***SOFTWARE TEST SPECIFICATION***

***Version 1.0***

***By: Christen Ford***

**1. Introduction**

This testing document outlines the goals and objects, scope, and any major constraints placed on the testing process. It also details the strategies used for unit and integration testing as well as validation testing. A comprehensive testing procedure is provided that lists the components covered by the test plan as well as how they will be tested. This information is summarized in a table that explicitly states every set of test cases for each component. Validation testing is also covered by this test plan. This testing will be conducted using both black box and white box testing methodologies. There will also be extensive testing of the front-end of the software.

**1.1 Goals and Objectives**

The goal of the testing process specified here is to guarantee the system functions as specified and caters to the needs of the users of the software. This will be accomplished by providing as close to 100% code coverage as possible. This process will be automated as much as possible because I am the only person developing the software as well as the sole person responsible for testing it.

**1.2 Statement of Scope**

The testing process is meant to ensure that the components that make up this system provide the functionality specified in the system requirements specification as well as the functionality expected by the users of the software. However, the primary purpose of the testing process as it applies to this software is to root out any bugs or defects in the software as soon as possible.

Optimally, this document would provide 100% code coverage, testing all the functionality provided by the software’s implementation. However, this is infeasible as at the time of writing this document, as only a portion of the software’s functionality has currently been implemented. Test cases will be provided for each requirement as stated in the system requirements specification, however requirements that have already been implemented may receive more specific test cases than requirements that are not completed.

That said, the testing performed in this document will cover both unit and integration tests. Both types of testing are covered in more detail later in this document. The primary motivation is to ensure that no unexpected behavior arises in any of the components in the software or during their interactions.

In addition, a testing log will be kept (specified in a separate document) that keeps track of the success of the test cases detailed in this document. Test case attempts will be specified with a date, time, test case identifier, input, expected output, actual output, and result of the test.

**1.3 Major Constraints**

This software is user-centric. To completely test this software, I would need to employ user testing to do so. I do not know anyone that would be willing to conduct user testing of this software and even if I did it would be difficult for me to do so.

This software utilizes a web front-end as a means of interacting with the software. Normally, this would make user testing easy, as I would just have to create test accounts for users on the web site and direct them to it. This approach is infeasible for reasons discussed below.

Threading and concurrency control mechanisms have been built into the backend to handle requests from multiple users. Also caching has been built in to ensure minimum access to the database. The SQL server express database that provides data persistence can be easily migrated to a server using Microsoft SQL Server software or a Microsoft Azure web instance, but I do not believe it is worth the time or cost to do so now.

I do not have a server of my own to host this software on for user testing, nor do I want potential users to host the software on their own machines to conduct user testing. This compromise means that the unit and integration testing I perform must be as thorough as possible.

**2. Test Plan**

This section describes the overall testing strategy and the project management issues that are required to properly execute effective tests.

**2.1 Unit Testing**

Unit testing will be conducted using the built-in unit testing framework inside Visual Studio. This process will be expedited using NUnit which integrates directly into Visual Studio’s testing framework. NUnit is a clone of JUnit with all of the same functionality but is intended for use in testing .NET managed code, primarily C# and Visual Basic.

The goal is for each component within the system to be tested fully. However, this may be impossible with the current implementation of the software. Certain components feature additional functionality that were included to make the software more extensible in the future as new features are added to the front-end web site. Testing this functionality may be possible within the bounds of unit testing, however the front end lacks the proper mechanisms to test the components from the perspective of an end user.

The components to be tested include:

* All data bearing classes.
  + Building: Stores building information.
  + Course: Stores course information.
  + Department: Stores department information.
  + Major: Stores major information.
  + School: Stores school information.
  + Section: Stores section information.
  + User: Stores user information.
* The cache management classes.
  + MemoryCache: Thread-safe, memory-based cache that implements reading, writing, deleting, and updating. Also implements a page notification mechanism that allows subscribers to intercept items as they are paged out of cache.
  + CacheItem: Key/value pair that is stored inside the cache.
* Selected functionality from the data management classes.
  + CourseManager: Provides interaction for courses and sections between the front-end graphical user interface and the database manager.
  + DatabaseManager: Provides interaction with the logical layer manager classes and the database. Explicitly contains only database logic such as SQL bearing functions.
  + UniversityManager: Provides interaction for buildings, departments, and schools between the front-end graphical user interface and the database manager.
  + UserManager: Provides interactions for the user class between the front-end graphical user interface and the database manager. Also provides authentication and session management functionality.

**2.2 Integration Testing**

As defined by the software design specification, this software utilizes a layered architecture. This architecture connects the graphical user interface in the presentation layer to the data management classes in the logical layer to the database management class in the database layer. Each layer will be considered fully tested once each component that comprises it passes all unit tests.

Integration testing will be performed that tests the interactions between directly interacting layers. Front-end integration testing will take place between the interactions in the presentation and logical layer. While back‑end integration testing will take place between the logical layer and the database layer. No integration testing will take place between the presentation and database layers, as the successful completion of the previous two integration test suites form a transitive relationship that ensures the database layer will interact correctly with the presentation layer.

The front-end test integration test suite will use mock data and testing stubs that replicate functionality from the database layer. The back-end integration test suite will use live data from the database, as the functionality in the database layer handles SQL query requests to the database.

**2.3 Validation Testing**

This section states the requirements to be tested.

(REQ) Functional requirements that will be tested include:

**2.3.1** The system shall have three distinct roles for users: Student, Instructor, and Administrator.

**2.3.2** The system shall allow users to have only one role.

**2.3.3** The system shall allow students to enroll in unarchived courses that have at least one (1) not full section they meet the prerequisites for.

**2.3.4** The system shall allow students to unenroll from course sections they are currently enrolled in.

**2.3.5** The system shall allow students to view their schedule.

**2.3.6** The system shall allow students to view their academic record.

**2.3.7** The system shall allow instructors to view the courses they are scheduled to teach.

**2.3.8** The system shall allow instructors to view the number of students who are signed up for their courses.

**2.3.9** The system shall allow instructors to view a student’s record only if they have an advisor-advisee relationship.

**2.3.10** The system shall allow instructors to view a student’s schedule only if they have an advisor-advisee relationship.

**2.3.11** The system shall allow administrators to create new courses in the course catalog.

**2.3.12** The system shall allow administrators to remove courses from the course catalog.

**2.3.13** The system shall allow administrators to modify courses in the course catalog.

**2.3.14** The system shall allow administrators to archive courses in the course catalog.

**2.3.15** The system shall allow administrators to unarchive courses in the course catalog.

**2.3.16** The system shall allow administrators to create new users.

**2.3.17** The system shall not allow anyone other than administrators to create new users.

**2.3.18** The system shall not allow users to change their role.

**2.3.19** The system shall allow all users to view the course catalog.

**2.3.20** The system shall require all users to login with a university identifier and password.

**2.3.21** The system shall allow users to change their password.

**2.3.22** The system shall allow users to logout of the system.

**2.3.23** The system shall allow administrators to remove a course section.

**2.3.24** The system shall allow administrators to add a course section.

**2.3.25** The system shall allow administrators to modify a course section.

(NFREQ) Non-functional requirements that will be tested include:

**2.3.1** The system shall require all users to be associated with a first name, last name, university identification number, email address, and a role.

**2.3.2** The system shall require students to be associated with a major field of study.

**2.3.3** The system shall validate the user’s university identification number and password prior to allowing them to login to the system.

**2.3.4** The system shall allow students to only register for eighteen (18) credit hours maximum per term.

**2.3.5** The system shall register students for courses within three (3) seconds after the student submits their course registration information.

**2.3.6** The system shall require passwords to be at least eight (8) characters in length.

**2.3.7** The system shall not store the user’s password in plain text.

**2.3.8** The system shall hash and salt the user’s password before storing it in the user data store.

**2.3.9** The system shall hash and salt the user’s password before validating it with the password stored in the user data store.

**2.3.10** The system shall use the same hash and salt mechanism for storing the user’s password and validating the user’s password.

**2.3.11** The system shall require all courses to be associated with a department, name, description, and number of credit hours.

**2.3.12** The system shall require all course sections to be associated with dates and times offered as well as at least one (1) instructor.

**2.3.13** The system shall not allow course section start times to be later than course section end times.

**2.3.14** The system shall require course section start times and course section end times to differ by at least 50 minutes.

**2.3.15** The system shall display the section number in the format SO‑### where ### is the section number.

**2.3.16** The system shall display the course number in the format [Department Abbreviation]‑### where ### is the course number.

**2.3.17** The system shall not allow a class to be a prerequisite of itself.

**2.3.18** The system shall require all university identification numbers to be unique.

**2.3.19** The system shall not allow section numbers to be less one (1), or greater than 999.

**3. Test Procedure**

The testing procedure will be largely automated for unit tests. This testing utilizes the NUnit automated testing framework for .NET applications. NUnit integrates seamlessly into Visual Studio’s testing framework and utilizes similar syntax to Java’s JUnit.

The components in the system are too large to test entirely by hand. Many of the functionalities promised by the software will be tested as they are implemented utilizing the GUI front-end. At this point it is too time consuming to implement software tests that test the integration of the components.

I have outlined all of the test cases for integration and system testing that I will utilizes to determine whether the system functions as a whole. However, as stated the primary testing mechanism will simply be myself interacting with the system from all three user types. If the system behaves correctly for all of the functionality promised of the three user types, then I will assume that it is functioning correctly.

Specific emphasis will be placed on tests that involve inputting data into the system. This will ensure that the database cannot be tampered with from the web site. It will also ensure that the system does not accept bad input that could potentially corrupt the rest of the system.

**3.1 Software Components to be Tested**

Components in this section are listed from data holding units (classes that exists just to hold data) to data manipulation components (components that allow interaction between the front-end and back-end) i.e. units to components. Note that all unit-level, data holding classes have their testing fully automated by NUnit (and are suffixed by a (N)).

**3.1.1 Building (N)**

This component is responsible for holding information relating to a building.

**3.1.1.1 Testing Procedure**

Testing of this component is fully automated using NUnit.

**3.1.1.2** **Unit Test Cases**

All test cases utilized by NUnit are listed below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test ID | Name | Purpose | Input | Expected Output |
| BLD001 | Test Equals | Asserts that two buildings are equal. | Building(2, “Stillwell Hall”, “SH”), Building(2, “Rockwell Row”, “RR”) | True |
| BLD002 | Test Not Equals | Asserts that two buildings are not equal. | Building(0, “Chadwick Hall”, “CH”), Building(1, “Fenwick Hall”, “FH”) | False |
| BLD003 | Test Get Hash Code | Asserts that a building generates the correct hash code. | Building(0, “Chadwick Hall”, “CH”) | 0 |
| BLD004 | Test Compare to Less Than | Asserts that one buildings name alphabetically comes before another. | Building(0, “Chadwick Hall”, “CH”), Building(1, “Fenwick Hall”, “FH”) | < 0 |
| BLD005 | Test Compare to Equals | Asserts that one buildings name is alphabetically the same as another. | Building(2, “Stillwell Hall”, “SH”), Building(3, “Stillwell Hall”, “SH”) | == 0 |
| BLD006 | Test Compare to Greater Than | Asserts that one buildings name is alphabetically greater than the others. | Building(2, “Stillwell Hall”, “SH”), Building(0, “Chadwick Hall”, “CH”) | > 0 |

**3.1.2 CourseTest (N)**

This component is responsible for holding information relating to a course.

**3.1.2.1 Testing Procedure**

Testing of this component is fully automated using NUnit.

**3.1.2.2** **Unit Test Cases**

All test cases utilized by NUnit are listed below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test ID | Name | Purpose | Input | Expected Output |
| COS001 | Test Equals | Asserts that two courses are equal. | Course(0, “Intro to Programming”), Course(0, “Artificial Intelligence”) | True |
| COS002 | Test Not Equals | Asserts that two courses are not equal. | Course(0, “Intro to Programming”), Course(1, “Computer Networks”) | False |
| COS003 | Test Get Hash Code | Asserts that a course generates the correct hash code. | Course(0, “Intro to Programming”) | 0 |
| COS004 | Test Compare to Less Than | Asserts that one courses name alphabetically comes before another. | Course(1, “Computer Networks”), Course(0, “Intro to Programming”) | < 0 |
| COS005 | Test Compare to Equals | Asserts that one courses name is alphabetically the same as another. | Course(0, “Intro to Programming”), Course(3, “Intro to Programming”) | == 0 |
| COS006 | Test Compare to Greater Than | Asserts that one courses name is alphabetically greater than the others. | Course(4, “Graphics Design”), Course(0, “Intro to Programming”) | > 0 |
| COS007 | Test Add Section | Asserts that a section can be added to a course. | {Course(0, “Intro To Programming”), 0, Sections:{0}}, 0 | Has Section(0) return True |
| COS008 | Test Remove Section | Asserts that a section can be removed from a course. | {Course(0, “Intro to Programming”), 0, Sections:{}}, 0 | HasSection(0) returns False |
| COS009 | Test Has Section | Asserts that a course can state that it has a section based on the section’s identifier. | {Course(0, “Intro to Programming”), Sections:{1}}, 1 | True |
| COS010 | Test Get Sections | Asserts that a course can provide an iterable for its sections. | Course(0, “Intro to Programming”) | An iterable containing the set of {0, 1}. |

**3.1.3 DepartmentTest (N)**

This component is responsible for holding information relating to a department.

**3.1.3.1 Testing Procedure**

Testing of this component is fully automated using NUnit.

**3.1.3.2** **Unit Test Cases**

All test cases utilized by NUnit are listed below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test ID | Name | Purpose | Input | Expected Output |
| DEP001 | Test Equals | Asserts that two departments are equal. | Department(0, “Computer Science”), Department(0, “Engineering”) | True |
| DEP002 | Test Not Equals | Asserts that two departments are not equal. | Department(0, “Computer Science”), Department(1, “Engineering”) | False |
| DEP003 | Test Get Hash Code | Asserts that a department generates the correct hash code. | Department(0, “Computer Science”) | 0 |
| DEP004 | Test Compare to Less Than | Asserts that one departments name alphabetically comes before another. | Department(0, “Computer Science”), Department(1, ”Engineering”) | < 0 |
| DEP005 | Test Compare to Equals | Asserts that one departments name is alphabetically the same as another. | Department(0, ”Engineering”), Department(1, “Engineering”) | == 0 |
| DEP006 | Test Compare to Greater Than | Asserts that one departments name is alphabetically greater than the others. | Department(1, ”Engineering”), Department(0, “Computer Science”) | > 0 |
| DEP007 | Test Add Course | Asserts that a course can be added to a department. | {Department(0, “Computer Science”), Courses:{}}, 0 | HasCourse(0) returns True |
| DEP008 | Test Remove Course | Asserts that a course can be removed from a department. | {Department(0, “Computer Science”), Courses:{0}}, 0 | HasCourse(0) return False |
| DEP009 | Test Has Course | Asserts that a department can state that it has a course based on the course identifier. | {Department(0, “Computer Science”), Courses:{0}}, 0 | True |
| DEP010 | Test Get Courses | Asserts that a department can provide an iterable over the department’s courses. | {Department(0, “Computer Science”), Courses:{0, 1, 2}} | Collection containing the set of {0, 1, 2} |

**3.1.4 MajorTest (N)**

This component is responsible for holding information relating to a major.

**3.1.4.1 Testing Procedure**

Testing of this component is fully automated using NUnit.

**3.1.4.2** **Unit Test Cases**

All test cases utilized by NUnit are listed below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test ID | Name | Purpose | Input | Expected Output |
| MAJ001 | Test Equals | Asserts that two majors are equal. | Major(0, “Computer Science”), Major(0, “Computer Information Systems”) | True |
| MAJ002 | Test Not Equals | Asserts that two majors are not equal. | Major(0, “Computer Science”), Major(1, “Computer Information Systems”) | False |
| MAJ003 | Test Get Hash Code | Asserts that a major generates the correct hash code. | Major(0, “Computer Science”) | 0 |
| MAJ004 | Test Compare to Less Than | Asserts that one majors name is alphabetically less than another majors name. | Major(1, “Computer Information Systems”), Major(0, “Computer Science”) | < 0 |
| MAJ005 | Test Compare to Equals | Asserts that one major’s name is alphabetically the equal to another major’s name. | Major(0, “Computer Science”), Major(1, “Computer Science”) | == 0 |
| MAJ006 | Test Compare to Greater Than | Asserts that one majors name is alphabetically greater than another majors name. | Major(0, “Computer Science”), Major(1, “Computer Information Systems”) | > 0 |

**3.1.5 Section (N)**

This component is responsible for holding information relating to a section.

**3.1.5.1 Testing Procedure**

Testing of this component is fully automated using NUnit.

**3.1.5.2** **Unit Test Cases**

All test cases utilized by NUnit are listed below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test ID | Name | Purpose | Input | Expected Output |
| SEC001 | Test Equals | Asserts that two sections are equal. | Section(0, 0, “Spring”, 2019), Section(0, 1, “Spring”, 2019) | True |
| SEC002 | Test Not Equals | Asserts that two sections are not equal. | Section(0, 0, “Spring”, 2019), Section(1, 1, “Spring”, 2019) | False |
| SEC003 | Test Get Hash Code | Asserts that a section can generate a correct hash code. | Section(0, 0, “Spring”, 2019) | 0 |
| SEC004 | Test Compare to Less Than | Asserts that a section’s start date and time is less than another section’s start date and time. | Section(0, 0, “Spring”, 2019), Section(1, 1, “Spring”, 2019) | < 0 |
| SEC005 | Test Compare to Equals | Asserts that a section’s start date and time is equal to another section’s start date and time. | Section(0, 0, “Spring”, 2019), Section(1, 1, “Spring”, 2019) | == 0 |
| SEC006 | Test Compare to Greater Than | Asserts that a section’s name is greater than another section’s start date and time. | Section(0, 0, “Spring”, 2019), Section(1, 1, “Spring”, 2019) | > 0 |
| SEC007 | Test Add Day | Asserts that a day can be added to a section. | {Section(1, 1, “Spring”, 2019), Days:{}}, “Monday” | HasDay(“Monday”) returns True |
| SEC008 | Test Remove Day | Asserts that a day can be removed from a section. | {Section(1, 1, “Spring”, 2019), Days:{“Monday”}}, “Monday” | HasDay(“Monday”) returns False |
| SEC009 | Test Has Day | Asserts that a section can state that it has a day based on the day’s name. | {Section(1, 1, “Spring”, 2019), Days:{“Monday”}}, “Monday” | True |
| SEC010 | Test Get Days | Asserts that a section can provide an iterator over its days. | {Section(1, 1, “Spring”, 2019), Days:{“Monday”, “Wednesday”, “Friday”}} | Collection over the set {“Monday”, “Wednesday”, “Friday”) |
| SEC011 | Test Add Instructor | Asserts that an instructor can be added to a section. | {Section(1, 1, “Spring”, 2019), Instructors:{}}, “000010” | HasInstructor(000010“) returns True |
| SEC012 | Test Remove Instructor | Asserts that an instructor can be removed from a section. | {Section(1, 1, “Spring”, 2019), Instructors:{“000010”}}, “000010” | HasInstructor(“000010”) returns False |
| SEC013 | Test Has Instructor | Asserts that a section can state that it has an instructor based on the instructor’s university identifier. | {Section(1, 1, “Spring”, 2019), Instructors:{“000010”}}, “000010” | True |
| SEC014 | Test Get Instructors | Asserts that a section can provide an iterator over the section’s instructors. | {Section(1, 1, “Spring”, 2019), Instructors:{000000, 000001, 000002}} | Collection over the set {000000, 000001, 000002) |
| SEC015 | Test Add Student | Asserts that a student can be added to a section. | {Section(1, 1, “Spring”, 2019), Students:{}}, “000001” | HasStudent(“000001”) returns True |
| SEC016 | Test Remove Student | Asserts that a student can be removed from a section. | {Section(1, 1, “Spring”, 2019), Students:{“000001”}}, “000001” | HasStudent(“000001”) returns False |
| SEC017 | Test Has Student | Asserts that a section can state that it has a student based on the student’s university identifier. | {Section(1, 1, “Spring”, 2019), Students:{“000001”}}, “000001” | True |
| SEC018 | Test Get Students | Asserts that a section can provide an iterator over its students. | {Section(1, 1, “Spring”, 2019), Students:{“000001”, “000002”}} | A collection over the set {“000001”, “000002”} |
| SEC019 | Test Current Enrollment | Asserts that a section can keep track of its enrollment. | {Section(1, 1, “Spring”, 2019), Students:{“000001”, “000002”}} | 2 |

This component is responsible for holding information relating to a user.

**3.1.6.1 Testing Procedure**

Testing of this component is fully automated using NUnit.

**3.1.6.2** **Unit Test Cases**

All test cases utilized by NUnit are listed below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test ID | Name | Purpose | Input | Expected Output |
| USR001 | Test Equals | Asserts that two users are equal. | User(“000001”, “Claptrap”, “Stewardbot”, “cl4ptbot@hyperion.com”, “Instructor”), User(“000001”, “Handsome”, “Jack”, “handsome@hyperion.com”, “Administrator”) | True |
| USR002 | Test Not Equals | Asserts that two users are not equal. | User(“000001”, “Claptrap”, “Stewardbot”, “cl4ptbot@hyperion.com”, “Instructor”), User(“000002”, “Ugly”, “Jack”, “ujack@hyperion.com”, “Administrator”) | False |
| USR003 | Test Get Hash Code | Asserts that a user generates the correct hash code. | User(“000001”, “Claptrap”, “Stewardbot”, “cl4ptbot@hyperion.com”, “Instructor”) | True |
| USR004 | Test Compare to Less Than | Asserts that one user’s last name is alphabetically less than another user’s last name. | User(“000002”, “Handsome”, “Jack”, “handsome@hyperion.com”, “Administrator”), User(“000001”, “Claptrap”, “Stewardbot”, “cl4ptbot@hyperion.com”, “Instructor”) | < 0 |
| USR005 | Test Compare to Equals | Asserts that one user’s last name is alphabetically the equal to another user’s last name. | User(“000002”, “Handsome”, “Jack”, “handsome@hyperion.com”, “Administrator”), User(“000003”, “Ugly”, “Jack”, “ugly@hyperion.com”, “Student”) | == 0 |
| USR006 | Test Compare to Greater Than | Asserts that one user’s last name is alphabetically greater than another user’s last name. | User(“000001”, “Claptrap”, “Stewardbot”, “cl4ptbot@hyperion.com”, “Instructor”), User(“000002”, “Handsome”, “Jack”, “handsome@hyperion.com”, “Administrator”) | > 0 |
| USR007 | Test Add Major | Asserts that a major can be added to a student. | {User(“000003”, “Zero”, “Unknown”, “zero@unknown.com”, “Student”), Majors:{}}, 0 | HasMajor(0) returns true |
| USR008 | Test Remove Major | Asserts that a major can be removed from a student. | {User(“000003”, “Zero”, “Unknown”, “zero@unknown.com”, “Student”), Majors:{0}}, 0 | HasMajor(0) returns false |
| USR009 | Test Has Major | Asserts that a student can state that it has a major based on the major identifier. | {User(“000003”, “Zero”, “Unknown”, “zero@unknown.com”, “Student”), Majors:{0}}, 0 | True |
| USR010 | Test Get Majors | Asserts that a student can provide an iterator over its majors. | {User(“000003”, “Zero”, “Unknown”, “zero@unknown.com”, “Student”), Majors:{0}} | Collection containing the set of {0} |
| USR011 | Test Add Relationship | Asserts that a relationship can be added to a user. | {User(“000003”, “Zero”, “[Unknown]”, “zero@unknown.com”, “Student”), Relationships:{000001}}, 000001 | HasRelationship(000001) returns true |
| USR012 | Test Remove Relationship | Asserts that a relationship can be removed from a user. | {User(“000003”, “Zero”, “[Unknown]”, “zero@unknown.com”, “Student”), Relationships:{000001}}, 000001 | HasRelationship(000001) returns false |
| USR013 | Test Has Relationship | Asserts that a user can state that they have a relationship with the given user (indicated by the user’s university identifier). | {User(“000003”, “Zero”, “Unknown”, “zero@unknown.com”, “Student”), Relationships:{000001}}, 000001 | True |
| USR014 | Test Get Relationships | Asserts that a user can provide an iterator over their relationships. | {User(“000003”, “Zero”, “Unknown”, “zero@unknown.com”, “Student”), Relationships:{000001}} | Collection containing the set of {000001} |

**4. Validation Testing**

Each test below corresponds to a requirement in section 2.3. This is indicated by the numeric format of the test identifier. REQ indicates that the test is for a functional requirement while NREQ indicates that the test is for a non‑functional requirement. Each requirement is suffixed by a number corresponding to the requirement in section 2.3. In certain cases there is an additional letter that indicates that the test corresponds to testing part of a requirement.

|  |  |  |  |
| --- | --- | --- | --- |
| Test ID | Purpose | Input | Expected Output |
| REQ001 | Tests that the system only allows users have a role of Student, Instructor, or Administrator. | Administrator navigates to the ‘User’ page and selects the ‘Create User’ option. | Dropdown provided to select the user’s role only contains options for ‘Administrator’, ‘Instructor’, and ‘Student’. |
| REQ002A | Tests that a user can be created. | User(“000001”, “Claptrap”, “Stewardbot”, “cl4ptbot@hyperion.com”, “Instructor”), Password=”password” | User navigates to the ‘User’ page and selects the ‘Create User’ option. User enters the provided input and clicks on the ‘Create’ button. The system displays a message that the user was created successfully. |
| REQ002B | Tests that a user can only have one role. | User(“000001”, “Claptrap”, “Stewardbot”, “cl4ptbot@hyperion.com”, “Student”) | System outputs “Error: Cannot create user: 000001, user already exists!” |
| REQ003A | Tests that a student cannot enroll in a course section they do not meet the prerequisites for. | Student navigates to the ‘Enroll’ page. The student then selects a school, department, and course. They then click ‘View Sections’, choose a section, and click enroll. | Error: Unable to enroll in section, prerequisites not met. |
| REQ003B | Tests that a student cannot enroll in a course section that is full. | Student navigates to the ‘Enroll’ page. The student then selects a school, department, and course. They then click ‘View Sections’, choose a section, and click enroll. | Error: Unable to enroll in section, course is full! |
| REQ003C | Tests that a student can enroll in a course section that is not full, and that they meet the prerequisites for. | Student navigates to the ‘Enroll’ page. The student then selects a school, department, and course. They then click ‘View Sections’, choose a section, and click enroll. | Successfully enrolled in section! |
| REQ003D | Tests that advisors can override prerequisites for their advisees. | Instructor navigates to the ‘Manage Advisees” page, selects a semester, year, school, department, and course. Instructor selects an advisee, then selects a section. Instructor clicks the “Override” button. | Advisee is successfully enrolled in the course. |
| REQ004 | Tests that students can unenroll from sections. | Student navigates to the ‘Schedule’ page, selects the section, and clicks the ‘Unenroll’ button. | Successfully unenrolled from section. |
| REQ005 | Tests that students can view their schedule. | Student navigates to the ‘Schedule’ page. | The students schedule for the current term is shown. |
| REQ006 | Tests that students can view their academic record. | Student navigates to the ‘Grades’ page. | The student’s academic record is shown. |
| REQ007/008 | Tests that instructors can view their scheduled courses. | Instructor navigates to the ‘Schedule’ page. | Each section the instructor is scheduled to teach is shown along with the enrollment for the section. |
| REQ009 | Tests that an advisor can view their advisee’s grades. | Instructor navigates to the ‘Advisees’ page. Instructor selects the advisee using the provided dropdown, selects the ‘View Record’ tab and clicks on the ‘Load Advisee Record’ button. | The advisee’s academic record is displayed on screen. |
| REQ010 | Tests that advisors can view advisee’s schedules. | Instructor navigates to the ‘Advisees’ page. Instructor selects the advisee using the provided dropdown, selects the ‘View Schedule’ tab and clicks on the ‘Load Advisee Schedule’ button. | Page refreshes and the advisee’s schedule is shown on screen. |
| REQ011 | Tests that administrators can create new courses. | User with the administrator role selects the ‘Add Course’ tab on the ‘Courses’ page, inputs valid course information, and clicks the submit button. | Course was successfully created. |
| REQ012 | Tests that administrators can modify courses. | User with the administrator role selects the ‘Modify Course’ tab on the ‘Courses’ page, selects a course to modify, modifies it in some way and clicks the submit button. | Page is refreshed with the updated course information. |
| REQ013 | Tests that administrators can archive courses. | User with the administrator role selects the ‘Modify Course’ tab on the ‘Courses’ page, selects a course to modify, ticks the ‘Archive’ checkbox so it is on, and clicks the submit button. | Page is refreshed with the updated course information. |
| REQ014 | Tests that administrators can unarchive courses. | User with the administrator role selects the ‘Modify Course’ tab on the ‘Courses’ page, selects a course to modify, ticks the ‘Archive’ checkbox so it is off, and clicks the submit button. | Page is refreshed with the updated course information. |
| REQ015 | Tests that administrators can create new users. | User with the administrator role navigates to the ‘Manage Users’ page. | ‘Manage Users’ page loads and the administrator is presented with the option to create a new user. |
| REQ016A | Tests that students cannot create new users. | User with the student role navigates to the ‘Manage Users’ page. | User is redirected to their home page. |
| REQ016B | Tests that instructors cannot create new users. | User with the instructor role navigates to the ‘Manage Users’ page. | User is redirected to their home page. |
| REQ017 | Tests that users cannot change their role. | User navigates to the account page. | User’s role is displayed in a readonly textbox with no way to change it. |
| REQ018 | Tests that users can view the course catalog. | User clicks on the ‘View Catalog’ link. | User is redirector to the ‘View Catalog’ page. |
| REQ019 | Tests that users can login using their university identifier and password. | 000001, password | User is redirected to their home page. |
| REQ020 | Tests that users can change their password. | password, P@$$w0rd, P@$$w0rd | Password changed successfully! |
| REQ021 | Tests that users can logout. | User clicks logout button. | User is logged out and redirected to the system’s home page. |
| REQ022 | Tests that administrators can add a course section. | Administrator navigates to the ‘Courses’ page and clicks on the ‘Add Section’ tab. The administrator selects a school, department, course. The administrator enters the section information clicks the ‘Add Section’ button. | The page is refreshed. The section iappears in the listing of sections for the course. |
| REQ023 | Tests that administrators can modify a course section. | Administrator navigates to the ‘Courses’ page and clicks on the ‘Modify Section’ tab. The administrator selects a school, department, course. The administrator modifies the section information clicks the ‘Modify Section’ button. | The page is refreshed. The updated section information appears when a user views the section. |
| NFREQ001A | Tests that an administrator must provide a first name when creating a user account. | First Name:””, LastName:”Wadden”, UnivID:”000999”, Email: wadd@mail.com, Role:”Student”, Major:”Biology” | Error: You must enter a first name to create the user’s account! |
| NFREQ001B | Tests than an administrator must provide a last name when creating a user account. | First Name:”John”, LastName:””, UnivID:”000999”, Email: wadd@mail.com, Role:”Student”, Major:”Biology” | Error: You must enter a last name to create the user’s account! |
| NFREQ001C | Tests that an administrator must provide a university identification number when creating a user account. | First Name:”John”, LastName:”Wadden”, UnivID:””, Email: wadd@mail.com, Role:”Student”, Major:”Biology” | Error: You must enter a university identifier to create the user’s account! |
| NFREQ001D | Tests that an administrator must provide an email address when creating an account. | First Name:”John”, LastName:”Wadden”, UnivID:”000999”, Email:””, Role:”Student”, Major:”Biology” | Error: You must enter an email to create the user’s account! |
| NFREQ001E | Tests that an administrator must provide a role when creating an account. | First Name:”John”, LastName:”Wadden”, UnivID:”000999”, Email:” wadd@mail.com”, Role:””, Major:”Biology” | Error: You must enter a role to create the user’s account! |
| NFREQ002 | Tests that students are required to have a major when their account is created. | First Name:”John”, LastName:”Wadden”, UnivID:”000999”, Email:”wadd@mail.com”, Role:”Student”, Major:”” | Error: You must provide a major for the student! |
| NFREQ003A | Tests that a user must enter a university identification number before they can login. | Username:””, Password:”password” | Error: Invalid university ID or password entered! |
| NFREQ003B | Tests that a user must enter a password before they can login. | Username:”100000”, Password:”” | Error: Invalid university ID or password entered! |
| NFREQ004 | Tests that students cannot register for more than 18 credits per term. | CarryingCredits: 16, AddedCredits:3 | Error: Cannot enroll in section, students are not allowed to enroll in more than 18 credits per semester! |
| NFREQ005 | Tests that a system loads the home page within 3 seconds of the user logging into the system | UnivID:”000001”, Password:password | User is redirected to home page within three seconds of clicking the login button. |
| NFREQ006 | Tests that a user’s password is at least 8 characters in length at account creation. | Password:Hello | Error: Password must be at least 8 characters in length. |
| NFREQ011A | Tests that a course must belong to a department. | DepartmentID:(Not entered), Name: “Data Structures”, Description:”Blah blah blah”, Number: 250, StartDateAndTime: 2019, 1, 21, 8, 0, 0, EndDateAndTime: 2019, 5, 8, 8, 50, 0, Credits: 4 | Error: Course must belong to a department! |
| NFREQ011B | Tests that a course must have a name before it can be created. | DepartmentID:(Not entered), Name: “”, Description:”Blah blah blah”, Number: 250, StartDateAndTime: 2019, 1, 21, 8, 0, 0, EndDateAndTime: 2019, 5, 8, 8, 50, 0, Credits: 4 | Error: Course must have a name! |
| NFREQ011C | Tests that a course must have a description before it can be created. | DepartmentID:(Not entered), Name: “Data Structures”, Description:””, Number: 250, StartDateAndTime: 2019, 1, 21, 8, 0, 0, EndDateAndTime: 2019, 5, 8, 8, 50, 0, Credits: 4 | Error: Course must have a description! |
| NFREQ011D | Tests that a course must have a number before it can be created. | DepartmentID:0, Name: “Data Structures”, Description:”Blah blah blah”, Number: (Not entered), StartDateAndTime: 2019, 1, 21, 8, 0, 0, EndDateAndTime: 2019, 5, 8, 8, 50, 0, Credits: 4 | Error: Course must have a number! |
| NFREQ011E | Tests that a course must state its number of credit hours before it is created. | DepartmentID: 0 Name: “Data Structures”, Description:”Blah blah blah”, Number: 250, StartDateAndTime: 2019, 1, 21, 8, 0, 0, EndDateAndTime: 2019, 5, 8, 8, 50, 0, Credits: (Not entered) | Error: Course must state its number of credit hours! |
| NFREQ012A | Tests that a course section must list a date and time offered before it can be created. | DepartmentID: 0 Name: “Data Structures”, Description:”Blah blah blah”, Number: 250, StartDateAndTime: (Not entered) EndDateAndTime: 2019, 5, 8, 8, 50, 0, Credits: (Not entered) | Error: Course section must have a start date/time and end date/time! |
| NFREQ012B | Tests that a course section must be taught by at least one instructor before it is created. | {Section(0, 0, “Summer”, 2019), StartDateAndTime: DateTime(2019, 5, 31, 8, 0, 0), EndDateAndTime: DateTime(2019, 8, 1, 8, 50, 0), Instructors: {}} | Error: Course must have at least one instructor! |
| NFREQ013 | Tests that course section start times must be earlier than course section end times. | {Section(0, 0, “Summer”, 2019), StartDateAndTime: DateTime(2019, 5, 31, 8, 50, 0), EndDateAndTime: DateTime(2019, 8, 1, 8, 0, 0), Instructors: {0}} | Error: Unable to create section, start time must be earlier than end time! |
| NFREQ014A | Tests that a course section cannot be created if its start, and end times do not differ by at least 50 minutes. | {Section(0, 0, “Summer”, 2019), StartDateAndTime: DateTime(2019, 5, 31, 8, 0, 0), EndDateAndTime: DateTime(2019, 8, 1, 8, 49, 0), Instructors: {0}} | Error: Unable to create section, start and end times do not differ by at least 50 minutes! |
| NFREQ014B | Tests that a course section can be created if its start, and end times differ by at least 50 minutes. | {Section(0, 0, “Summer”, 2019), StartDateAndTime: DateTime(2019, 5, 31, 8, 0, 0), EndDateAndTime: DateTime(2019, 8, 1, 8, 50, 0, Instructors: {0})} | Section created successfully! |
| NFREQ015 | Tests that section identifiers are displayed as SO‑### where ### is the section number. | {Section(0, 0, “Summer”, 2019), Number: 100, StartDateAndTime: DateTime(2019, 5, 31, 8, 0, 0), EndDateAndTime: DateTime(2019, 8, 1, 8, 50, 0, Instructors: {0})} | SO-100 |
| NFREQ016 | Tests that course identifiers are displayed as [Department Abbreviation]‑### where ### is the course number. | {Department(0, “Accounting”), Abbreviation: “ACTG”}, {Course(10, “Personal Accounting 1”), DepartmentID: 0, Number: 225} | ACTG-225 |
| NFREQ017 | Tests that a course may not be a prerequisite of itself. | {Course(5, “Artificial Intelligence”), Prerequisite: (5, 0), DepartmentID: 0, Number: 400} | Error: Unable to add prerequisite, course cannot be a prerequisite of itself! |
| NFREQ018 | Tests that a user’s account cannot be created if the university identification number already exists. | 2741896 | Error: Unable to create user, university identification number already exists! |
| NFREQ019A | Tests that section numbers not in the range [1, 999] are not accepted when a section is created. | {Section(1, 0, “Summer”, 2019), Number: 0, StartDateAndTime: DateTime(2019, 5, 31, 8, 0, 0), EndDateAndTime: DateTime(2019, 8, 1, 8, 50, 0, Instructors: {0})} | Error: Unable to create section, section number must be between 1 and 999! |
| NFREQ019B | Tests that section numbers in the range [1, 999] are accepted when a section is created. | {Section(0, 0, “Summer”, 2019), Number: 1000, StartDateAndTime: DateTime(2019, 5, 31, 8, 0, 0), EndDateAndTime: DateTime(2019, 8, 1, 8, 50, 0, Instructors: {0})} | Error: Unable to create section, section number must be between 1 and 999! |

**4. Test Log**

A test log will be maintained by myself and will become part of the documentation for the system. Attaching the testing log to this document is unnecessary as I have not completely implemented the system yet. Therefore, I have not completed testing yet.

Note that each entry in the testing log will have the following structure and will be organized into a tabular format: Test case attempts will be specified with a testing identifier, date, time, test case identifier, input, expected output, actual output, and result of the test.