University of Worcester

COMP2331 Object Oriented Design and Development

Portfolio Design Documentation

Student Number: [redacted]

University of Worcester

Introduction

This document is an analysis of the brief for the Milestone 3, along with a design for the software solution. This document focuses on producing a cohesive and loosely coupled solution, adhering to several OO design principles.

Problem Analysis

Building on the second prototype from Milestone 2, the client requires a greater depth of functionality from the aquarium solution. The client asks that the aquarium contains a minimum of 13 creatures, as opposed to the 10 required in the previous prototype. These 13 include at least 10 'JavaFish', along with a minimum of one seahorse, urchin and piranha respectively. As with the previous prototype, each creature's width should not exceed one tenth of the display window. Additionally, JavaFish should be scaled by a value of 0.15 or less. Much of the brief is consistent with the previous prototype, with each creature requiring the ability to generate a random speed, between 0.005 and 0.05 per frame, on instantiation, and move along the horizontal axis. However, the seahorse also requires the ability to move on the vertical axis to produce diagonal movement. Each of the creatures, other than urchins, should be placed in the aquarium at a randomly generated position. When a creature reaches the bounds of the screen it will turn and face the other direction, as to remain within the window. As in the previous prototype, any creatures with visible mouths should emit bubbles. Additional requirements given by the client for this prototype include the ability to place 'FishFood' anywhere within the virtual aquarium at the position of a mouse click. The fish food should then fall to the bottom of the aquarium at an appropriate speed, giving the appearance of it slowly sinking deeper into the water.

Further to this brief, the client has also offered some desirable criteria. The client requests that an attempt be made to meet any of the desirable criteria given in the previous brief. I The client requests that an attempt be made to meet any of the desirable criteria given in the previous brief. In addition, the client asks that the JavaFish enact a 'shoaling' behaviour.
Where a creature collides with fish food, it should appear that the fish eats the food. This results in the fish 'growing', and the food be removed from the screen.

Assumptions

Where there is some ambiguity in the brief, several assumptions should be made:

- The Seahorse's diagonal movement suggests a continuous movement along both the Y and X axis.
- The seahorse's speed should be between 0.005 and 0.05 per frame in each axis.
- Choosing a creature's speeds on 'Start-Up' refers to the instantiation of the object, as opposed to the 'main' method of the program.

- The Urchin moving 'along the bottom' of the tank suggests a horizontal movement. Furthermore, the urchin must continue in the other direction once it reaches the edge of the tank.
- All sea creatures should not exceed one tenth of the width of the aquarium, not just the fish.
- Seahorses and Urchins do not have visible mouths.
- Urchin does not face any particular direction.
- Bubbles will travel upward after emission before disappearing.
- The shoaling behaviour should be reminiscent of how fish 'flock' in real life, where fish group and move in unison.
- A creature 'growing' should be achieved by an increase in the entity's scale.

Decomposing the Problem Through Class Discovery

Key	
Noun	Verb
Class	Method

Client Brief

Your client enjoyed the second prototype aquarium simulation immensely. They are really clear now on what they want, and you are required to design and produce a final prototype aquarium simulation according to the following revised brief:

1) Essential:

a. The aquarium must have a minimum of ten moving 'JavaFish' tokens, as well as at least one 'SeaHorse', one 'Urchin', and one 'Piranha' – the client has provided appropriate textures in 'JavaFish3' software project, please use the 'billboard' model provided.

b. The size of each fish must be such that its width does not exceed one tenth of the width of the display window. In addition, the JavaFish tokens should be scaled by 0.15 or less.

c. As before, each 'SeaHorse' should appear to swim diagonally, with a randomly chosen speed at start-up between 0.005 and 0.05 per frame. When a 'SeaHorse' collides with an edge of the aquarium, it should change direction so that it remains within the visible area of the aquarium. Each SeaHorse's starting position should be randomly chosen to be anywhere within the visible area of the aquarium.

d. Each 'Urchin' should appear to move along the bottom of the aquarium, again with a randomly chosen speed at start-up between 0.005 and 0.05 per frame. When an Urchin collides with an edge of the aquarium, it should change direction so that it remains within the visible area of the aquarium.

e. Each 'Piranha' should appear to swim horizontally backwards and forwards, always facing its direction of travel. Its speed should be randomly chosen at start-up, and should be within the limits of 0.005 and 0.05 per frame. When a Piranha collides with an edge of the aquarium, it

should change direction so that it remains within the visible area of the aquarium. Each Piranha's starting position should be randomly chosen to be anywhere within the visible area of the aquarium.

f. Each 'JavaFish' should also appear to swim horizontally backwards and forwards, always facing its direction of travel. Its speed should be randomly chosen at start-up, and should be within the limits of 0.005 and 0.05 per frame. When a JavaFish collides with an edge of the aquarium, it should change direction so that it remains within the visible area of the aquarium. Each JavaFish's starting position should be randomly chosen to be anywhere within the visible area of the aquarium.

g. Any virtual fish with a visible mouth should emit bubbles (texture available in JavaFish3 software project, please use the built-in 'sphere' model).

h. The user can place 'FishFood' anywhere in the virtual aquarium (texture available in JavaFish3 software project, please use the built-in 'sphere' model) by pointing the mouse and clicking the left mouse key. The FishFood will then fall to the bottom of the aquarium at an appropriate speed – it should give the appearance of something (eg a breadcrumb) falling through water.

i. The virtual fish do not interact with each other in any way; when they collide, they simply pass over each other.

2) Desirable (bonus – you can attempt to meet any or all of these if you have time!):

a. Meet any of the 'desirable' criteria given for the second milestone.

b. Give the group of 'JavaFish' a 'shoaling' behaviour (this is commonly referred to as 'flocking') – it is entirely up to you how you implement this, but obviously the more 'realistic' this behaviour is, the better.

c. Whenever an Urchin, a SeaHorse, a JavaFish, or a Piranha touches a FishFood, the FishFood is 'eaten', removing the FishFood in question from the virtual aquarium. The fish should grow slightly as a result of eating the FishFood.

d. Whenever a Piranha touches another pet, the Piranha should 'eat' the other pet, removing it from the virtual aquarium. The Piranha should grow slightly as a result. Whenever a pet is eaten, a new one should be 'spawned'.

List of Classes Discovered and Their Methods

- Simulation (simulation)
    - populate(); (have)

- World (aquarium)

- JavaFish
    - move(); (swim horizontally)
    - flip(); (face)
    - setSpeed(); (chosen speed)
    - setPosition(); (chosen position)
    - wiggle(); (wiggle)

- Seahorse
    - move(); (swim diagonally)
    - setSpeed(); (chosen speed)
    - setPosition(); (chosen position)
    - flip(); (face)

- Piranha
    - move(); (swim horizontally)
    - setSpeed(); (chosen speed)
    - setPosition() (chosen position)
    - flip(); (face)
    - eat(); (eat)
    - setScale(); (grow)

- Bubble
    - addBubble(); (emit)

- FishFood
    - fall(); (fall)
    - remove(); (eaten)

- Entities (fish)
    - CollisionHandler(); (touch)

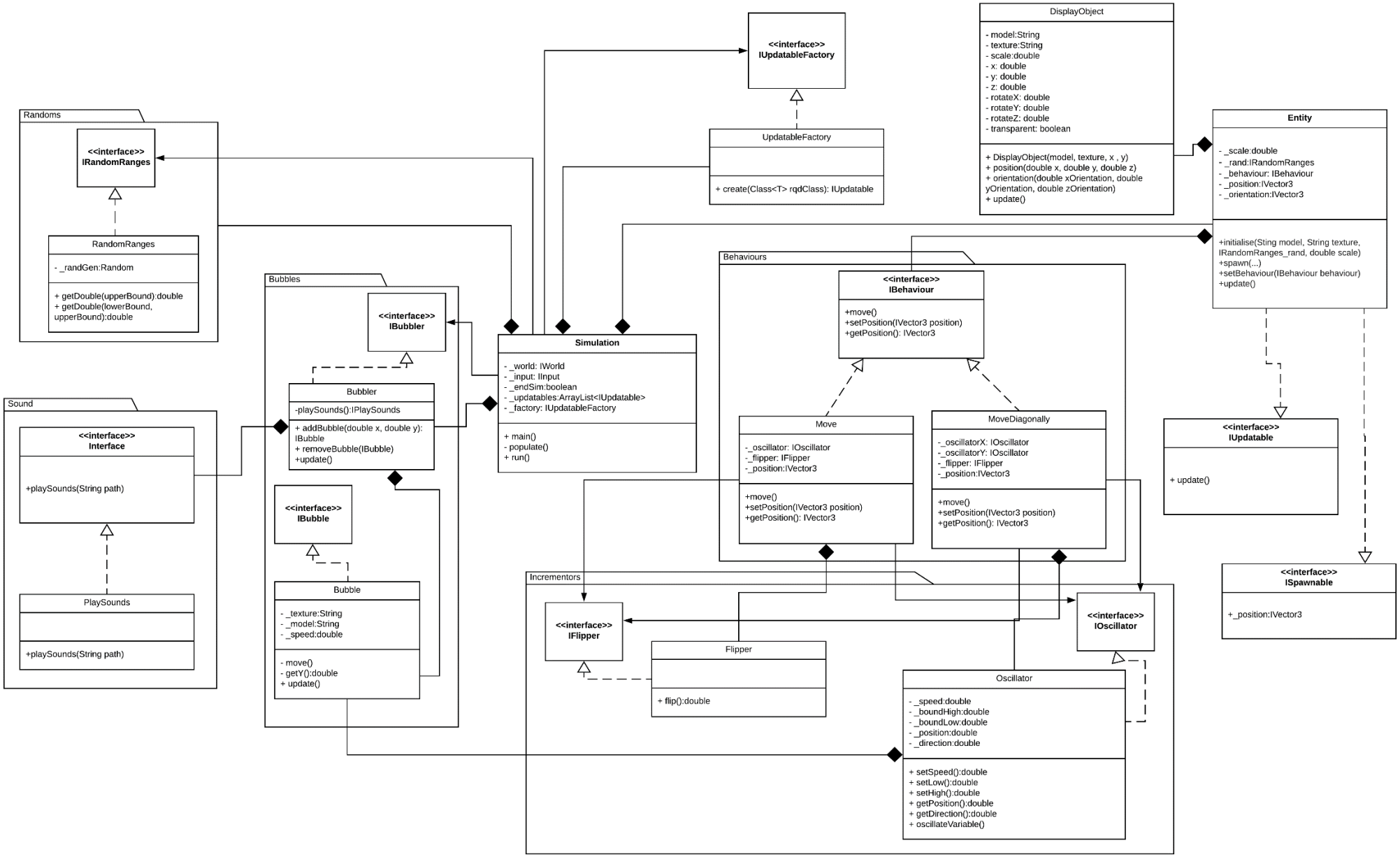
    - spawn() (spawned)
    - remove() (eaten)

- Random
    - generateSpeed(); (randomly chosen speed)
    - generatePosition(); (Randomly chosen position)

OO Software Design

COMP2331 Object Oriented Design & Development 2018-19
Assessment 1: third milestone

University of Worcester



COMP2331 Object Oriented Design & Development 2018-19
Assessment 1: third milestone

Explanation

I aim to achieve cohesion and reusability through smart use of packages. I have opted to use a strategy pattern in order to implement behaviours, this way they can be changed during runtime. The oscillator and flipper packages are composed within behaviours to prevent bloating.

Test Strategy

During Implementation, BlueJ's ability to test individual methods interactively will be utilised to perform ad-hoc testing. A Unit Test will also be undertaken to thoroughly test a crucial class in isolation. This will likely be the Oscillation class due to its use in different ways by multiple classes. If time allows, it would be ideal to perform unit test on all classes, integration tests and validation tests in order to test the software thoroughly.

Learning Journal

OO Design Principles

When designing, implementing and refactoring the solution, consideration was given to OO design principles. Application of design principles should produce code that is well encapsulated, with loose coupling and strong cohesion. The five SOLID principles (detailed below) were utilised, along with the dependency injection principle.

SOLID

The SOLID principles are a set of five OO design principles, introduced by Robert C. Martin (often referred to as Uncle Bob) in his 2000 paper 'Design Principles and Design Patterns' (Simon LH, 2019).

Single Responsibility Principle (SRP)

The single responsibility principle states that every module or class should have responsibility over a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated by the class (Despoudis, 2017). In simpler terms, classes should only have one reason to change. This allows for the for the avoidance of god classes. Unfortunately, the simulation class is a god class in my implementation. Through abstraction much of the in cohesion in the simulation class could be rectified. An entity manager could be utilised to control the population of the aquarium, leaving simulation to strictly focus on the running of the program.

That being said, there are several good examples of SRP within the solution. The IDespawnable class has a single method that simply removes a display object from the scene, as shown below.

```
public interface IDespawnable
{
    /**
     * METHOD: Remove the displayObject from the aquarium screen
     *
     */
    public void despawn() throws WorldDoesNotExistException;
}
```

Open/Closed Principle

Adapted from Martin (2000), a functional use of the open closed principle has two fundamental components. The first being that objects should be open for extension, and therefore be made to behave in a different manner to ensure it meets the current requirements of the application. The second is that objects are also closed for modification, where changes to public interface are not allowed. It could be said that these two fundamentals are conflicting, as an extension could be considered a change of the object itself. However, through abstraction, you can create derivative classes that build on the superclass, while leaving its public interface untouched.

This is such an asset as a change in public interface can have catastrophic effects on functionality. Once a public interface is defined, a contract is created with all users of the class. Therefore, if a class must change, functionality should be added, rather than modified. Open/Closed Principle and Single Responsibility Principle work in tandem, as a minimal interface is much less likely to require modification.

I utilised this principle in my work when implementing the Mouthed class, an extension of Entity that has additional bubble blowing functionality. I created an interface that contained the additional functionality required to pass information about an entity to a bubbler class (see below.)

```
public interface IMouthed
{
    /**
     * METHOD: DECLARE a method that return position IVector3, call it getPosition.
     *
     * @return     the new position of the object
     */
    public IVector3 getPosition();

    /**
     * METHOD: DECLARE a method that returns an x coordinate, call it getX.
     *
     * @return     x coordinate of object
     */
    public double getX();

    /**
     * METHOD: DECLARE a method that returns a y coordinate, call it getX.
     *
     * @return     y coordinate of object
     */
    public double getY();
```

The class then implemented this interface, as well as IEntity, I order to extend the functionality. This prevented any unwanted changes to the rest of my coded solution.

Liskov's Substitution Principle

Barbara Liskov states that, 'Derived types must be completely substitutable for their base types' (Signh *et al.*, 2018). The principle itself is strongly linked to subtype polymorphism. The principle states that it must be ensured that derived types are suitable from a behavioural standpoint. Singh states that, when a derived type can behave as its supertype, without breaking its behaviour, this can be referred to as strong behavioural subtyping.

Interface Segregation Principle

Robert Martin explains that it is not necessary for a client to depend on methods which they do not use. Larger interfaces should be split into more manageable and specific categories in order to ensure that clients are only aware of methods relevant to their own needs (Martin and Martin, 2006).

Interface Segregation Principle relates to certain disadvantages that larger interfaces may have, such as a lack of cohesion. It is also important to acknowledge that it can be beneficial for clients to understand them as abstract base classes, as opposed to a single class. Another important distinction to make is the separation of clients and their respective interface. If they are used by entirely different clients, then interfaces should remain separate in order to avoid difficulty.

The interface Segregation Principle was utilised within the Mouthed class in my code, as shown below.

```
public class Mouthed implements IUpdatable, ISpawnable, IEntity, IMouthed
{
```

Functionality for the class was split into distinct interfaces: IUpdatable, relating to the ability to update the class per tick; ISpawnable, relating to the objects ability to be spawned onto the screen; IEntity, containing all functionality required for the class to enact behaviours and IMouthed, containing functionality regarding bubble emission.

Dependency Inversion Principle

Dependency Inversion Principle states that both higher and lower level modules should rely on abstractions, rather than higher level objects depending on lower levels (Noback, 2018). Here, the use of the word abstraction relates to the creation of an interface or an abstract class. Adhering to this principle should achieve loose coupling, as there is a removal of direct dependencies between sub and super classes (DEVIQ, 2020).

Dependency inversion principle has been utilised in the solution through the implementation of factory classes. Factory classes remove the dependency of both high- and low-level objects on concrete classes for instantiation, and instead instantiate by means of abstraction. If a class were to be instantiated directly with the simulation class, the new keyword immediately creates a dependency between the class and simulation. Preventing this greatly decouples the simulation class from objects within the aquarium.

Dependency Injection

Dependency Injection is a programming technique that makes a class independent of its dependencies (Janssen, 2018). This is achieved by decoupling an objects usage from its creation. Objects are injected into their subclasses by the calling class, and then utilised within these subclasses.

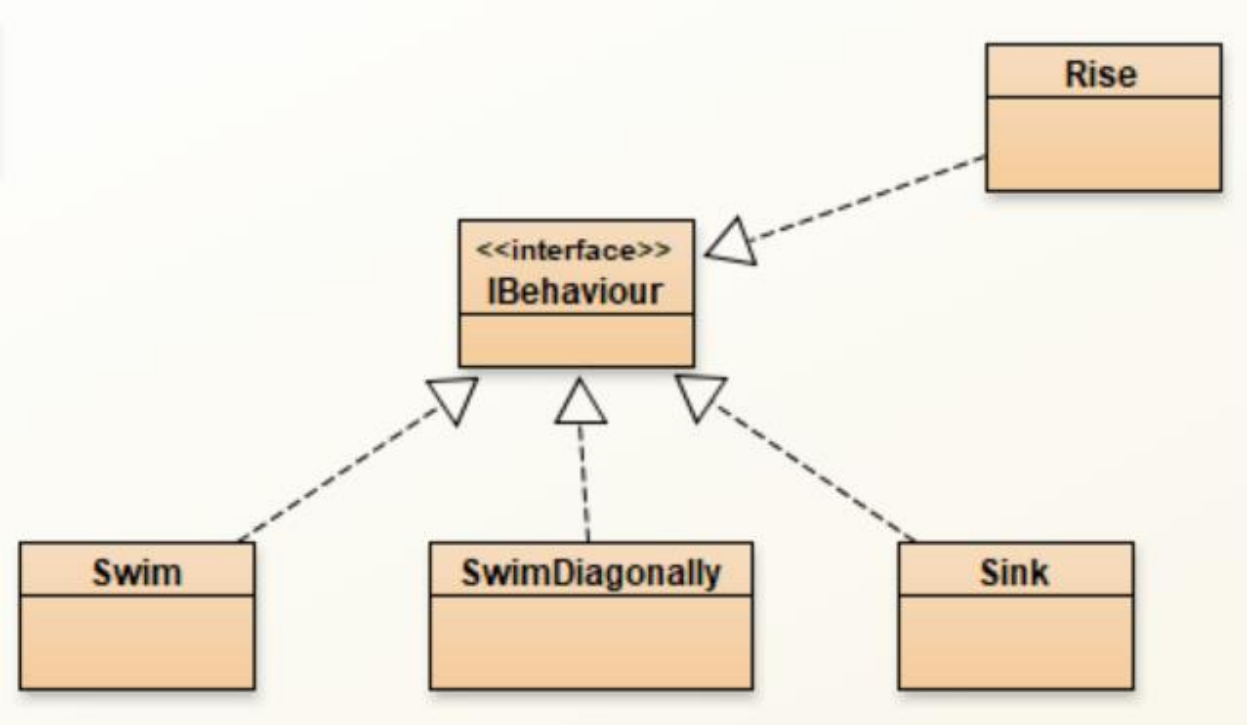
This technique has been utilised in my code when passing the RandomRanges random number generator to classes.

Design Patterns

Strategy Pattern

Adapted from RefactoringGuru (Unknown), the intent of a strategy pattern is to define a family of encapsulated algorithms, store them within a separate class and make their object's substitutable.

The strategy pattern enables the selecting of an algorithm at runtime (Nnamdi, 2018), especially useful in my code for the implementation of behaviours within the Entity class. This is because, in the brief, several creatures could require the ability to change behaviour, such as the JavaFish switching from shoaling to swimming behaviours. Rather than breaking the Interface Segregation Principle, where no client should be forced to depend on methods it doesn't use, entities can enact a behaviour in complete abstraction from the inner workings of the concrete base classes. The encapsulation also allows the simple extension of behaviours, where another behaviour class can be added, and its behaviour enacted immediately by entities within the program.



The figure shows the structure of the strategy pattern within my program. Each behaviour class implements an interface (shown below) that couples to clients only through an abstraction. Furthermore, clients aren't even coupled to a realisation of the abstraction, minimising coupling to an even greater degree (SourceMaking, Unknown).

```
public interface IBehaviour
{
    /**
     * METHOD: declare method to advance the client's position in varying ways, call it move().
     *
     */
    public void move();

    /**
     * METHOD: declare method to set both orientation and position within a specific behaviour, call it setSituation.
     *
     * @param  position    the position of the client, passed to the behaviour
     * @param  position    the orientation of the client, passed to the behaviour
     */
    public void setSituation(IVector3 position, IVector3 orientation);

    /**
     * METHOD: declare method to set the speed within a behaviour, call it setSpeed.
     *
     * @param  speed    the speed to be passed to the behaviour
     */
    public void setSpeed(double speed);

    /**
     * METHOD: declare method to set the speed within a behaviour, call it setSpeed.
     *
     * @param  speed    the speed to be passed to the behaviour
     */
    public IVector3 getPosition();

    /**
     * METHOD: declare method to get the orientation from a behaviour a behaviour, call it getOrientation.
     *
     * @return     the orientation vector from the behaviour
     */
    public IVector3 getOrientation();
```

Observer Pattern

The observer pattern is classified as a behavioural pattern, where it is more specifically concerned with communication between classes (Poyias, 2019). The intent of the Observer Pattern is to define one to many relationships between objects so that when an object's state changes, all its dependents are notified (SourceMaking, Unknown).

The observer pattern has been utilised in my code in order to handle mouse click inputs. A package has been created called InputHandling, containing a class called MouseHandler. This class contains an array, which stores a list of objects subscribed to receive notification of a mouse click should it occur (see code below.)

```
public void update(){
    int[] mouseVal;

    try
    {
        // If we have a mouse click
        if (0 == _inputCapture.getMouseClicked()){
            // Capture mouse coordinates
            mouseVal = _inputCapture.getMousePointer();

            // Report mouse click!
            for (IInputListener l : _listeners)
            {
                l.onInput(mouseVal);
            }
        }

    }
    catch (Exception e)
    {
        // do nothing

    }
}
```

The mouse handler continually checks whether a click has occurred in its update loop. If a click occurs, the value is stored as an array, before being passed to all objects subscribed to the event (stored in the listeners array). This value is then outputted to the simulation class, where its coordinates are passed into a display object in order to produce FishFood at the site of the click.

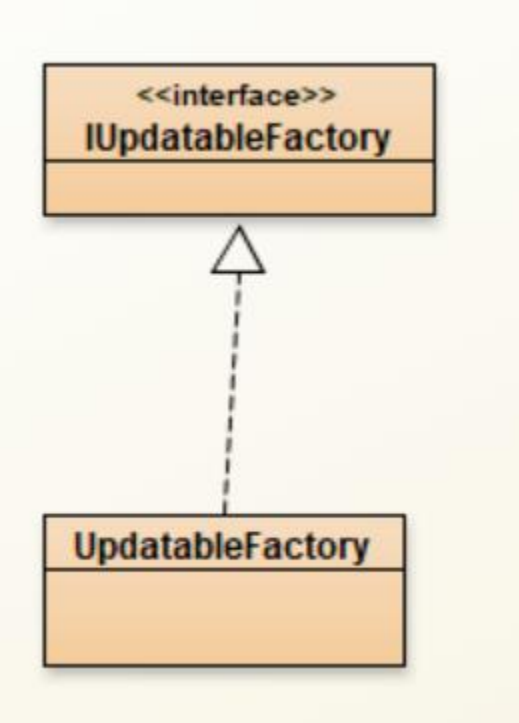
By utilising a listener, the job of monitoring a mouse click is taken out of simulation and abstracted out into a separate class. This reinforces strong cohesion within the code, as it is not the job of Simulation to monitor mouse clicks.

Factory Pattern

The factory pattern enables the creation of an object without exposing the creation logic to the client, whilst also being able to refer to the newly created object using a common interface (Rai, 2018). The intent of the pattern is to define an interface for instantiating an object, but allowing subclasses decide which class to instantiate (Gamma et al., 2009).

Factory classes are more flexible than constructors, offering an ability to create objects without knowing what will be required beforehand. They are useful in my work as they have allowed IUpdatable creation to be abstracted out of the Simulation class, aiding cohesion. A single factory has allowed the creation of multiple objects that otherwise would have been dependent on simulation for construction, aiding decoupling.

I have created my factory within package, as shown below.



The factory in instantiated within Simulation, and its use ensures abstraction is used to create every updatable object within the solution. I would have liked to have implemented further factories within my code, including one for the purpose of creating DisplayObjects within the entity class.

Efficient Design

Divide and Conquer

Adapted from Makabee (2011), the divide and conquer approach seeks to reduce a problem's complexity by continually dividing it into more manageable sub-sections. Handling these sub-sections independently is simpler than attempting to form a solution to the problem as a whole. This hierarchical sub-dividing of problems was utilised in, among other places, the design and implementation of entity behaviours. In order to compose a single behaviour, such as swimming horizontally, the split was split into parts. Behaviours utilise a package called Incrementors (shown below), containing coded solutions for the mathematics of various aspects of what makes up each entity's behaviour. For example, the swim behaviour is made up of a flipper, controlling the reversal of facing direction, and an oscillator, controlling the oscillation of position within the bounds of the screen.

Agile Discovery

To summarise the writing of Farias (2018), discovery, a phase of agile development, sets out goals through analysis and dialogue between stakeholders and developers. It prevents misaligned goals, where the client's needs are not met by the final solution, and incomplete

requirements, where the developer must add features to the program in order to meet the requirement brief.

References

Despoudis, F. (2017) *Understanding SOLID Principles: Single Responsibility* Available at: https://codeburst.io/understanding-solid-principles-single-responsibility-b7c7ec0bf80 (Accessed 11th March 2020)

DEVIQ. (2020) *Dependency Inversion Principle: Depend on abstractions, not concretions.* Available at: https://deviq.com/dependency-inversion-principle/ (accessed 15th March 2020).

Farias, H. (2018) *Discovery is the most underrated phase of software development* Available at: https://www.infoworld.com/article/3294201/discovery-is-the-most-underrated-phase-of-software-development.html (Accessed 11th March 2020)

Gamma *et al.* (2009) *Design Patterns 15 Years Later: An Interview with Erich Gamma, Richard Helm, and Ralph Johnson* Available at: https://www.informit.com/articles/article.aspx?p=1404056 (Accessed 15th March 2020)

Janssen, T. (2018) *Design Patterns Explained – Dependency Injection with Code Examples* Available at: https://stackify.com/dependency-injection/ (Accessed 15th March 2020)

Makabee, H. (2011) *Divide-and-Conquer: Coping with Complexity* Available at: https://effectivesoftwaredesign.com/2011/06/06/divide-and-conquer-coping-with-complexity/ (Accessed 11th March 2020)

Martin, R. (2000) *More C++ Gems. Compiled and Introduced by Robert C. Martin* Cambridge University Press.

Martin, R. and Martin, M. (2006) *Agile Principles patterns and practices in C#* Prentice Hall Publishing

Nnamdi, C. (2018) *Keep it Simple with the Strategy Design Pattern* Available at: https://blog.bitsrc.io/keep-it-simple-with-the-strategy-design-pattern-c36a14c985e9 (Accessed 15th March 2020)

Noback, M (2018) *Principles of Package Design: Creating Reusable Software Components* Apress Publishing.

Poyias, A. (2019) *A Quick Guide to the Observer Pattern* Available at: https://medium.com/datadriveninvestor/design-patterns-a-quick-guide-to-observer-pattern-d0622145d6c2 (Accessed 15th March 2020)

Rai, Y. (2018) *Factory method Design Pattern* Available at: https://dzone.com/articles/factory-method-design-pattern (Accessed 15th March 2020).

RefactoringGuru. (Unknown) *Strategy* Available at: https://refactoring.guru/design-patterns/strategy (Accessed 15th March 2020)

Simon LH. (2019) *SOLID Principles: Explanation and examples* Available at: https://itnext.io/solid-principles-explanation-and-examples-715b975dcad4 (Accessed 11th March 2020).

Singh, K *et al.* (2018) *Design Patterns and Best Practice in Java: a Comprehensive Guide to Building Smart and Reusable code in Java* PACKT Publishing Birmingham.

SourceMaking (Unknown) *Observer Design Pattern* Available at: https://sourcemaking.com/design_patterns/observer (Accessed 15th March 2020).

SourceMaking (Unknown) *Strategy Design Pattern* Available at: https://sourcemaking.com/design_patterns/strategy (Accessed 15th March 2020)

Appendix: Design Document 2

Please see 'previous milestone' folder as cannot append PDF at home.