Unit Test Plan

In our Unit Test Plan, the classes that will be tested will be the functional classes of the system:

- HomeAudioSystemController by testing each methods of the controller (i.e. addSong)
- PersistenceHomeAudioSystem by making sure the classes are written the right way in the persistence

While the following classes won't be tested since they are the presentation and design classes:

- HomeAudioSystemPage (View)
- HAS: Song, Location, Artist, Playlist, Album, AlbumGenre (Models)
- DateLabelFormatter, InvalidInputException (helper classes)

In other words, the classes that will be tested in the Unit Test Plan will mostly be the classes in the controller and the persistence while the classes in the model and the view will not be tested. The reason is that the goal of the Unit test is to test the program components, i.e. the different functions and methods, and making sure they return the desired result using different input parameters. In our system, these methods and functions are mostly present in the controller, and most interactions are handled by the controller as well. While the view contains all the information presented to the users, it does not perform any modifications on this information. The view sends all its information to the controller, where it is processed and transferred back to the view. Therefore, the view don't need to be unit tested (the GUI will be tested in another plan). In the case of the model, it consists of classes that dictate the structure of the system, and contain its logic and rules. However, as with the view, the model which contains the information on the system's behavior is mostly modified and instantiated by the controller. It does not perform any change or provide any functionality by itself. Therefore, the controller is still in charge of the user interaction and the model does not need to be unit tested.

To derive the test cases for the Unit Tests, there are two techniques that we will be using: The Partition Testing and Guideline-based Testing. First, the Partition Testing makes group of inputs to our tests with common characteristics that will have a valid output and group of inputs that will have invalid outputs. Each group only needs tests for one input, thus saving testing time. For example, when we change the volume to a location in the application, the volume input would have to be a positive integer with a value below 100. This would represent a valid group of input, while an invalid group of inputs would be a negative integer or greater than 100. The second testing strategies, Guideline-based Testing, will consist of writing test cases based on testing guidelines reflecting common errors found by programmers such as changing the size of the collection, choosing inputs generating error messages, making the buffer overflow, test the one-value collections and so on. An example in our application would be to test the functionality of the addSong method by entering an empty string or a spacebar in the song name when deriving the test case (this should output an appropriate error message).

The tool used for the unit testing will be JUnit and PHPUnit both specialized unit testing frameworks based on xUnit. The JUnit framework gives us the ability to automate the unit testing in Java. By using an automation framework, we can create generic test classes containing test cases that can be all run automatically with a detailed report at the end detailing which methods passed or failed, and if it is the case, why it failed. This gives us the possibility to run an entire record of tests after a change has being amended in the program. The PHPUnit has exactly the same functionality as the JUnit, but in the PHP language

The test coverage of our application should be at least 90% for the classes of the controller (as explained some classes don't need to be tested). The % of test coverage for Unit Testing is the higher compared to integration testing and system testing, since Unit testing is a low level testing method. It aims for precise methods and components of different classes, thus it is easier to test precise line of codes and to simulate an error in this environment. However, it is not necessary to reach the 100% test coverage as the focus should also be put into logic and functionality testing instead of testing every lines of the source code.

These tests should be run in two occasions: When a pertinent change is brought to the program, and before the program is launched into production. Since the unit cases are automated, all the tests cases can be run rapidly. Therefore, they can be prompted after each pertinent change without creating a time issue.

Finally, the Unit Test for the desktop app and the mobile app should essentially be the same, as they use the same jar file (they have the same model and controller). However, the difference will be while testing the view (the GUI), which will be done manually, not in the Unit Test. A difference to note will be between the desktop app and the web app, since they use two different automation frameworks. Basically, the testing content will be the same, but it will be written in two different fashions.