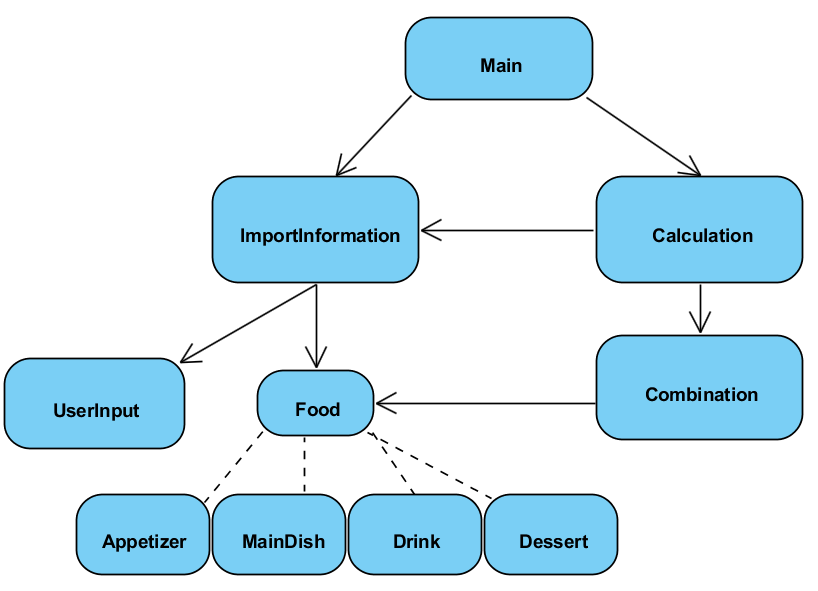
Test report – Release 2.0

1. Dependency diagram
2. Testing method
3. Test plan
4. Unit test
5. Integration test
6. System test
7. Branch coverage
8. C/DC coverage
9. Loop coverage

Dependency diagram

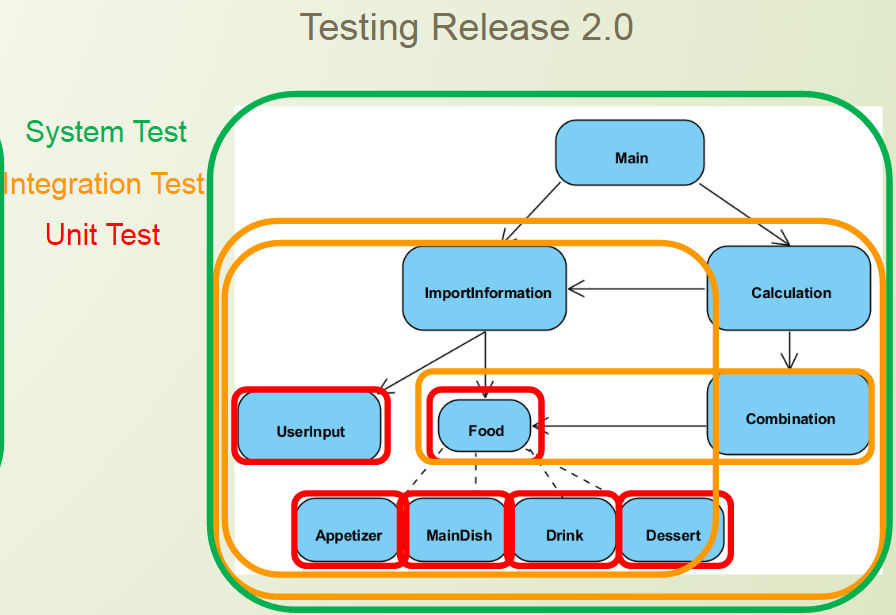


This diagram show the dependency relation between each classes and the general execution flow of the program. Although the number of class had been increased from 7 to 10 after performing debugging and refactoring process in the release 1.0 of the program, the main algorithm of calculating combination and importing user input remain mostly the same. The main class which control the main execution flow call the functions from two classes namely “ImportInformation” which help to import the menu and users input and “Calcualtion” which is mainly responsible for calculating valid food combination based on the imported user’s preferences and the imported menu. However, the testing method that we used in release 2.0 is different from release 1.0 which was used “Big Bang” approach. Instead, we implemented “Bottom-up” approach which is a more systematic and effective approach to help us test the program.

Testing method

Since the “Big Bang” approach that we used in release 1.0 has increased complexity and efforts to locate the bugs causing code in the program, more time is needed to spend on testing and debugging which in turn the project plan had been slightly delayed. In order to give greater confidence that program is being test properly and increase the ability to trace the bugs more easily, the “Bottom up” approach is then being used in the program testing of the release 2.0. The “Bottom-up” strategy required integration testing before performing system testing while all bottom level classes had been tested by unit testing. Hence, we can ensure that the lower level classes were being tested and confirmed that the code in those classes were performed as expected and prevent fault causing section from propagating to other classes, while the integration test can help to isolate and locate the bugs causing section in the sub-system. Unlike other testing strategies such as “Top-down”, ”Sandwich”, “Modified top-down”…etc. which test tubs have to be created when doing integration test, the “Bottom-up” approach requires no test stub during the testing process which means some efforts and time can be saved while the program is still being tested properly. Therefore, we decided to use “Bottom-up” strategy due to limited development time for our project.

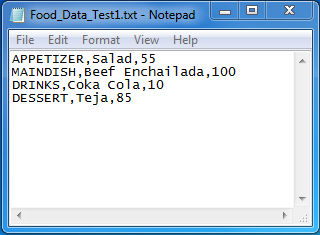
Test plan



The diagram from above showed a general view of the testing sequence and classes involved in unit-testing, integration and system testing. Since we were using “Bottom-up” strategy to test our program, there will be a total of 6 unit tests, 3 integration tests and 1 system test need to be validated and checked individually. The testing will begin with unit testing, followed by integrated testing and finally system testing. The following table shows the test sequence:

|  |
| --- |
| 1. Unit Testing:  Food (Stub: N/A)  Appetizer (Stub: N/A)  MainDish (Stub: N/A)  Drink (Stub: N/A)  Dessert (Stub: N/A)  UserInput (Stub: N/A) |
| 2. Integration Testing:  Combination + Food (Stub: N/A)  Information + UserInput + Food + Appetizer + MainDish + Drink + Dessert (Stub: N/A)  Calculation + Combination + Information + UserInput + Food + Appetizer + MainDish + Drink + Dessert (Stub: N/A) |
| 3. System Testing:  Main + Calculation + Combination + Information + UserInput + Food + Appetizer + MainDish + Drink + Dessert (Stub: N/A) |

The testing was conducted by using Eclipse with Java Runtime Environment 1.7 installed and external Junit 4 library. The program will be tested by mostly using “assertEquals” function which provided by Junit library to validate the program output and expected output. Different form the program that run in deployment, the program that we used in testing will import a menu that only contains a single food for each of 4 categories of food types. This help us to decrease the complexity of the test algorithm while it is sufficient to satisfy the requirements to test the program properly. The following screen capture shows that test data that we used throughout our testing process.



**Unit testing**

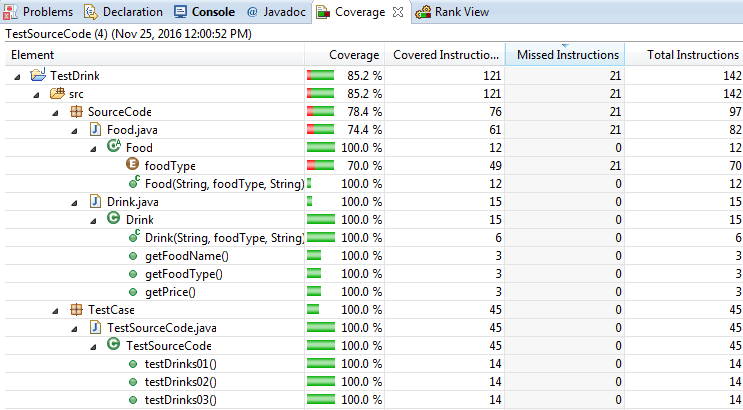
Similar to release 1.0, unit tests for bottom level classes such as (Food, MainDish, Drink…etc.) are required before conducting integration test for higher level classes. This can help us to confirm that the bottom classes are working as expected and the bugs will least likely caused by these classes. The following screen capture shows one example of the six unit tests that being tested in similar way.



The method that we implemented to test these classes is to directly compare the string output of the function. Since these classes mostly consist of getter and setter functions, we directly observe and access the parameter during run time. We can test the program output by creating a new object of that class and compare the expected output which is set separately as a string object and compare them by using “assertEquals” function.

**Coverage**

Due to the fact that the bottom classes were mostly composed of simple getter or setter functions, there is no conditional statement or for-loop statement in those classes. Hence, we cannot measure the effectiveness of the test by counting branch coverage, loop coverage or condition coverage expect statement coverage. The following screen capture shows the statement coverage of the bottom level classes.



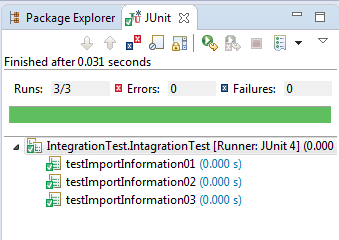
Note that the statement coverage of “Food” class is failed to reach 100% coverage, this is because the “Food” class itself is an abstract class and also the super class for “Drink”, “Appetizer”, “MainDish” and “Dessert” class. Hence, some of the statement in “Food” class cannot be directly accessed and decreases the statement coverage slightly. However, we can obtain 100% statement coverage in other classes which means our unit tests are successful.

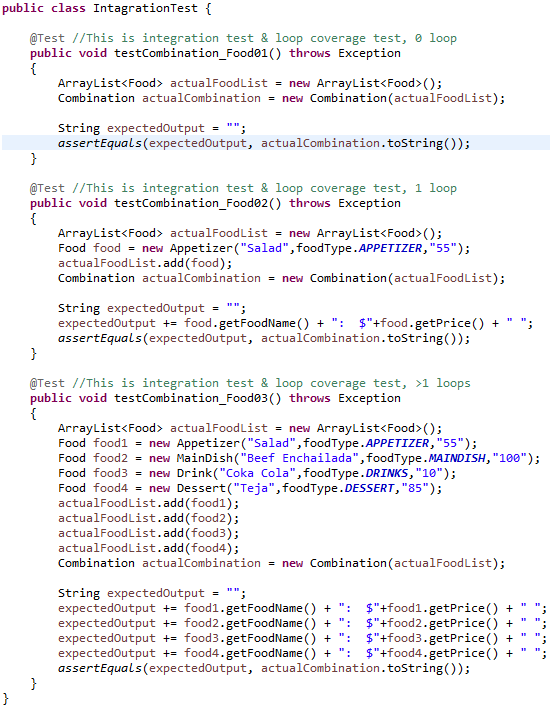
**Integration testing**

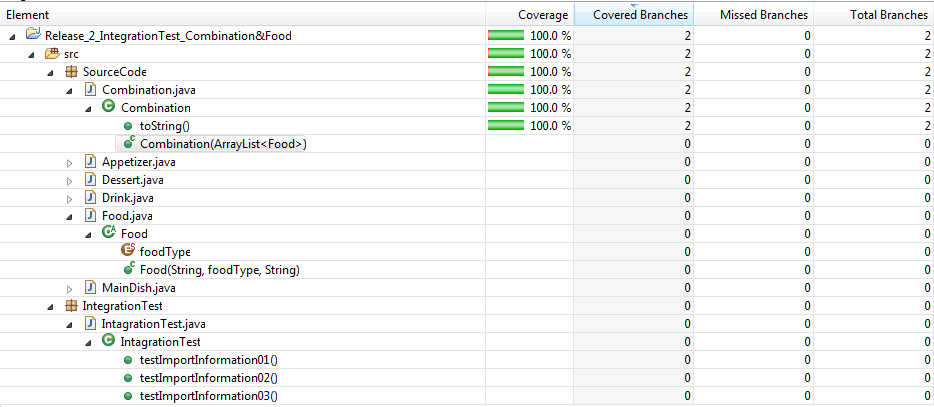
The integration test itself is more like a logical extension of unit testing, multiple units will be combined and being tested together instead of testing a single unit. The purpose of performing integration test to check if the combinations of different individual units of the program behave as expected and eventually all modules and sub-system which construct the program will be tested together. One of the advantages of performing integration test is that potential faults that only existed when different units are being tested as modules can be identified at early stage. If bugs were found in these tests, they are more likely caused by the interface or interaction between units. Hence, bugs or faults can be easily located in specific sub-system.

For the second release version of our program, there are a total of three integration tests which mentioned in test plan. The following screen captures show the number of test cases and its implementation used to those three integration tests separately.

**1. Combination + Food (Stub: N/A)**

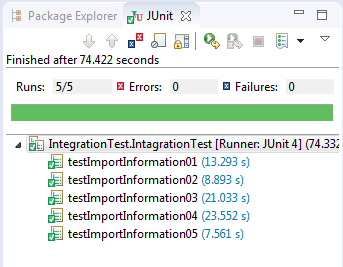


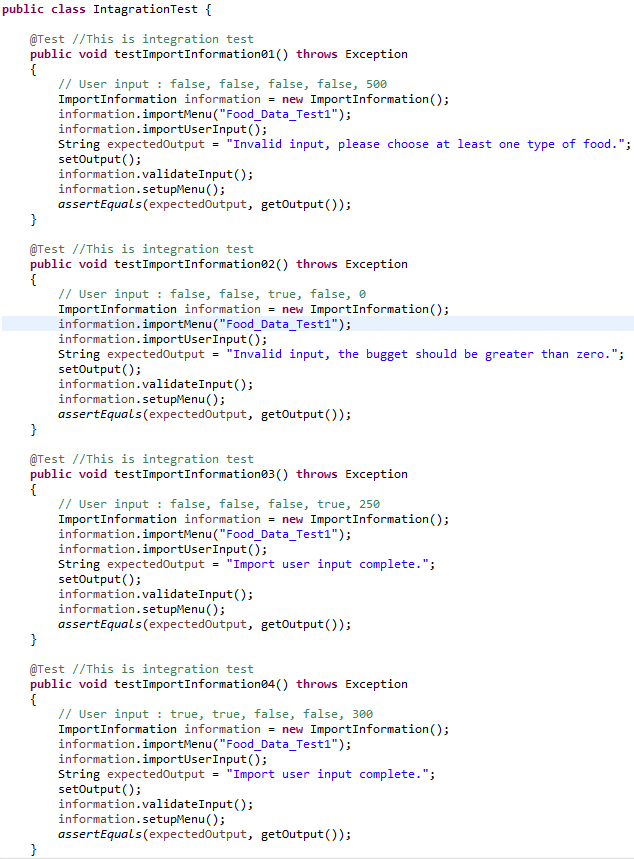


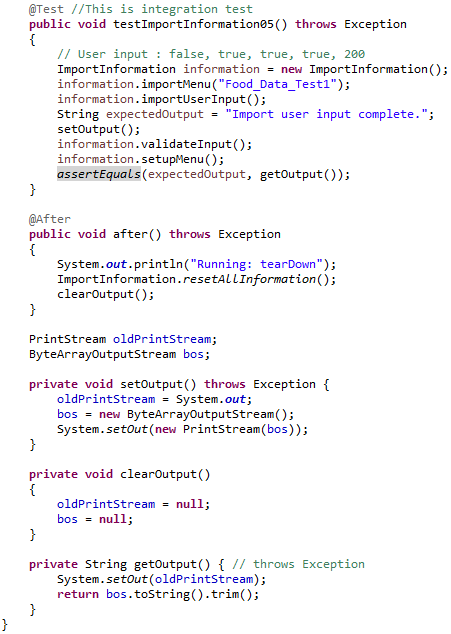


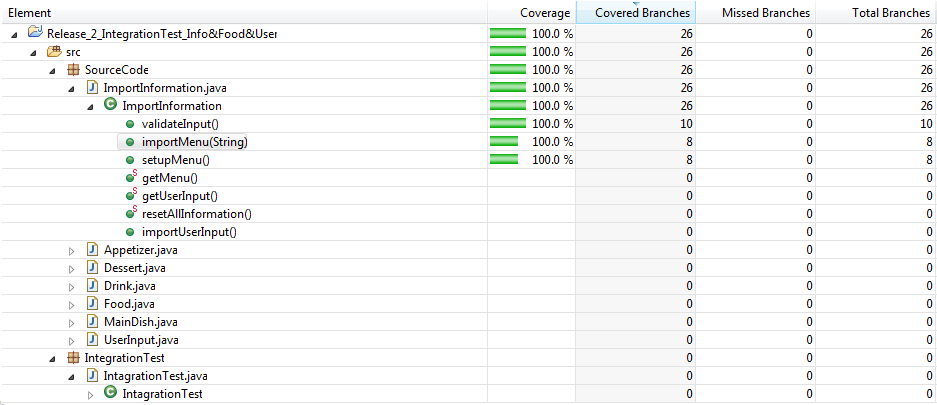
Loop and branch coverage

As shown from the screen capture above, this integration test only need 3 test cases to complete to task, this is because the “Combination” class is rather a simple class which only consist of a constructor and a “toString” function. While the “toString” has a simple for loop to iterate a list and produce string output. Hence, both of the loop and branch coverage can be achieved by only doing 3 test cases. The first test case will not enter the loop, while the second test will only iterate the loop body once and the third test will execute the loop body for multiple times. The tests were validate by entering on element into the list, entering a single element into the list and entering multiple element into the list. Proved by the screen captures from above, our test cases were successfully pass without error and it is able to achieve full branch coverage. From our analysis, it also able to achieve full loop coverage.

**2. Information + UserInput + Food + Appetizer + MainDish + Drink + Dessert (Stub: N/A)**







Branch coverage

C/CD coverage

