Testing Plan – H11B_ECLAIR

Our testing plan emphasis test-driven development. This is a software development process relying on software requirements being converted to test cases before software is fully developed. In other words, we will first plan our testing approach and code our tests before we begin coding functionality.

Our testing will compromise of unit testing, integration testing and system testing to gain a wholistic overview of our code functionality, not only on its own but with other functions within the system. Our testing model employs a zoom-out approach by commencing with unit tests to isolate each object within the system and to ensure their functionality and to consider micro-edge cases. Once we have done this, we move to group individual components and test them with each other; integration testing. This ensures that objects are functional individually as well as with other objects in a group setting. Finally, we test the system as whole by system testing. This is the most holistic form of testing as it not only involves testing groups of objects together but those groups with other groups (the system). To be more technical, we will test every package within our backend framework together to ensure they are compatible with one another.

Unit tests are those aimed at testing the smallest piece of code that can be (logically) isolated. In the purposes of OOP, we will aim these tests at the methods and returns of classes and subclasses. An example of this is in the first task delegation, we aim to complete the implementation of the Entity class and subclasses. As such, a unit test would be to create the object, and see that the object's method's return matches the expected return, thus signifying that the backend implementation matches out expectations of how the objects work.

Following unit testing, we have integration testing. These focus on testing different components as a combined body. An example of integration testing within our project would be to test the interaction between Entity subclasses, such as how the Player entity can utilize methods that affect the fields of Collectible entities. Integration testing would thus aim to determine that the correct interaction occurs within these components and thus check the backend functionality cross classes.

Finally, system testing is done to evaluate the entirety of the project as a further extrapolation of integration testing, Within the project, this would be the testing of functionality as the game system runs, with each possible class and interaction examined to assess that the backend returns as expected. This would be the running of the Dungeon Mania Controller, which allows for the invoking of all possible classes while running through a game, and thus we can system test the overall project system.

In summation, it is especially important to include many testing techniques within the testing plan. This ensures that any given software functions as intended from top to bottom. Additionally, by testing many different situations (whether these may be unlikely or not), especially edge cases and unique scenarios, we increase our code coverage. This is critical as code coverage refers to the percentage of source code executed during testing. In other words, the greater the test coverage, the more lines of code executed and by extension of this the less errors that can occur. By executing all lines of code and testing that they function as intended, the margin for error becomes slim and hence, including multiple forms of tests during test-driven development is critical.