Marcel Kuipers	4389042	Remi Flinterman	4362950
Nils Hullegien	4389069	Luka Miljak	4368916
Damian Voorhout	4388550		

Architecture Design - Carried Away

1. Introduction

This document contains a description of our context project on computer games. We will talk about the design goals and the software architecture. This document will be modified when needed.

1.1 Design goals

Availability
Following the *Scrum* principle, at the end of each *sprint* a working product should be ready for users to try out. This way testers can give us feedback on the current system each week and the developers can adjust the system accordingly. It also makes sure that the game studio doesn't deviate the game from what the users actually want.

Performance
What every user wants in a game is very good performance. The Oculus Rift can be quite performance intensive for older systems and non gaming laptops. Therefore it's imperative that we focus on higher performance to make sure the game runs very smoothly.

Reliability
Just like performance, running into bugs during their gameplay experience ruins the fun. With each new feature, the amount of entropy in the system increases drastically. It should be the goal of the developers to keep that as low as possible. Every developer is responsible for their own code, so each of them will have to make sure that all their code is nearly bug free.

Manageability
When we want to change a part of the game, we want to be able to do this without any problems. To accomplish this, we want our game to be playable after every sprint, as well as keep our code modifiable. Every developer is responsible for their own code, so they'll each have to make sure all their code is kept manageable.

Scalability
As new features are added, the program not only becomes less and less manageable, the scalability is also compromised. When a developer adds a new feature, he or she should keep in mind if it affects the program as a whole for later expansion. Sometimes spending some more time on a feature to make the program future proof is very much worth it in the long run. Developers should consider this when implementing a feature and sometimes put in some extra work.

Securability

Since our game will be played on LAN only, we don't have to secure our game. When the internet connection that our game will be played on is safe, our game is safe.

2. Software architecture views

In this chapter the architecture is explained in the form of the components of the system, which will be split into subcomponents and subsystems.

2.1. Subsystem decomposition

The system has been divided into three subsystems: the Client Interface, Server Interface and Server Resources. The interaction between these systems is illustrated in figure 1.

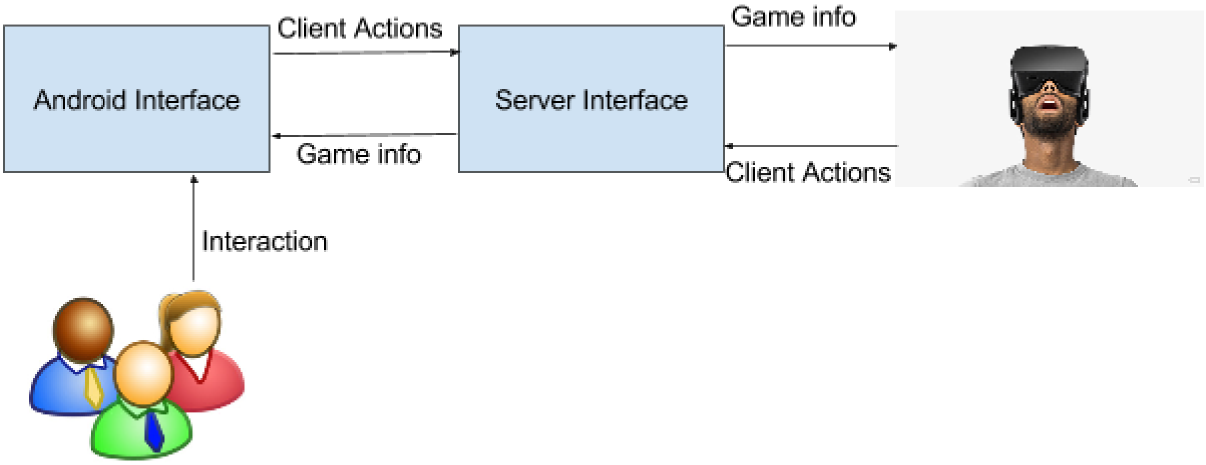

Figure 1. Architecture Diagram

In this section the subsystems are described in short:

- The Android Interface can be interacted with. The actions the users perform are sent to the server in order to influence the world.

- The Server Interface is used to connect every client with each other. It is responsible for handling the entire game. It sends information (for example the location of the player in the world) from the game to the Android users and the Oculus Rift user. What information the server sends to whom depends on the game design.

The Oculus Rift should be directly connected to the machine the server interface is running on. So everything done by the VR-user is directly being handled by the server interface, instead of through an extra interface like the Android interface.

2.2. Hardware-Software mapping

Both the hardware and the software for the server are not the same as the software and hardware for the Android devices. The Oculus Rift should be directly connected to the pc, which functions as a server, using an HDMI and a USB cable. Through the Server UI a lobby can be started, with which the Android users can connect using the Android UI and the Local Area Network everyone is connected to.

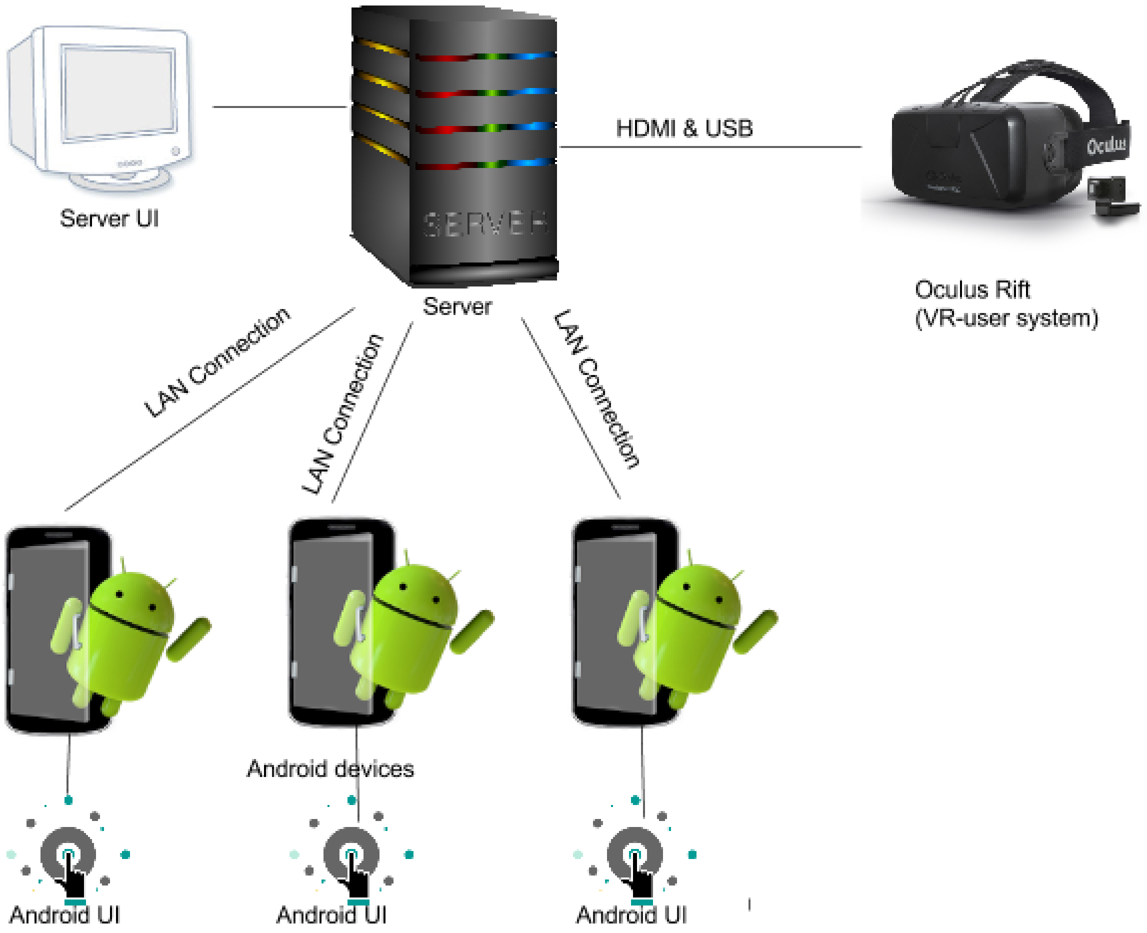

Figure 2. Illustration of the Hardware-Software mapping

2.3. Persistent data management

We won't use large amounts of memory, but we will keep our models, textures, animations, etc. in an assets folder. Scores are also saved in a designated text file, also located in the assets folder.

2.4. Concurrency Control

The Android devices repeatedly send messages to the Server and back. This may cause some concurrency problems if the Server receives too many messages at once. However, this application does not need to worry about that, as it uses the JMonkey networking framework. This framework ensures that messages are being sent by a single thread. Furthermore, the order in which messages are being sent is not important in this context.

3. Project Modules

Because of the clear distinction between the desktop application and the Android application, the project has been split up into three modules. These modules and their relations to each other are discussed in this section.

The modules are contained each within their own separate Maven project. A fourth Maven project has all three of these projects in its project folder and builds them consecutively. This setup is very simple and it easily allows the addition of more modules.

The Desktop Module
The Maven module ContextDesktop is meant for the server. This is what the computer, which has the Oculus Rift connected to it, must have.

The App Module
The ContextApp module is meant for Android devices. This module is used to deploy the application on an Android device.
Unlike the Desktop module, this one runs on *Java 1.7* instead of *Java 1.8*, because many Android devices do not support *Java 1.8*.
In order for this module to be installed, you need the Android SDK on your computer (which is automatically installed when also installing Android Studio).

The Messages Module
The ContextMessages module, unlike the other two modules, is more of a library than an application. Both the Desktop and the App module require the Message classes stored in here. These modules have this module as a dependency, so this module requires to be installed for the other two to even work.

The jme3-networking library requires the Message classes to be in both the server and the client, even though they're two completely different application. So instead of copy and pasting Message classes in those modules, having them in this module is a much cleaner way.

4. Class structure and implementation methods.

In this section, the structure of the implementations of many features of the game are described.

4.1 The desktop module

4.1.1 Main

For the main game, we need to configure the main to be compatible with VR, this is done in the VRConfigurer class.
We also need to handle key events, for example when switching to the debug camera, moving this camera around or exiting the game. This is all done within the InputHandler class, which is called with each update from the Main class.

4.1.2 Level generation

The following steps show the process of creating a world:

1. Initially, five random level pieces are generated.
2. The player is generated in the first level piece in line. In this context, the player is defined as the commander, the camera attached to the commander and the platform the commander is standing on.
3. The fly cam is generated for debugging purposes. The fly cam is explained in the subsection below.
4. The game starts.
5. While the game is running, the player will move forward at a constant speed. Whenever the player reaches the middle of the second level piece in the environment, the first piece is removed and a new random level piece is added. This way, there will always be five level pieces in the game.

4.1.3 Enemies

Enemies are created by the EnemyFactory, which can create 3 different enemies depending on where they need to be spawned. The EnemySpawner determines randomly when enemies need to spawn and when they need to be deleted. Only once per level piece enemies can spawn, the generateEnemies method determines on which level piece the commander currently is. The requirements for deleting an enemy are either the enemy being dead, or the enemy being left to far behind the player. Both the generating and destroying of enemies is done every game tick. A list of enemies is kept in the EnemySpawner class to keep track of how many enemies currently reside in the game. The maximum is twelve. The two methods return a list of enemies to be deleted or added by the Environment. The Environment is responsible for moving and displaying them.

4.2 The android module

This section will talk about the structure of the Android application.

4.2.1 The android Manifest

This file, located in *ContextApp/src/main* is used to configure the android application. So far, it is only used to define the class that is launched when the application is deployed, which right now is *MainApplication*.

4.2.2 MainActivity

This the activity that is started when the application is deployed. An activity is a class that implements the Activity superclass, which is imported. Activities are responsible for visualization and interaction with the hardware. *MainActivity* is a special activity, since it implements the *AndroidHarness* class, which is provided by jMonkey. This allows the activity to serve as a wrapper for a jMonkey application, which can be defined. MainActivity also projects the UI on the screen and contains the logging function for the buttons.

4.2.3 UI

The UI consists of multiple screens which all have their own XML file. A few screens are dedicated to loading and will show the player what is happening while they have to wait. The UI screens that are projected during gameplay show which position the player has carrying the platform, contain buttons used to defend against enemies, and have three hearts that represents the player's health. There's one in-game UI screen for each player/position.

4.2.4 Automatic Connection on LAN servers

One of the features on the Android, is that it can automatically connect to an open server lobby, that is running on the same LAN. Three classes are responsible for this. On the server side, there is the *ClientFinder* class, and on the client side there is the *ServerFinder* and *AutoConnector* class. Running both "*Finder*" classes (on different instances of the application) will make the *ServerFinder* find the ip addresses of every server that is currently running the *ClientFinder* class.
The following steps show the process of the client finding a server:

1. (ServerFinder) Sends a message with a certain password to every device on every network interface.
2. (ClientFinder) Receives a message and verifies using the password that the message really came from a *ServerFinder*.
3. (ClientFinder) Sends a message with a certain password to the device that sent the previous message.
4. (ServerFinder) Receives a message and verifies using the password that the message really came from a *ClientFinder*.
5. (ServerFinder) Can now do something useful with the ip-address of the server, such as connecting to the lobby.

So the responsibility of the *Finder* classes is finding ip addresses of servers that are running on the same LAN.

If an android device can't connect to the server within about 5 seconds, the app will be turned off by the alert function in MainActivity, which also shows a popup telling the player that the game should be turned on before trying to access the app.

The *AutoConnector* uses those two classes, to find the ip address of a running server. Then it uses that ip address to automatically connect to them.

4.2.5 Random events

Every once in a while, a random event will start for the android players. In this random event, a bug will have to be removed. This bug is found on one of the screens of the four android players. There will also be a spray. When the spray and the bug are on the same screen, the event will end and the main screen will be shown again. To get the spray and the bug together the players have to swipe the spray around. For example: when the spray is at position back left and that player swipes up, the spray will be at position front left. The bug is static on the screen it spawned on, so the spray has to be moved towards the bug.

4.3 The Message Module

4.3.1 Traffic between server and client.

The jme3-networking library is mainly used for the sending of messages between a server and client. The following steps need to be taken in order to send a certain message, the *AccelerometerMessage* is used as an example here:

1. Create a Message class that extends *AbstractMessage*. This class requires an @Serializable annotation and a public empty constructor (other constructors are still allowed, but this one is required).
2. Add any kind of attributes that serve as data that has to be sent. For example the *AccelerometerMessage* has x_force, y_force and z_force as attributes.
3. In both the *ServerWrapper* and *ClientWrapper* class (in the Desktop and App module respectively), the created message needs to be registered to the *Serializer.* This is done in the static initializer (*static { /* code here */}*) of the classes. The *AccelerometerMessage* is already there if an example is needed.
4. Now an instance of the message class can be sent from anywhere using the availible methods in the server (in the Desktop module) or in client (in the App module), which are both accessible from their respective wrapper classes. *AccelerometerSensor* for example repeatedly sends AccelerometerMessages.
5. Now a *MessageListener* is needed in the module that receives the message. To do this, the abstract *MessageListener<T>* class needs to be extended. *AccelerometerListener<AccelerometerMessage>* is such a class. The *messageReceived* method gets called every time it receives a message from a connection. Then the contents of that message can be used to do some intresting things, such as moving the platform.

There is a reason that this module contains a *MessageListener<T>* class while the jme3-networking library already contains such a class. The problem with the one from jme3 is that the *messageReceived* method gets called when ANY type of message is received. So an *instanceof* statement and a cast is required to process this message. This is a very bad code practice. This *MessageListener<T>* class however, only listens to one specific message.

4.4 Several class diagrams

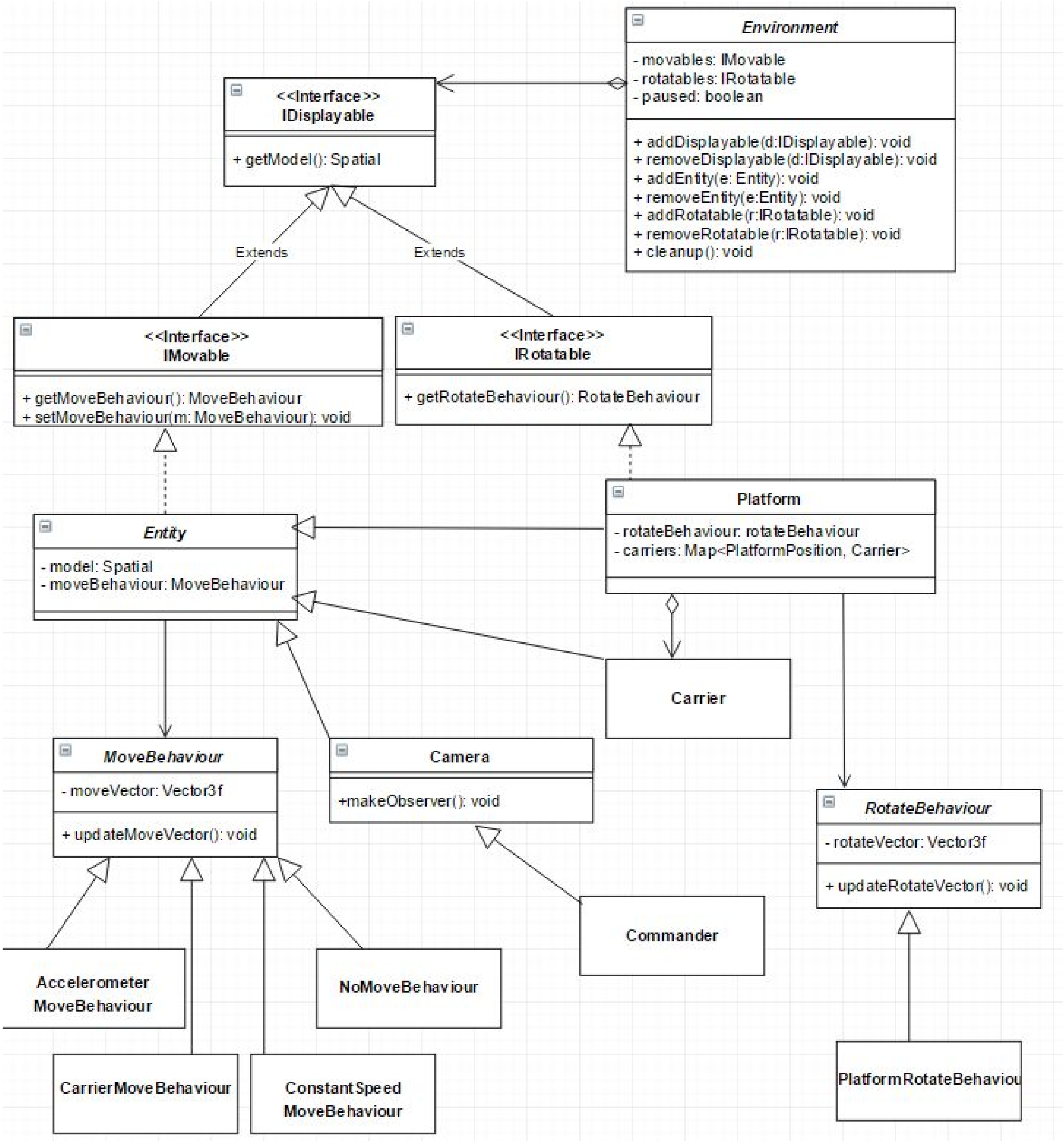
Environment
- movables: IMovable
- rotatables: IRotatable
- paused: boolean
+ addDisplayable(d:IDisplayable): void
+ removeDisplayable(d:IDisplayable): void
+ addEntity(e: Entity): void
+ removeEntity(e:Entity): void
+ addRotatable(r:IRotatable): void
+ removeRotatable(r:IRotatable): void
+ cleanup(): void
<<Interface>>
IDisplayable
+ getModel(): Spatial
Extends
Extends
<<Interface>>
IMovable
+ getMoveBehaviour(): MoveBehaviour
+ setMoveBehaviour(m: MoveBehaviour): void
<<Interface>>
IRotatable
+ getRotateBehaviour(): RotateBehaviour
Platform
- rotateBehaviour: rotateBehaviour
- carriers: Map<PlatformPosition, Carrier>
Entity
- model: Spatial
- moveBehaviour: MoveBehaviour
Carrier
MoveBehaviour
- moveVector: Vector3f
+ updateMoveVector(): void
Camera
+makeObserver(): void
RotateBehaviour
- rotateVector: Vector3f
+ updateRotateVector(): void
Commander
Accelerometer
MoveBehaviour
NoMoveBehaviour
CarrierMoveBehaviour
ConstantSpeed
MoveBehaviour
PlatformRotateBehaviou

Diagram 1. The Environment and the strategy pattern.
This diagram shows the basics of how the Environment is set up.

A *strategy* design pattern is also hidden in this diagram. Each *Entity* contains an abstract *MoveBehaviour* (the strategy), which contain some concrete strategy implementations (*AccelerometerMoveBehaviour, NoMoveBehaviour, etc*). The same applies to the *RotateBehaviour*, which is another strategy.

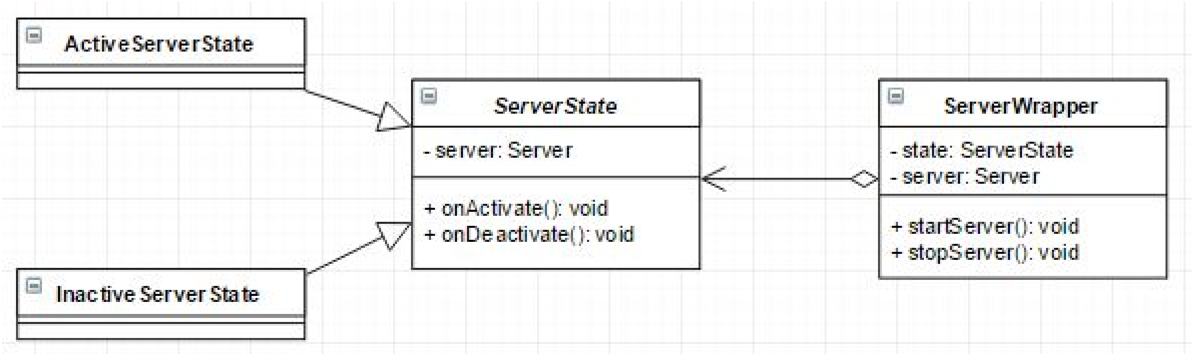

Diagram 2. The Serverwrapper and the state pattern.
This diagram shows the basics of the ServerWrapper and its states. The *ServerWrapper* contains two different states, the *ActiveServerState* and *InactiveServerState*, both which have their own implementation of what happens when it's activated and deactivated.
The *ClientWrapper* in the *ContextApp* module has the exact same design.

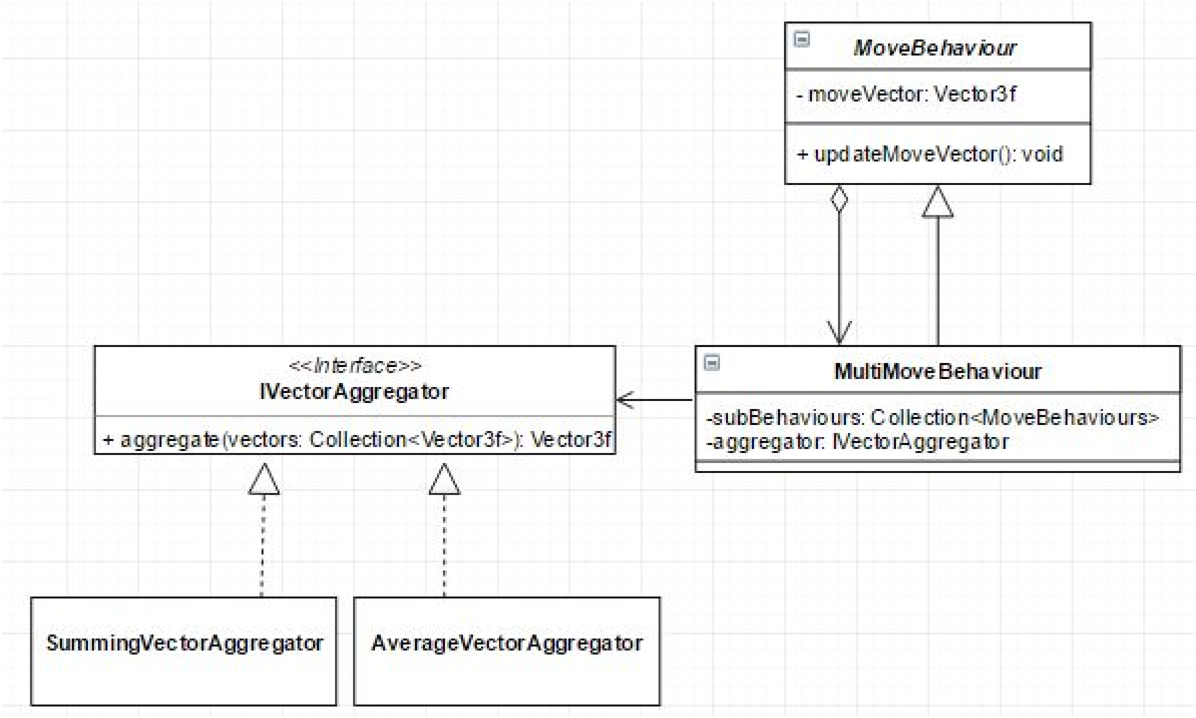
MoveBehaviour
- moveVector: Vector3f
+ updateMoveVector(): void
<<Interface>>
IVectorAggregator
+ aggregate(vectors: Collection<Vector3f>): Vector3f
MultiMoveBehaviour
-subBehaviours: Collection<MoveBehaviours>
-aggregator: IVectorAggregator
SummingVectorAggregator
AverageVectorAggregator

Diagram 3. The MultiMoveBehaviour and the composite + strategy pattern
This diagram shows the basics of the *MultiMoveBehaviour* class. It's a *MoveBehaviour* that is composed of multiple other behaviours, which allows for more intresting and complex behaviours for certain entities. This relates to the composite design pattern, because the *MoveBehaviour* can be seen as the component, the *MultiMoveBehaviour* as the composite and all other behaviours as the leaves.

The *MultiMoveBehaviour* combines all the other behaviours using an aggregator, which is there to ‘summarize’ a collection of vectors into a single vector. Examples of such aggregators are the *SummingVectorAggregator*, which takes the sum of all vectors and the *AverageVectorAggregator*, which takes the average. Thes *IVectorAggregator* can be seen as a strategy, giving the strategy pattern, mixed with the composite pattern.

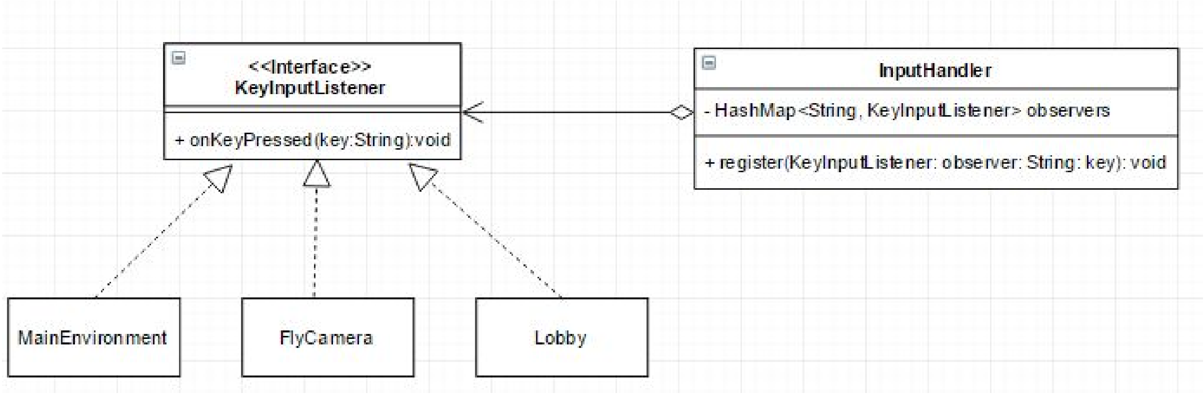
<<Interface>>
KeyInputListener
+ onKeyPressed(key:String):void
InputHandler
- HashMap<String, KeyInputListener> observers
+ register(KeyInputListener: observer: String: key): void
MainEnvironment
FlyCamera
Lobby

Diagram 4. The InputHandler and the observer pattern.

The *InputHandler* (a singleton class) listens to KeyEvents. However these events aren’t being processed within this class. Classes that implement the *KeyInputListener* interface do that. They can register themselves to the *InputHandler* using a specific key and their onKeyPressed method gets called when a key event happens with the registered key. This simulates the observer pattern as the InputHandler represents the subject and the listeners represent the observers. The only difference between many generalized observer patterns and this one, is that the *InputHandler* does extends/implement a Subject class/interface.

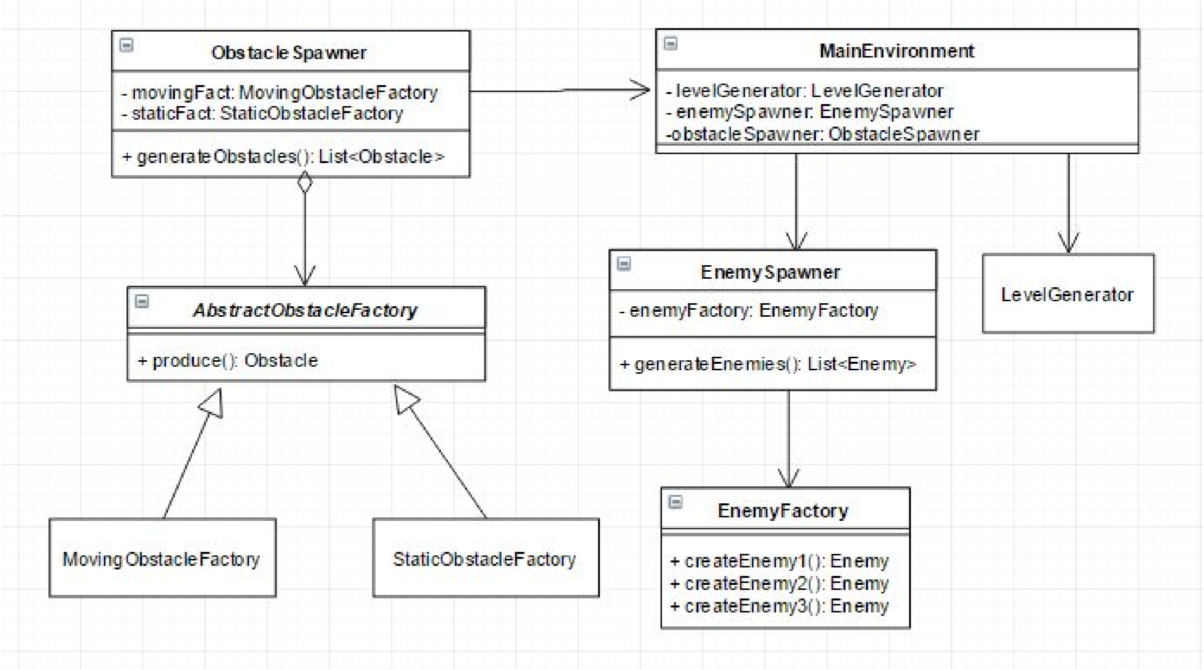
ObstacleSpawner
- movingFact: MovingObstacleFactory
- staticFact: StaticObstacleFactory
+ generateObstacles(): List<Obstacle>
MainEnvironment
- levelGenerator: LevelGenerator
- enemySpawner: EnemySpawner
-obstacleSpawner: ObstacleSpawner
AbstractObstacleFactory
+ produce(): Obstacle
EnemySpawner
- enemyFactory: EnemyFactory
+ generateEnemies(): List<Enemy>
LevelGenerator
MovingObstacleFactory
StaticObstacleFactory
EnemyFactory
+ createEnemy1(): Enemy
+ createEnemy2(): Enemy
+ createEnemy3(): Enemy

Diagram 5. The spawners and the Abstract factory pattern.

The MainEnvironment repeatedly needs to produce and remove enemies, obstacles and levelpieces. It has three classes that does the job for him: The *ObstacleSpawner*, *EnemySpawner* and *LevelGenerator*. The LevelGenerator is the simplest one, is just generates several LevelPieces. The creation of enemies is slightly more complicated, that's why EnemySpawner makes use of an *EnemyFactory*, which creates the enemies.
The *ObstacleSpawner* is more complicated as it contains multiple factories for different types of obstacles. That's why there is an abstract *ObstacleFactory* class created that represents these.

5. Glossary

Scrum: A software development framework that uses agile designing and rapid prototyping. It requires a working version of the software to be delivered at the end of each 1-week sprint.

Sprint: An iteration in the scrum process. Every sprint has the same length, with a maximum of time one month. This project uses sprints of 1 week each.

Appendix A: CRC Cards

A.1 ContextApp

Class name: MainActivity	
Superclasses: Activity	
Subclasses: x	
Responsibilities: To set all processes in motion to load the app and its functions. To set the main UI and its functionality and project it on the screen. To update the player’s hearts.	*Collaborators:* AccelerometerSensor AttackMessenger HealthMessageHandler HitMissMessageHandler MakeButtonFunctions SfxPlayer StartBugEventMessageListener

Class name: AccelerometerSensor	
Superclasses: Activity	
Subclasses: x	
Responsibilities: To send the server data from the android device’s accelerometer.	*Collaborators:* MainActivity

Class name: AttackMessenger	
Superclasses: x	
Subclasses: x	
Responsibilities: To send the server a message when the player presses an attack button	*Collaborators:* MainActivity

Class name: HealthMessageHandler	
Superclasses: MessageListener	

Subclasses: x	
Responsibilities: To listen for messages with health data and call the functions that use it.	*Collaborators:* MainActivity

Class name: HitMissMessageHandler	
Superclasses: MessageListener	
Subclasses: x	
Responsibilities: To listen for messages with with a boolean that represents whether or not the player's attack hit an enemy and make the right consequences happen.	*Collaborators:* MainActivity

Class name: MakeButtonFunctions	
Superclasses: x	
Subclasses: x	
Responsibilities: To set the text and functions of the attack buttons in the UI.	*Collaborators:* MainActivity

Class name: PositionHolder	
Superclasses: MessageListener	
Subclasses: x	
Responsibilities: To receive messages from the server with the player's PlatformPosition and hold this data.	*Collaborators:* MainActivity

Class name: SfxPlayer
Superclasses: x
Subclasses: x

Responsibilities: To play sound effects.	*Collaborators:* MainActivity

Class name: StartBugEventMessageListener	
Superclasses: MessageListener	
Subclasses: x	
Responsibilities: To listen for messages that tell it to initiate the bug minigame.	*Collaborators:* MainActivity

Class name: EnableSprayMessageHandler	
Superclasses: MessageListener	
Subclasses: x	
Responsibilities: To listen for messages that tell whether or not the player has the bugspray for the bug minigame.	*Collaborators:* RotateBugSprayActivity

Class name: RotateBugSprayActivity	
Superclasses: Activity	
Subclasses: x	
Responsibilities: To start the bug minigame, its functions and UI.	*Collaborators:* MainActivity StopAllEventsMessageListener

Class name: StopAllEventsMessageListener	
Superclasses: MessageListener	
Subclasses: x	
Responsibilities: To stop the bug minigame when the message is received.	*Collaborators:* RotateBugSprayActivity

Class name: ActiveClientState	
Superclasses: ClientState	
Subclasses: x	
Responsibilities: To represent the client that is in an active state.	*Collaborators:* ClientWrapper

Class name: AutoConnector	
Superclasses: x	
Subclasses: x	
Responsibilities: To automatically connect to the server when initiated.	*Collaborators:* ClientHub

Class name: ClientHub	
Superclasses: x	
Subclasses: x	
Responsibilities: To allow for both Activities in the collaborators section of this card to use the same client.	*Collaborators:* MainActivity RotateBugSprayActivity

Class name: ClientState	
Superclasses: x	
Subclasses: ActiveClientState, InactiveClientState	
Responsibilities: To represent an abstract state for the client.	*Collaborators:* ClientWrapper

Class name: ClientWrapper
Superclasses: x

Subclasses: x	
Responsibilities: To create a client. To start or stop a client. To switch the client’s state.	*Collaborators:* AccelerometerSensor AttackMessenger MainActivity RotateBugSprayActivity AutoConnector ClientHub

Class name: InactiveClientState	
Superclasses: ClientState	
Subclasses: x	
Responsibilities: To represent the client that is in an inactive state.	*Collaborators:* ClientWrapper

Class name: ServerFinder	
Superclasses: x	
Subclasses: x	
Responsibilities: To find a server to connect to. To allow for communication with the server.	*Collaborators:* AutoConnector

A.2 ContextDesktop

Class name: Main	
Superclasses: VRApplication	
Subclasses: x	
Responsibilities: To set everything in motion so the game starts running.	*Collaborators:* HUDController Environment MainEnvironment Carrier InputHandler

<table>
<tr><td></td><td>VRConfigurer
Messengers and MessageListeners</td></tr>
</table>

<table>
<tr><td colspan="2"><i>Class name:</i> AudioController</td></tr>
<tr><td colspan="2"><i>Superclasses:</i> x</td></tr>
<tr><td colspan="2"><i>Subclasses:</i> x</td></tr>
<tr><td><i>Responsibilities:</i>
To play background music.</td><td><i>Collaborators:</i>
Environment</td></tr>
</table>

<table>
<tr><td colspan="2"><i>Class name:</i> Enemy</td></tr>
<tr><td colspan="2"><i>Superclasses:</i> Entity</td></tr>
<tr><td colspan="2"><i>Subclasses:</i> x</td></tr>
<tr><td><i>Responsibilities:</i>
To be an entity that targets carriers and attacks them, making the carriers take damage, and that can take damage and get killed.</td><td><i>Collaborators:</i>
EnemyFactory
EnemyMoveBehaviour
EnemySpawner
MainEnvironment
Carrier
EnemySpot</td></tr>
</table>

<table>
<tr><td colspan="2"><i>Class name:</i> EnemyFactory</td></tr>
<tr><td colspan="2"><i>Superclasses:</i> x</td></tr>
<tr><td colspan="2"><i>Subclasses:</i> x</td></tr>
<tr><td><i>Responsibilities:</i>
To make enemies spawn on level pieces, which can then move according to their AI.</td><td><i>Collaborators:</i>
EnemySpawner</td></tr>
</table>

<table>
<tr><td colspan="2"><i>Class name:</i> EnemySpawner</td></tr>
<tr><td colspan="2"><i>Superclasses:</i> x</td></tr>
<tr><td colspan="2"><i>Subclasses:</i> x</td></tr>
<tr><td><i>Responsibilities:</i>
To spawn enemies using the</td><td><i>Collaborators:</i>
MainEnvironment</td></tr>
</table>

EnemyFactory and random number generation.	

Class name: EnemyMoveBehaviour	
Superclasses: EntityMoveBehaviour	
Subclasses: x	
Responsibilities: To control how an enemy moves	*Collaborators:* Enemy

Class name: HUDController	
Superclasses: x	
Subclasses: x	
Responsibilities: To project the HUD of the game on the screen.	*Collaborators:* MainEnvironment

Class name: InputHandler	
Superclasses: x	
Subclasses: x	
Responsibilities: To make inputs have functionality.	*Collaborators:* Main

Class name: LobbyHUDController	
Superclasses: x	
Subclasses: x	
Responsibilities: To project the title screen and connected players on the screen.	*Collaborators:* LobbyEnvironment

Class name: VRConfigurer

Superclasses: x	
Subclasses: x	
Responsibilities: To set configurations so the game works with the Oculus Rift VR headset.	*Collaborators:* Main

Class name: Environment	
Superclasses: AbstractAppState	
Subclasses: LobbyEnvironment, MainEnvironment	
Responsibilities: To represent an Environment which consists of the 3D environment and its HUD	*Collaborators:* x

Class name: LevelGenerator	
Superclasses: x	
Subclasses: x	
Responsibilities: To randomly generate the level during gameplay using levelPieces	*Collaborators:* MainEnvironment

Class name: LevelPiece	
Superclasses: x	
Subclasses: x	
Responsibilities: These are the pieces the level consists of	*Collaborators:* EnemyFactory EnemySpawner LevelGenerator MainEnvironment

Class name: LobbyEnvironment	
Superclasses: Environment	

Subclasses: x	
Responsibilities: This is the environment that contains the lobby screen.	*Collaborators:* Main

Class name: MainEnvironment	
Superclasses: Environment	
Subclasses: x	
Responsibilities: This is the environment the game is played in.	*Collaborators:* Main Carrier Platform CarrierMoveBehaviour MovingObstacleFactory ObstacleSpawner

Class name: Path	
Superclasses: Entity	
Subclasses: x	
Responsibilities: To be an entity that is a part of the level and is visible to the commander.	*Collaborators:* LevelGenerator MainEnvironment

Class name: Camera	
Superclasses: Entity	
Subclasses: Commander	
Responsibilities: To be the point and angle in the environment from which the level is projected on the screen.	*Collaborators:* MainEnvironment

Class name: Carrier	
Superclasses: Entity	

Subclasses: x	
Responsibilities: To represent a carrier with all the data bound to him. To change the data of the carrier, such as their enemySpots or their health	*Collaborators:* Enemy EnemyFactory EnemyMoveBehaviour EnemySpawner EnemySpot MainEnvironment Platform CarrierMoveBehaviour AttackMessageHandler

Class name: CarrierAssigner	
Superclasses: x	
Subclasses: x	
Responsibilities: To assign carriers (the real life players using an android device) to the positions and vice versa.	*Collaborators:* Main LobbyEnvironment MainEnvironment Platform

Class name: Commander	
Superclasses: Camera	
Subclasses: x	
Responsibilities: To be the viewpoint of the commander.	*Collaborators:* EnemySpawner MainEnvironment EnemySpot CarrierMoveBehaviour ObstacleSpawner

Class name: EnemySpot	
Superclasses: x	
Subclasses: x	
Responsibilities: To allow for an enemy to target and attack	*Collaborators:* Enemy

a carrier from their location, which is relative to the carrier.	EnemyMoveBehaviour Carrier

Class name: Entity	
Superclasses: x	
Subclasses: Enemy, Path, Camera, Carrier, Platform, Obstacle	
Responsibilities: To represent an element in the environment that has a model and MoveBehaviour	*Collaborators:* Environment MainEnvironment

Class name: Platform	
Superclasses: Entity	
Subclasses: x	
Responsibilities: This is the platform the commander is (virtually) carried on.	*Collaborators:* MainEnvironment Commander

Class name: AcceleratingMoveBehaviour	
Superclasses: EntityMoveBehaviour	
Subclasses: x	
Responsibilities: To let the platform (with commander and carriers) slowly accelerate.	*Collaborators:* Platform

Class name: AccelerometerMoveBehaviour	
Superclasses: MoveBehaviour	
Subclasses: x	
Responsibilities: To allow the accelerometer data from the carriers affect the movements of the platform.	*Collaborators:* Platform PlatformRotateBehaviour

Class name: CarrierMoveBehaviour	
Superclasses: EntityMoveBehaviour	
Subclasses: x	
Responsibilities: To control the movements of the carriers.	*Collaborators:* x

Class name: ConstantSpeedMoveBehaviour	
Superclasses: EntityMoveBehaviour	
Subclasses: x	
Responsibilities: To allow entities to move at a constant speed.	*Collaborators:* x

Class name: EntityMoveBehaviour	
Superclasses: MoveBehaviour	
Subclasses: EnemyMoveBehaviour, AcceleratingMoveBehaviour, CarrierMoveBehaviour, ConstantSpeedMoveBehaviour	
Responsibilities: To represent an abstract MoveBehaviour that is used by entities.	*Collaborators:* MainEnvironment

Class name: MoveBehaviour	
Superclasses: x	
Subclasses: AccelerometerMoveBehaviour, EntityMoveBehaviour, MovingObstacleMoveBehaviour, MultiMoveBehaviour, StaticMoveBehaviour	
Responsibilities: To represent an abstract MoveBehaviour, the way an object moves and with what cause.	*Collaborators:* Entity MultiMoveBehaviour PlatformRotateBehaviour

Class name: MovingObstacleMoveBehaviour	

Superclasses: MoveBehaviour	
Subclasses: x	
Responsibilities: To represent the MoveBehaviour of MovingObstacles.	*Collaborators:* MovingObstacle

Class name: MultiMovebehaviour	
Superclasses: MoveBehaviour	
Subclasses: x	
Responsibilities: To let multiple MoveBehaviours control how an object moves.	*Collaborators:* Platform

Class name: PlatformRotateBehaviour	
Superclasses: RotateBehaviour	
Subclasses: x	
Responsibilities: To allow the platform to rotate according to the accelerometer data from the carriers.	*Collaborators:* Platform

Class name: RotateBehaviour	
Superclasses: x	
Subclasses: PlatformRotateBehaviour	
Responsibilities: To allow for objects to rotate.	*Collaborators:* Commander Platform

Class name: StaticMoveBehaviour	
Superclasses: MoveBehaviour	
Subclasses: x	

Responsibilities: To represent the movements of static objects (they don’t move)	*Collaborators:* Entity StaticObstacle

Class name: AbstractObstacleFactory	
Superclasses: x	
Subclasses: MovingObstacleFactory, StaticObstacleFactory	
Responsibilities: To represent an abstract obstacle factory.	*Collaborators:* x

Class name: MovingObstacle	
Superclasses: Obstacle	
Subclasses: x	
Responsibilities: This is a moving obstacle.	*Collaborators:* MovingObstacleMoveBehaviour MovingObstacle

Class name: MovingObstacleFactory	
Superclasses: AbstractObstacleFactory	
Subclasses: x	
Responsibilities: To produce MovingObstacles	*Collaborators:* ObstacleSpawner

Class name: Obstacle	
Superclasses: Entity	
Subclasses: MovingObstacle, StaticObstacle	
Responsibilities: To be an entity that damages all carriers when collided with.	*Collaborators:* MainEnvironment AbstractObstacleFactory MovingObstacleFactory ObstacleSpawner

Class name: ObstacleSpawner	
Superclasses: x	
Subclasses: x	
Responsibilities: To place obstacles in the level while it is generated.	*Collaborators:* MainEnvironment

Class name: StaticObstacle	
Superclasses: Obstacle	
Subclasses: x	
Responsibilities: This is a static Obstacle	*Collaborators:* StaticObstacleFactory

Class name: StaticObstacleFactory	
Superclasses: AbstractObstacleFactory	
Subclasses: x	
Responsibilities: To produce static obstacles.	*Collaborators:* ObstacleSpawner

Class name: Score	
Superclasses:	
Subclasses: x	
Responsibilities: To hold the score data, which consist of the score (integer) and the score holder (string)	*Collaborators:* MainEnvironment ScoreController ScoreReader ScoreWriter

Class name: ScoreController	
Superclasses: x	

Subclasses: x	
Responsibilities: To read the high scores from a file and project them on the lobby/title screen.	*Collaborators:* LobbyHUDController LobbyEnvironment MainEnvironment

Class name: ScoreReader	
Superclasses: x	
Subclasses: x	
Responsibilities: To read the high scores from a file	*Collaborators:* ScoreController

Class name: ScoreWriter	
Superclasses: x	
Subclasses: x	
Responsibilities: To project high scores onto the UI in the lobby/title screen.	*Collaborators:* ScoreController

Class name: ActiveServerState	
Superclasses: ServerState	
Subclasses: x	
Responsibilities: To represent a server state that is active	*Collaborators:* ServerWrapper

Class name: AttackMessageHandler	
Superclasses: MessageListener	
Subclasses: x	
Responsibilities: To listen for messages that contain a direction, which is used to execute an	*Collaborators:* Carrier

attack.	

Class name: ClientFinder	
Superclasses: x	
Subclasses: x	
Responsibilities: To look for android devices running the app to connect with.	*Collaborators:* Main

Class name: EnableSprayToVRMessageHandler	
Superclasses: MessageListener	
Subclasses: x	
Responsibilities: To listen for messages that tells the program to move the bugspray over to another carrier.	*Collaborators:* Main

Class name: HealthMessenger	
Superclasses: x	
Subclasses: x	
Responsibilities: To send a message to the client with an integer representing the carrier's new health value.	*Collaborators:* Carrier

Class name: HitMissMessenger	
Superclasses: x	
Subclasses: x	
Responsibilities: To send a message to a client telling it whether or not he/she has succesfully hit an enemy.	*Collaborators:* Carrier

Class name: InactiveServerState	
Superclasses: ServerState	
Subclasses: x	
Responsibilities: To represent a server state which is inactive	*Collaborators:* ServerWrapper

Class name: ServerState	
Superclasses: ActiveServerState, InactiveServerState	
Subclasses: x	
Responsibilities: To represent an abstract server state	*Collaborators:* ServerWrapper

Class name: ServerWrapper	
Superclasses: x	
Subclasses: x	
Responsibilities: To create a server. To start and stop the server To switch between server states.	*Collaborators:* Main CarrierAssigner HealthMessenger HitMissMessenger

Class name: StopEventMessageHandler	
Superclasses: MessageListener	
Subclasses: x	
Responsibilities: To listen for messages that tell it that the bug minigame event has ended, so that the other android devices can be informed of this as well.	*Collaborators:* Main

Class name: AverageVectorAggregator	
Superclasses: x	
Subclasses: x	
Responsibilities: To average a collection of vectors into one vector.	*Collaborators:* Platform DistanceVectorAggregator

Class name: DistanceVectorAggregator	
Superclasses: x	
Subclasses: x	
Responsibilities: To give a vector as indication for the similarity between all the vectors that are received (used for steering and rotating)	*Collaborators:* PlatformRotateBehaviour

Class name: ProjectAssetManager	
Superclasses: x	
Subclasses: x	
Responsibilities: To make the asset manager accessible from any class.	*Collaborators:* Every class that pulls assets from the asset manager.

Class name: SummingVectorAggregator	
Superclasses: x	
Subclasses: x	
Responsibilities: To give a vector that is the sum of all vectors that are received.	*Collaborators:* Platform AverageVectorAggregator.

A.3 ContextMessages

Class name: AccelerometerMessage	
Superclasses: AbstractMessage	
Subclasses: x	
Responsibilities: To contain the Accelerometer data from the android device the message is sent from.	*Collaborators:* AccelerometerMoveBehaviour ServerWrapper AccelerometerSensor ClientWrapper

Class name: AttackMessage	
Superclasses: AbstractMessage	
Subclasses: x	
Responsibilities: To contain a Direction and PlatformPosition from a carrier who attacked, which allows for the attack to be executed on the server.	*Collaborators:* AttackMessageHandler ServerWrapper AttackMessenger ClientWrapper

Class name: EnableSprayToAppMessage	
Superclasses: AbstractMessage	
Subclasses: x	
Responsibilities: To allow for the server to give bug spray to one carrier during the bug minigame event.	*Collaborators:* Main ServerWrapper EnableSprayAppMessageHandler ClientWrapper

Class name: EnableSprayToVRMessage	
Superclasses: AbstractMessage	
Subclasses: x	
Responsibilities: To allow the client to send a message to the server, so that the bugspray can be moved from one player to another.	*Collaborators:* EnableSprayToVRMessageHandler ServerWrapper RotateBugSprayActivity

	ClientWrapper

Class name: HealthMessage	
Superclasses: AbstractMessage	
Subclasses: x	
Responsibilities: To contain a health value which is used by the client to update its hearts.	*Collaborators:* HealthMessenger ServerWrapper HealthMessageHandler ClientWrapper

Class name: HitMissMessage	
Superclasses: AbstractMessage	
Subclasses: x	
Responsibilities: To contain a boolean which is used by the client to check whether or not hit an enemy or not	*Collaborators:* HitMissMessenger ServerWrapper HitMissMessageHandler ClientWrapper

Class name: MessageListener	
Superclasses: com.jme3.network.MessageListener	
Subclasses: AttackMessageHandler, EnableSprayToVRMessageHandler, StopEventMessageHandler, HealthMessageHandler, HitMissMessageHandler, PositionHolder, StartBugEventMessageListener, EnableSprayAppMessageHandler, StopAllEventsMessageListener	
Responsibilities: To allow for receiving messages which are used to call methods to manipulate the data on their side.	*Collaborators:* x

Class name: PositionMessage	
Superclasses: x	

Subclasses: x	
Responsibilities: To contain the PlatformPosition so that the android device carriers can be identified and that the UI can show the players in which position they are.	*Collaborators:* CarrierAssigner ServerWrapper PositionHolder ClientWrapper

Class name: StartBugEventMessage	
Superclasses: AbstractMessage	
Subclasses: x	
Responsibilities: To allow for the server to communicate with the android devices, so the bug minigame event can be initiated.	*Collaborators:* MainEnvironment ServerWrapper StartBugEventMessageListener ClientWrapper

Class name: StopAllEventsMessage	
Superclasses: AbstractMessage	
Subclasses: x	
Responsibilities: To allow for the server to communicate to the android devices that the bug minigame event has ended.	*Collaborators:* Main ServerWrapper StopAllEventsMessageListener ClientWrapper

Class name: StopEventToVRMessage	
Superclasses: AbstractMessage	
Subclasses: x	
Responsibilities: To allow for the client to communicate to the server that the bug minigame event has ended.	*Collaborators:* ServerWrapper StopEventMessageHandler RotateBugSprayActivity ClientWrapper