# COMP2331 Object Oriented Design and Development

## Portfolio Design Documentation

Student Number: 18010943

### Introduction

*All documentation has some kind of introductory text. We usually write this last. Don’t spend too long on it, just a brief summary of the contents of the document*.

This documentation describes and analyses a problem given by the client, which asks for a simulation of a aquarium full of moving fish, bubbles and fish food.

### Problem Analysis

*This should be revised from milestone 2 to accommodate the new client brief, applying responsibility driven design (including class discovery) and hierarchical software design as part of the analysis.*

Client Brief

Key:

Commonalities across all fish

Commonalities across most fish

**Actions/Behaviours (Verbs)**

*Classes(Nouns)*

*JavaFish*

Must:

* Have at least 10 JavaFish tokens.
* Scaled by 0.15 or less.
* **Swim** horizontally back and forth.
* **Face** in the direction of *movement.*
* **Blow** *Bubbles* from mouth.
* **Randomly Select** speed between 0.005 and 0.05.
* **Randomly Select** *Position* within the bounds of the aquarium.
* Switch direction of *movement* when screen edge hit.

Desirable:

* If fish **collides** with *FishFood*, the food should be removed from the 3D world
* Fish will **grow** slightly after **colliding** with *FishFood*

*SeaHorse*

* **Swim** diagonally up and down the screen.
* **Face** in the direction of *movement*.
* **Blow** *Bubbles* from mouth.
* **Randomly Select** speed between 0.005 and 0.05.
* **Randomly Select** *Position* within the bounds of the aquarium.
* Switch direction of *movement* when screen edge hit.

Desirable:

* If fish **collides** with *FishFood*, the food should be removed from the 3D world
* Fish will **grow** slightly after **colliding** with *FishFood*

*Urchin*

Must:

* **Swim** horizontally back and forth.
* **Face** in the direction of *movement*.
* **Randomly Select** speed between 0.005 and 0.05.
* **Randomly Select** *Position* within the bounds of the aquarium.
* Switch direction of *movement* when screen edge hit.
* **Spawned** at the bottom of the aquarium

*Piranha*

Must:

* **Swim** horizontally back and forth.
* **Face** in the direction of *movement*.
* **Randomly Select** speed between 0.005 and 0.05.
* **Randomly Select** *Position* within the bounds of the aquarium.
* Switch direction of *movement* when screen edge hit.

Desirable:

* If fish **collides** with *FishFood,* the food should be removed from the 3D world
* Fish will **grow** slightly after **colliding** with *FishFood*

*FishFood*

Must:

* **Spawn** at *mouse* *position* when left click is **pressed**
* *FishFood* will **fall** to the bottom of the aquarium at an appropriate speed

Desirable:

* If fish **collides** with *FishFood,* the food should be removed from the 3D world

After creating the class discovery for the client brief, I noticed a lot of commonalities between each entity.

Each fish must

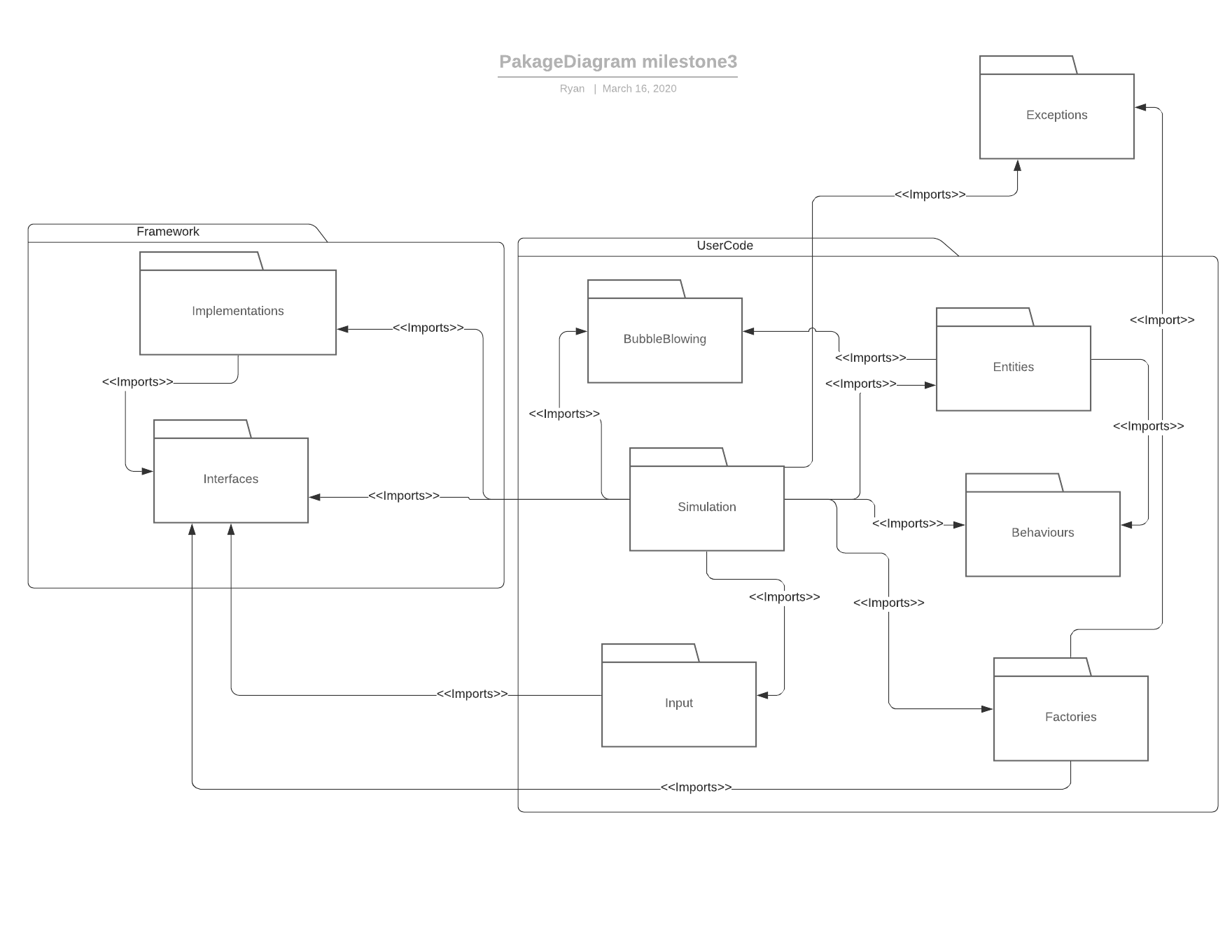
* Swim either horizontally or vertically
* Blow bubbles (except Urchin)
* Spawn into the 3D world
* Have a Position and orientation
* Random Speed and position

Because all entities need to spawn into the 3D world and they need a position and orientation to show up after spawning, I can use an interface that each entity that can spawn uses. Also due to the only things separating the fish from each other being their texture, size and behaviour, I do not need to create a separate class for each fish, but instead I can create one generic class for fish. There still remains the problem of some of the fish having different swim behaviours. To overcome this, I could make the fish’s behaviours externally, so a horizontal behaviour class and a vertical behaviour class.

There still remains a problem with the urchin being the only fish that can’t blow bubbles. One resolution could be to create 2 swimmable classes one which is for fish with a mouth and one without. This would mean a lot of repeated code for 1 difference and this does not seem to be practical. I could create an abstract class that both classes inherit from to prevent duplicate code, but this will introduce some unwanted dependencies between my entities, which in turn couples my code. To conclude, I believe that the first solution where I create 2 fish classes which both implement the same interface will be the better one for my project, as one of the principles I am looking to follow is the favour composition over inheritance principle. This solution includes one class for fish with mouths and one for Urchin.

For blowing bubbles and mouse input, I could use an observer pattern and use Simulation as a listener, so that the bubbles and fish food can be added to the IUpdatables list inside simulation. For an observer pattern I will need a publisher class and a listener class, so I can create a class for each of these . The publisher will subscribe the listener to the event and the listener will be listening for the event.

Due to the amount of passing position and orientation variables, I will create a Vector3 class which holds 3 doubles in one object, making it easier to pass data, as it is just one variable.

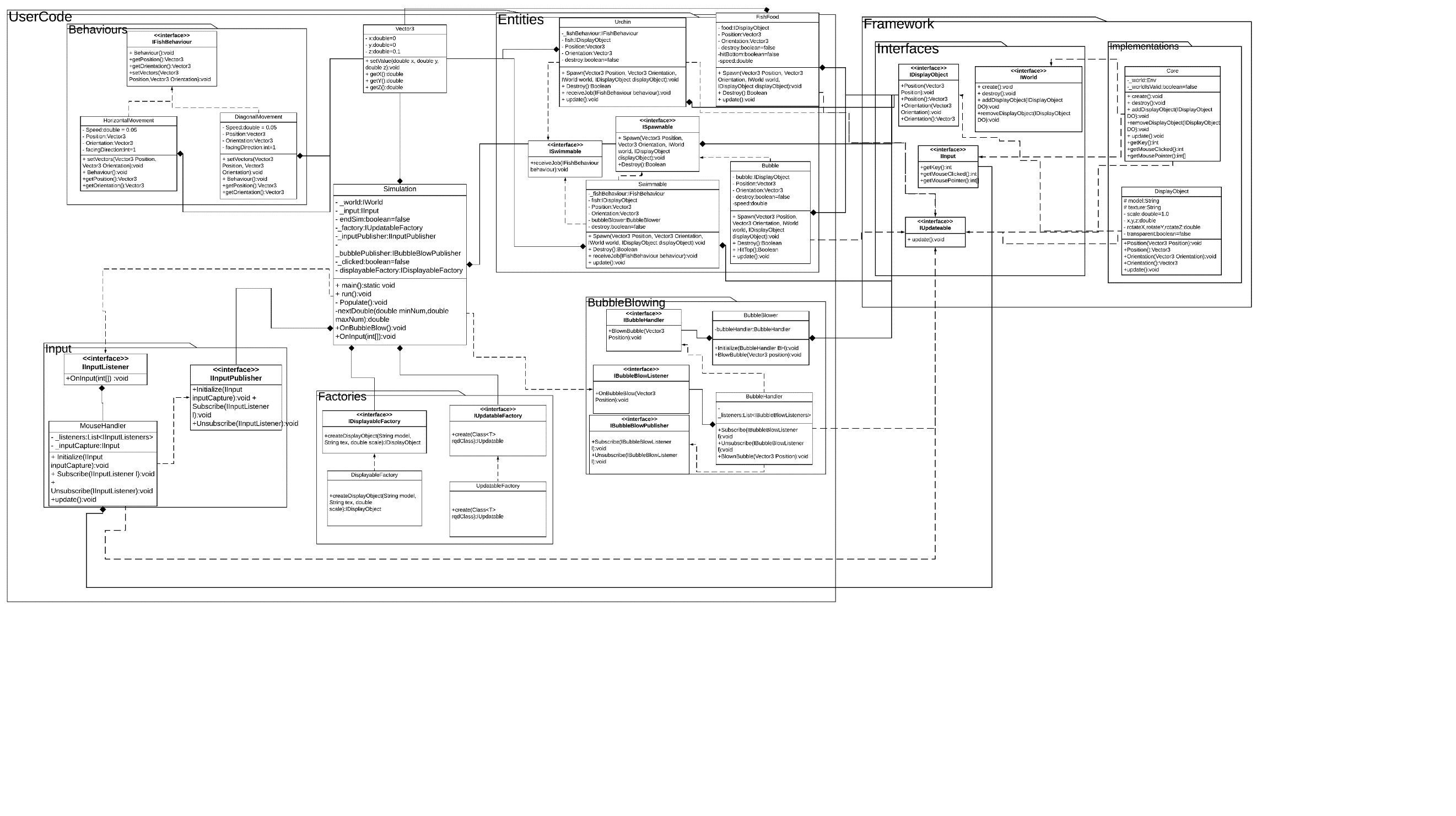


Hierarchical software design

The above image is a package diagram of my software. It shows the dependencies between each package and which package will import which. Behaviours will contain all the fish’s behaviours such as diagonal movement and horizontal movement. Entities will contain all of the entities including bubble, Swimmable, FishFood and Urchin. It also will contain the Interfaces for class for these entities, for example all fish will implement the ISwimmable interface and all spawnable entities will implement ISpawnable . UserCode will contain the simulation and the Vector3 class. Framework contains the base code that uses Env3D to create the simulation. IDisplayObject, DisplayObject, IWorld , IUpdatable, IInput and Core are all classes that reside in the Framework package. These are all spit into a package for implementations and one for interfaces. The Factories package contains a factory for IDisplayableObjects and one for IUpdatableObjects. Input contains all the classes involved in input management such as MouseHandler, IInputPublisher and IInputListener. The only class that imports this package is simulation class. The BubbleBlowing package includes everything needed for blowing bubbles.

## 3.OO Software Design

*This should be revised from milestone 2 according to the above analysis.*



In my design, I have gone with the approach of favouring composition over inheritance, which means that I have used interfaces where posible to encapsulate my classes to at best hide the content of the classes that I am composing. I have chose to create a factory for both displayobject and Iupdatable object, this is so that I can follow the dependency inversion principle so that any dependencies that the objects would have had with simulation if they were instantiated inside simultion will be avoided. This will help with the coupling of my code aswell. The way I have design my behaviours is that they are using the strategy behaviour pattern, so that the behaviours are interchangable at runtime. I will use an observer pattern for the input and blowing bubbles. This means that I will need to make simulation as a listener for both events, so that the simulation can deal with creating fishfood and bubbles by adding them to the updatable list. For The fish I have designed a generic class which works for all fish that can blow bubbles, this is called swimmable, which implements the ISwimmable, IUpdatable and ISpawnable interfaces.

## Test Strategy

*This should be the same as used in milestone 2 – for completeness, you may wish to change it in response to any feedback given.*

## Learning Journal

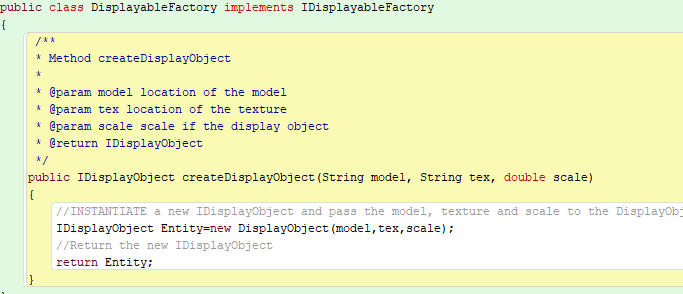
*In this section, you need to evidence knowledge and application of the concepts being assessed. Follow the guidance under each of the sub-headings below.*

#### OO Design Principles

*Explain the five SOLID principles +1 - for each principle: name it; state what the principle is; explain what it achieves; and illustrate this discussion by highlighting an element of your software that adheres to it.*

**S**ingle Responsibility Principle

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.). Also, other classes that are dependent on a class with multiple responsibilities might need to be updated or recompiled when you alter one of the responsibilities, even though they are only dependent on one of the other responsibilities within the class Janssen, 2018.). Having a class that only has one job to do also improves your code’s cohesion and prevents classes from becoming an unmaintainable, monolithic mess that is difficult to re-use Janssen, 2018.).

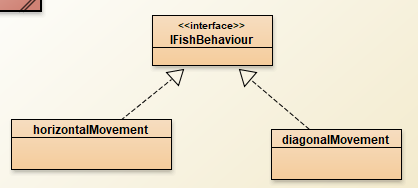


Here is an example from my code where I am adhering to the Single responsibility principle. The classes Single responsibility create an IDisplayObject and return it.

**O**pen/Closed Principle

Open/Closed principle is based around the concept that that objects should be open for extension and closed for modification (Janssen, 2018.). What this means is that your code should be able to add new functionality without modifying the already existing code Janssen, 2018.). There are two approaches to this principle. The first being inheritance, which entails having an abstract version of a behaviour, then having children add functionality to the behaviour without changing the contract between users of the class Janssen, 2018.). There is an issue with using this approach however, as due to the side effect of using inheritance, your code will become tightly coupled if the subclasses depend on the implementation of the parent class Janssen, 2018.). The second approach would be to use polymorphism, which uses interfaces instead of super classes to implement new behaviours without worrying about changing the code to use them Janssen, 2018.). The use of interfaces allows for a new level of abstraction which provides loose coupling to your code. This is the most optimised version of open/closed principle, as interfaces are closed for modification and you can create new implementations to extend the functionality of the software Janssen, 2018.).

In my code I have implemented the open/closed principle for my fish behaviours as I can add new behaviours (open for extension) without having to modify any current code (closed for modifications ). This is fantastic for fish behaviour as I can re-use the same structure for another project and add new functionalities and behaviours without having to change any code.



**L**iskov Substitution Principle

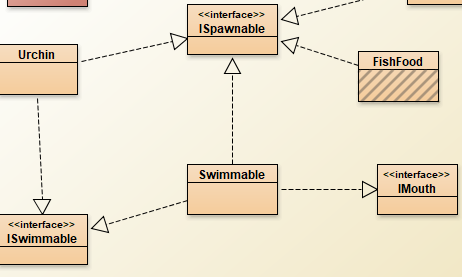
Liskov Substitution Principle states that derived classes must be substitutable for the base class(Janssen, 2018.). This requires the child classes to act in a similar way as the objects of the superclass(Janssen, 2018.). The key benefits of this principle are that is loosens coupling and improves code reuse(Janssen, 2018.).

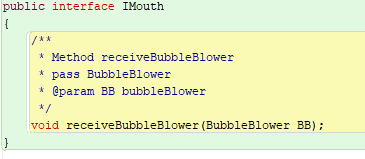
My project does not use the Liskov substitution principle, but If I was to use it, an example of this would be to add a base class of each of the behaviours and then use overriding to override the behaviour of the base class to make it so that they can be interchangeable with the base class.

**I**nterface Segregation Principle

The Interface Segregation Principle’s one rule is that “*clients should not be forced to depend upon interfaces that they do not use”* Janssen, 2018.). The goal of this principle is very similar to the goal of Single Responsibility Principle, where reduction of side effects and frequency of required changes are prioritised by splitting the software into several individual parts Janssen, 2018.). An example of a violation of this principle would be if an animal interface had three methods: Chew(),Run() and Bite(). The class that implements this interface is called Snake, but Snake does not use the Run() method that it is implementing. To overcome this, you would need to create other interfaces that snake can implement for example Reptile and make the animal interface more abstract, so it works with all animal implementations.

In my code I used the interface segregation principle to stop the Urchin class from implementing a bubbleBlower, as it does not need to receive a reference to bubble blower because it does not have a visible mouth. So to prevent Urchin from implementing a bubbleBlower receiver through ISwimmable, made a separate interface that is only being used on fish with mouths. This interface is called IMouth and It contains one method which gets a new bubble blower. Because Urchin does not implement IMouth, it no longer has to worry about implementations that it is not using.



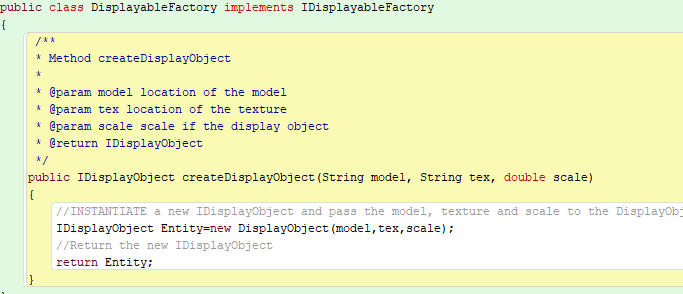


**D**ependency Inversion

Dependency Inversion’s general idea is that

* *high-level modules shouldn’t depend on low-level modules. Both Should depend on abstractions(* Janssen, 2018.).*.*
* *Abstractions should not depend on details. Details should depend on abstractions* (Janssen, 2018.).

Following this principle will lead to your code being reusable and unaffected by any alterations made in low-level modules. The Abstraction that is required of this principle will decouple the high-level and low-level modules from each other because they are not depending directly to each other but instead depend on the abstraction of the modules(Janssen, 2018.). One key part to note about this principle is that it helps us respect all the other SOLID principles but primarily the Liskov substitution principle and the Open/Close principle(Janssen, 2018.). Due to the abstractions that dependency inversion creates and the different implementations of these abstract dependencies, natural extension points are put in place, which allow for the modification of behaviours without altering the class itself(Janssen, 2018.). This is a requirement of the Open/Closed principle and in turn almost enforces this principle naturally. Dependency inversion also makes you use subtyping in the correct manor where you are programming to the high-level module rather than the class itself, which follows the Liskov Substitution principle so that you can replace the implementations with others of the same interface without the code breaking(Janssen, 2018.).



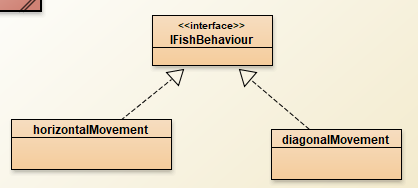


One place where I have adhered to the dependency inversion principle is the IDisplayableFactory. Inverting the dependencies from classes like simulation, where they are calling new DisplayObject themselves which makes DisplayObject dependent on what ever class in calling new DisplayObject. But by implementing a factory that creates DisplayObject from an abstraction, this means that both the low-level module and high-level module are depending on abstractions.

Favour Composition over inheritance

This Principle tries to enforce a rule where communication between objects must be handled with composition rather than using inheritance. This will benefit your code greatly as it will reduce the dependencies between classes and loosen the coupling of the code.

My software uses 0 inheritance, which means that I have been following this principle by using composition for communication instead.



Here is an example where I could have used inheritance by making fishBehaviour a parent class instead. But this would tighten the coupling in my code and create unwanted dependencies. So instead I have used interfaces and composition for when I want to use the behaviours.

#### Design Patterns

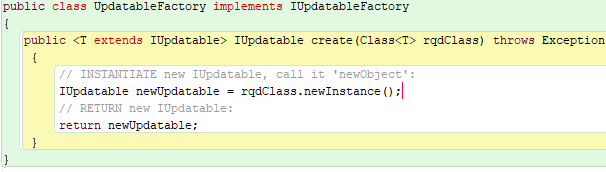
*Name three of the design patterns that you have employed in your design, and highlight where these patterns are implemented in your software. Explain each design pattern by stating the ‘Name’, ‘Classification’, ‘Intent’, and ‘Structure’.*

Factory Pattern

Classification-Creation pattern (Carr, 2009)

Intent-Used to replace constructors, taking on the job of creating a class and determining the type of that instantiated object at run-time (Carr, 2009).

Structure-Has a return type of the object that will be instantiated.





In my code I have used generics so that I can instantiate any class by name as long as they are an IUpdateable object. The factory pattern is extremely useful for dependency inversion has they are taking away any dependencies from non-abstract classes. I have used this factory in the making of all my IUpdatableObjects including Swimmable, mouse handler and bubble.

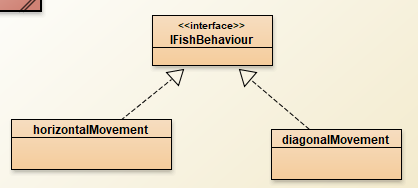
Strategy Pattern

Classification-behaviour pattern (Carr, 2009)

Intent- Used for interchangeable behaviours or algorithms which can be switched between at run time (Carr, 2009).

Structure-At least 2 classes of the same type, which can be substituted with each other, usually be subtype polymorphism.

In my code I have used this pattern for the behaviours of the fish, because I realised that some fish have the same movement behaviour, where some do not. With this in mind I needed a way to use the same interface for all behaviours but have the ability to change the instance of the behaviour at runtime. So, my implementation of this pattern included a IFishBehaviour interface and 2 different behaviours one for horizontal movement and one for diagonal movement. When I instantiate a new fish, I also give them a behaviour and depending of the fish that is being instantiated the movement behaviour will be different.



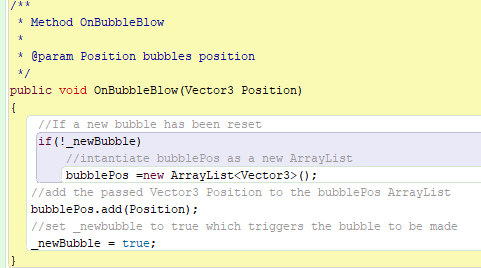


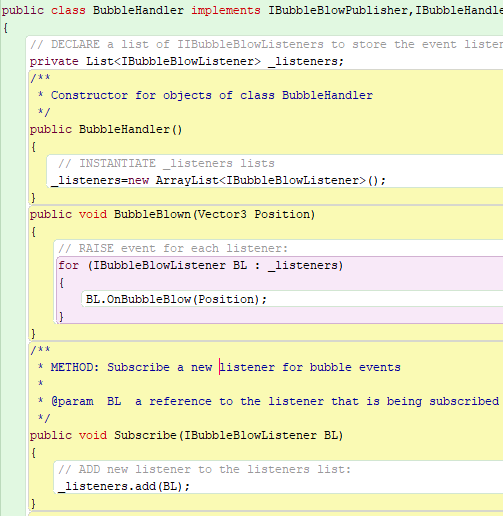


Observer Pattern

Classification-behavioural pattern (Carr, 2009)

Intent-The observer allows an object to publish its state to other objects and these objects can subscribe to the event to be notified with any changes (Carr, 2009). This can be extremely useful for behaviours that only happen every now and again.

Structure-There needs to be at least 2 entities involved in this pattern. One which publishes the event to listeners and listener, which subscribe to the event and get notified when something changes 



In my code I have used this pattern for blowing bubbles and for handling mouse input. For blowing bubbles, the listener is the simulation class. This is so that the Simulation can add the bubbles to the Updatables list when they are created, which is need if I want the bubbles to move. There is a bubble blower which counts to a certain number, then tells the bubble handler that it can (which is the publisher) fire the bubble blow event and the handler will notify any listeners subscribed to the event(Simulation) Simulation then creates the bubble and adds it to the updatable list.

#### Efficient Design

* *Briefly explain the ‘divide and conquer’ approach to software design, and highlight how this approach influenced your design (eg in terms of package structure).*

Divide and conquer is the process of break a problem apart and dividing it into simpler, more maintainable problems. A problem is divided into separate parts, so that you can handle each part separately. Each of the smaller parts of the problem being handled is much less complex than dealing with the problem as a whole, thus we can “conquer” it.(Makabee, 2011). In my design I have used this to work out how to trigger an event that told the simulation to create a bubble, then add it to the updatables list. At first this problem seemed impossible, but I split the problem into different sections that could be comprehended easily.

* *Briefly explain what ‘agile discovery’ is, and highlight any parts of your software design that were achieved through use of this process.*

## References

*List your references here using the Harvard referencing style.*

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].

Makabee, H., 2011. *Divide-And-Conquer: Coping With Complexity*. [online] Effective Software Design. Available at: <https://effectivesoftwaredesign.com/2011/06/06/divide-and-conquer-coping-with-complexity/> [Accessed 16 March 2020].

Janssen, T., 2018 *SOLID Design Principles Explained: The Open/Closed Principle With Code Examples*. [online] Stackify. Available at: <https://stackify.com/solid-design-open-closed-principle/> [Accessed 16 March 2020].

Janssen, T., 2018. *SOLID Design Principles Explained: The Liskov Substitution Principle With Code Examples*. [online] Stackify. Available at: <https://stackify.com/solid-design-liskov-substitution-principle/> [Accessed 16 March 2020].

Janssen, T., 2018. *SOLID Design Principles Explained: Interface Segregation With Code Examples*. [online] Stackify. Available at: <https://stackify.com/interface-segregation-principle/> [Accessed 16 March 2020].

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].

Carr, R., 2009. *Gang Of Four Design Patterns*. [online] Blackwasp.co.uk. Available at: <http://www.blackwasp.co.uk/gofpatterns.aspx> [Accessed 16 March 2020].

## Appendix: Design Document 2

*Append here your Design Document for the second milestone.*

### Introduction

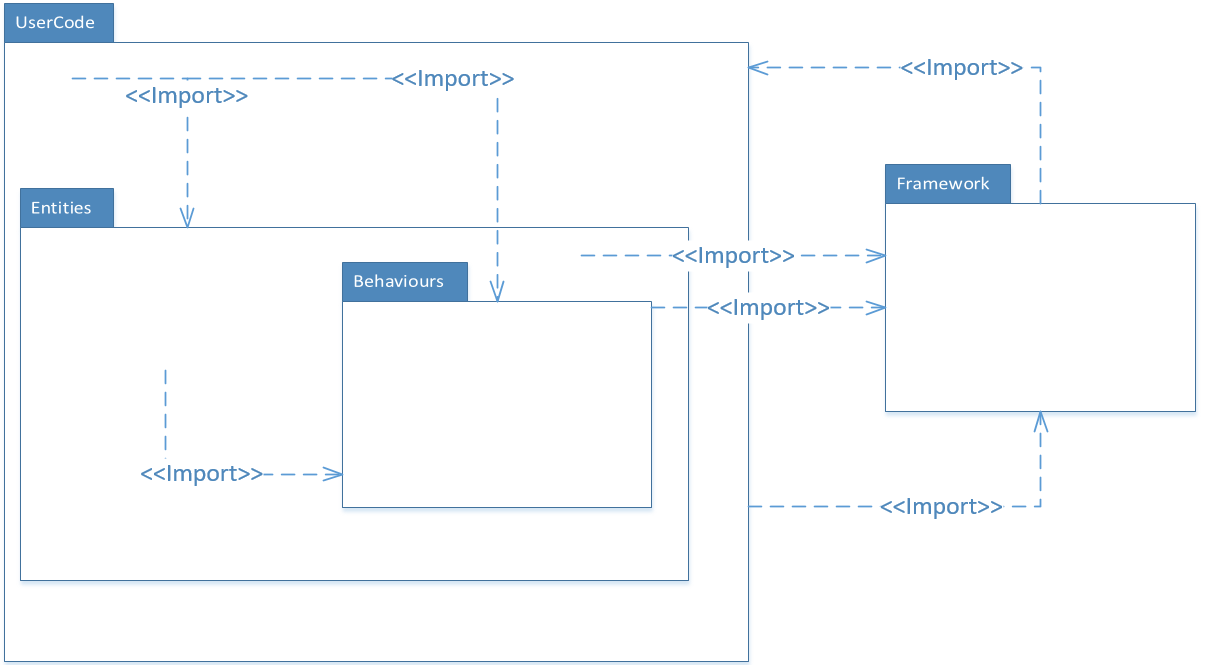
This document contains a detailed analysis on the software that has been outlined in the specification. The program essentially displays 10 different aquatic creatures within an aquarium and allows them to all swim around within that aquarium while blowing bubbles.

### Problem Analysis

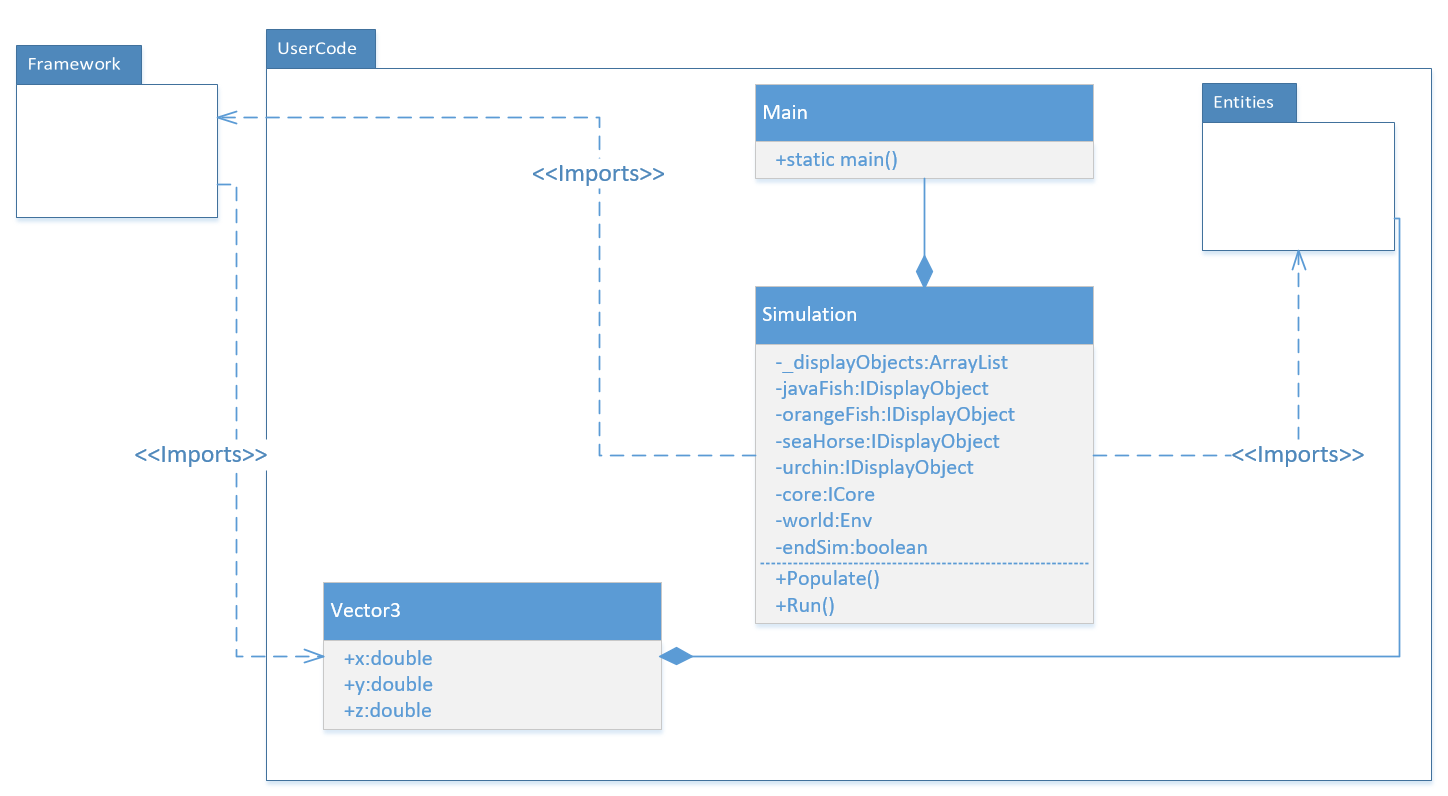
*This should be revised from milestone 1 to accommodate the new client brief.*

The client brief states that there must be 10 moving fish with at least a JavaFish, SeaHorse and an Urchin that all move within the aquarium. The fish do not interact with each other in anyway, so there is no need to implement collision between the fish. The client brief also states that each fish must be facing the direction that they are swimming, therefore I must add some code that alters the fish’s orientation once they have successfully reached the edge of the screen. The JavaFish and Urchin must both swim horizontally, while the SeaHorse must swim diagonally. All fish’s speed must be randomly chosen on start-up and must be between 0.005 and 0.05. All of the above problems have been solved in the first milestone; therefore, I will use similar solutions within this program. The SeaHorse and JavaFish must be randomly positioned within the boundaries of the aquarium. This problem could be solved in a similar manor to how I solve the random speed problem. In which case I will create a method that is shared by all entities, which randomly selects numbers for the x and y based on the certain parameters given. Meanwhile, the client brief does not state whether they want the Urchin to have a random x position, only that they want the urchin to be placed at the bottom of the screen. Therefore, an assumption here would be that I can make the Urchin have a set position instead of a random one. And because there will already be a method named setPosition for the random position, I will be able to use method overloading to use the same method name but use different parameters and return value. Possibly the biggest problem to solve within the client brief is to make each fish with a visible mouth emit bubbles. Assuming that this problem will require the use of collections (lists, arrays), there will need to be a function that allows the removal of the bubbles from the array and the world. Otherwise, this will cause the program to crash due to too many objects being left unmanaged. Luckily, this method already exists within the core framework, so I will just need to call upon this method to remove the bubbles. The desirables within the client brief are to give the aquarium some appropriate sounds, for example the bubbles making a sound when they pop. Solving this should prove to be an easy task, because in theory all I need to do is play a sound during the event of the bubble reaching the top of the screen. The other desirable is to allow some of the fish to gain a burst of acceleration at random intervals. Thankfully, in milestone one I gave a similar solution, where I made my orange fish have random bursts of acceleration and deceleration.

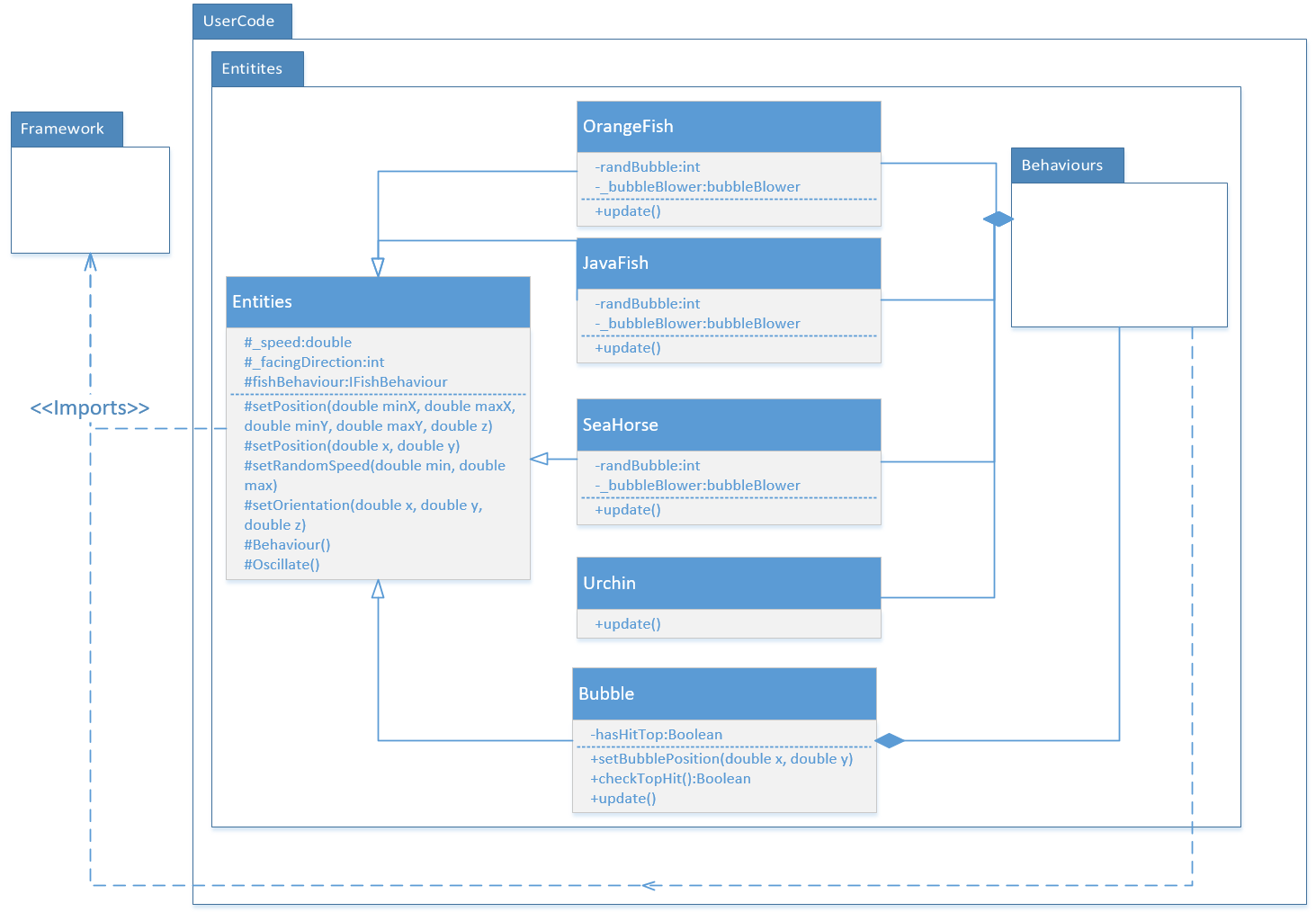
## OO Software Design



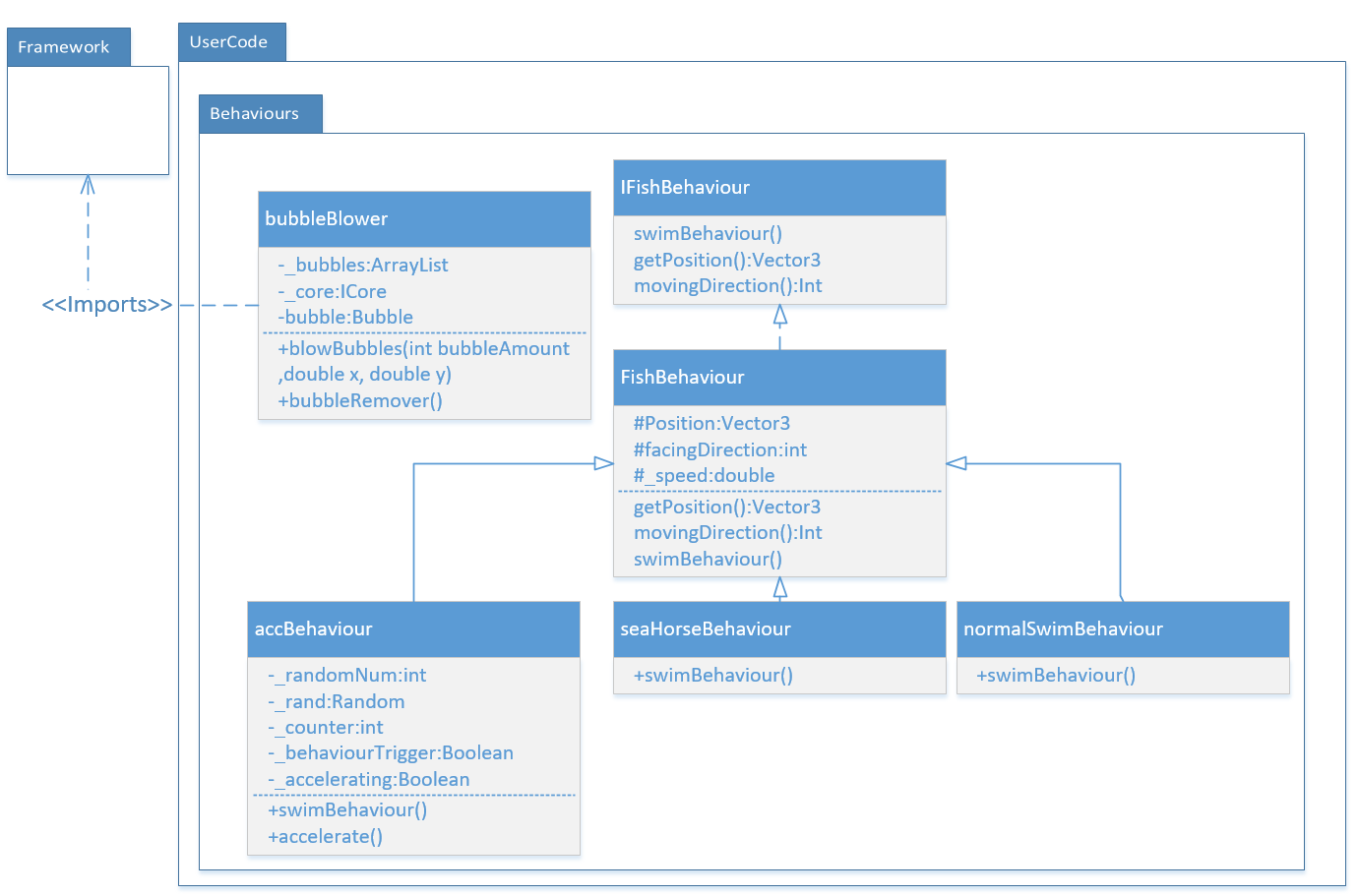
This is a package diagram of my software. It shows the dependencies between each package and which package will import which. Behaviours will contain all the fish’s behaviours such as acceleration and bubbleBlower. Entities will contain all of the entities including bubble, JavaFish, OrangeFish, SeaHorse and Urchin. It also will contain the parent class for all these entities and the class is an abstract class called entity. UserCode will contain the simulation, main class and the Vector3 class. Framework contains the base code that uses Env3D to create the simulation. IDisplayObject, DisplayObject, ICore and Core are all classes that reside in the Framework package.



Here is a class diagram show the hierarchal structure of my UserCode package. There will be 3 different classes called Main, simulation and Vector3. Vector3 holds the x,y and z position of an entity. Therefore, all the entities will have a ‘has a’ relationship with Vector3. The Main class contains a static main method, will makes the software executable. It composes Simulation. Simulation is the main hub for creating the fish. Therefore, Simulation will have a ‘has a’ relationship with most of the entities within the software. All the fish will have a static type of IDisplayObject, which means the Simulation class will import IDisplayObject. It also will contain a reference to ICore and so will need to import Framework.ICore. The Populate method will call upon a method inside Core to add all the entities to a list and the scene. The DisplayObject class will import Vecor3, so that all the DisplayObjects can use the same Vector3 member.



The Entities package resides inside the UserCode package and contains the OrangeFish, JavaFish, SeaHorse, Urchin, Bubble and Entities classes. All the fish have similar members and methods, because they inherit a lot of methods from the Entities Class. All the fish also will have a ‘has a’ relationship will most of the behaviours. Entities imports DisplayObject, because it will extend DisplayObject from the Framework package. bubble Behaviour, which resides in the behaviours package will have a ‘has a’ relationship with the Bubble class.

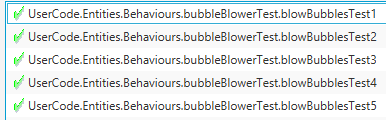


The behaviour package will use an interface called IFishBehaviour which will be implemented by the fishBehaviour class. The fishBehaviour class will have 3 child classes that have an ‘is a’ relationship. These will be called seaHorseBehaviour,accBehaviour and normalSwimBehaviour. These will all control how the fish will swim. The bubbleBlower class will import the Framework package to use core and IDisplayObject. This is so that the bubbles can be placed on the scene and added to the DisplayObjects array inside DisplayObjecs.cs.

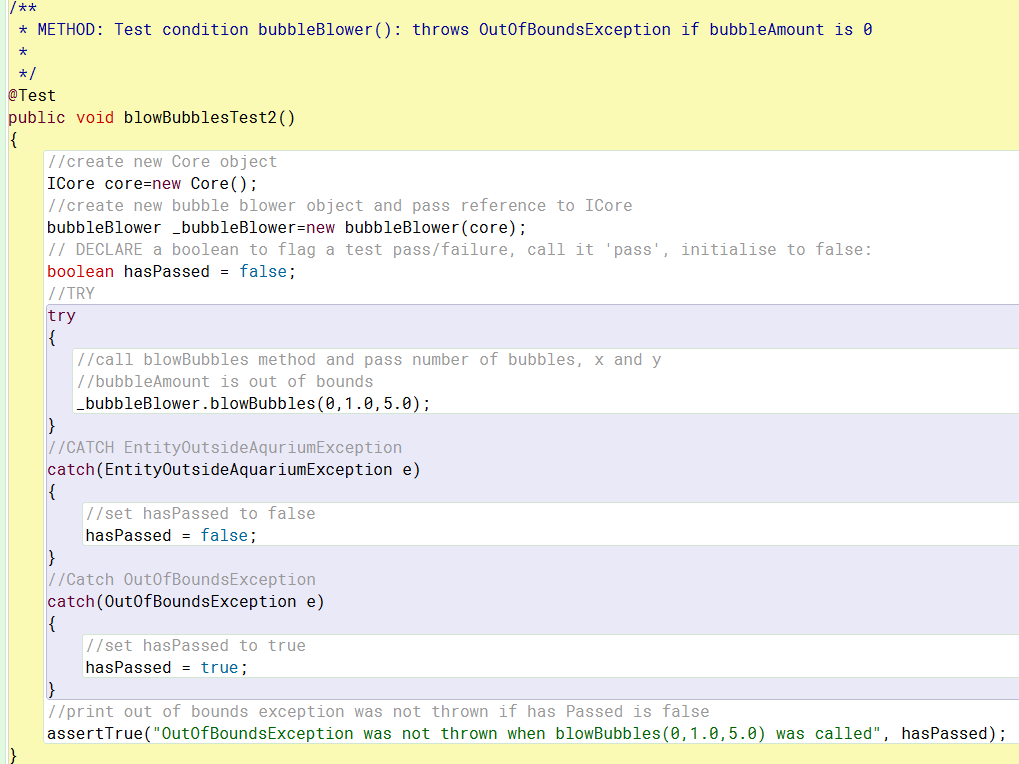
## Test Strategy

This software will include a unit test, which tests the bubbleBlower class and the parameter that will be passed to the blowBubble() method. The parameters are bubbleAmount which is an integer that defines how many bubbles will be created. The other 2 parameters are x and y values, and these are both double. To test these parameters, I will need to check if these parameters are in between a certain amount and if the value goes outside this amount, then I will throw and exception. Therefore, I will need to create some user exceptions for each parameter. For the bubbleAmount I will use a simple out of bounds exception that is thrown when the bubbleAmount is too high or too low. For the x and y values, to test this, I can just check to the if they are less that the lowest x and y values or more than the highest x and y values of the aquarium. Then if the position of the fish is outside these bounds, then I will throw an EntityOutsideAquariumBounds exception. The unit tests will test a number of different combinations that the parameter could contain, in hopes to test for every possible outcome. This way I will receive an exception no matter what numbers are given to the parameters of blowBubbles().

Unit tests in action



The Unit test contains 5 different tests , testing against conditions to see of the exceptions get thrown or not. All 5 tests have passed and work as expected.



## Learning Journal

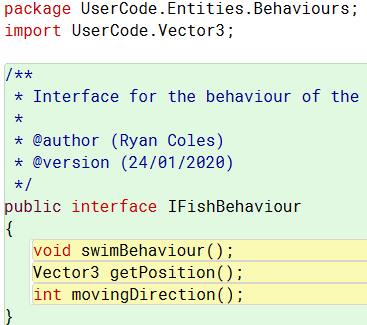
#### Abstract Classes and Interfaces

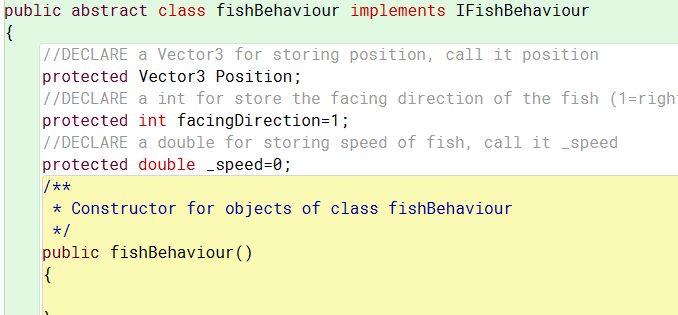
*Explain what Abstract Classes and Interfaces are, why they are used, and indicate where in your documentation and/or software they are used.*

**Abstract classes** are classes that cannot be instantiated, but they can be subclassed. What this means is that an abstract class cannot be used by another class, unless you use inheritance and/or interfaces. Abstract classes become useful when you expect the child classes of the abstract class to use common methods and fields (Deep, 2019).

**Interfaces** are similar to abstract class, as they also cannot be instantiated. Methods inside an interface do not have a body. This job is for the class that is implementing the interface. Interfaces cannot contain a constructer. This is because the interface cannot directly be used in the creation of objects (W3schools.com, n.d.).

The reasons for using an interface are to make the program more secure, by hiding certain details and only displaying the details that are important (W3schools.com, n.d.). Another really useful feature of an interface is that you can have one class that implements multiple interfaces. Therefore, a class that instantiates the class that is implementing multiple interfaces will have access to methods and members from each interface. Most languages, including Java does not support multiple inheritance, which would give the same results has implementing multiple interfaces, which is another reason to use interfaces (W3schools.com, n.d.).

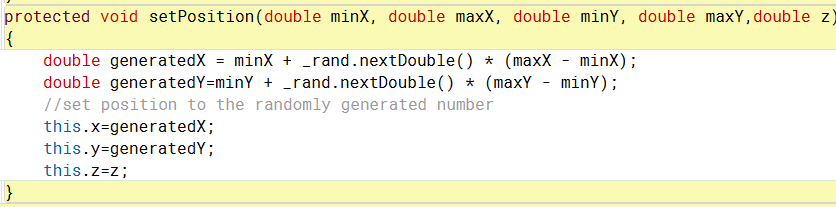
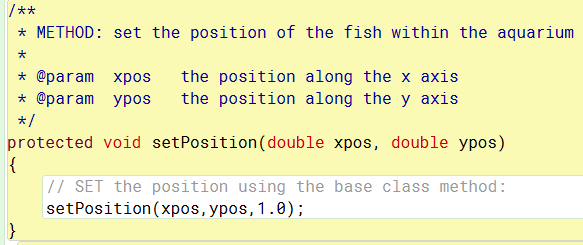




Here are examples of where I have used an interface and an abstract class. When using a fishBehaviour I’m able to use the interface to hide certain attributes and use the abstract class to have all my child class inherit from. Using these two methods of OOP improve my reuse of code and boost the efficiency of my code.

**Polymorphism** is a concept of Object-Oriented-Program, which allows objects of different types to be accessed through the same interface and each object type can provide its own, independent implementation of this interface.

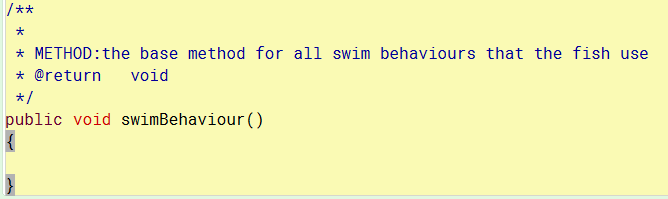
**Method overloading** is a static form of polymorphism, which allows the implementation of multiple classes with the same name, but with different parameters/body (Sharma, 2019). An example of where I have used this form of polymorphism is below.



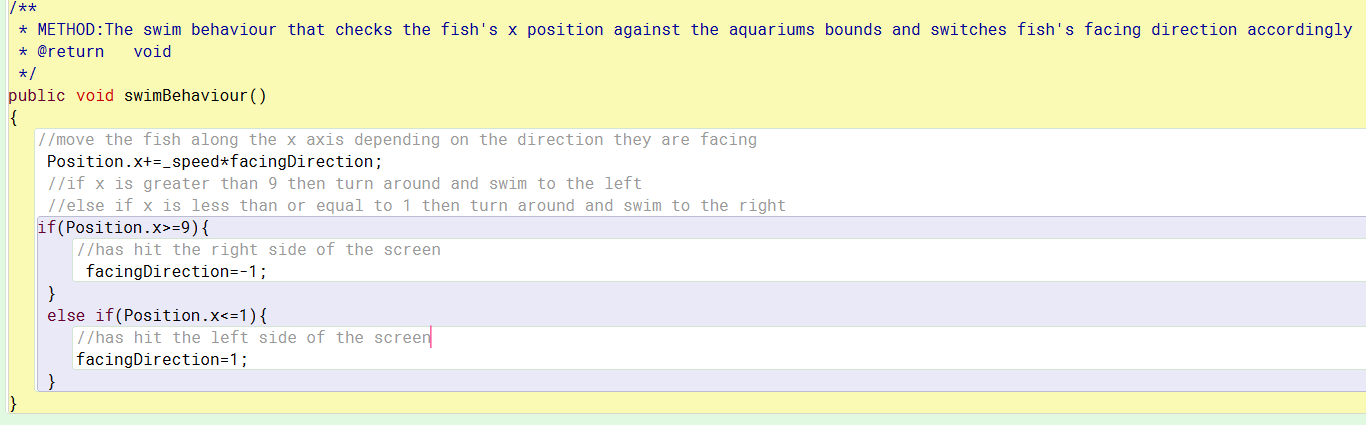
Here are two completely different methods inside the same class (Entity). They both share the same name, so they can be called in the same way, setPosition(), but require a different number of parameters to be passed. Overloading is very useful when the developer wants to use the simplest version of a method, which takes less parameters and makes the code more efficient. Then when it’s not possible to use the default values from the simplest version of the method, you can make a call the same method name and input different parameters.

Another form of polymorphism is **method overriding** and this allows a child class to use a method with the same name, parameters and return type as a method from its parent class. This proves to be very useful when you want the child class to have a method that includes in own implementation of an inherited method, without worrying about changing the method in the parent class (Singh, n.d.)

fishBehaviour.cs



normalSwimBehaviour.cs



seaHorseBehaviour.cs

Here are 3 different methods from 3 different classes. The method without any content in its body is the parent class. The other 2 are in the child classes. The reason why I have used swim behaviour in this manor is because, I can make a call to the swimBehaviour method and depending on which fish class I call it from, the behaviour will be different. For example, the JavaFish class uses the overridden method inside the normalSwimBehaviour class. However, if used in the seaHorse class, it will enact the behaviour from the seaHorseBehaviour Class.

Polymorphism also allows use to do something called **subtyping**. This is the act of changing the typing of an object reference on the fly to a typing that is derived from itself. In relation to my program, the JavaFish is ultimately derived from the IDisplayObject class. Therefore, JavaFish is an IDisplayObject and I can create a reference to IDisplayObject and instantiate it as a dynamic type of JavaFish. This I have created a new JavaFish object, but I can still pass it as an IDisplayObject. This proves to be extremely vital when adding each object to the \_IDisplayObjects list inside DisplayObjects, as managing objects becomes a lot easier, because I can iterate over one single List of IDisplayObjects instead of creating a list for each derived typing of IDisplayObject .

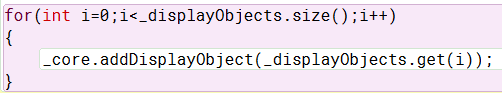
IDisplayObject object called javaFish

Simulation.cs



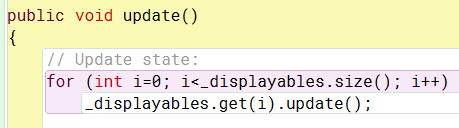
Java fish’s new dynamic type is JavaFish.





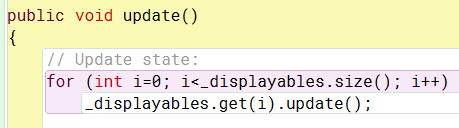
JavaFish can now be added to a list of IDisplayObject typing

Core.cs

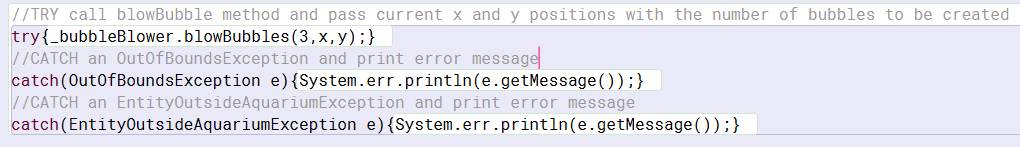


Now java fish is a displayObject , it can be used in Core without a reference to its own class because it’s an IDisplayObject and can be iterated over along with a collection of other IDisplayObjects.

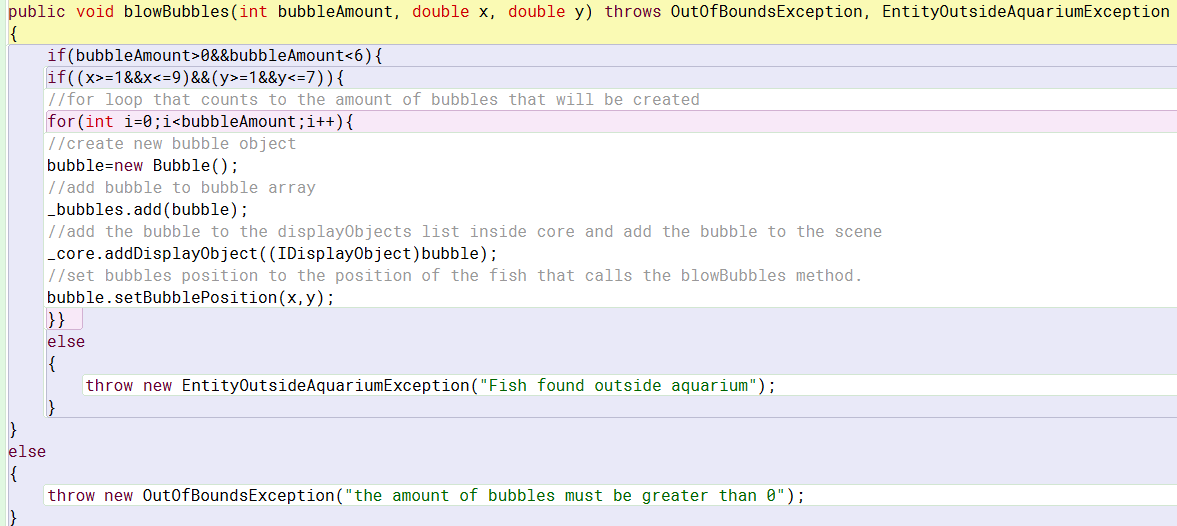
**A collection class** is used as a representation of multiple items of the same typing, formed together to create a single unit. There are many different types of a collection class, but the ones I have used in my software are the ArrayList class and List class. The purpose of a collection class is to make data manipulation and data store more efficient (Techopedia.com, n.d.). One the ways I have used it in my software is to store my entities within an ArrayList and then iterate over it to call the update method across each of my entities.



In programming an exception is an anomaly that occurs after program execution, for example if the software is looking for a file that doesn’t exist, it will print a read error (Computerhope.com, 2019). **Exception handling** is the procedure of responding to exceptions on execution of the software (Computerhope.com, 2017). The method I used to handle exceptions is the try and catch method. This involves putting the code that you think may cause an error in a try block, then use a catch block to collect any exception that occurs, then respond by displaying an error (Techopedia.com, n.d.).



Here I have a try and 2 catch blocks, which attempts to call upon blowBubbles inside the bubbleBlower class, passing 3 different values: the amount of bubbles, current x position and current y position.

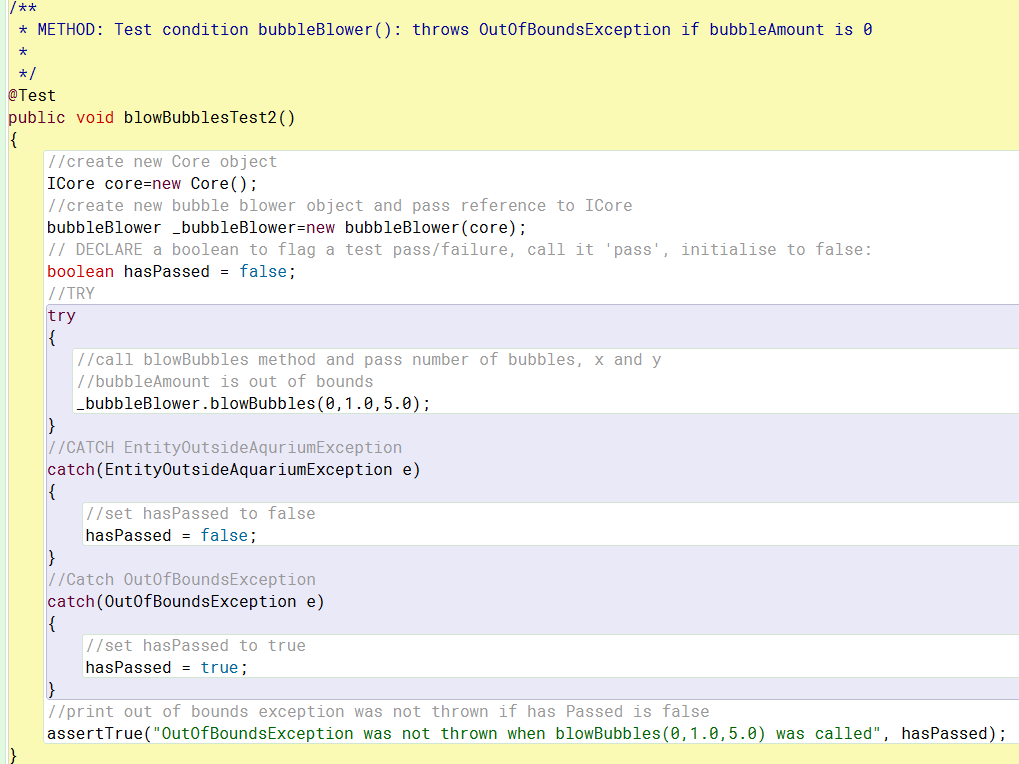


This is the code which throws the exceptions when certain conditions are met. So, of the bubble amount is in between 1 and 5, then the next line will be run, but if not, then the OutOfBoundsException will be thrown. The next line will check if the fish is within the bounds of the aquarium, but if not, then it will throw an EntityOutsideAquariumException.

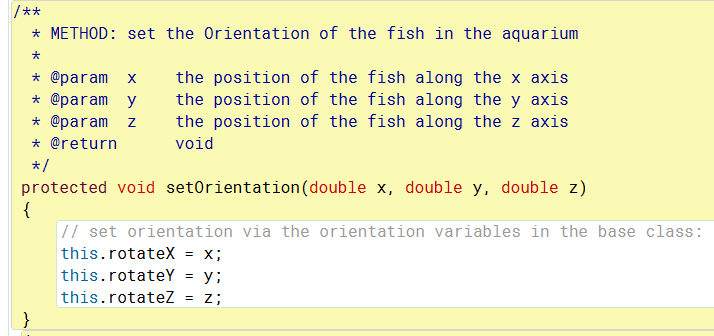
In programming a **Test strategy** is a protocol that must be followed in order to accomplish the test objectives and execution of the test types mentions in the test plan. The steps included in a test strategy are test objective test environment, test approach, automation tools and strategy, contingency play and risk analysis. The reasons to have a test strategy in place is to find the highest possible number of errors within software with a manageable amount of effort applied over a feasible amount of time (TechRepublic, n.d.).

**Unit Testing** is the testing of individual components within a piece of software. Unit testing is the first level of testing and is implemented before integration testing. The purpose of unit testing is to eradicate any bug within the software by testing every possible outcome for each unit until the program works as designed. Unit testing make code more reliable, reusable and more maintainable (Software Testing Fundamentals, n.d.).

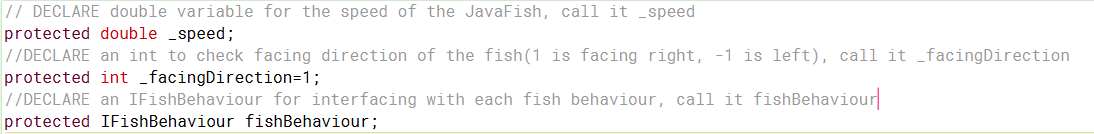
The unit test that I have employed into my software test the bubbleBlower class



For this unit test I am calling the blowBubbles method and testing the passed values against different conditions. In the image above I am making a Boolean and setting it to true after using the try block to pass a 0 as the bubbleAmount, which is out of bounds and therefore will be caught with an OutOfBoundsException. The assertTrue function will print a message saying that the exception was not thrown after testing it with parameters that should fail the test. I believe that implementing this test is ideal, because if the bubbleAmount was something like 20, then it would probably end with the software crashing, so having a test, that will test for this will be great. Also, if the value was 0, then there will be no bubbles being created, so having an exception that tells you exactly what’s going wrong with the code it perfect. And in relation to my test strategy, this is the perfect solution for testing my program.



This image shows Javadoc tagging, consistent code layout, parameter passing and appropriate comments. The @param is the Javadoc tagging and it allows the code documented easier, as the Javadoc will have to do let automated set up of the documentation.



This also shows appropriate comments and consistent code layout, as I am using DECLARE to declare member variable and the same layout for all my member variables. This is important as, for other developers looking at my code, having a consistent layout and appropriate comments makes it extremely easy for them to manage and understand.

## References

W3schools.com. (n.d.). *Java Interface*. [online] Available at: https://www.w3schools.com/java/java\_interface.asp [Accessed 26 Jan. 2020].

Janssen, T. (n.d.). *OOP Concepts for Beginners: What is Polymorphism*. [online] Stackify. Available at: https://stackify.com/oop-concept-polymorphism/ [Accessed 26 Jan. 2020].

Sharma, S. (2019). *Method Overloading In Java*. [online] C-sharpcorner.com. Available at: https://www.c-sharpcorner.com/UploadFile/fd0172/method-overloading-in-java/ [Accessed 26 Jan. 2020].

Singh, C. (2014). *Method overriding in java with example*. [online] beginnersbook.com. Available at: https://beginnersbook.com/2014/01/method-overriding-in-java-with-example/ [Accessed 26 Jan. 2020].

Deep, A. (2019). *When to Use Abstract Class and Interface - DZone Java*. [online] dzone.com. Available at: https://dzone.com/articles/when-to-use-abstract-class-and-intreface [Accessed 26 Jan. 2020].

Techopedia.com. (n.d.). *What is a Collection? - Definition from Techopedia*. [online] Available at: https://www.techopedia.com/definition/25317/collection [Accessed 26 Jan. 2020].

Computerhope.com. (2019). *What is an Exception?*. [online] Available at: https://www.computerhope.com/jargon/e/exceptio.htm [Accessed 26 Jan. 2020].

Computerhope.com. (2017). *What is Exception Handling?*. [online] Available at: https://www.computerhope.com/jargon/e/exception-handling.htm [Accessed 26 Jan. 2020].

Techopedia.com. (n.d.). *What is a Try/Catch Block? - Definition from Techopedia*. [online] Available at: https://www.techopedia.com/definition/25641/trycatch-block [Accessed 26 Jan. 2020].

Guru99.com. (n.d.). *How to create Test Strategy Document (Sample Template)*. [online] Available at: https://www.guru99.com/how-to-create-test-strategy-document.html [Accessed 26 Jan. 2020].

TechRepublic. (n.d.). *Testing Strategies for Object Oriented Systems*. [online] Available at: https://www.techrepublic.com/resource-library/whitepapers/testing-strategies-for-object-oriented-systems/ [Accessed 26 Jan. 2020].

Software Testing Fundamentals. (n.d.). *Unit Testing - Software Testing Fundamentals*. [online] Available at: http://softwaretestingfundamentals.com/unit-testing/ [Accessed 26 Jan. 2020].

## Appendix: Design Document 1

### Introduction

This document contains a detailed analysis on the software that has been outlined in the specification. The program essentially displays 4 different aquatic creatures within an aquarium and allows them to all swim around within that aquarium.

### Problem Analysis

The client brief states that there must be at least a JavaFish, SeaHorse and an Urchin that all move within the aquarium. The fish do not interact with each other in anyway, so there is no need to implement collision between the fish. The Urchin description does not state whether it is supposed to move diagonally or horizontally, so the assumption here would be to make the Urchin swim horizontally along the bottom of the screen. The client brief also states that each fish must be facing the direction that they are swimming, therefore I must add some code that alters the fish’s orientation once they have successfully reached the edge of the screen. A desirable aspect of the program would be to add an extra fish token and/or make use of advanced techniques to give the fish random movements. This does not state how often this random movement must appear, so I must assume the numbers myself.

## OO Software Design



Each fish will have a random speed between 0.005 and 0.05 per frame. Similar to the way I will set the position of each fish on start up, I can use inheritance to accomplish this task. If I make a parent class and make all the fish children to this class, then I can make the random speed generator inside the parent class. As all the fish need to change direction when they hit the edge of the screen, I can use a composition class to tell each of the fish to turn around when they’ve reached the boundaries of the aquarium. The class diagram contains this class and contains 2 methods. One for getting the position of the fish and one for checking where the fish is on the screen. The OrangeFish class is the extra ‘desirable’ token that I have added. This will have a random generator that randomly selects when the OrangeFish should start its acceleration behaviour. Urchin, JavaFish and Seahorse will only have an update function that calls upon one method inside the fish’s parent class (Pet). This method allows each fish to move back and forth along the aquarium. The relationships between each class are: Urchin is a Pet, SeaHorse is a Pet, JavaFish is a Pet, OrangeFish is a Pet, Pet has a FishBehaviour and Pet is a DisplayObject

## Learning Journal

#### Problem Analysis

In programming, problem definition is when the developer creates a logical response to the requirements of the program. This usually in the form of flow charts and logic diagrams, that clearly describes the problems with the software to the developer in a logical format. It is very important to use problem definition before writing the code for the software, so that the developer has a clear scope of the problem. They will usually ask them selves a few questions to find a solution e.g. What is the problem? What is the input and expected output? I mainly used this technique within the documentation, particularly the problem analysis and Design, as it allowed me look at the brief in a logical format and analyse each component of the brief step by step.

Decomposition is the act of taking one large problem and splitting it up into smaller, more manageable problems. So, decomposition by class discovery is the act of breaking up a set of requirements for the software into each class and the purpose of each class. A common way this is done within the industry is to use the verb/noun method. This requires the developer to look at the client brief and search for any nouns or verbs. The nouns will be the classes and the verbs will be the actions that take place within those classes. For example, the brief uses words such as ‘Seahorse’ and ‘JavaFish’. As these are nouns they would become the names of my classes. A verb related to JavaFish would be swim, which means that this would be and action inside this class. This this case, I noticed that all the fish have the verb ‘swim’ connected to them. Therefore, I was able to make a class that all the fish can use to swim, which is called fishBehaviour. I was able to make use of this technique when creating a class diagram too, as it made it much easier to map out how each class connect with each other and the job of each class.

Abstraction is the isolation of a specific segment within a problem, then proceeding to solve this segment. To accomplish this, you must first identify attributes of the problem and filter out the attributes that are relevant, then solely work on finding a solution for these parts (Alexis, 2016). In my code I have demonstrated this technique as I realised that each fish has similar moving patterns and understood that there was a better, more efficient way of doing this. I used composition to abstract all the movement code from each fish class and use one class, which holds all this movement code and allows each fish to call upon this code. (behaviour method inside pet class lines 96-112)

#### Composition

Composition is an Object-Oriented Programming concept that allows the re-use of existing code. In composition you create instance variables of one or more objects inside a class that are a reference to other classes. This forms a “has a” relationship between the objects (Baeldung.com, 2019). A real-world example of this would be that a chair would have other objects attached to it such as a leg or cushion. So, you could say that chair “has a” leg. This is useful in terms of software development, as you can have many classes use the same method as each other, while only having to type the code once. Therefore, creating a more efficient way of developing software. I have found this concept immensely advantageous throughout the development of my software. While reading the brief I noticed that all the fish have similar behaviours and so I took advantage of this detail and created a class named fishBehaviour. This contains a method that checks if the fish has reached the edge of the screen and if it has, then it tells the fish to turn around. By creating an instance of this class in my Pet class, I was able to use this same method to check for the screen bound, while only having the one method that resides inside the fishBehavour class.

This is the line of code I used to call upon the method called swimBehaviour, inside fishBehaviour class.

#### Inheritance

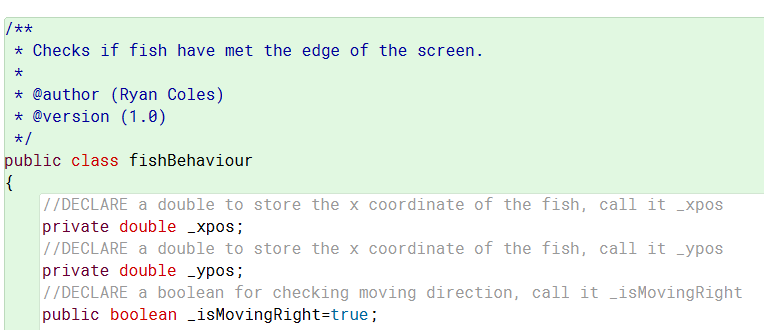
Inheritance in programming is the use of hierarchy between classes. In inheritance, there will be a class that is on top of the hierarchy tree and this class is called the parent class. The parent class offers classes that are lower in the hierarchy tree (also known as the child classes) some variables and methods that sit inside the parent class (Baeldung.com, 2019). As long as these methods and variables don’t have a private scope on them, the child class is free to use this code themselves. This is extremely useful when creating a program, as the developer can re-use code in the child classes, without having to type the same code twice, which makes the code extremely efficient.

Inheritance has been very useful when creating the program for the assignment, as I have made one class called Pet, that sets the position and orientation of each fish in the aquarium. The less efficient way of doing this would be for each fish to set their own position and orientation, which means without inheritance, I would have had to re-type all this code for each fish in the aquarium.

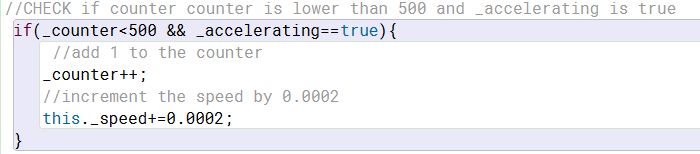
Inheritance is an “is a” relationship, whereas composition is a “has a” relationship. For example, a person “has a” leg (for this we would use composition) and a person “is a” mammal (for this we would use inheritance). You would use composition if you wanted to make an object that is made up of other objects. Ideally if you can ever get a “has a” relationship between objects, this is always favoured choice between composition and inheritance. (Baeldung.com, 2019) The main reasons to not inheritance are that composition relationships are a lot easier to maintain and some languages including Java, won’t allow the use of multiple inheritance. This means that you can only extend one class at a time, whereas you can have multiple references to different classes inside a singular class. This means that class can reuse code from anywhere as long as they have a reference to that class and the scope of the code they are trying to call is public.

#### Efficient Code

In order to keep a consistent code lay-out, I have followed the coding standard given by the university. This include a header for each class to describe its function, all class member variables to have a comment that follows a specific format and each member variable to have an underscore before the name (\_example Variable).



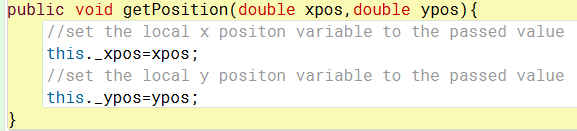
Each line must also include a comment that states the intent of the logic. The code below is an example of this.



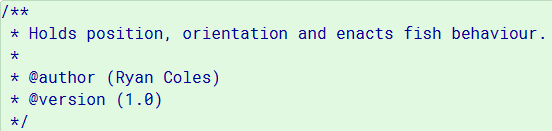
I have also used the camelCase naming convention throughout my code to keep a consistent code lay-out.



This code exemplifies the camelCase naming convention, but it also shows how I’ve used parameter passing in my code. In this example I am passing a specific minimum and maximum value to the random speed generator method, so that the value of the speed can only be between the numbers given in the client brief. To avoid confusion when parameter passing, I use an underscore for the data member variables and for variables in the parameters of the method, I don’t use an under score. So, when assigning local variable to parameter variables, they won’t have the same name.



Javadoc tagging is when you use a certain format when commenting, so that your code will have a structure that Javadoc can process and present information to the user. (Oracle.com, 2019)



The above image is an example of Javadoc tagging in my code. This will let the Javadoc know the author, version and the purpose of the class.

## References

*List your references here using the Harvard referencing style.*

Baeldung.com. (2019). *Inheritance and Composition (Is-a vs Has-a relationship) in Java*. [online] Available at: https://www.baeldung.com/java-inheritance-composition [Accessed 17 Nov. 2019].

Alexis, R. (2016). *What is abstract thinking? How is it used in problem solving?”*. [online] Quora. Available at: https://www.quora.com/What-is-abstract-thinking-How-is-it-used-in-problem-solving [Accessed 17 Nov. 2019].

Oracle.com. (2019). *Proposed Javadoc Tags*. [online] Available at: https://www.oracle.com/technetwork/java/javase/documentation/proposed-tags-142378.html [Accessed 17 Nov. 2019].