Assignment 3 - Group 26

Exercise 1 - 20-Time, Reloaded

1) We had two different requirements. One for the Settings feature and one for the Audio feature:

Functional Requirements for settings

1.1 Must haves

- The game shall have some settings that can be changed to change the behavior or look of the game.
- These settings must be observable values, so that parts of the game can attach listeners to them and be notified when the value of the setting changes.
- The settings should be persistent, meaning that when the game is closed and re-opened, the settings won't change.
    - The settings must be stored in a human-readable format.
- There must be a (settings-) screen in the game where the user can see the values of all the settings.
- Multiple settings-types must be supported:
    - double
    - integer
    - boolean
- The following settings must exist:
    - screen size
    - movement speed

1.2 Should haves

- The settings screen should allow the user to change values of all settings.
- More setting types should be supported:
    - KeyCode
    - double (slider from 0 to 1)
- The settings for sound should be supported.
- There should be settings for the controls of the game.
- Most of the constants in the game that should be a setting are replaced by a setting.

1.3 Could haves

- Settings for all the values in the game that could be a setting.

1.4 Won’t haves

- There won't be a setting for every 'magic number' in the source code, because not all of those values are a setting.

Non-functional requirements settings

- The settings classes will have to adhere to the same non-functional requirements of Fish.io itself (except for the date of delivery).
- A first, fully working, version of the settings shall be ready and submitted to the staff of the course on the 9th of October.

Functional Requirements for audio

1.1 Must haves

- The game shall have background music that will be played while the program is running, unless disabled with the settings.
- The game shall cycle through different background music tracks.
- The game shall have sound effects that will be played for different events in the program.
  - These include (but are not limited to):
    - Clicking a button
    - Eating a fish
    - Getting eaten by an enemy fish
    - Eating a powerup
- The Audio Engine will support three volume types, represented as percentage values in doubles (0.0 to 1.0):
  - Master volume, affects all audio played.
  - Music volume, affects background music
  - Sound effects volume, affects sound effects
- There will be a mute button for music displayed during the game, with which the user can toggle between the game playing all sounds, only sound effects and no sounds.
- The game will be able to play at least 10 sounds at the same time.
- The game shall be able to play sounds in mp3 format.

1.2 Should haves

- The settings screen should have settings for controlling the aforementioned volumes.

1.3 Could haves

- More sound effects could be added for other events in the game.

1.4 Won’t haves

- The game will not play a different background music track each time it is started.

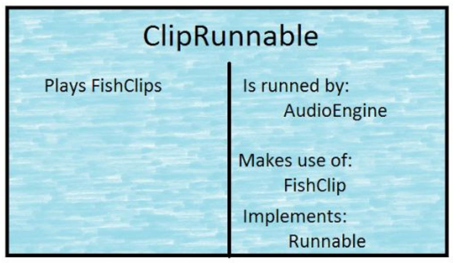
Non-functional requirements for audio

- The audio engine classes will have to adhere to the same non-functional requirements of Fish.io itself (except for the date of delivery).
- A first, fully working, version of the audio engine shall be ready and submitted to the staff of the course on the 9th of October.

2)

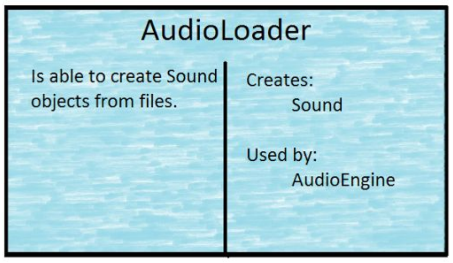
The simplest way to show the responsibilities of both the audio and settings feature is through CRC cards and UML.

Audio CRC cards



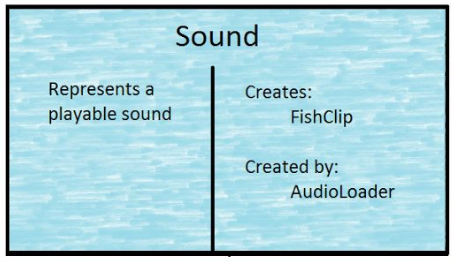
ClipRunnable

Plays FishClips	Is runned by: AudioEngine Makes use of: FishClip Implements: Runnable



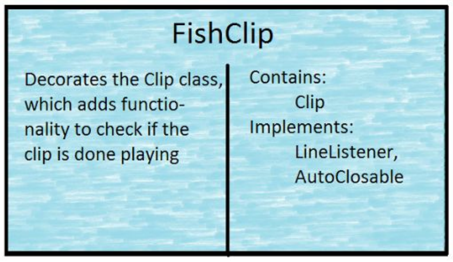
AudioLoader

Is able to create Sound objects from files.	Creates: Sound Used by: AudioEngine



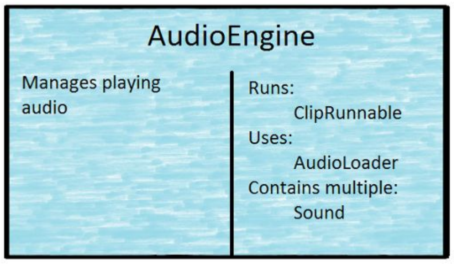
Sound

Represents a playable sound	Creates: FishClip Created by: AudioLoader



FishClip

Decorates the Clip class, which adds functio-nality to check if the clip is done playing	Contains: Clip Implements: LineListener, AutoClosable



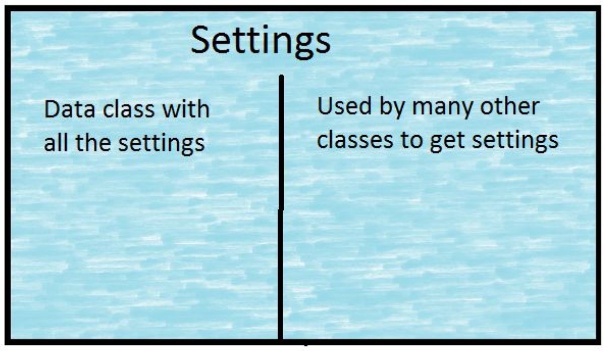
AudioEngine

Manages playing audio	Runs: ClipRunnable Uses: AudioLoader Contains multiple: Sound

Audio UML

Because the UML would be a little too large to even be a little bit readable, the UML of the audio should be viewed through the GitHub repository.

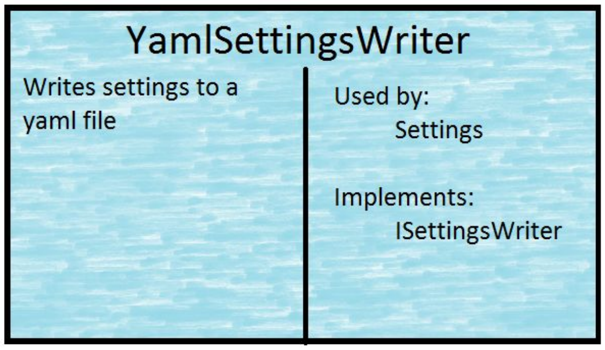
Settings CRC cards



Settings

Data class with all the settings

Used by many other classes to get settings

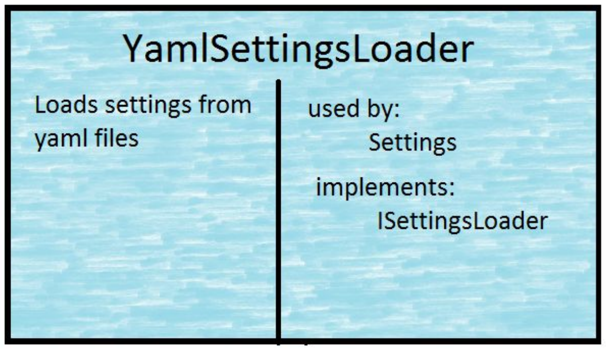


YamlSettingsWriter

Writes settings to a yaml file

Used by:
Settings

Implements:
ISettingsWriter



YamlSettingsLoader

Loads settings from yaml files

used by:
Settings

implements:
ISettingsLoader

Settings UML

View it from the GitHub repository.

Exercise 2 - Design patterns

Singleton Pattern - Logger

1) Natural language description of implementation.
Why: When we were designing our logging system, we decided to look at examples of logging that already existed. Most of these logging systems used a simple Singleton Logger to log all the events. There were also other solutions, but most of these either seemed to require an entire restructure of the project and were to hard to implement (example: Aspect-Oriented Programming).

Using the singleton patterns means that each class that wants to log something can simply call the logger by including Logger log = Log.getLogger();. There is only only one instance of the logger, which means that if the logger has to log information to a file, a buffered writer can be used. Also it is easy to access the logger and change its settings from anywhere in the code as long as it was set up during the program start. And lastly, we are able to change the behaviour of the Log class at runtime, which could be important for certain classes.

How: As said the logger uses the Singleton pattern. This means that the Logger has a private constructor and only the logger itself can initialize itself. This way there will only be a single instance of the logger.

In our logging API we decided to use *eager initialization*, this means that we initialize the logger as an attribute in its own class. We call the logger from the main class as soon as the program starts running. This means that there is only one thread at that time, so we don't have to worry about concurrency issues at that point, thus it is safe to use eager initialization. The main reason for eager initialization is that we assume that the logger is callable in the rest of the code. This makes logging worry-free for the development team.

The logging-process works like the Java.util logging API, but of course does not makes use of an pre-existing classes. This means that we have a single logging object (as the singleton pattern demands). This logging object has the following attributes: the Log object, the LogLevel and an ArrayList containing Handlers.

The Log object is the logger itself. It can be retrieved by the getLogger() method for use in other classes. Like said before the Log class uses eager initialization, so this attributed is set as soon as the log object is called for the first time.

The LogLevel attribute indicates what level of logging has been set. This LogLevel is an ENUM which can take the following values: NONE, ERROR, WARNING, INFO, DEBUG and TRACE. The importance of these LogLevel is as listed, from high to low. Setting the Log class to a specific log level means that all logs of LogLevels below the set LogLevel are ignored.

The ArrayList that contains Handlers has to be initialized after the logger is called. An ArrayList is used to hold multiple Handlers, meaning that multiple ways of logging can be employed at the same time. The Log classes iterates over this list so that every included handler can log the log message.

Each Handler can employ a single format to handle the log message. Handlers and formats implement their own interface. The Handler handles the log message in a way that it is designed for. Example: the ConsoleHandler logs to the console. After handling the message, the Log class will iterate over the next Handler until all Handlers have been executed.

Concluding, we believe this approach makes for a robust logging API for us to use in our game.

2) Class diagram of implementation.

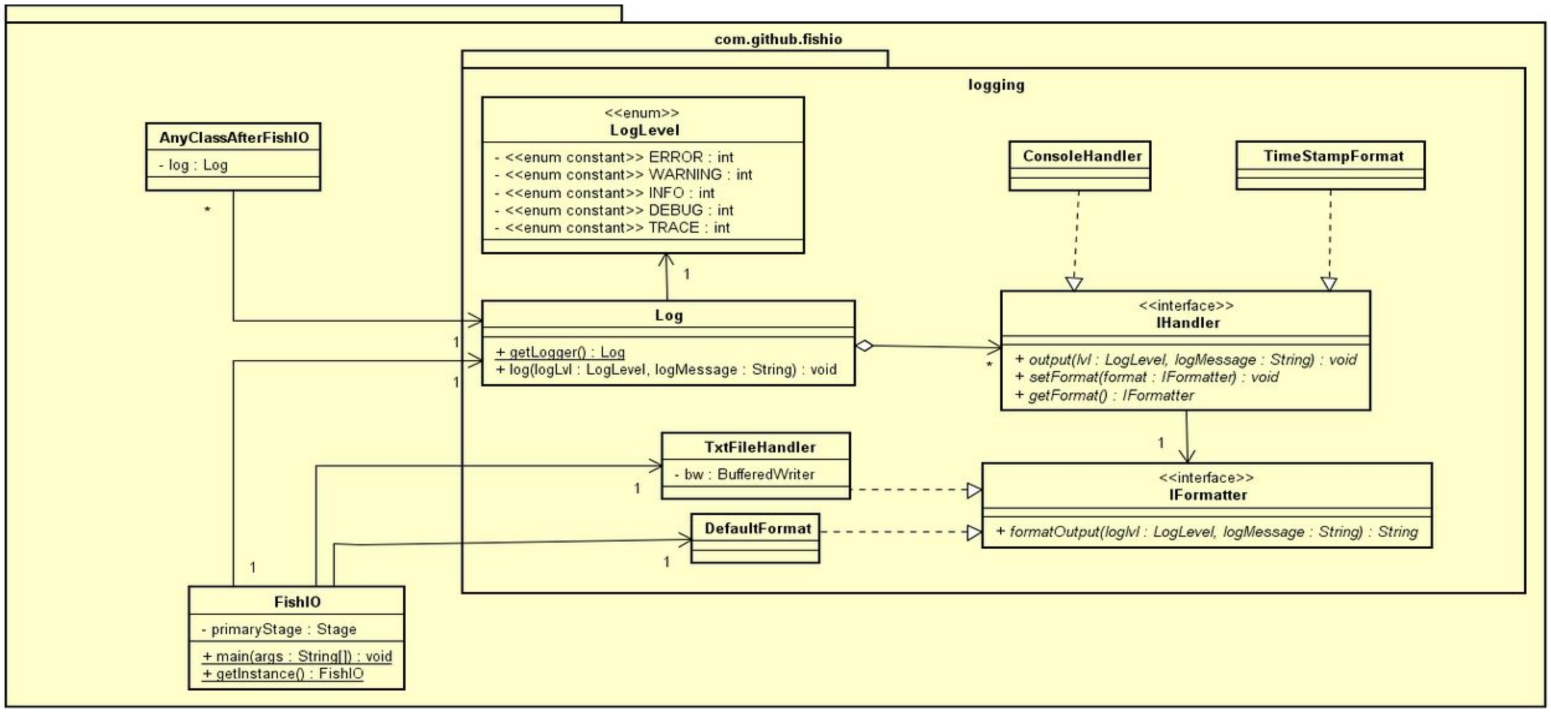

com.github.fishio
logging
AnyClassAfterFishIO
- log : Log
<<enum>>
LogLevel
- <<enum constant>> ERROR : int
- <<enum constant>> WARNING : int
- <<enum constant>> INFO : int
- <<enum constant>> DEBUG : int
- <<enum constant>> TRACE : int
ConsoleHandler
TimeStampFormat
Log
+ getLogger() : Log
+ log(logLvl : LogLevel, logMessage : String) : void
<<interface>>
IHandler
+ output(lvl : LogLevel, logMessage : String) : void
+ setFormat(format : IFormatter) : void
+ getFormat() : IFormatter
TxtFileHandler
- bw : BufferedWriter
<<interface>>
IFormatter
+ formatOutput(loglvl : LogLevel, logMessage : String) : String
DefaultFormat
FishIO
- primaryStage : Stage
+ main(args : String[]) : void
+ getInstance() : FishIO
*
1
1
1
*
1
1
1
1

Figure 1: The class diagram of the logger implementation. For the explanation of the Logging API please see the previous exercise. A bit larger version can be found on the GitHub repository.

3) Sequence diagram of example of use case of implementation.

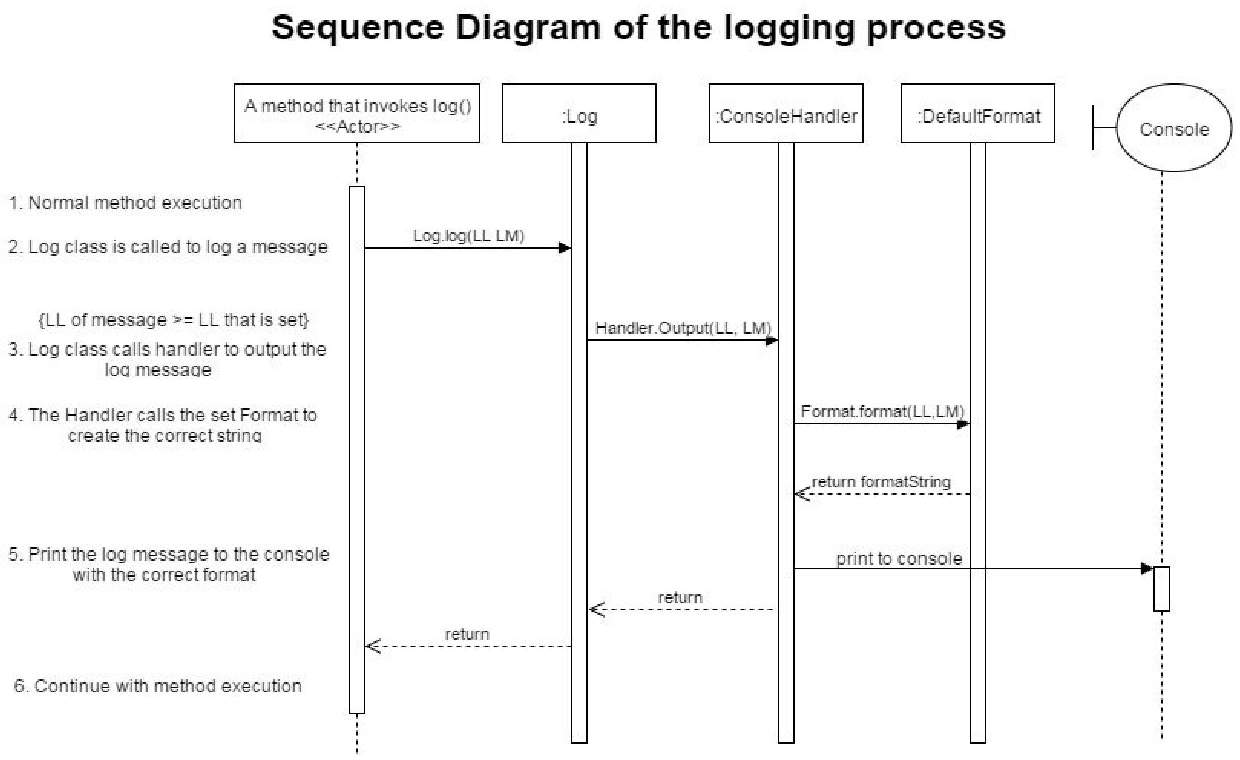

Sequence Diagram of the logging process
A method that invokes log() <<Actor>>
:Log
:ConsoleHandler
:DefaultFormat
Console
1. Normal method execution
2. Log class is called to log a message
Log.log(LL LM)
{LL of message >= LL that is set}
3. Log class calls handler to output the log message
Handler.Output(LL, LM)
4. The Handler calls the set Format to create the correct string
Format.format(LL,LM)
return formatString
5. Print the log message to the console with the correct format
print to console
return
return
6. Continue with method execution

Figure 2: A sequence diagram of a use case of a logging process. LM stands for Log Message and LL stands for Log Level. In this process the Log class has been initialized with a ConsoleHandler which uses the DefaultFormat. Note that Log, ConsoleHandler and DefaultFormat have live lines that exist before and after the method that calls the logger. This is thanks to the singleton pattern of the Log class. The Log class is initialized in the main method of the game.

Strategy pattern - Behaviours

1)

Before the implementation, we had something quite similar to the strategy pattern. Each subclass of the Entity class (EnemyFish, PlayerFish and PowerUp) had their own way of moving around the PlayingField. So we had a special interface IMovable designed for entities that can move around in the PlayingField. This interface had the most important method *getSpeedVector() : Vec2d* which returned the direction in which the entity is moving and how fast. PlayingField could then use that speedVector to move the entity.

Then PowerUps were implemented. One of the PowerUp subclasses is PuFreeze, which is supposed to make every EnemyFish stop moving. In other words, the PuFreeze needed to be able to make the *getSpeedVector()* method return (0, 0). In order to do this we added a boolean attribute to the EnemyFish that, if true, would make the *getSpeedVector()* return (0, 0). This increased the cyclomatic complexity of the EnemyFish and also made it very ugly. EnemyFish needed to have special extra code, just so that PuFreeze could mess with it.

A better solution would be the strategy pattern. Instead of making the all the entity classes implement IMovable, they can just have an IMovable attribute! We renamed the IMovable interface to IMoveBehaviour, because that would make a little bit more sense. The IMoveBehaviour has the same *getSpeedVector()* method as the old IMovable.

Four classes that implement IMoveBehaviour were added. The RandomBehaviour class makes the entity move in a certain direction, but can sometimes randomly slow down or speed up in the x - or y-direction. This behaviour is used by the EnemyFish. VerticalBehaviour is a simple behaviour with a constant speed that makes the entity move straight down, which is what all PowerUps do. KeyListenerBehaviour is for the PlayerFish, which makes it listen to the user input from the keyboard. Finally the FrozenBehaviour, which simply returns (0, 0) when *getSpeedVector()* is called.

This made it very easy for the PuFreeze class to change the behaviour of and EnemyFish. All it had to do now is *enemyFish.setBehaviour(new FrozenBehaviour())* and the EnemyFish would stand still. Not only will it help for this, but also for future PowerUps that change the behaviour of entities. For example: A PowerUp that makes all EnemyFishes smaller than the PlayerFish swim towards the PlayerFish like a magnet. The strategy design pattern saves us from creating a lot of messy code for future implementations.

Apart from the *getSpeedVector()* method, the IMoveBehaviour also has a method *preMove() : void* which gets called by the PlayingField each tick, just before getting the speedVector. In this method the behaviour can update its speedVector. For example, KeyListenerBehaviour updates its speedVector depending on which keys are pressed on that moment. Some behaviours, like VerticalBehaviour, are very simple and don't need to make use of *preMove()*, but this method is useful for complex behaviours.

2) Class diagram of implementation

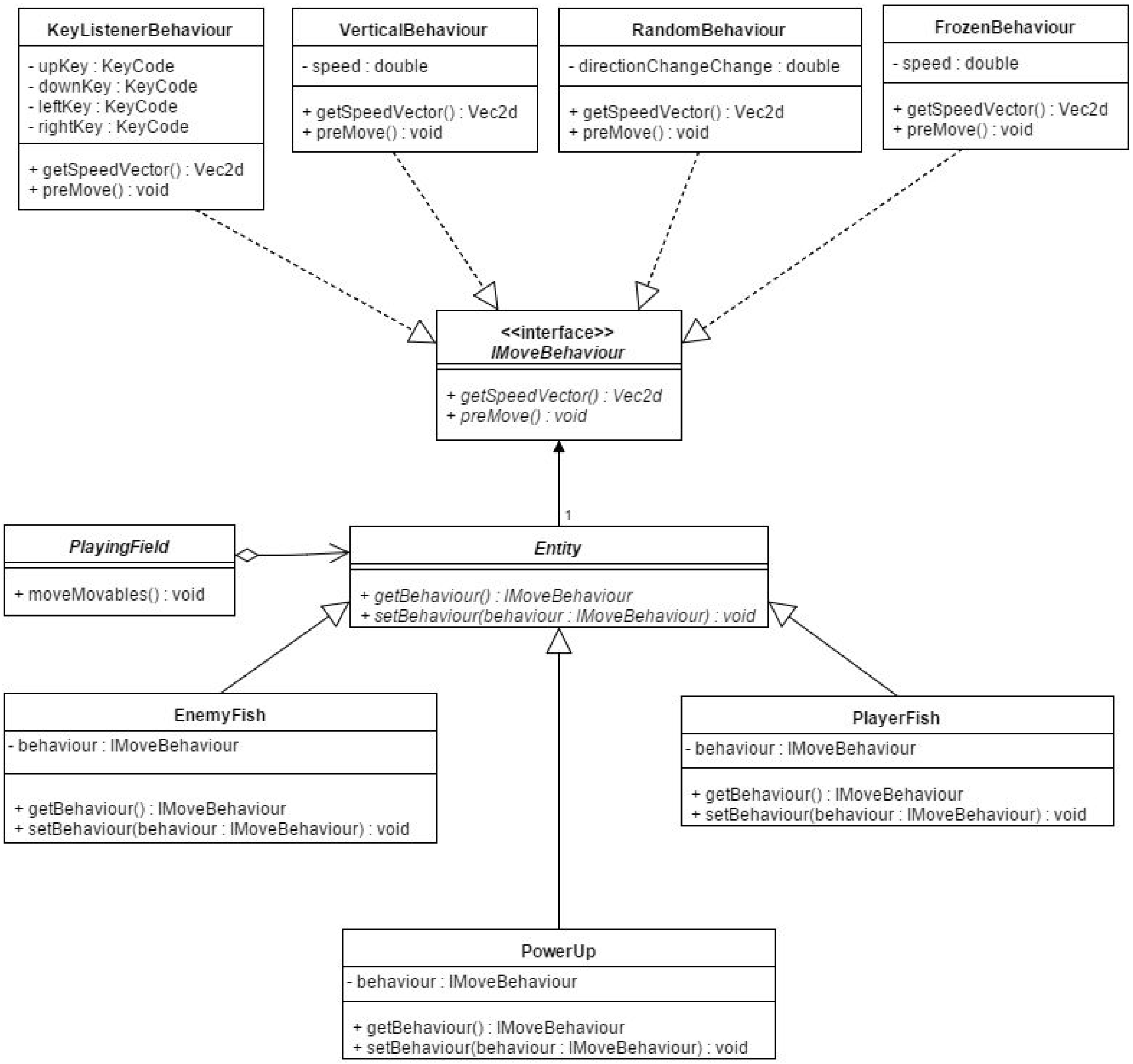

KeyListenerBehaviour
- upKey : KeyCode
- downKey : KeyCode
- leftKey : KeyCode
- rightKey : KeyCode
+ getSpeedVector() : Vec2d
+ preMove() : void
VerticalBehaviour
- speed : double
+ getSpeedVector() : Vec2d
+ preMove() : void
RandomBehaviour
- directionChangeChange : double
+ getSpeedVector() : Vec2d
+ preMove() : void
FrozenBehaviour
- speed : double
+ getSpeedVector() : Vec2d
+ preMove() : void
<<interface>>
IMoveBehaviour
+ getSpeedVector() : Vec2d
+ preMove() : void
1
PlayingField
+ moveMovables() : void
Entity
+ getBehaviour() : IMoveBehaviour
+ setBehaviour(behaviour : IMoveBehaviour) : void
EnemyFish
- behaviour : IMoveBehaviour
+ getBehaviour() : IMoveBehaviour
+ setBehaviour(behaviour : IMoveBehaviour) : void
PlayerFish
- behaviour : IMoveBehaviour
+ getBehaviour() : IMoveBehaviour
+ setBehaviour(behaviour : IMoveBehaviour) : void
PowerUp
- behaviour : IMoveBehaviour
+ getBehaviour() : IMoveBehaviour
+ setBehaviour(behaviour : IMoveBehaviour) : void

*Figure **3**: The class diagram of the behaviour implementation. This includes only the things that have changed as a result of the strategy pattern. The dead : boolean attribute of Entity is not included because it hasn't changed.*

3) Sequence diagram of example of use case of implementation.

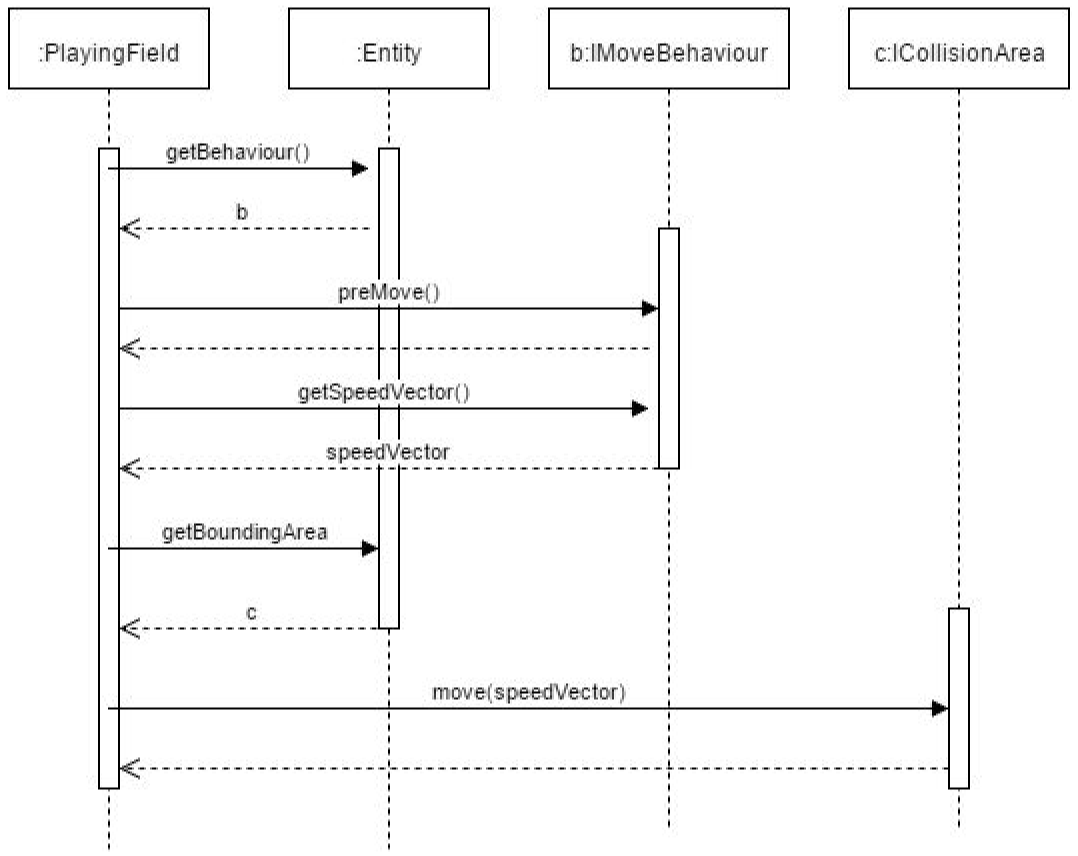


Figure 4: The sequence diagram of how behaviours are used to move entities. This entire sequence diagram is pretty much how the moveMovables() method works in the PlayingField class.

Exercise 3 - Software Engineering Economics

1) In the paper “How to Build a Good Practice Software Project Portfolio?” the authors used quite an extensive process to identify good and bad practises. The authors had a repository available of 352 projects. Of each project a lot of data was available, and with that data the authors tried finding the good and bad practises.

In order to make be able to identify what project factors were good and bad practises the authors had to prepare their data first, this took four steps:

Firstly, the authors analyzed the performance of all projects.They took the project size, cost and duration and computed an average of all the projects.

Secondly, they categorized each project into ‘good practise’ and ‘bad practise.’ The authors did this by comparing each project to the average performance of all projects. Each project has a different project size, measured in function points. In order to compare projects in a fair way, only projects with the same relative project size were considered. The authors created a Cost/Duration matrix from which it was possible to identify each project.

Thirdly, the authors used 56 different project factors that they thought could be important to project success and indexed these in the four quadrants of the Cost/Duration matrix. Each factor then had percentages per quadrant due to the different projects being in different quadrants in the C/D matrix and it was known what projects had what factors. The authors then assessed the outcome of that step by performing a calculation on the outcomes based on the chi-square test. This test allowed them to say with certainty whether certain factors belonged in the good or bad practise categories.

And fourthly, the authors identified factors strongly related to good or bad practises by analyzing specific subsets of factors in different research areas. For this analysis, the authors defined being strongly related as when the percentage of good or bad percentages was more than 50%.

After the authors found which factors were strongly related to good or bad practises they constructed a null and alternative hypothesis for each factor F:

1. H(F,0) means that F has no influence on good or bad practise.
2. H(F,1) means that F does influence good or bad practise.

In order to test these hypotheses the authors used the binomial distribution to estimate the chance that the observation done during factor categorization took place under the null hypothesis. The authors used the following formula:

$$p = (n\ nCr\ k) \cdot prob(F,\ n)^k \cdot (1 - prob(F,\ n))^{n-k} \ .$$

Where *p* is the chance of the null hypothesis being correct, n the total number of projects, *prob(F, n)* is the number of projects where factor F holds divided by n and k the number of projects for which factor F is good/bad practise holds.

The authors chose to reject the null hypothesis if p was smaller than 0.05, which means that a factor F could be classified as good/bad practise. So this entire process is the answer to the question.

2) In the study, visual basic only appeared on 6 projects out of 352. While visual basic scored really high in the good practise factors, such a small sample size might not give a good overview of visual basic projects.

Continuing on that, the authors explained that the visual basic projects were relatively small in size (27 to 587 function points). Visual basic projects also seem to be relatively simple because of the nature of the programming language.

All in all, visual basic does seem to provide a higher success rate in projects, but since it's only a small fraction of all projects and the majority of projects depend on another programming language, visual basic is not that interesting.

3) Three other factors that could have been studied in the article:

1. **Factor name:** Test Driven Design (TDD)
   Factor explanation: Test driven design is the methodology of designing test cases before
2. writing the code that is needed to pass them.
   Why would this factor be interesting to study: Test driven design is a topic that was thought to us in our first year and it seems to be a hot topic in development teams in the past few years. However, a lot of developers do not seem to like to work in the way that TDD demands. Seeing the actual impact of TDD would maybe teach us whether it is good to employ TDD development.
   Good or bad practise: TDD will likely result in more robust system, so less failing projects, as the system should have been tested thoroughly. But it seems that TDD might take more time to implement, so the duration might be longer than projects of relative size that follow another development cycle. Though we think that TDD should be counted as <u>good practise</u>.
3. **Factor name:** Use of UML in Object Orientated languages
   Factor explanation: UML documentation that took place before and during the project.
   Why would this factor be interesting to study: Because UML is an industry standard. To see whether companies really use it and how it impacts project success is interesting to both us students and companies to either encourage the use of UML,
   Good or bad practise: We think that good use of UML should result in good practise. Plus, since we think most companies will use UML in the correct way (to reduce risks by documenting assumptions and describing the main classes/points of the code base and or design), UML will end up in mostly the <u>good practise</u> quadrant.

4. **Factor name:** Code reviews
   Factor explanation: Making reviewing code mandatory before a specific code is implemented in the code base.
   Why would this factor be interesting to study: This factor is already used widely in developments teams, however it would be interesting to see if the use of code reviews has real impact on code quality. As an example: most developers think they use code review mainly for discovering bugs, but they actually seem to use it mainly for code maintainability!
   Good or bad practise: This factor should belong in the good practise category. Code reviews are used to improve the overall quality of the code, so this should be the expected outcome.

4) Three bad practises in explained in detail:

1. **Factor:** Many team changes, inexperienced team
   Details: Having a solid team of people that have built up experience in the system/development of a specific product is a really good practise for projects. When people are people in the team are replaced, a bit of that experience is lost. And when working with a team that has no experience with a certain language or system, there is a learning curve to overcome. If people are not given time to bridge over this gap than projects will surely cost more and take longer in the end.
2. **Factor:** Rules & Regulations driven
   Details: Each rule and regulation that is added to a system will restrict the development of that system by reducing the degree of freedom. Rules and regulations can make it harder to create solutions and sometimes require tedious functionalities to be implemented before a product can be shipped. It's no surprise to see this factor as bad practise in projects.
3. **Factor:** Dependencies with other systems
   Details: Dependencies with other systems also reduce the freedom of the solutions to a problem, just like rules and regulations. Even more problems can occurs as the systems, for which the product is build upon, can change, get outdated or even contain bugs themselves. It can be expensive and tedious to solve these errors as sometimes the current development team has no influence on the existing systems. Dependencies with other systems is bad practise when all these negative aspects are taken into account.