ECSE 321

Introduction to Software Engineering

Deliverable 3

Quality Assurance Plan

March 18, 2018

Group 11

Alexander Harris - 260688155

Abbas Yadollahi - 260680343

Filip Bernevec - 260689062

Yunus Can Cukran - 260669715

Gareth Peters - 260678626

Table Contents

Unit Test Plan

Tested Classes

The classes that will be thoroughly tested throughout the development of the TreePLE application will be the following:

- TreePLEService.java (service class in the backend)
- TreePLERestController.java (controller class in the backend)
- SQLiteJDBC.java (relational database)

The rationale for testing the classes above is that they make up the core implementation use cases of the application. Each time a new method in either the service, the controller or the database is implemented, a test suite covering the boundary and middle cases must be applied to insure the expected functionality of the method.

Non-Tested Classes

The classes that won't be tested during the development of the TreePLE application will be the following:

- TreePLESpringApplication.java (Spring bootup class)
- DTO classes (Location, Municipality, Species, SurveyReport, Tree, User)
- Model classes (Location, Municipality, Species, SurveyReport, Tree, User, SustainabilityCalculator)

The rationale for not testing the TreePLESpringApplication class is that it is required for running the TestTreePLERestController, as a result it needs to be well-implemented for the test to bootup properly. For the model classes, we don't test them since they are all autogenerated by the Umple, thus there is a low probability of encountering errors. As for the DTO classes, the getters and setters methods inside each class are also autogenerated, apart from the constructors, hence the methods are trivial to test.

Techniques and Tool Support

The unit test techniques used to verify the implementation of the business methods are the structural and functional unit testing. Being a white-box approach with selected inputs and expected outputs, the structural technique, more specifically path testing and statement testing, are used to derives the test suites. These tests will ensure a proper coverage by going through all the paths and statements of the methods are executed at least once. For the functional testing techniques, the boundary value and input domain testing are used to guarantee that every scenario of input is tested, as well as boundary conditions on specific values. With all the testing techniques mentioned above, the test suites created will increase the coverage of the overall code.

The tools used are JUnit in Eclipse, as well as Eclemma as a plugin to get a code coverage report on the tests. Jenkins is also used to continuous delivery the project on the hub with automated testing.

Test Coverage

The test coverage for unit testing will be ideally around 100%, as with the techniques mentioned previously, every statement of a method is tested. The tests are executed every time a new method in the business logic is implemented, even in its starting stage. This way, test-driven development will help change the method to ensure it covers all the tests related to it. Trigerring the tests is done manually for now, but with Jenkins later on, the continuous integration will trigger the tests after every commit. This way, all the tests created will validate that the code is still fully functional if a new feature has been added.

List of business methods to Test

Here is a list of all the business methods in the service class covered by tests:

- createUser()
- createSpecies()
- createMunicipalities()
- createTree()
- getTreeById()
- getUserByUsername()
- getMunicipalityByName()
- deleteUser()

Here is a list of all the business methods in the database class covered by tests:

- insertUser()
- getUser()

- getAllUsers()
- deleteUser()
- updateUserPassword()
- updateUserRole()
- updateUserAdresses()
- updateUserTrees()
- insertSpecies()
- updateSpecies()
- getAllSpecies()
- getSpecies()
- deleteSpecies()
- insertLocation()
- updateLocation()
- getAllLocation()
- getLocation()
- deleteLocation()
- insertMunicipality()
- updateMunicipality()
- getAllMunicipalities()
- getMunicipality()
- deleteMunicipality()
- insertTree()
- updateTree()
- getAllTrees()
- getTree()
- deleteTree()

Integration Strategy

This project will utilize a bottom-up approach towards integration testing to match our development strategy. This will require us to write multiple drivers to simulate higher-level components. Tools used to perform the integration testing will include JUnit and a custom tool for generating a pre-populated database for testing purposes. It is difficult to evaluate test coverage using tools like EclEmma when it comes to integration testing, so we will outline below the specific goals of our integration tests and expected behaviour.

Backend Services

1 - Integration between the SQLiteJDBC class containing the logic for querying and updating the SQLite database, and the TreePLEService class responsible for the main business logic. Goals of this testing would be to verify that data stored in the database is correct and persistent. Namely the service is able to correctly update database records, add new ones, remove them and retrieve data corresponding to the desired query. Types of data would include trees, user data, species, municipalities, locations and surveys. Very important to test this integration as otherwise this could result in incorrect/corrupted user data, inability for users to access the application, access to false data or data from other users.

2 - Integration between TreePLEService and TreePLERestController class. Goals of this test would be to verify the proper function of the DTO Conversion APIs and the REST APIs, namely verifying that the controller is properly converting data provided by the TreePLEService into the required DTO, and that the data received when calling the service is correct.

Umple-generated models and corresponding DTO classes, as well as the TreePLESpringApplication class will not be subjected to their own integration testing, as they are quite simple and used exclusively in the classes mentioned in the above integration tests, and as such should experience sufficient coverage when including the unit testing.

Backend and Frontend Integration

The process for testing the backend and frontend integration will be broken down into the following test cases:

- User login:
    - User needs to enter all required/correct credentials.
    - Correct user data is retrieved.
- User registration:
    - User can not create account with same username as an existing one.
    - User is added to database with correct data.
- Planting tree:
    - User needs to enter all required data before tree is added.
    - Tree is added to database with correct data matching what was entered by user.
    - Tree is added to list of trees associated with user.
- Cutting down tree:
    - User can only cutdown trees that are “owned”.
    - Correct tree is removed from database.
    - Tree is removed from users list of owned trees after being cut down.
- Populating map with existing trees:
    - Frontend retrieves all trees contained in the database and populates map with them.
    - All trees on the map display proper information corresponding to what is contained in the database (i.e. height, diameter, species, etc.).

System Test Plan

Create User

Similarly to actual operation, the first functionality a user would interact with would be to create a user in order to login to the application. This is required for most of our primary use cases, such as plantTree and cutDownTree. Because of this, system test cases must first be implemented for creating a user. These test cases include:

- Checking if a newly created user (with valid inputs) is properly saved in the database.
    - This would need to be done by getting the user back from the database to the frontend through the use of getAllUsers and getUserByUsername through an HTTP call.

- Checking if an invalid user is not saved into the database from the frontend as a valid user
    - This includes the testing of: null username, null password, empty username, empty password, username with only spaces, password with only spaces, username that already exists in the system, null addresses, empty addresses, addresses with only spaces. This allows us to test our error reporting on the frontend services of our application for creating and getting a user.

Two separate test cases would need to be performed for creating a user: one for testing the interaction of the Android frontend with the rest of the system and one for testing the interaction of the Web frontend.

Get All Trees

To later test the plant tree system function, get all trees must first be tested since plant tree is retrieved from the database using get all trees. These test cases involve:

- Populating the database with tree JSON Objects, and then viewing on the web frontend if those tree objects are retrieved. This requires the database to be also populated with at least one user, municipality and species.
    - This involves comparing each of the attributes of each tree, so the location, height, diameter, municipality, land, ownership, status, species, survey report, and date planted.

Create Municipality

The test cases for creating a municipality in the system should pass before testing plant tree. This is preferable to populating the database with a municipality before plant tree since the system will eventually have to be able to create a municipality anyways. Creating municipality will eventually only be a function for scientists, but for now is a universal function for any user. This requires an unpopulated database just for testing purposes.

- A test case needs to be performed to test creating a valid municipality. Creating a municipality requires inputs for the borders (more than 3 locations), and the name of the municipality.
    - The municipality can then be retrieved from the database by making an HTTP Get call at the moment this document is written, but will eventually be able to be seen by seeing an area highlighted by a colour. This will be able to be clicked on to view certain sustainability attributes as well as the name of the municipality.

    - The test case is fulfilled if the border locations are all correct and the name is correct. The total amount of trees should be zero.
- Invalid test cases must also be performed to test error reporting. This includes:
    - No border locations
    - Less than 3 border locations
    - Null name
    - Empty/Only spaces municipality name
    - Municipality already exists with name specified

Create Species

The test cases for creating a species in the system should pass before testing plant tree. This is preferable to populating the database with a species before plant tree since the system will eventually have to be able to create a species anyways. Creating species will eventually only be a function for scientists, but for now is a universal function for any user. This requires an unpopulated database just for testing purposes

- A test case needs to be performed to test creating a valid species. Creating a species requires inputs for the genus, the species itself and the common name of the species.
    - The species can then be retrieved from the database by making an HTTP Get call.
    - The test case is fulfilled if the HTTP request properly retrieves the genus, the species and the common name of the species.
- Invalid test cases must also be performed to test error reporting. This includes:
    - Null genus name, empty genus name, only spaces in genus name
    - Null species name, empty species name, only spaces in species name
    - Null common name, empty common name, only space in common name
    - Species already exists with given common name.

Plant Tree

In order to later be able to test the system for cutting down a tree, plant tree must first be tested since it is not possible to cut down a tree unless it is first planted.

- A detailed description of a test case for testing a valid creation of a tree is seen below.
- Many system test cases are required for testing error reporting. This includes test cases for:

- empty height, empty diameter
- null species, non existent species
- null municipality, non existent municipality
- null land, null ownership, null status

Cut Down Tree

Cut Down Tree needs to be tested since it is one of the main use cases of the application.

- A test case is made to test if cutting down of a tree is valid. A user needs to be created and be given a tree. If a user owns a tree, only they can cut down the tree. For residents, this only includes trees that are on their property. For scientists they should be able to cut down any tree.
    - The test case is passed if the tree disappears from the Google Maps UI on the web frontend. This implies that the tree has successfully been deleted from the database.

Two Detailed Test Cases

1. Testing if planting a tree in the Android frontend creates a tree in the database:
    a. Start with a completely empty database or a database that already has existing municipalities and species.
    b. Create a valid user account with a Scientist role so as to allow for the user to plant anywhere.
    c. Interact with the Google Map UI to choose a location that the tree will be planted at. This can be done by holding down on the location where the tree is to be planted.
    d. Provide all valid information for the tree to be planted. This information includes the ownership, land, height, diameter, species, municipality and date planted. This might involve creating a new species and a new municipality that the tree will have as characteristics.
    e. Create the tree by confirming the tree that is wished to be planted.
    f. If everything is working, the tree will appear at the location specified as an icon.
    g. If the tree is selected, a popup message should appear that contains all the information of the newly created tree.
    h. The test case is successful if the popup window provides all the characteristics of the tree correctly. This includes the correct height, diameter,

species, municipality, location, land, ownership, status that is Planted, date planted, and survey report with one entry that contains the user's information and the date planted. Sustainability attributes will also be present and can have their values confirmed by manually performing the calculations.

i. Lastly, switch to the Web frontend and utilize the function to list all of the trees to see if the newly created tree is viewable on the Map with the same attributes as stated above.

2. Testing if a customer was able to post a survey from the Android frontend:
    a. Start with a completely empty database or a database that already has existing municipalities and species.
    b. Create a valid user account with a Scientist or a Resident role
    c. Interact with the tree on Google Map UI, i.e choose a tree that will be reported upon
    d. Check for authorizations. If the user is a scientist they can mark any tree to be struck down. Residents have limited jurisdiction over their privately owned trees and they can't mark public trees for cutdown.
    e. Given correct authorization, check if a valid type of report appeared in the database
    f. The test is successful if upon switching back to Android we see a visible survey when tapped on the tree icon.

Expected Coverage

Expected coverage should be about 90% of the system so far. We base this estimate on how much of the use cases we are covering. Since we are treating the whole system as a black box we might be losing some coverage on some internal methods. The loss of coverage would also be caused by the fact that some of the branching statements can never be reached from the frontend. An example of this is the if statements checking if the inputted enumeration is part of the actual enum. The frontend only allows for the user to choose from those few entries in the enum, so it is not possible to actually enter those if statements.

Several of our delete methods in the service have not actually been implemented in the frontend yet, so that also reduces the coverage by quite a bit.

Work Plan

Work Plan Week 6

KEY DEADLINES

- February 11th : Deliverable 1
- February 25th : Deliverable 2
- March 18th : Deliverable 3
- March 29th : Deliverable 4
- April 8th : Deliverable 5
- April 11th : Deliverable 6

WORKING HOURS

Team Members	Week 1	Week 2	Week 3	Week 4	Week 5	Week 6	Week 7	Week 8	Week 9	Week 10
Filip Bernevec	6	4	5	4	5	6				
Abbas Yadollahi	6	5	10	6	5	7				
Alexander Harris	6	5	7	6	6	7				
Gareth Peters	6	4	5	5	5	6				
Yunus Cukran	6	5	5	4	5	6				

TASK ALLOCATIONS

WEEK 1: February 5th – 11th		
Team Members	**Leadership Roles**	**Main Task**
Filip Bernevec	Project Manager	Requirements-level activity diagram

Abbas Yadollahi	Back-end developer	Domain level statechart + workplan
Alexander Harris	Android developer	Use case diagram + use case specifications
Gareth Peters	Web developer	Functional and non-functional system requirements
Yunus Cukran	Back-end developer	Domain model in Umple and class diagram

WEEK 2: February 12th – 18th		
Team Members	**Leadership Roles**	**Main Task**
Filip Bernevec	Project Manager	Sequence diagrams + Work Plan
Abbas Yadollahi	Back-end developer	Backend Business Logic + Database
Alexander Harris	Android developer	Android Front-end
Gareth Peters	Web developer	Detailed Design of proposed solution + class diagram
Yunus Cukran	Back-end developer	Architecture of proposed solution + block diagram

WEEK 3: February 19th – 25th		
Team Members	**Leadership Roles**	**Main Task**
Filip Bernevec	Project Manager	Sequence diagrams + Work Plan
Abbas Yadollahi	Back-end developer	Backend Business Logic + Database
Alexander Harris	Android developer	Android Front-end
Gareth Peters	Web developer	Detailed Design of proposed solution + class diagram

Yunus Cukran	Back-end developer	Architecture of proposed solution + block diagram

WEEK 4: February 26th – March 4th		
Team Members	**Leadership Roles**	**Main Task**
Filip Bernevec	Project Manager	Web front-end start
Abbas Yadollahi	Back-end developer	Backend + Database
Alexander Harris	Android developer	Android Front-end
Gareth Peters	Web developer	Back-end bugs
Yunus Cukran	Back-end developer	Database bugs

WEEK 5: March 5th – 11th		
Team Members	**Leadership Roles**	**Main Task**
Filip Bernevec	Project Manager	Unit Testing
Abbas Yadollahi	Back-end developer	Unit Testing
Alexander Harris	Android developer	Unit Testing
Gareth Peters	Web developer	Unit Testing
Yunus Cukran	Back-end developer	Unit Testing

WEEK 6: March 11th – 18th		
Team Members	**Leadership Roles**	**Main Task**
Filip Bernevec	Project Manager	Unit Test Plan + Unit Testing
Abbas Yadollahi	Back-end developer	Unit Test Plan + Unit Testing
Alexander Harris	Android developer	Integration Test Plan + Unit Testing

Gareth Peters	Web developer	System Test Plan + Unit Testing
Yunus Cukran	Back-end developer	System Test Plan + Unit Testing

WORK PLAN

RED = Task completed

YELLOW = Modification to last work plan

Week 1 (February 5th – 11th)

Key Deadline: Deliverable 1 Feb 11th

- Determine the functional and non-functional requirements.
- Determine the use cases with diagram and specifications.
- Design the activity diagram for entire scenario.
- Design the domain model in Umple with class diagrams
- Design the statechart for class Tree.

Week 2 (February 12th – 18th)

- Develop a system architecture including block diagrams.
- Describe a detailed solution including class diagrams.
- Develop a prototype including “Plant Tree” and “Cut Down Tree” use cases for the Java Spring backend and the Web and Android front-end.

Week 3 (February 19th – 25th)

Key Deadline Deliverable 2 Feb 25th

- Develop a prototype implementation containing the “List All Trees” use case in the backend and frontend.
- Create an implementation-level sequence diagram from “Plant Tree” and “List All Trees” use cases covering all architectural layers.
- Implementation for Android frontend pushed to next week.

Week 4 (February 26th – March 4th)

- Android front-end implementation with Google Maps API.
- Fixing bugs with the database code and back-end code of several methods.
- Forecast use case implementation pushed to later weeks.

Week 5 (March 5th – 11th)

-Develop test cases for unit testing, system testing and component testing.

- Start testing the software prototype.

Week 6 (March 12th – 18th)

Key Deadline: Deliverable 3 Mar 18th

- Write-up of deliverable 3 report and implement test cases for business methods.

- Description of release pipeline pushed to next week.

Week 7 (March 19th – 25th)

- Finalization of the Android and Web front-end.
- Implementation of extra features and testing extra features.

Week 8 (March 26th – April 1st)

Key Deadline: Deliverable 4 Mar 29th

- Create a presentation for the TreePLE application.

Week 9 (April 2nd – 8th)

Key Deadline: Deliverable 5 Apr 8th

- Review source code of full implementation.

Week 10 (April 9th – 11th)

Key Deadline: Deliverable 6 Apr 11th

- Submit source code and commit history.