Assignment 4 - Group 26

Exercise 1 - Your wish is my command, Reloaded

Requirements for Multiplayer

The requirements are listed in the same way as our game, so we will use the MoSCoW method. Firstly, the functional requirements of the logging are listed as must, should, could and won't haves. Secondly, the non-functional requirements are listed.

Functional Requirements

1.1 Must haves

- The game Fish.io must have non-local multiplayer.
  - This must support multiplayer for at least 3 clients.
  - The connection will find place through TCP/IP connections.
- The implementation will be socket-based and will use a server that distributes the necessary information and environment variables.
- The following actions must be implemented:
  - Every client must be able to control its own playerfish.
  - Players should be able to eat another player if they are big enough according to the game rules.
  - Smaller AI fish must spawn and move so that the players can eat those instead if they cannot eat players.
- Multiplayer must be implemented into the GUI.
  - There must be a multiplayer screen on which the player can connect to a multiplayer game by connecting to a specific IP.
  - There must be a button in the main menu that links to the multiplayer screen.

1.2 Should haves

- The settings should be the same on all clients.
  - The server can send a settings file to all clients that the clients will use.
- The playing field should be bigger than a single player playing field..
- The user's view of the playing field should be able to move within the boundaries of the playing field.
  - This is called a scrolling screen.

1.3 Could haves

- The server could spawn certain power ups that players can collect.
  - These might have to be rebalanced in order to be fair to other players.

- There could be extra entities like a large fish that is able to eat all players.

1.4 Won’t haves

- The multiplayer implemented won’t be local multiplayer version (2 players on one machine).
    - (The items below are requirements of local multiplayer):
    - The user shall be able to start a local multiplayer game using a “multiplayer” button on the start menu.
    - The local multiplayer version shall work similar to the single player version. A few differences:
        - The controls for the extra player(s) will differ from the controls for the first player.
        - Different player fish can have overlapping bounding boxes without any result.
- There won’t be any extra entities like:
    - Sea urchins: When the player is bigger than a sea urchin and collides with it, the player gets killed instead of eating it.
    - Obstacles: the player must move around obstacles and cannot move through them.

Non-functional requirements

- A first, working, version of the multiplayer mode shall be ready and submitted to the staff of the course on the 16th of october.
- The multiplayer will not be tested with the same coverage as the total game (75%), as this is very difficult to do right on travis and it would take too much time to implement.
- The components that can be tested easily on travis will be tested with the same coverage as the entire game (75%).
- The multiplayer implementation will have to adhere to the same non-functional requirements of Fish.io itself, with exception of the modified non-functional requirements in this document.

Why we didn’t fully finish multiplayer in time:

We weren’t able to completely finish the multiplayer feature in time (we also didn’t expect it), because simply: multiplayer requires many additions and changes to the code. You would also need experience with sockets in order to do it fast. We only have one team member who has (a little bit) experience with sockets and the rest just had to follow what the person was doing. We also had to decide to use the Netty dependency, which took time to choose and also to understand (following Netty documentation, tutorials and such).

However, we surprisingly got very far with the multiplayer feature. Though there are countless bugs and things that can be improved, this is a case of “done is better than perfect”. These things should all be resolved in future sprints, including writing tests!

We came to the conclusion that, especially because of the singleton patterns of FishIOClient and FishIOServer, that testing the multiplayer components is not feasible. Any component that is non-trivial to test, uses these unmockable singleton classes. The only multiplayer classes that are really testable, are the PlayingFields, FishServerSettingsMessage and FishServerEntitiesMessage.

CRC cards of the main classes of the multiplayer feature

FishClientMessage

Interface for classes that represents a message sent from the client.	Child of: FishMessage Located in: Multiplayer.client Implemented by: FishClientRequestPlayerMessage FishClientPlayerFishMessage

FishServerMessage

Interface implemented by classes representing a message from the server.	Child of: FishMessage Implemented by: FishServerEntitiesMessage, FishServerSettingsMessage, FishServerPlayerMessage Located in: Multiplayer.server

FishClientHandler

Handles all FishClientMessages	Child of: SimpleChannelInboundHandler <FishServerMessage> Changes: FishIOClient

FishServerHandler

Handler used by the server for messages sent from the client.	Child of: SimpleChannelInboundHandler <FishClientMessage> Located in: Multiplayer.server

MultiplayerClientPlayingField

A playing field used by a client in a multiplayer game. Updates the fish on the screen to match the server.	Child of: MultiplayerPlayingField Located in: Multiplayer.client

MultiplayerServerPlayingField

A PlayingField that is used by the server in a multiplayer game. Sends the entities to the connected clients. Receives information about the connected players.	Child of: MultiplayerPlayingField Makes changes to: ServerGameThread, Renderer Located in: Multiplayer.server

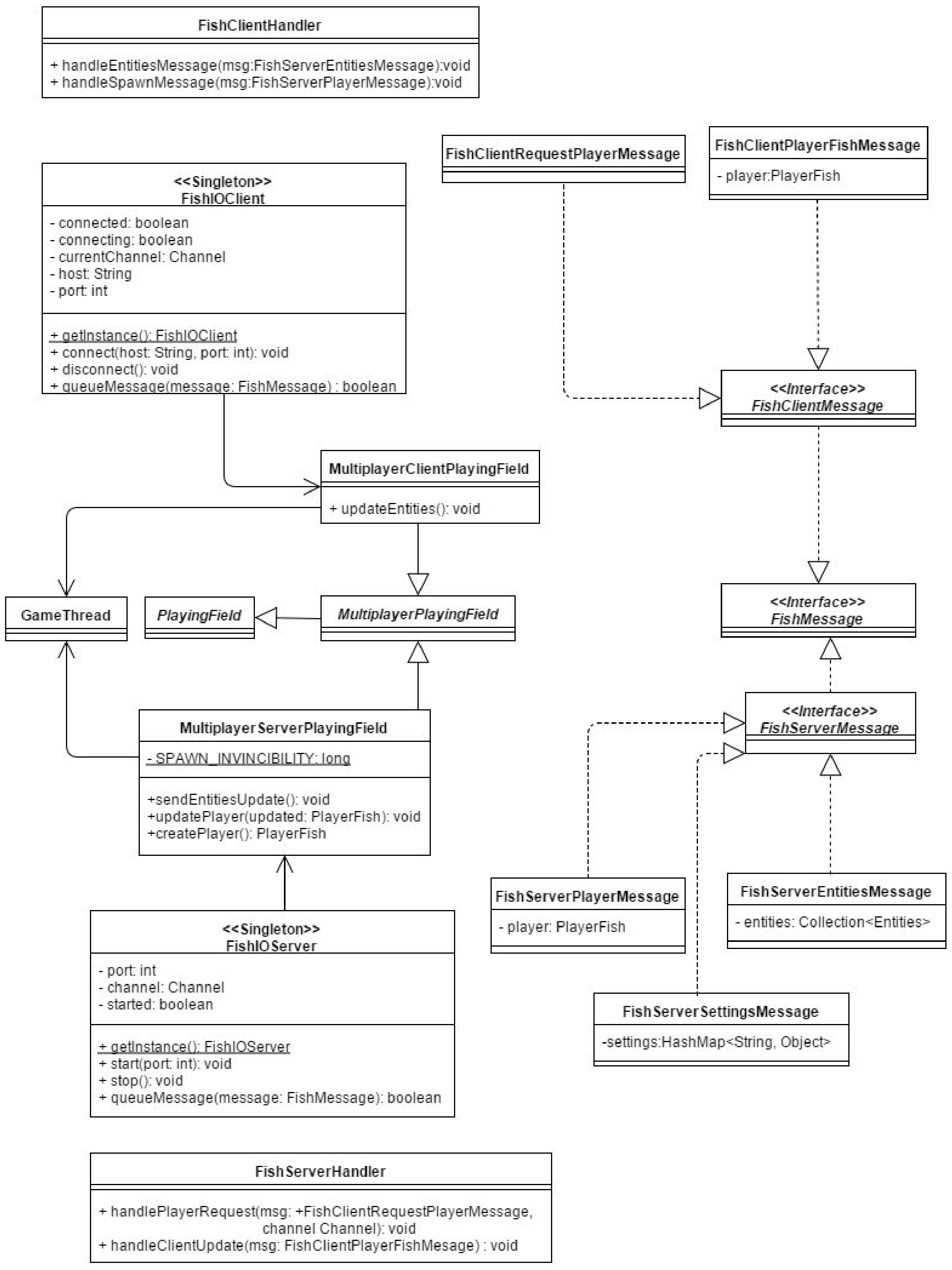
FishIOClient

Singleton class that represents this client in a multiplayer game. Connects to / disconnects from a server	Implements: Runnable Located in: Multiplayer.client

FishIOServer

Singleton class that represents a server in a multiplayer game. Sends messages to the connected clients.	Implements: Runnable Located in: Multiplayer.server

Class diagram of the overall multiplayer feature


FishClientHandler
+ handleEntitiesMessage(msg:FishServerEntitiesMessage):void
+ handleSpawnMessage(msg:FishServerPlayerMessage):void
FishClientRequestPlayerMessage
FishClientPlayerFishMessage
- player:PlayerFish
<<Singleton>>
FishIOClient
- connected: boolean
- connecting: boolean
- currentChannel: Channel
- host: String
- port: int
+ getInstance(): FishIOClient
+ connect(host: String, port: int): void
+ disconnect(): void
+ queueMessage(message: FishMessage) : boolean
<<Interface>>
FishClientMessage
MultiplayerClientPlayingField
+ updateEntities(): void
GameThread
PlayingField
MultiplayerPlayingField
<<Interface>>
FishMessage
<<Interface>>
FishServerMessage
MultiplayerServerPlayingField
- SPAWN_INVINCIBILITY: long
+sendEntitiesUpdate(): void
+updatePlayer(updated: PlayerFish): void
+createPlayer(): PlayerFish
FishServerPlayerMessage
- player: PlayerFish
FishServerEntitiesMessage
- entities: Collection<Entities>
<<Singleton>>
FishIOServer
- port: int
- channel: Channel
- started: boolean
+ getInstance(): FishIOServer
+ start(port: int): void
+ stop(): void
+ queueMessage(message: FishMessage): boolean
FishServerSettingsMessage
-settings:HashMap<String, Object>
FishServerHandler
+ handlePlayerRequest(msg: +FishClientRequestPlayerMessage, channel Channel): void
+ handleClientUpdate(msg: FishClientPlayerFishMesage) : void

Exercise 2 - Software Metrics

The analysis file can be found in our GitHub Repository. (Taeir/fish-io/docs/Code Metrics)

According to inCode, the classes that contain design flaws are *BoundingBox, YamlSettingsLoader, PuFreeze, PuSuperSpeed, TestPuFreeze and TestPuSuperSpeed*, all with severity 1.
For this assignment we've chosen the *BoundingBox, YamlSettingsLoader* and both *PuFreeze and PuSuperSpeed*. These two PowerUp classes have exactly the same problem so we've chosen to fix those at the same time (they're both our "third" chosen class).

BoundingBox

According to inCode, the BoundingBox class is a so called schizophrenic class. This means that the class has a lot of public methods and they can be seperated in clusters by looking at the methods that call them. Some of the methods also do not use local data but are using only input data from other methods. Atleast, that is what inCode says is wrong with the BoundingBox class.
However, when looking critically at the different specific reasons inCode specifies as 'causes to why BoundingBox is a schizophrenic class', some things are fishy.

Firstly, inCode states that there are quite a few methods which are not used, namely:

- equals(Object ob),
- getBottomLeft(),
- getRotation(),
- getSize(),
- getTopRight(),
- setRotation(double angle),
- setSize(double size).

However, the equals(Object ob) is extensively used in the assertEquals() and assertNotEquals() methods when testing the BoundingBox and is therefore very important to keep.
The getBottomLeft() and getTopRight() methods are, like their counterparts getBottomRight() and getTopLeft(), used in the Renderer class to be able to draw fish sprites which are not horizontally alligned to the playing field.
The getRotation() and setRotation() methods are used by the Renderer class to render the sprites which are not horizontally alligned under the right angle.
The getSize() and setSize() methods are both used within the class and again also in the Renderer.
To conclude, inCode has some problems in finding out which methods are used, as most of the aforementioned methods are actually very vital for the performance of the game due to theiir usage in the Renderer class. These methods are used inside the working of the

Renderer methods and we get the feeling that inCode did not have a look inside these methods to see whether they were actually using any other methods.

Secondly, inCode also states that there are a few methods inside the BoundingBox class which do not use local data of the class but other data. According to inCode, these methods are:

- getHeight(),
- getRotation(),
- getWidth()
- intersects(ICollisionArea other).

These four methods again made us wonder whether inCode actually looks at the inside of the methods of classes at all. Out of those four methods, three actually do have their own local data which they use:

- getHeight() uses the **private double** height field.
- getRotation() uses the **private double** rotation field.
- getWidth() uses the **private double** width field.

It is true that instersects(ICollisionArea other) does not have a local data field which it used. This method is used to calculate whether an ICollisionArea has any intersection with another ICollisionArea object. Combined with the last complain inCode had about this class, we decided to remove the intersects(ICollisionArea other) method.

Lastly, inCode mentioned that our code was separated in two clusters of classes which did not use the same fields and were not used by the same methods or classes. In reality, one of the clusters only contained the intersects(ICollisionArea other) and it’s corresponsing TestIntersects() method. This has been fixed by removing the intersects(ICollisionArea other) method and it’s testmethod alltogether

We refactored the game in a such a way that the PlayerFish always has the subtype CollisionMask instead of ICollisionArea, and therefore removing the instersects(ICollisionArea other) method from both the BoundingBox class as the ICollisionArea interface. Since the game does not use BoundingBoxes for intersections anymore, this is fine.
During the process of doing so, we found out that our TestEnemyFish class still used BoundingBoxes and their intersect methods. Therefore we deleted these tests to be able to rewrite them using CollisionMasks (as every EnemyFish created in the game nowadays does).

YamlSettingsLoader

According to inCode, the *YamlSettingsLoader* class is a data class. This is because the feature this class is part of has everything to do with data: Settings. The first idea of settings is that everything should be saved in a single file and that all those values should be loaded in the game. How to load them however and where should those settings be stored? To hold on to the single responsibility per class design principle, it was decided that a single class should be responsible for loading the settings.

That's when the *YamlSettingsLoader* was created. For perfomance reasons, the settings file should only be loaded once and as soon as the application starts. For the *YamlSettingsLoader* to accomplish this, it needed several methods to load the settings (once for each type like double, String, boolean, etc). However these methods should not be accessed by other classes, as we only want to load once. What needed to happen is store all those settings in variables (or HashMaps like we did, to make it more organized) at the constructor of *YamlSettingsLoader* and then the *Settings* class accesses those fields once before storing it himself. However, all those fields that store the settings in the *YamlSettingsLoader*, made inCode falsely accusing him of being a data class.

Whether *YamlSettingsLoader* is even a data class is debatable. According to Lanza & Marinescu - 2006, date classes are "dumb data holders". The *YamlSettingsLoader* clearly is a data holder, but it is not dumb. It has the responsibility to load data, not to store it, that is the *Setting* class' job. It is true that it does store data, but that is because it must be able to do that in order to fulfill its responsibility.

Another feature data classes have, that *YamlSettingsLoader* doesn't have, is loose coupling with other classes. It couples only with one class, the *Settings* class. While this coupling isn't very strong, it is still a valid association. They have a "used by" association, where *YamlSettingsLoader* is used by *Settings*.

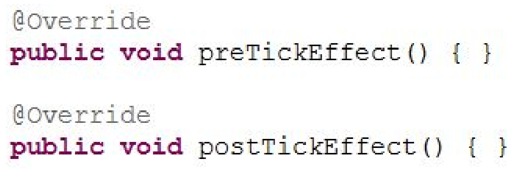
We could give the loading methods of the class a return value that contains a list of loaded settings. While that would reduce the amount of fields, it is still a horrible idea. The settings would need to be loaded every time a class has need of it, which would destroy the performance of the application.

PuFreeze and PuSuperSpeed

These two classes both extend the *DurationPowerUp* class. This class exists to “help” PowerUp classes that want something to do with a duration. It extends the executeEffect method in the *PowerUp* class and splits it up in more methods. A startEffect method that gets called when the PowerUp gets actived and endEffect method when the PowerUp is over. These two methods are required to be implemented by its subclasses, otherwise those subclasses might as well extend *PowerUp* instead of *DurationPowerUp*.

When this code was implemented, the author thought ahead: “What if we want a PowerUp that has to do something every tick?” For example a PowerUp that makes all smaller EnemyFishes swim towards the PlayerFish needs that feature. You want every time an EnemyFish spawns to get that behaviour, so the PowerUp would need to keep checking for spawned EnemyFishes every tick.

Because of that thought, two extra methods were added in *DurationPowerUp*: preTickEffect and postTickEffect, which act as an extra service to the subclasses in case they want it. The problem was that now we have two tradition breaking classes, *PuFreeze* and *PuSuperSpeed*. They are tradition breaking because they didn’t implement the preTick and postTick method of *DurationPowerUp*, because they don’t need them. This leads to ugly and unnecessary code.



```
@Override
public void preTickEffect() { }

@Override
public void postTickEffect() { }
```

Fixing this is no problem at all. Removing the abstract methods preTickEffect and postTickEffect is enough (and adjusting the other methods accordingly). *PuFreeze* and *PuSuperSpeed* now can’t break a tradition that isn’t even there!

However, to ease the fears of the original author: “What to do when we want a PowerUp that does need to update every tick?”. The solution is simple and should have been thought of before. All that the PowerUp will need to do is implement the existing *TickListener* and register itself to the GameThread in its constructor. *DurationPowerUp* was pretty much stealing the responsibility of *TickListener*. So that is another flaw fixed!