course i), where i = COMP282 | System returns false when the course is found in the student's transcript but the student has failed | Pass |  |
| 3 | **Check postcondition 2** |  | Pass |  |
| 4 | Call coursesPassed(Course i), where i = MATH450 | System returns false if the course is not found in the student's transcript | Pass |  |
| 5 | **Check postcondition 3** |  | Pass |  |
|  |  |  |  |  |
| **Post-conditions** |  |  |  |  |
| 1 | The function confirms the student has taken COMP182 | |  |  |
| 2 | The function confirms the student has failed COMP282 | |  |  |
| 3 | The function confirms the student has not taken MATH450 | |  |  |

Defects Uncovered & Fixes Made

* Initially, the method would return true if the student took the course, even if they did not receive a passing grade. To fix it, the method was adjusted to read the student’s grade as well.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test Case #:** | 6.2 |  | **Test Case Name:** | Check met prerequisites |
| **System:** | AI^2 |  | **Subsystem:** | Schedule |
| **Designed by:** | Team 6 |  | **Design Date:** | 12/1/2016 |
| **Executed by:** | Sina Eradat |  | **Execution Date:** | 12/1/2016 |
| **Short Description:** | Tests that a student has met all prerequisites | |  |  |
|  |  |  |  |  |
| **Pre-conditions** |  |  |  |  |
| A student object exists and is open | |  |  |  |
| The student object possesses an up-to-date transcript list containing course objects to be evaluated | | | |  |
| A course object with prerequisites exists | |  |  |  |
|  |  |  |  |  |
| **Step** | **Action** | **Expected System Response** | **Pass/Fail** | **Comment** |
| 1 | Call checkPrerequisites(Course i), where i = CourseA | System returns true | Pass | Sample student is a Computer Science major who has taken COMP182, failed COMP282, and has not taken MATH450.  Sample CourseA has a prerequisite of COMP182. |
|  | **Check postcondition 1** |  | Pass |  |
| 2 | Call checkPrerequisites(Course i), where i = CourseB | System returns false | Pass | Sample CourseB has a prerequisite of COMP182 and COMP282. |
| 3 | **Check postcondition 2** |  | Pass |  |
| 4 | Call checkPrerequisites(Course i), where i = CourseC | System returns false | Pass | Sample CourseC has a prerequisite of IS451. |
| 5 | **Check postcondition 3** |  | Pass |  |
|  |  |  |  |  |
| **Post-conditions** |  |  |  |  |
| 1 | The function confirms the student has met the prerequisites for CourseA | | |  |
| 2 | The function confirms the student has not met the prerequisites for CourseB | | |  |
| 3 | The function confirms the student has not met the prerequisites for CourseC | | |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test Case #:** | 7 |  | **Test Case Name:** | Calculate difficulty |
| **System:** | AI^2 |  | **Subsystem:** | Difficulty Rating |
| **Designed by:** | Brandon Garcia |  | **Design Date:** | 11/14/2016 |
| **Executed by:** | Brandon Garcia |  | **Execution Date:** | 11/20/2016 |
| **Short Description:** | Test that the difficultyRating function calculates values correctly | |  |  |
|  |  |  |  |  |
| **Pre-conditions** |  |  |  |  |
| Courses offered by CSUN are listed in the sample data | | |  |  |
| The tester can modify the code directly (e.g. using NetBeans without requiring an interface to access the code) | | | | |
|  |  |  |  |  |
| **Step** | **Action** | **Expected System Response** | **Pass/Fail** | **Comment** |
| 1 | Set sample data files to '1hoursSurvey' and '1gradeDistribution' | The IDE compiles | Pass | Data file 1 possesses at least one data point for the COMP182 course |
| 2 | Call calculateDifficultyRating function for the course 'COMP182' | The system sums the total hours spent studying per week in the specified class, counts the applicable number of data points, and displays the numbers. The same applies for grades (where 'A' is considered 4, 'B' is considered 3, and so forth). At the end, the rating is output. | Pass | Rating is calculated by average hours/average grade [points]. |
| 3 | **Check post-condition 1** |  | Pass |  |
| 4 | Repeat steps 1 and 2 using the '2hoursSurvey' and '2gradeDistribution' data files | The IDE compiles | Pass | Data file 2 possesses no data points for the COMP182 course |
| 5 | Call calculateDifficultyRating function for the course 'COMP182' | The system sums and counts as applicable; however, it uses the specified default values | Pass |  |
| 6 | **Check post-condition 2** |  | Pass |  |
|  |  |  |  |  |
| **Post-conditions** |  |  |  |  |
| 1 | The difficulty rating is returned using the provided data | |  |  |
| 2 | The difficulty rating is returned using the default data | |  |  |

Defects Uncovered & Fixes Made

* The IDE did not compile initially. To fix it, the function was modified to throw an IOException.
* Incorrect answers were being calculated (no decimal places). To fix it, data types for variables used to calculate the final output were changed from Integer to Double.
* Incorrect answers were still being calculated (decimals ended with a .0). To fix it, calculations using a Double variable were multiplied by 1.0 to ensure proper type casting.
* Difficulty rating of 0 returned if the course had no data points; while the calculation was correct, it was not something we wanted to keep. To fix it, default values were added to the code.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test Case #:** | 8 |  | **Test Case Name:** | Log section data |
| **System:** | AI^2 |  | **Subsystem:** | Section |
| **Designed by:** | Jonathan Carrasco |  | **Design Date:** | 11/20/2016 |
| **Executed by:** | Jonathan Carrasco |  | **Execution Date:** | 11/20/2016 |
| **Short Description:** | Test reading and storing sample section data | |  |  |
|  |  |  |  |  |
| **Pre-conditions** |  |  |  |  |
| Sample list of sections available | |  |  |  |
| The tester can modify the code directly (e.g. using NetBeans without requiring an interface to access the code) | | | | |
|  |  |  |  |  |
| **Step** | **Action** | **Expected System Response** | **Pass/Fail** | **Comment** |
| 1 | Set sample data file to 'sampleSections' | The IDE compiles | Pass | sampleSections is delimited by commas |
| 2 | Call logSections() | The system creates a section object, reads comma delimited lines, and stores information inside the object. It does so for each line. | Pass |  |
| 3 | Call displaySections() | The system prints out a list of every section | Pass |  |
| 4 | **Check post-condition 1** |  | Pass |  |
|  |  |  |  |  |
| **Post-conditions** |  |  |  |  |
| 1 | The sample sections have been stored in section objects | |  |  |

# Appendix

All source code has been provided in a separate ZIP file, in two options:

* The DirectSource folder provides them as a collection of .java and .png files
* The NetBeans folder provides them directly from a NetBeans export