System Design Document for PixelWarfare

Version: 1.0
Date: 2017-05-29
Author: Group 26 - Codetroopers
This version overrides all previous versions.

Introduction

Design Goals:

PixelWarfare is an Augmented-reality multiplayer online game for smartphones. It consists of two scopes, the first one is Server and the other is that of the Client. At any given moment one instance of the Server should be running to serve connected clients.
Both implementations of Server and Client must be decoupled to guarantee freedom of implementation.
To ensure the robustness of PixelWarfare's Server and Client parts, designing and implementing the logical components shouldn't in any way depend on:

- The method of communicating: *WebSockets*
- Data representation: *JSON, XML etc*
- Client-part platform: *Android or iOS*.
- Client-part specific technologies *e.g. GoogleMaps*
- Server-side's Model independence and ease of modification.

It is focal to keep the Model parts easily testable, to ensure desired performance quality and robustness.

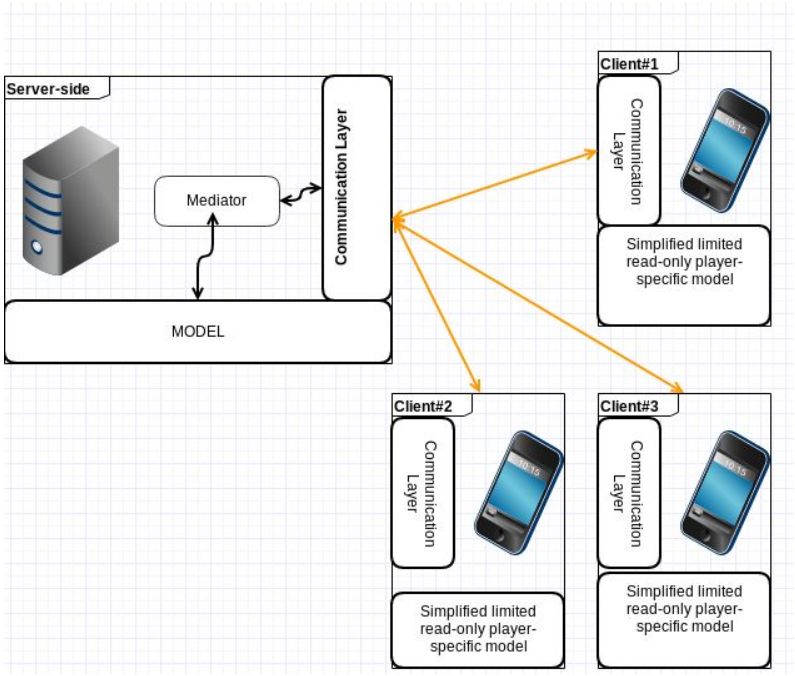
Definitions, acronyms and abbreviations

Server: The program that contains the Model part and handles incoming requests and sends updates.
Client: The Android app that is used by one and only player that reflects the player's current state, and transfers commands to the Server's model.
Connection Layer: The part that lies on top of the Server and Client and manages the incoming/outgoing data.
Exp: Experience points gained or lost at different points during the game.
Hp: Health points, a percentage.
Radar Status: Binary state (online/offline) of being able to be seen and being able to see and interact with other in-game objects.
GUI: Graphical User Interface.
MVC: Model-view-controller pattern.
JVM: Java Virtual Machine.
POJO: Plain Old Java Object.

System Architecture

As mentioned, and as the case is for almost every multiplayer game, this project consists of two parts, each with its own set of goals and functions. The design of those two scopes has been in its own a project. In the following sections, the general structure and design choices for the two parts are explained, followed by an explanation of how those parts are supposed to communicate.

The game is played on smartphones, and the implemented client is written for Android, whilst the Server-side part is written in Java and can be hosted on any machine that runs JVM.



In this software there are two main parts, namely the Model one and the part that controls all incoming and outgoing information. To avoid direct dependency between those two parts, a third component -Mediator- is added to act as a connecting ring between the Model and Server and to make sure that those two are separated. The model of the game refers to all classes needed to represent the game.

The server-side is event-driven i.e. the model remains in its state unless is asked upon to do something. There is no main loop that updates all connected clients with a certain interval, rather every client is informed when they're supposed to be updated. This strategy leads to significant drop in CPU and network usage on both ends.

The whole game can be seen as an MVC-based system, where the parts are distributed onto different machines: As stated earlier, the Model lies on the server side, whilst the View is hosted on the clients (permitting different visual representations of the game). The Controller is split between the Server-side and the Client-side. This level of abstraction allows for varying implementations of the Client, whereas the Server-side can stay the same.

A key guideline is to rely on interfaces whenever possible due to the fact that it separates what the caller expects from the implementation. Having a pure set of methods one can call without any knowledge of the implementation is a strength in the meaning that it makes extensibility possible and feasible.

Subsystems

Server side

Server-side is the main component serving data to clients and is where the Model lies. Any actions that pertain to the Model can only be performed on the Server-side.

The Model part of the Server-side consists of the following main classes:

- World class: It's an aggregator of players and loot-boxes and any other possible extensions to the game. World class serves as the direct interface to the game logic as seen from the Mediator i.e. it acts on the requests coming from the Mediator. Having stated that, this does *not* mean that the World class relies on a Mediator to function properly, the IWorld interface provides all the needed functionality to

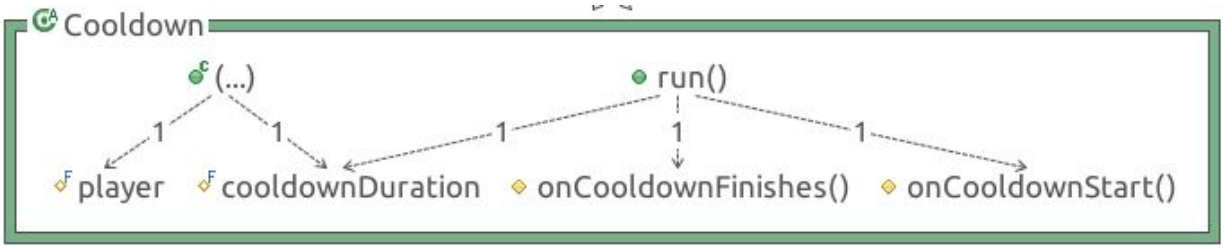
use World in a context of a text-based game instead. The class is responsible for:

- Registering new players.
- Retrieving a player object via their Id.
- Performing an attack between two players of the same world.
- Updating in-world objects a player can see given their new position.
- Updating an avatar for a player.
- Prompting a player to consume a certain loot-box.
- Spawning loot-boxes in the world.
- Prompting a player to change weapon.
- Provide a list of all items in the Shop.

Player-objects in the World class are mapped to their unique Id, which facilitates quick data fetching, instead of looking players up in a linear list.

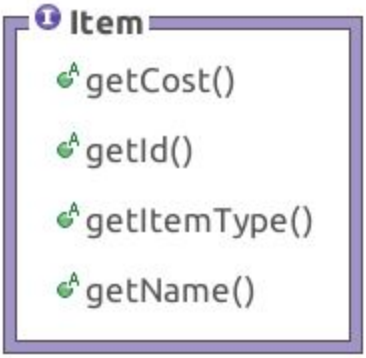
World-class performs actions on players even if it's not directly asked to, such actions include informing nearby players by that a player is revived, and deducing experience points from a player who attacks an unarmed player.

- Player class: It implements the interface IPlayer, providing all the functionality needed to query and change the state of a player. A player object can exist independently from a World-object. It also allows for and handles all edge-case scenarios such as buying, selling items and switching weapons.
    - Cooldown-class: Implementing Runnable, this class is a parent class to the two different in function but similar in structure cooldowns used by a Player-object, namely Radar-cooldown and Respawn-cooldown. It first executes an action then waits for a certain period of time before performing a second action. It uses Template Method Pattern to delegate the implementation of its pre- and post-methods to its subclasses.

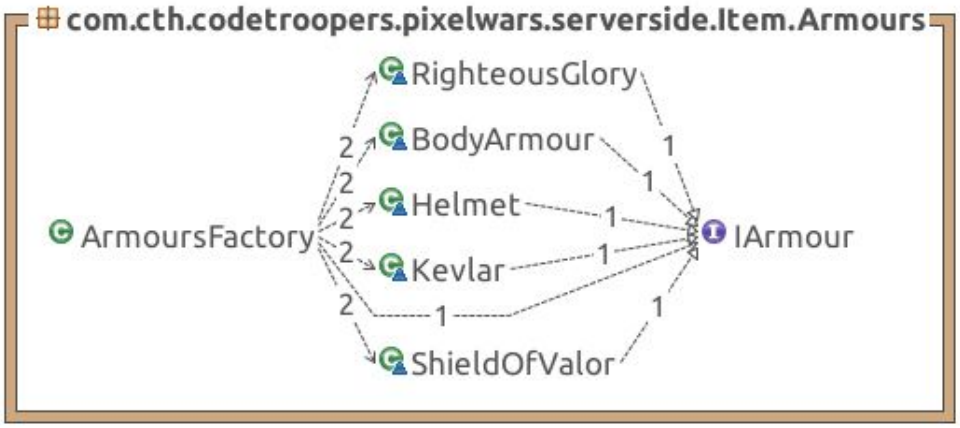


- Any gain or loss in experience points is calculated by the subpackage Experience and its abstract class Exp to keep Player-class as neat as possible.
- No constants concerning the Player-class are hardcoded into the Player-class but are instead stored in a subclass under the name PlayerConstants containing several initial values.

- Item-Package: It contains the interface Item abstracting all possible Items that can be created. All ownable items must have some sort of an implementation of the Item-Interface

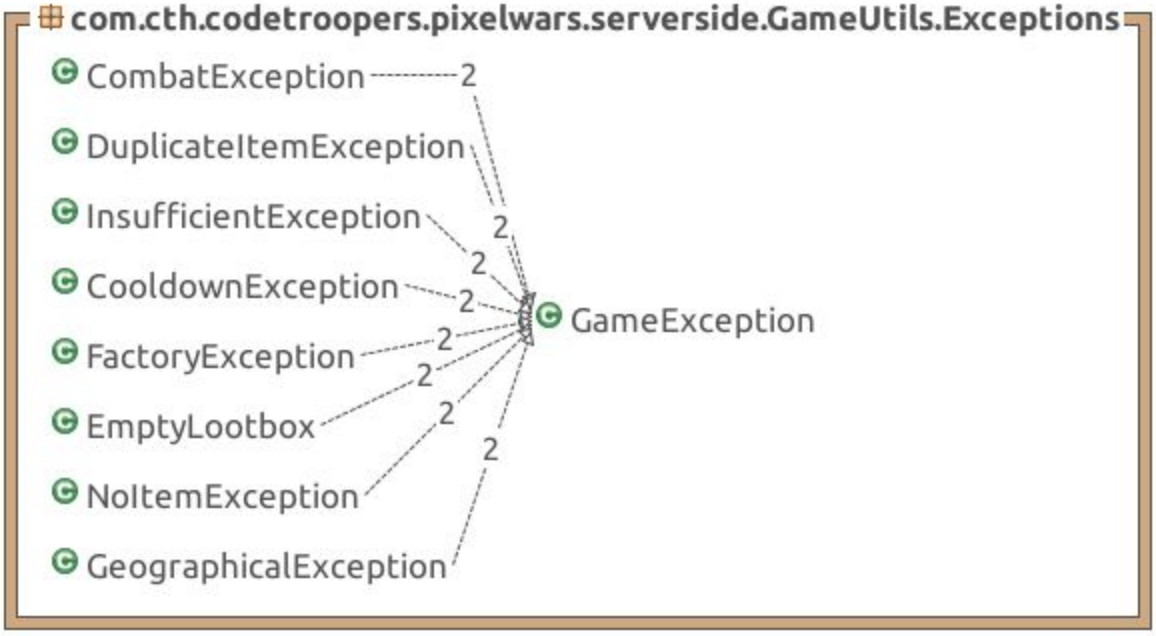


The main two types of Items in the game are weapons and armours, each of which is contained in its subpackage. All subclasses cannot be directly accessed and instantiated, for that a class styled according to the Factory pattern is used. Passing the item's Id to the factory which returns the desired item, this makes for more secure and organized instantiation.



Both packages has the same basic structure, and new weapons/armours can be added with ease, by just extending the base class and overriding its method.

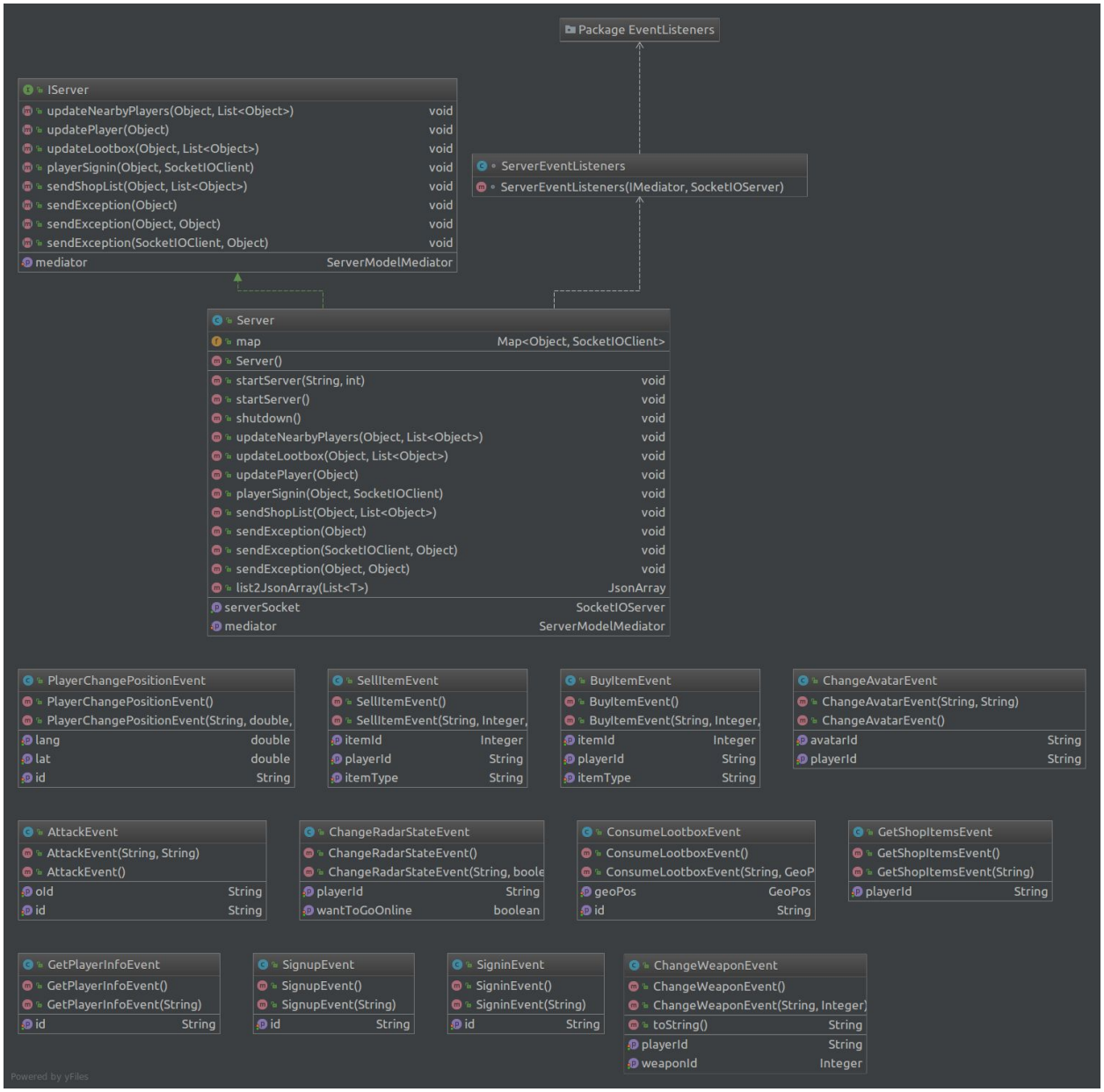
- GameUtils-package: This package contains multipurpose classes that are used in different places throughout the lifecycle of the Server-side. The class GeoPos is a simple data class containing the location of players, loot-boxes and any other in-world entities. This class is used in the GeoDistance helper class that calculates the distance between two GeoDistance-objects in meters. One other key component in Exceptions-package is the parent class GameException that extends Throwable, and is inherited by different sub-exceptions.



All of the classes mentioned above act as the Model part of our distributed MVC-design.

Communication Layer

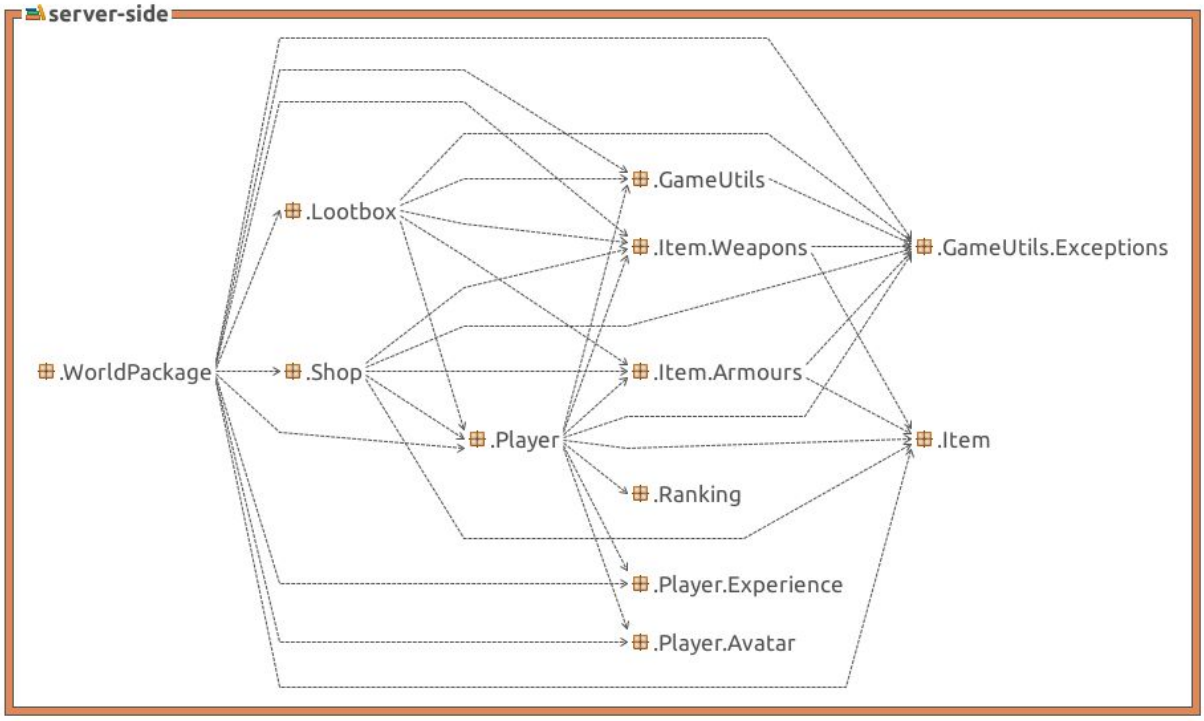
One other necessary component is the communication layer that oversees the updating of data located on remote clients, and responds to requests from clients querying the Model or asking to perform a certain action.
This layer cannot be separated from technical implementations, however it is completely separated from all Model-classes.



The ServerController-package contains a Server class that implements the interface IServer, which can perform the following actions:

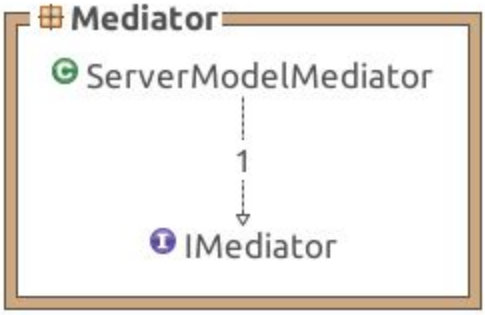
- Update a client with information about nearby clients.
- Update the data about the player status for a remote client.
- Send updates about loot-boxes.
- Registering a connecting client.
- Inform client(s) of a game-logic related exception.

The ServerController uses WebSockets to communicate with remote clients. When started on a host device, it listens to a certain port, where the clients are expected to send their requests.
It passes over objects as JSON objects which makes sent data readable on both ends.
ServerController class is tasked with listening to inquiries from clients to perform on the model or inform it.
Incoming requests are formed as JSON objects, each has its own structure. The Framework SocketIO converts all incoming events into a POJO, such a POJO should be defined so that the framework knows at which events it should convert the incoming JSON object to the suitable POJO. The skeleton for the requests are placed in subpackage EventObjects. Following the framework guidelines those objects should be typical data classes, meaning that they should only be used to store and retrieve data.
For every event object, there exists a corresponding Event Listener that is called with the event object passed as an argument.
The class ServerEventListeners maps all the event listeners and their associated event objects to a string tag. When an event is received, the event listener that corresponds to its tag is prompted to handle the event.
More details on how the events are sent back and forth comes in later section.
The diagram for the Server-side’s packages looks like this:



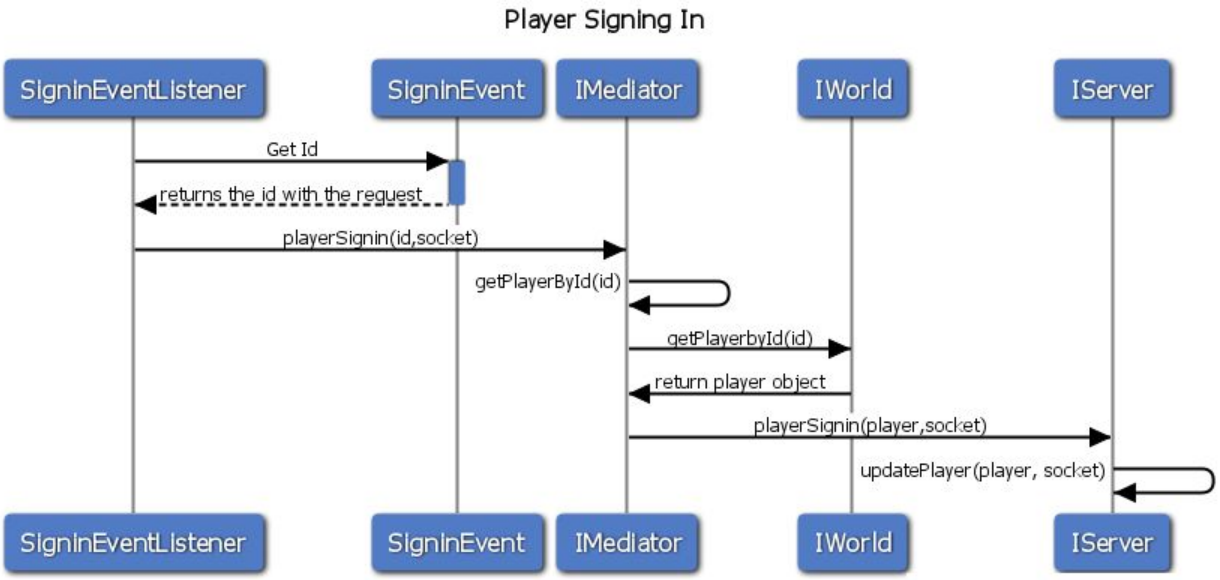
A well functioning model and a structured, neutral to design-details communication layer is all good. We need a third component to organize communication between the two. This third component can be easily replaced, since all dependencies are on interfaces and not concrete implementations.

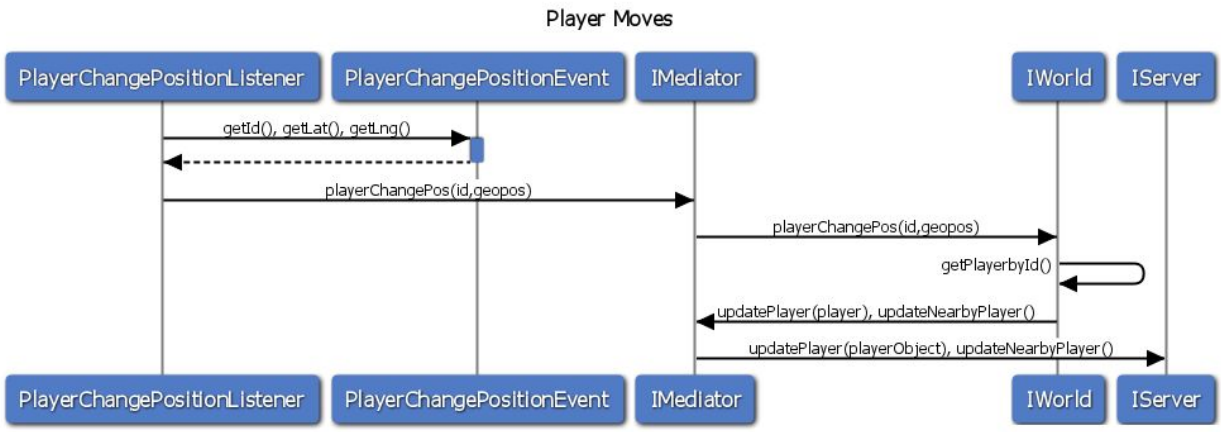
The IMediator interface offers useful functions needed by both the ServerController part and the Model-classes to ensure full collaboration on operating the game. It relies on two dependencies one of the IWorld type to delegate all Model-altering procedures to, and the other is that of the IServer type, which is used to send updates to remote client(s). Dependency Injection is used to connect the mediator with the other two components. This ensure easier debugging and more comprehensive ability to test.



The IMedaitor interface might cause circular dependency between the ServerController and the Model parts, but it is contained in its own package, and is easily replaceable extensible and self-contained.

Example Sequence Diagram:





Client-side

The Client-side android application is a separate project which uses WebSocket and HTTP-requests to communicate with the Server-side. The Client-side implements the View part of the MVC of the project as a whole, it also has a Controller that handles sending requests to Server as well as receiving incoming updates.

Within the Client-side yet another but local MVC-model is used to implement the actual inner structure of the android application.

Implementing MVC for Android.

In a typical, MVC-model, the View and Controller parts are as separated as possible. Android Studio doesn't out of the box support this mentality since it combines the View and Controller into single Activity-entity. This design is to be avoided when implementing a modular program.

Separating View and Controller

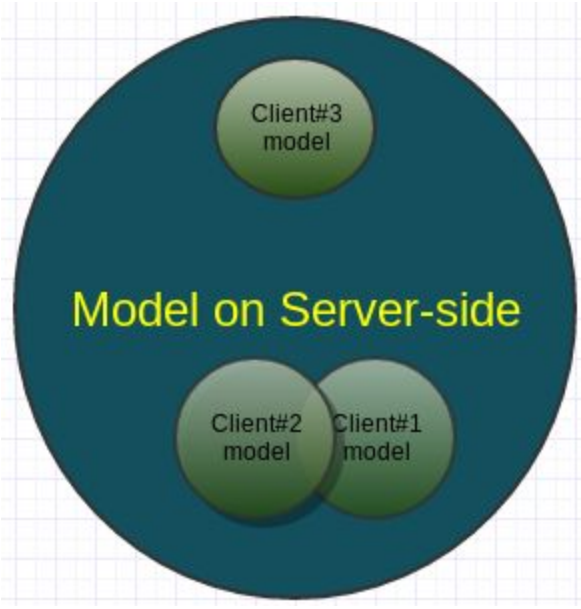
The used approach is having all views implement a parent interface that has the basic functionality of populating an xml-layout.
All inheriting views are customized to reflect their function. A view interface should contain two main things: a method to be invoked when data change,

and a view-specific listener interface, so that its corresponding controller will be notified when a user clicks on a control element e.g. a button in the view.
In Controllers-package, control-classes are created that implement their view's interface listener, and on those events they prompt the local client model to do an action.
Controllers only contain the views, and are injected into them as listeners to be called upon when a visual component is interacted with.

```
logInView.setListener(this);
```

Control classes do not fetch any data back from the Model, nor prompt their views to. Interaction between model and controllers is kept to the minimal of asking the model to do some action.

Client-side's Model
The term Model does not truly reflect the function of the local model, because it's just a reflection of a specific part of the remote model on the Server, it does not have any game logic in it.
However, it does inform the views when new information is fetched, and can take requests from the controllers, so all in all it is a rather limited representation and aggregation of data.

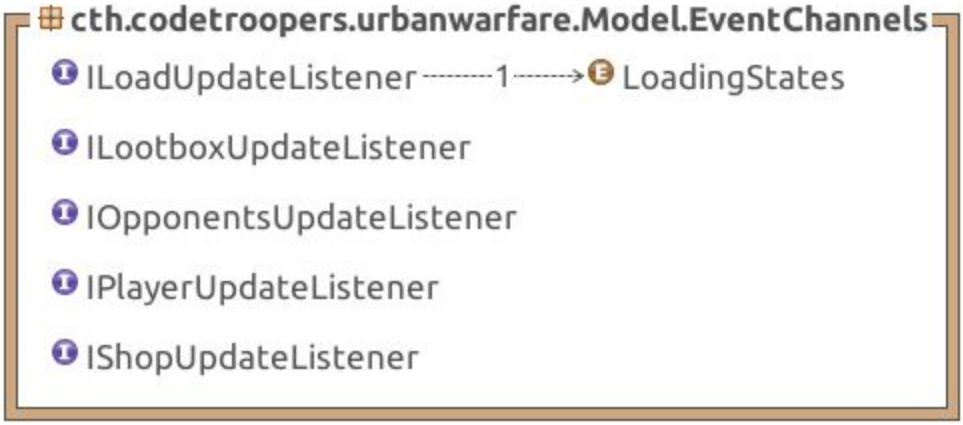


When two players are within each other's vision range, in other words they can see and interact with one another, their models overlap somewhat, whereas when they are completely separate, the models are separate.

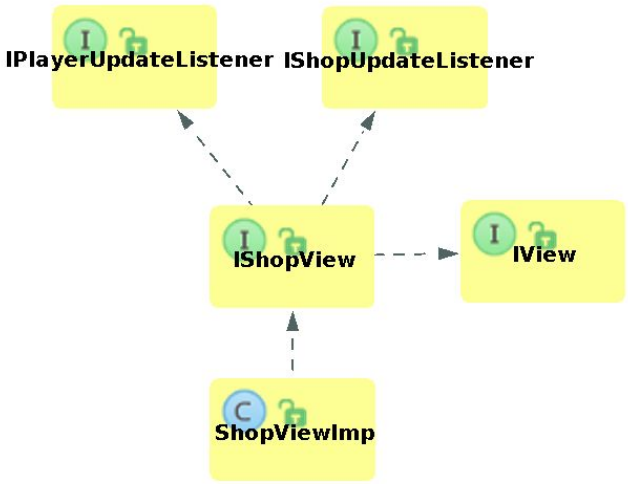
Channels Pattern

On the Client-side we implement an MVC-design which inclines the model to inform its view, when a change occurs. Given that the model does contain several types of data, and different views anticipate updates about different kind of information.
The best way to loosen the couplings is to use an observator pattern, where the view observe the model, and once a certain part of the model changes, they get updated right away.

As discussed above, the observator pattern guarantees observing one type of information, but in our case more than just one is required. To overcome that, the model provides a certain type of channels to which the views can subscribe, one view might subscribe to one or more channels, as it sees fit.



Whenever the model gets an update relating to any of these channels, all subscribing views are informed.It is up to the views to subscribe to the channels that concern them.



The IShopView inteface, subscribes to two channels on the Client model, as well as the superparent IView interface. The current implementation of IShopView, namely the class ShopViewImp simply just implements the readily existing IShopView interface.
By having established strong groundings and good programmatically infrastructure, it becomes easier to extend and improve on the controllers,

views and models, separately, knowing that we achieved a high level of modularity.

Connectivity Layer on Client-side

This subsystem serves the purpose of interacting with the remote server on behalf of the Android-client, as a result it contains all the technical details needed to formulate a proper JSON-request and send it properly to the remote server. On the other side, the request is handled, and sometimes, it sends back information. When such information is received, the Connectivity Layer informs the Client model of the incoming data. It also converts the incoming objects into POJO's so that the JSON representation is completely taken care of by the Connectivity Layer.

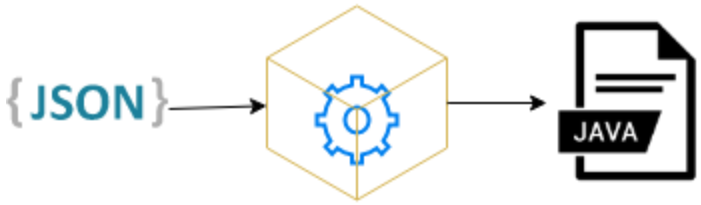
The Client-Model requires an IConnectivityLayer-implementing object, to flush data into as well as to listen to any events started by it. The Connectivity Layer does not depend on the Model, the model needs to implement an interface (ConnectivityListener) so that the Connectivity Layer can work with.

Skeletons and Skeleton Factory

Since the local representation of the data is just a static one, with no real functionalities. Skeleton classes have the sole purpose of storing data and making it available by getters, so that the views can interpret them separately. Having a JSON object passed around is not a viable option considering it would make the views dependent on a technical

representation, and if a decision is taken to move to XML, the views would need to be readjusted.

A Skeleton class resolves this dependency by taking in a JSON object and converting it into a POJO.



The Skeleton Factory is just an typical factory used to to aggregate the skeletons so that for example the Connectivity Layer can prompt it to create the corresponding objects before being passed over to the model.

Persistent Data Management

The Server-side does not store data upon shutdown, when starting up an instance of the server, it does not retrieve any previous information. As for the Client-side, any changes (change in experience, health points, weapons or gold) done to the player during the life-cycle of the server

instance, can be re-accessed since the data is always retrieved from server and never locally changed.

Server-side never send any data other than JSON-object, and as such all artwork are up to the Client, and they're managed by the Android Studio resource management system.

The final version of the system runs on localhost for both the Server-side as well as the Client. The client tries to connect to the address 10.0.2.2, this address is used by the emulator to connect to localhost on the hosting machine. When connecting to a remote server online, this URI address needs to be changed.

Access Control and Security

The person running the server-side can predefine certain constants to affect other player's playing experience, this functionality can be extended by building a UI for the server-side, which would be relatively easy considering the modularity of the World-package.

The current client and any future clients, cannot misuse the model on the server since they're limited to a set of requests.