We used OUnit and manual tests to test the functionality of our game. The basic functionality of the game was tested through glass box OUnit2 testing, however, the more complicated functionality of the game was tested manually.

Plant.ml

Most of the functions from Plant.ml were tested through glass box OUnit testing for each type of plant. For example, feed, water, and neglect functions were tested for all the plant types since all the plants have different attributes and were created differently. However, the random tragedies in Plant.ml were tested manually through the game because they take in randomly generated values. Those functions would be hard to test through OUnit testing because of it. While playing the game, we were able to manually test these types of functions. Glass box testing ensured that the functionality of this module was correct because we were able to cover most parts of the functions. For example, as mentioned, we were able to test the attributes of every plant we implemented, allowing us to confirm that the functions within the module worked for not just a few plants, but for all the plants we designed.

Garden.ml

For Garden.ml, we used OUnit testing to see if the feed, water, and neglect functions from Plant.ml would get applied in the proper way. In addition, we also tested whether a plant would get removed from the garden after calling the remove_plant function. The rest of the functions included some sort of visualization in the terminal, like print_garden. So, we mainly tested those functions manually. We went through all the menu options and tried all the possible options in order to see if the visualization parts would work. In addition, other functions like dragon and unicorn that included the use of a random number generator were tested through manual testing too. Completing all these steps ensured that the correctness of our system was met because we were able to confirm the expected results and outcome of the implementation. Also, for functions such as feed, water, and neglect, we had to implement test cases based on what we expected to happen in plant.ml. By confirming that the functions in garden.ml correctly affected the attributes in plant.ml, we were able to verify that the dependencies and interactions between the two modules were correct, confirming the correctness of the system as a whole.

Inventory.ml

For Inventory.ml, we used OUnit testing to check that plants would not be added to the inventory since they should be added directly to the garden instead. Additionally, we used OUnit testing to ensure that non-plant items (e.g., beef) would be added to the inventory after being purchased. We also tested that adding a ladybug to the garden would

increment its count in the inventory. I manually tested that item counts in the inventory were properly initialized and incremented. Confirming correctness within this module required verifying the expected behavior of the functions it implemented. Unit testing allowed us to write “simple” tests to determine if a component of our implementation correctly increased something that was expected. We were able to successfully verify that the functions produced the expected results in a predictable manner.

Store.ml

For Store.ml, we used OUnit testing to determine that buying plants incremented the plant_count attribute of the garden and that buying a ladybug would add it to the inventory. We manually tested that the store randomly applied discounts to items and that buying from the store would subtract money from the user as expected. Ensuring that the test cases within this module passed confirmed that the testing approach demonstrated the correctness of the system because it confirmed the logic we wanted to implement within this module. Since this module represented a store and the goal was to allow users to buy certain items based on different prices, the only way to confirm that our implementation successfully met this goal was to determine if items were actually added to the user’s inventory and if users lost money when they made a purchase. By using glassbox testing, we were able to create test cases based on these expectations, confirming the correctness of the system. Also, since we confirmed the correctness of other modules such as inventory.ml, we were able to use this module’s functionality to determine if store.ml was correct. As previously mentioned, we tested to determine if an item was added to the inventory after it was purchased by the user. This required the use of another module, verifying the dependencies and interactions between different modules of the system.

Recipe.ml

For Recipe.ml, all the functions (have_ingredients, missing_ingredients, update_inventory, create_recipe, and sell_recipe) were tested through glass box OUnit testing. To do this, we created our own recipes that aren’t part of the menu, and tested it. Additionally, we tested manually to see if after creating a recipe, it would add it to the inventory and subtract the ingredients from the inventory too. We also tested the sell function in a similar way, where we checked to see if the money in the garden would increase by the recipe sale price after selling it. By completing these test cases, we were able to ensure that the functions and implementation within this module were all correct. Correctness of the system was ensured within these test cases because we tested functionality of other modules based on what the functions in recipe.ml implemented. For example, when considering the creating recipe function, we tested to see if the inventory changed as expected, which is something we had to use from the inventory module. By not only testing the functions that affected the functionality within recipe.ml but also

testing the functionality of the modules that are affected by recipe.ml, we were able to ensure correctness of the system.