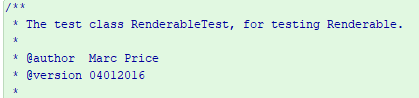
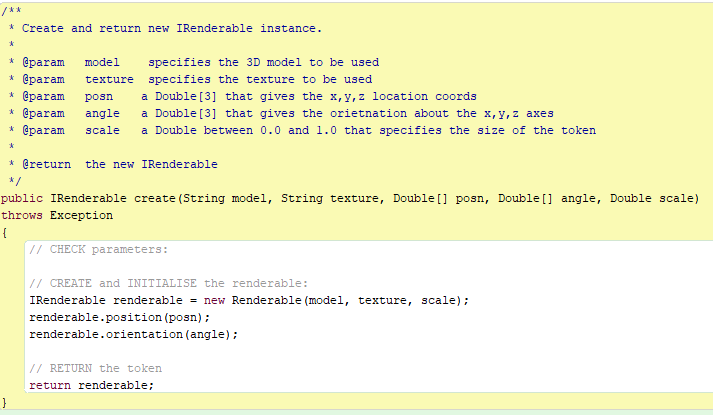
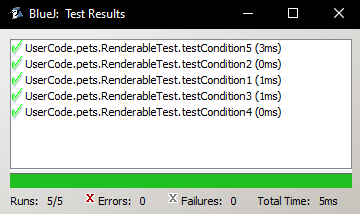
1. Make an initial assessment of the project so far: does it compile and execute without error, what does it do, does it appear to function correctly based on the description above, is it well-commented, does the Javadoc give sufficient details, does the code look efficient, has it been tested fully, anything else you would like to comment on? (5)

Once the code is first executed, I receive an AssetNotFoundException, which refers to the JavaFish/Food texture. The error comes from the run method inside the Simulation class

However, the commenting is up to standard with the Javadoc tagging system (Includes method comments with explanation of function and parameters). The Javadoc gives sufficient details, including class and method descriptors for each class. 

There are two unit tests included with this code. One which tests the Renderable class and the test conditions are about position, orientation and the constructions of a renderable object. All tests for this class seem to pass without error. 

The other test is about the mousehandler class, which is a test harness that tests if the listeners are called with the mouse pointer when clicked.

1. Does the UserCode.pets.FoodMgr class exhibit loose coupling? Explain your answer. (5)

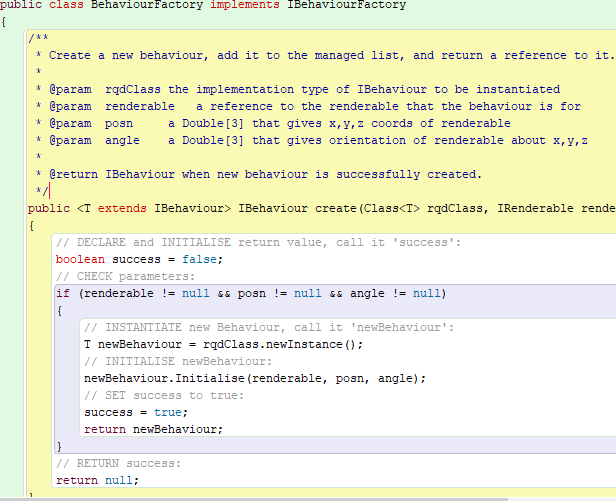
In OOP, coupling is the principle that states one object shouldn’t directly alter the state or behaviour of another class (Lambert, 2012). Objects which follow this principle by being independent from one another and don’t directly modify other objects are described as being *loosely coupled* (Lambert, 2012).

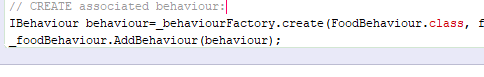
Having your code loosely coupled is a great way to make your code more flexible and easier to work with (Lambert, 2012). Tightly coupling a class is what happens when you don’t follow this principle, making your classes dependent on eachother and harder to reuse (Lambert, 2012).

The food manger class exhibits loose coupling, because the only knowledge that it has of other classes such as Renderable, comes from the interface of that object. The means that they are mostly independent of eachother.

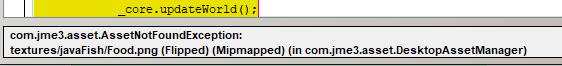
However, other classes are directly using the foodmanager class, rather than programming to the interface, which goes against the dependency inversion principle. The idea of the dependency inversion principle is to make sure that high-level modules don’t depend on low-level modules, instead these should both depend on abstractions(Janssen, 2018.).

1. The UserCode.pets.BehaviourMgr class violates the Single Responsibility principle. Implement a modification to the software that removes one inappropriate responsibility from that class. (5)

The single responsibility principle states that any class that the developer makes cannot have more than one responsibility Janssen, 2018.). For example, a class that creates an entity and enacts their behaviour would not be following this principle because it has two completely separate jobs Janssen, 2018.). A benefit of using this principle is that helps prevent unexpected side effects of future implementations. Because if you need to alter a responsibility within a class or the class itself, then having one responsibility per class means that other responsibilities won’t be affected Janssen, 2018.). The BehaviourMgr violates the single responsibility principle because it has the responsibility of updating the behaviours and creating the behaviours. To get around this I created a Behaviour factory which takes the creating behaviours responsibility away from the BehaviourMgr. 

Then in the food manager I create a behaviourfactory and use this to create a behaviour and add it to the list in behaviourMgr so it can be updated.

1. Diagnose the bug mentioned in the introduction above and implement a change to the code that resolves it. (5)

.



The way I fixed this exeption is by going into the foodmgr class and changing the string that holds the location of the fish food texture. Was “textures.javaFish/Food.png”.

References

Janssen, T., 2018. *SOLID Design Principles Explained: Dependency Inversion Principle With Code Examples*. [online] Stackify. Available at: <https://stackify.com/dependency-inversion-principle/> [Accessed 16 March 2020].

Lambert, S., 2012. *Quick Tip: The OOP Principle Of Coupling*. [online] Game Development Envato Tuts+. Available at: <https://gamedevelopment.tutsplus.com/tutorials/quick-tip-the-oop-principle-of-coupling--gamedev-1935> [Accessed 20 March 2020].

Janssen, T., 2018. *SOLID Design Principles Explained: The Single Responsibility Principle*. [online] Stackify. Available at: <https://stackify.com/solid-design-principles/> [Accessed 16 March 2020].