Assignment 1 - Group 26

Exercise 1 - Core

1. Following the Responsibility Driven Design, start from your requirements (without considering your implementation) and derive classes, responsibilities, and collaborations (use CRC cards). Describe each step you make. Compare the result with your actual implementation and discuss any difference (e.g., additional and missing classes).

First we underlined important words from our requirements:
- **In must haves**: game, start menu, playing field, fish, player fish, bounding box, collision detection, non-player fish, death screen
- **In should haves**: splash screen, sprites, score, lives, instructions page, options menu, music, sound.

We did not underline the could haves (yet), nor did we underline the would haves.

We then created CRC cards for the different classes, in 2 categories, and attempted to name their responsibilities:

GAME:
Fish - Generic fish class, used as a basis for both Player Fish and Non-Player Fish.
Player Fish - Represents the player, responsible for handling movement, keeping track of score and lives.
Non-Player Fish - Represents enemy fish.
Bounding Box - Responsible for describing the area where a Collidable is.
Collidable - Base for a collidable object, has a bounding box and checks for collisions.
Sprite - Represents an image that can be drawn (rendered) on the screen.
Playing Field - Represents the field that holds fish.
Game - Represents the game, responsible for moving Fish.

UI:
FishIO - Entry point of our application. Launches the GUI
Renderer - Responsible for rendering a playing field.
Main Menu - The main menu of the screen, has buttons that allow the user to go to different screens in the GUI.
Game Screen - A screen showing a playing field, where the game is run on.
Splash Screen - Shows up when the game is started, and switches to the main menu when done.
Instructions Screen - Shows instructions how to play the game.
Options Screen - Shows options (settings) and allows the user to change them. These changes are propagated to the correct parts.
Audio Engine - Responsible for playing sounds and background music.

We then ordered their collaborations (mainly for the Game)
Note: it is hard to represent this properly

$$\begin{tikzcd}
& \text{Bounding Box} & \\
& \text{Collidable} \arrow[u, "\text{(has)}"'] & \\
\text{Non-Player Fish} \arrow[r] & \text{Fish} \arrow[u, "\text{(is)}"'] \arrow[d, "\text{(part of)}"'] & \text{Player Fish} \arrow[l] \\
& \text{Playing Field} \arrow[dl] \arrow[dr] & \\
\text{Game} \arrow[dr] & & \text{Renderer} \arrow[dl] \\
& \text{Game Screen} &
\end{tikzcd}$$

The actual implementation, as seen by the class diagram below, has a few differences:

- The game and renderer class are both contained within the **Playing Field** class. There is not a good reason for this, and the **Playing Field** class should be split up into a **Playing Field**, **Game** and **Renderer** class.
- We have a **SinglePlayerPlayingField** class, which extends the **PlayingField** class. The reason a Playing Field is different for Single Player than for non single player, is because the responsibilities of **PlayingField**, **Game** and **Renderer** are all combined in **PlayingField** (see above).
- **Sprite** is implemented differently. We use the JavaFX Image class for our sprites, and have a **IDrawable** interface. This **IDrawable** interface simply defines a render method, where the drawable can decide how it should be rendered.
- **Non-Player Fish** is called **EnemyFish**
- There is no **Fish** class, but we have an **Entity** class, which fulfills the same role, but is more generic, allowing us to also use it as a base for all non fish entities in the playing field.
- In addition to an **ICollidable**, we also have an **IMovable**, representing a movable object and an **IPositional** interface, representing an object that has a position on the field.
  The **IPositional** interface could possibly be removed, since something that can move and/or collide, must have a position of some sort, so it is a bit redundant.
- We have a **LevelBuilder** class, which takes over some of the responsibilities of the playing field class: generating random fish and spawning them in the playing field.
- We also have a **TickListener** interface, which can be registered with a **PlayingField** to be called before/after a game tick or render tick.
- We still have a **Direction** enum, which is being phased out because we use non cardinal directions. It's purpose was to indicate what direction an **Entity** is moving in.
- We have a **Vec2d** class, which is used to represent 2 dimensional vectors. It is used by the **BoundingBox** and for **IMovable**s, for speed calculations.
- We have a **Preloader** class, which loads images (sprites) and screens in advance, to allow for fast switching between screens, and short load times for the game itself.

And for the UI

- There is no **Options screen**, since we don't have any options as of yet.
- There is no **Audio Engine**, since music and sound will be implemented at a later date.
- We have a **SinglePlayer Screen** instead of a **Game Screen**, for the **SinglePlayerPlayingField** extension of **PlayingField** (see above). It fulfills the same role: displaying a **Playing Field**, and passing on key input (movement) to the **PlayerFish**.
- We have also added an **Achievements Screen** and **Help Screen**, which both do not give much information. Achievements will be implemented slowly over the coming sprints, and the help screen will be filled with information in the future.

We also have an entire Logging framework, as is required for exercise 3 of this assignment. This is not in our requirements, and therefore not included in the RDD classes mentioned here.

2. Following the Responsibility Driven Design, describe the main classes you implemented in your project in terms of responsibilities and collaborations.
The main classes in our project are:

- **PlayingField**
    - Responsibilities:
        - Starting a game
        - Adding new entities to the field
        - Moving the entities on the field
        - Checking for collisions on the field
        - Calling ticklisteners
        - Removing dead entities from the field
        - Rendering all IDrawables on the screen
    - Collaborations:
        - LevelBuilder - Called for generating new enemies
        - IMovable - Move method called for moving movables in the field
        - ICollidable - Check for collision and call onCollide method.
        - IDrawable - Render method called to render on the screen
        - TickListener - Registered TickListeners are called before and after ticks.
        - Entity - Check if entity is dead, and call drawDeath method.
        - MainMenuController - Holds the GUI button to start a game, and is returned to after the game ends.
- **SinglePlayerController**
    - Responsibilities:
        - Displaying what's happening in the game to the user.
        - Allowing the user to interact with the PlayerFish
        - Showing a death screen on death.
        - Allowing the user to revive when he dies and has lives left
        - Showing the score and amount of lives
        - Allow the user to return to the main menu

        - Allow the user to restart and pause/unpause the game
    - Collaborations:
        - PlayerFish - Score and life changes are listened for and handled by the SinglePlayerController
        - MainMenuController - Holds the GUI button to start the game, and this screen is returned to when the user stops the game.
        - SinglePlayerPlayingField - Holds the fish that this screen should display, and the game thread and render thread.

3. Like was said in 1.2, the PlayingField class should be split up into 3 classes: Game, Renderer and PlayingField. Although not entirely separated, due to difficulties with splitting a core class, we made code changes to separate the rendering from the game thread. So it is still contained in the PlayingField class, but now separated therein, and can be separated fully in the future.

Direction has almost no responsibility, and was removed entirely. See https://github.com/Taeir/fish-io/pull/78

Other classes are important, but simply hold less responsibility. For example, Vec2d has little responsibility, representing a 2 dimensional vector, but is used by bounding boxes and by IMovable for movement. However, it does not make sense to merge this into either bounding box or IMovable, since it does have multiple collaborations with other classes.

4. Because of the class diagram being too large to fit into this report, here is a link to the diagram in our Github repository, where you can actually zoom in: class diagram.png.

5. Link to sequence diagram in our Github repository: sequence diagram.pdf.

Exercise 2 - UML in practice

1.
When a class is a part of another class and the first class is an independent part, you should use aggregation. An example in our code is the relation between the SinglePlayerPlayingField and the Player class. The player is a (important) part of the playing field but the player itself could function without a playing field.

Composition is a relation where one class is a part of another class and has no value by it's own. This relation is for example between the play button on the menu and the menu. They are both separate classes but a button without a screen makes no sense.
In Fish.io, most of the relations are aggregations. Most of these relations are in the PlayingField class. This is mainly because this class binds all the game parts together. Those parts are all independent and important by themselves. PlayingField has these relations with the TickListeners and, through multiple interfaces, with all the entities in the game.
One of the few compositions in Fish.io is the relation between an Entity and a CollisionArea. Without the Entity a CollisionArea has no meaning. Otherwise it would be possible for nothing to collide with something.

2.
No, our code does not contain any parameterized classes.
In the Vec2d class we could have used a parameterized class. We decided not to use generic types for this class. Because the class is mostly used for positions in the game, the class should work with doubles. A double however, can not be used as a type argument. Instead the class Double can be used, but this was not as easy to work with as a normal double. Therefore we decided not to use generic types in this case.

Because parameterized classes are very useful, we use some ‘standard’ Java classes that have generic types, for example the HashSet and ArrayLists.

Parameterized classes are useful when you need a class that has to work with many different objects, but only one (or a small amount) at a time. This is the easiest to visualize with some kind of data containing class, for example a list: If you want a list that works with every object, you implement a list of Objects. Getting an item from the list will return you an Object that you have to cast to the original type. Doing this wrong will result in a lot of exceptions. With the list parameterized you specify the type of objects used when instantiating the list. The lists contents are now all of that type.

An other benefit of generic types is that you can reuse a class. You can make a parameterized List class that works with different class types instead of multiple list classes that all work with one class (List<> instead of StringList, IntList, PlayerList, etc..) . This makes your code more organized and better maintainable.

3.
The inheritance/hierarchic diagram can be found in the UML folder.
Most of the hierarchies in our code are implemented using interfaces. This is because most of the classes aren't extensions of other classes, but do share some key properties. The classes that extend other classes are *SinglePlayerPlayingField*, *PlayerFish* and *EnemyFish*, they extend *PlayingField*, *Entity* and *Entity* respectively. These relations are 'is-a' relations, which means that a *PlayerFish* is an *Entity*. These extensions are made because the sub- and superclass share attributes and methods. An *Entity* has a position, a CollisionArea and can be alive or dead. Because a *PlayerFish* extends this by adding movement and scores, the classes are clearly not distinct and therefore have a is-a relation.

All the classes implementing interfaces are polymorphisms, which means that the classes implementing the same interface are clearly distinct but are interchangeable when used as part of something. In our game the *CollisionArea* is a good example of this definition. *CollisionMask* and *BoundingBox* both implement this interface, but the two classes are mostly different. The *CollisionMask* is basing collisions on the sprite of the entity, where the *BoundingBox* is basing collisions on a virtual box around the entity. Because the core of both classes is related, the rest of the game doesn't need to know which one is being used to function properly. This behavior is exactly the behavior that you would expect from an interface implementation, therefore all interface implementations are polymorphisms.

Exercise 3 - Simple logging

We implemented logging with the requirements stated in the email to the TA. The requirements can be found on the google drive, or on github in the docs/Requirements folder.

We implemented all the *Must* and *Should haves* in the logging API, however we did not implement two of the *Could have* requirements into the project, namely: 1)The logging could be included as a text window/console, which could be enable in the options menu. And 2) The logging could also return the class and method from which the logger was called. The reasons for exclusion for these methods are:

1) *The logging could be included as a text window/console, which could be enable in the options menu:*

    Logging is mostly for the development team. Logging in the GUI will probably look nice and sophisticated, however a file logger is implemented too. So it is possible for a user to send that log file instead of looking at the logs in the GUI. A user won't be able to fix the code if they are on the compiled package, most of the time. So there is not really a reason to include GUI logging on this project.

    Getting the logging to the point it is now also took a lot of time. Implementing a GUI handler would not have been possible in the timeslot that was created for logging in this sprint. Two new screens had to be created and probably a new thread to keep the GUI of both windows responsive.

    All in all, this feature was deemed not worth the effort and will not be implemented in the future.

2) *The logging could also return the class and method from which the logger was called:*

    While this can be deemed as pretty useful feature, it is actually quite hard to implement, as we found it out. It is certainly possible to do, but it would require to dig into the strack trace. This is quite risky and not trivial to implement.

    The advantage of seeing what method or class called the logger is that it is easier to see what method made a specific log at what time. But most of time, logs that are relevant to some one on the the development team are implemented by themselves, so they know what logs come from what classes, or at least know how to look it up.

    With these reasons in mind, we decided it was not worth implementing this feature for logging.