CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY

ASSIGNMENT OF BACHELOR'S THESIS

Title:	RPG game with augmented reality features - client part
Student:	Tomáš Zahálka
Supervisor:	Ing. Miroslav Balík, Ph.D.
Study Programme:	Informatics
Study Branch:	Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of winter semester 2018/19

Instructions

The aim of the thesis is to specify, design, and implement a functional prototype of the client part of a RPG game with features of augmented reality (AR).
1. Create a story line and rules for the game. Consider intensive usage of geolocation and AR features.
2. Review existing solutions. 3. Formalize the following requirements for the implementation of the client part:
- use of location sensors to determine the player's real world location,
- use microtransactions for purchasing items in the game,

4. Design the client part of the game, consider the following requirements.
5. Implement the functional prototype in the Unity engine, use OS Android as the target platform.
6. Document the prototype and perform suitable testing.
7. Tightly cooperate with Jakub Čech who works on the server part.

References

Will be provided by the supervisor.

Ing. Michal Valenta, Ph.D.
Head of Department

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague March 6, 2017

Czech Technical University in Prague
Faculty of Information Technology
Department of Software Engineering

Bachelor's thesis

RPG game with augmented reality features - client part

Tomáš Zahálka

Supervisor: Ing. Miroslav Balík, Ph.D.

May 15, 2017

Acknowledgements

I would like to thank my supervisor Ing. Miroslav Balík, Ph.D. for a great help with writing this thesis. Thanks go to my family and friends for support and to my colleague Jakub Čech working on the server part for an excellent cooperation.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on May 15, 2017 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Zahálka, Tomáš. *RPG game with augmented reality features - client part.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2017.

Abstrakt

Tato bakalářská práce se zabývá návrhem a vytvořením prototypu mobilní hry na hrdiny s prvky rozšířené reality. Hra se skládá ze dvou částí, klientské a serverové. Tato práce se zabývá klientskou částí. Mezi prvky rozšířené reality se řadí vykreslování herní mapy s generovanými herními objekty na ní zobrazovanými, se kterými lze interagovat. Přináší inovativní řešení pro zobrazování mapy herního světa, která je dlaždicová a používá skutečná vektorová data. Práce popisuje a implementuje některé běžné prvky her na hrdiny, jako je sbírání předmětů z herních objektů. Pro vývoj je použit Unity engine a operační systém Android je cílovou platformou. Prototyp představuje dobrý základ pro vývoj plnohodnotné hry.

Klíčová slova funkční prototyp, rozšířená realita, hra na hrdiny, mapa světa, vektorová data, využívající lokaci, mikrotransakce, nákupy v aplikaci, Unity, Android

Abstract

This Bachelor's thesis deals with designing and creating a prototype of a mobile role-playing game with augmented reality features. The game consists of two parts, the client part and the server part. This thesis deals with the client

part. The augmented reality features include displaying the game world map with generated game objects on top of it that can be interacted with. It brings an innovative solution for displaying the game world map, which is tile-based and uses real vector data. The thesis describes and implements some common role-playing game features, like collecting items from the game objects. The Unity engine is used for development, and the Android operating system is the target platform. The prototype presents a solid base for developing a fully-fledged game.

Keywords functional prototype, augmented reality, role-playing game, world map, vector data, location-based, microtransactions, in-app purchases, Unity, Android

Contents

List of Figures

List of Tables

Introduction

Location sensors in today's mobile devices present a very powerful feature. They enable creating a whole new gaming experience like never before. Nowadays virtually every smartphone has the means to determine the user's location.

This presents a potentially large user base, but surprisingly there is not a lot of games around taking advantage of this technology. There have been some reasonably successful attempts to fill this market gap. But while they attracted a large number of users in the beginning, some of them struggled to maintain these users in a longer term. I am going to look at some of these existing solutions.

The game is divided into two parts, a client part and a server part. In cooperation with my colleague Jakub Čech (who works on the server part), I am going to design a functional prototype of the client part of a role-playing game with features of augmented reality (AR). As part of these augmented reality features, I am going to create an innovative solution for displaying the world map. I am going to implement the prototype, perform suitable testing and release it.

Chapter 1

Goal of the thesis

The goal of the research part of the thesis is to analyze similar games on the market, to further understand the Unity engine and to research different map providers.

The goal of the practical part is to specify the features of the prototype, to look into options for displaying the world map, to gain knowledge about supporting microtransactions in a Unity-based game and to design the structure and components of the prototype. In the implementation phase, the goal is to add functionality for displaying the world map, obtaining the device's location, supporting microtransactions, to implement communication with the server, user authentication, parsing of the responses, displaying the game objects on the map and to add the required functionality for the game objects. Also, to create the user interface, test it and release it for the Android operating system.

Chapter 2

Game description

This chapter explains the game in more detail and describes some of the main features. Since this is a prototype of a mobile role-playing game, the player assumes the role of a character moving around a world with fictional elements. The augmented reality part of it lays in displaying a 2D map of the game world, but a one that is based on a real-world map. There are generated game objects being displayed on top of the map. The player is shown on the map along with the game objects, since the game obtains a location of the mobile device, and can interact with those game objects around him. The two types of objects in this prototype are a *monster* and a *shop*. There are multiple features available to the player, like buying items from a *shop* and equipping these items later. These features are detailed in the 4.1 (Requirements) and 5.1 (Actions) sections. The player has some basic attributes such as *gold* used for buying items. To make the game easier, the player can decide to buy products for real money, for example *gold* utilizing microtransactions (in-app purchases).

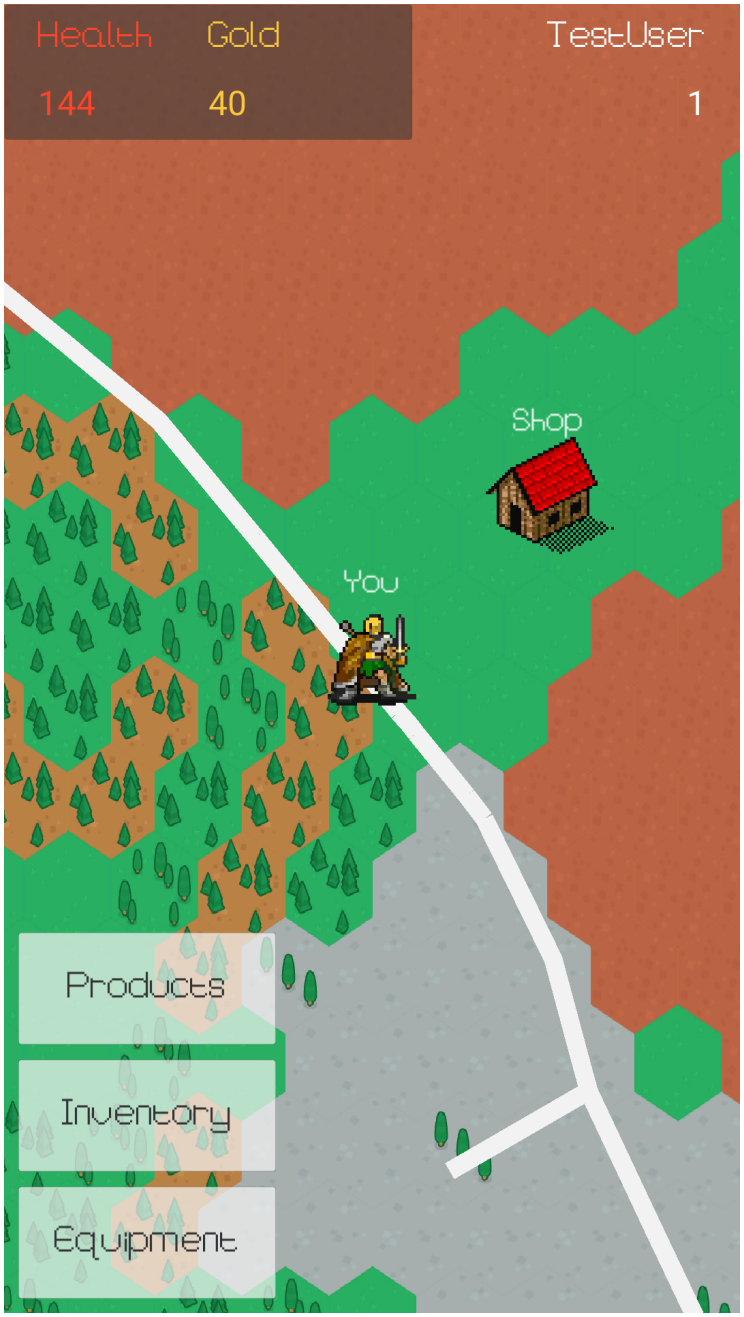

Figure 2.1: A screenshot of the game

Chapter 3

Existing games

3.1 Parallel Kingdom

This game is probably the most similar to our prototype. It was released in October 2008 and closed on November 1, 2016. Parallel Kingdom was quite successful in the beginning, reaching one million players in 2012. It also had its Facebook version [2].

"Parallel Kingdom is a mobile location based massively multiplayer game that uses your GPS location to place you in a virtual world on top of the real world."[3]

Based on their real-life location, players could claim territory anywhere on the map. They could discover different kinds of monsters, gather resources and build a city. They could choose from a variety of skills, weapons and armor. It included several types of resources used for building and trading. Since it was a massively-multiplayer game, players could interact with each other [4].

3.2 Ingress

The game was released in December 2013 for Android at first, the version for iOS came in July 2014. At the start of the game, the player chooses one of two factions. These factions compete against each other. The game presents the players with a real-world map. The map contains portals and their locations are based on real-life art places, such as statues and historic buildings.

"Move through the real world using your Android device and the Ingress app to discover and tap sources of this mysterious energy. Acquire objects to aid in your quest, deploy tech to capture territory, and ally with other players to advance the cause of the Enlightened or the Resistance."[5]

Figure 3.1: Parallel Kingdom - screenshot [1]

3.3 Pokémon GO

Just like Ingress, this game was developed by Niantic for Android and iOS. It was released in July 2016. Like the other mentioned games, it displays a map with the surrounding area. By using the location sensors, players move around the map, capture creatures, train them and battle with others. The map also features *PokéStops*, which provide players with different items, and gyms, which are battle locations. The game faced many technical issues at the time of the launch. Despite that, it has seen a huge success, and has been downloaded more than 500 million times [6].

3.4 Personal projects

I have attempted to create my own role-playing games utilizing location sensors in the past as well. The first idea was to simply use a mobile phone to walk around and interact with game objects on a real map.

For the first project, I decided to create a server for generating the game objects. I divided the map of the Czech Republic into rectangles of a small size (a few kilometers). Once a user wanted to obtain game objects from the server in a certain area, the server chose the corresponding rectangle based on the player's location and returned the containing objects. The game supported basic RPG features, such as killing monsters, purchasing items in shops, opening chests, repairing weapons and improving skills.

For the second project, I tried to create a completely offline game that would not require the players to have a data plan. The game objects generation worked in a similar way to the first project - by having rectangles and generating game objects for each of them separately. It also had similar features to the ones found in the first game, but added a few more, such as dialogues with people. A big improvement was the addition of dynamic game objects where people would move on the map and monsters would attack the players once they got too close. This game never got out of an alpha version. I created a *Google+ Community* page where I gained around 30 active users with some of them leaving feedback which led to adding more features [7].

Both games were built using the **Android SDK** [8]. For displaying the map, I used **Google Maps** available for Android with the **Google Maps Android API v2** [9].

Chapter 4

Analysis

The analysis part of the thesis lists the requirements of the game first and describes the use cases. Then it considers possible solutions for displaying the world map and looks into different map providers. It also chooses a technology for integrating in-app purchases in the game. The last part deals with proposing a look and a structure of the user interface.

4.1 Requirements

For this prototype, I agreed on several basic requirements in discussions with my colleague. These requirements are shown in figure 4.1.

4.1.1 Functional requirements

These requirements define the functionality that is provided to the user.

4.1.1.1 User registration

The game allows a user to register for the game with a chosen username.

4.1.1.2 Player authentication

The registered user (referred to as a player) must sign into the game.

4.1.1.3 Player's attributes

Each player has a set of basic attributes that are presented to the them along with the username. These are:

- health,
- experience,

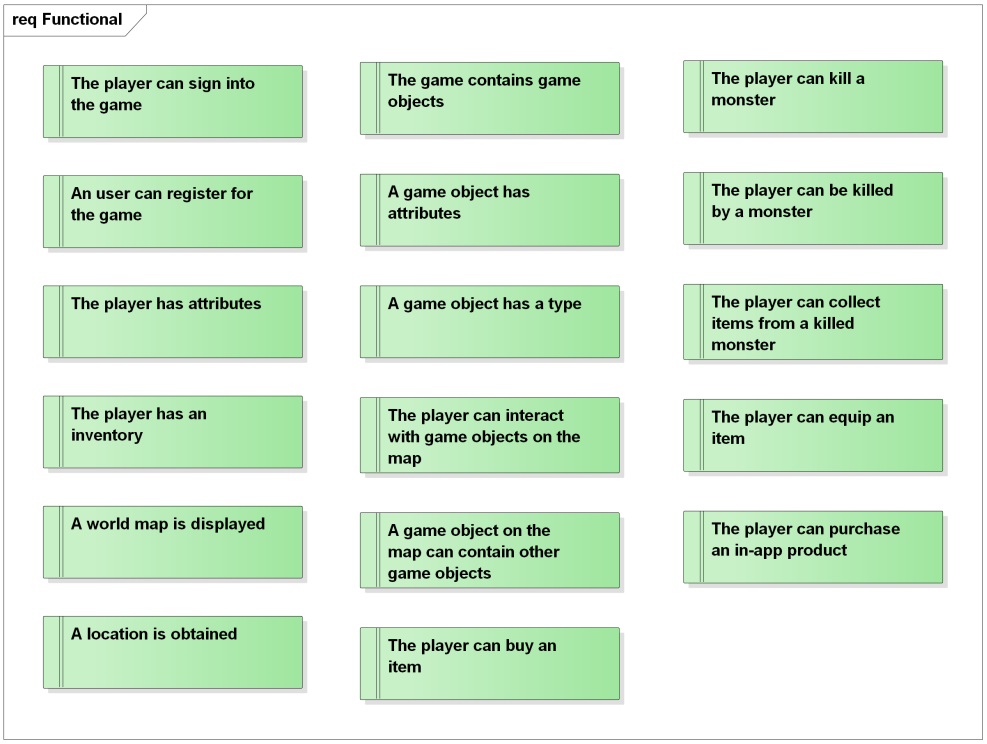

Figure 4.1: Functional requirements

- level,
- gold,
- gems.

4.1.1.4 Player's inventory

Each player has an inventory that can contain any game object.

4.1.1.5 World map

The game displays a world map of some form based on the real world.

4.1.1.6 Location

The device's location is obtained periodically.

4.1.1.7 Player on the map

The player is shown on top of the world map based on the obtained location.

4.1.1.8 Game objects

There are game objects in the game. They are generated on the server and requested by the client based on the player's location. Each game object can be displayed on top of the world map.

4.1.1.9 Game object attributes

A game object has several attributes, a type and optionally a name and a description. Each of those 2 optional attributes are used over the name and description attributes in the game object type, if set.

4.1.1.10 Game object type

A game object type contains attributes common to game objects of the same type. Each game object type can have an arbitrary number of attributes. It contains an icon for displaying the game object. It also has a name and a description in a case the game object does not specify them. There are five different types of game objects in the prototype:

- Skeleton,
- Goblin,
- Health Potion,
- Shop,
- Sword.

4.1.1.11 Interaction with game objects

The player can interact with game objects on the map, but only to a certain distance defined for each game object type. Each type of game object can have a different behavior and can display its details.

4.1.1.12 Game object's inventory

Each top-level game object (a one that is shown on the map) can contain other game objects.

4.1.1.13 Buying an item

The game allows the player to buy items in a *shop*. By doing so, the player loses some *gold* based on the item's price (a game object's attribute). The item is added to the player's inventory.

4.1.1.14 Killing a monster

The player can kill a *monster*. This action lowers the player's *health* based on the received damage (a *monster's* attribute). It adds *experience* and *gold* for the player based on the *monster's* attributes.

4.1.1.15 Killing the player

The player can be killed by a *monster* (die). This affects the player's attributes.

4.1.1.16 Collecting items

Items from a killed *monster* can be collected by the player. The *monster* is then removed from the game, and the items are added to the player's inventory.

4.1.1.17 Equipping an item

The player can equip items to different slots. Any game object can be equipped to a certain slot if a specific attribute is defined. The prototype has one available slot, the right hand.

4.1.1.18 Purchasing an in-app product

The player can purchase a product for real money that is consumed in the game in some form. In the case of this prototype, the player can purchase *gold*, which is added to the player's attributes.

4.1.2 Non-functional requirements

These requirements specify the criteria the prototype must meet. They are shown in figure 4.2.

4.1.2.1 Android support

The prototype runs on the **Android** operating system version 4.0 (API level 14) or higher. This means the game targets almost all Android users.

The **Google Play Games plugin for Unity** used in this project also requires an Android version of 4.0 or higher [10].

4.1.2.2 Display orientation

The game supports a portrait screen orientation.

4.1.2.3 Google Play Games authentication

The player is authenticated using **Google Play Games**.

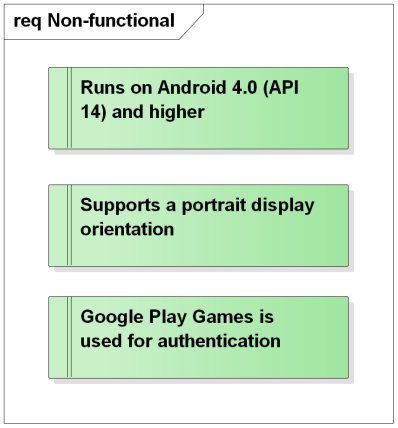

Figure 4.2: Non-functional requirements

4.2 Use cases

Use cases show specific applications of the requirements by the users.

4.2.1 Users

I define three types of users of the game. An unregistered user, an authenticated player and an unauthenticated player. The word player is only used for users that are already registered in the game. All user types are shown in figure 4.3.

4.2.2 Authentication

Figure 4.4 shows the actions used in the authentication process. An unregistered user can choose to register for the game. He is presented with a prompt to choose a username. After a successful registration, the unregistered user becomes a player, the unauthenticated player is then automatically signed in and becomes an authenticated player.

4.2.3 Game object actions

The actions involving game objects on the map are shown in figure 4.5. All actions involve opening a specific window for a game object.

4.2.4 Player actions

Figure 4.6 shows the actions the player can choose to do without interacting with the game objects on the map. Equipping an item involves showing the equipment slots, choosing a slot and opening the inventory to choose an item to assign to that slot.

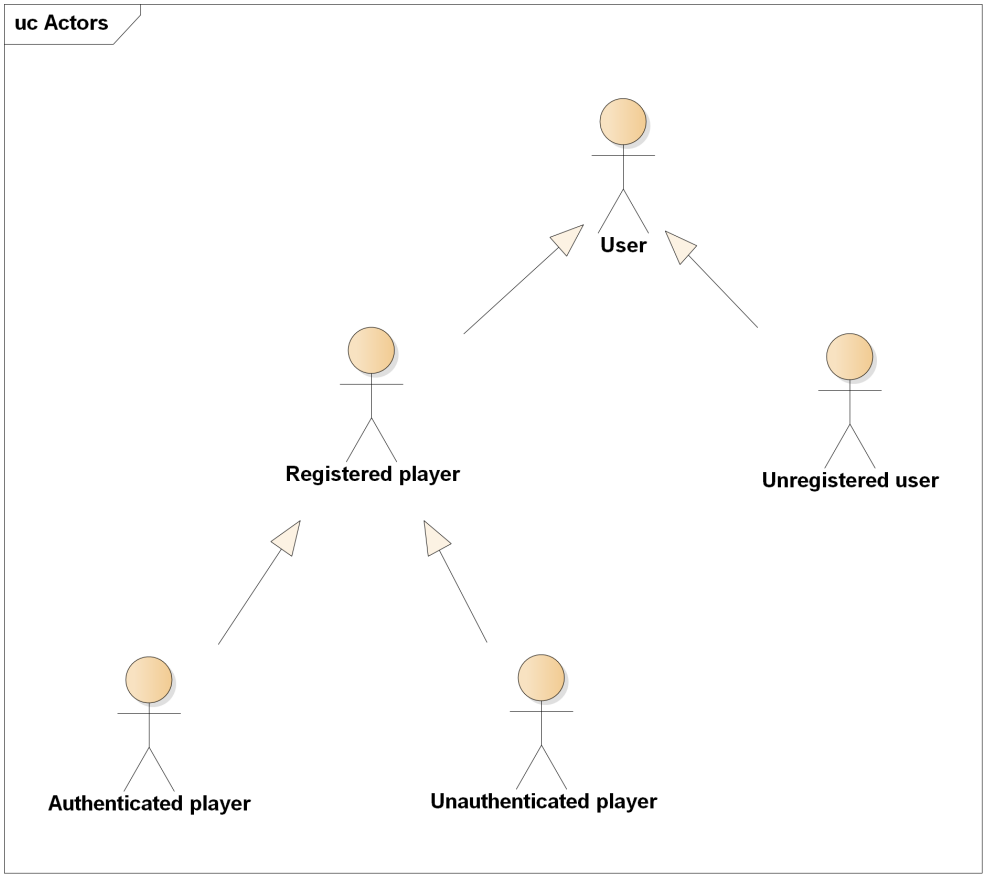

Figure 4.3: Users

4.3 World map

This part analyzes different map providers and chooses a solution for displaying the world map in the game.

4.3.1 Map providers

There are many map providers on the market. The biggest and most popular being the **Google Maps** [11]. Another provider is **Mapbox** [12] whose services are used by big companies. The **OpenStreetMap** [13] project provides free geographic data.

4.3.1.1 Google Maps

One of the APIs provided by Google Maps is the **Static Maps API**, which returns portions of the world map as raster images. There are several URL parameters that can be specified in an HTTPS request to the API. The center pa-

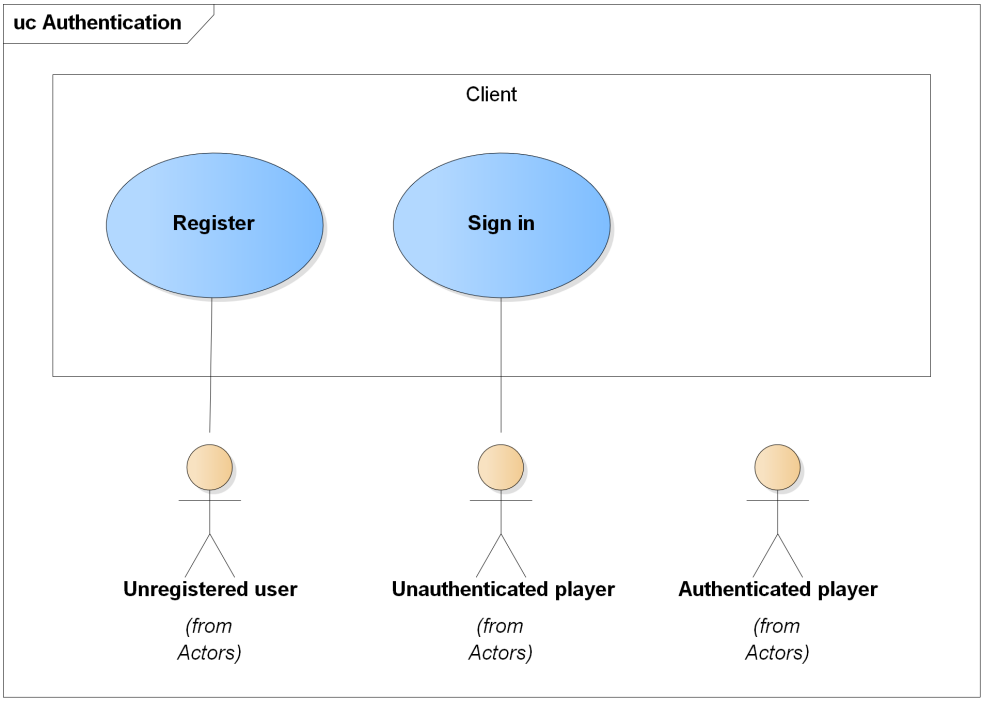

Figure 4.4: Authentication use case

rameter determines the location (the center of the image) and the zoom specifies the magnification level of the map. Then there is the size of the rectangle of the map image. It is specified in pixels in the format {width}x{height}. The maximum values for the standard version of the API are 640x640. As with most APIs, Google requires the developer to obtain a key to be able to use the Static Maps API. There are several usage limits as well [14].

4.3.1.2 Mapbox

This is one of the newer map providers (founded in 2010) but its services are already being used by companies like Foursquare [15] and Uber Technologies [16]. Mapbox offers many interesting APIs as well as a product called **Mapbox Studio**, which enables the users to design a custom map. The company offers three different pricing plans, the one called *Starter* is free with 50,000 map views (or mobile users) per month [17]. At the time of writing this thesis, Mapbox had a beta version of the **Unity SDK** available for download, which aims to make it easier for Unity developers to access their product [18].

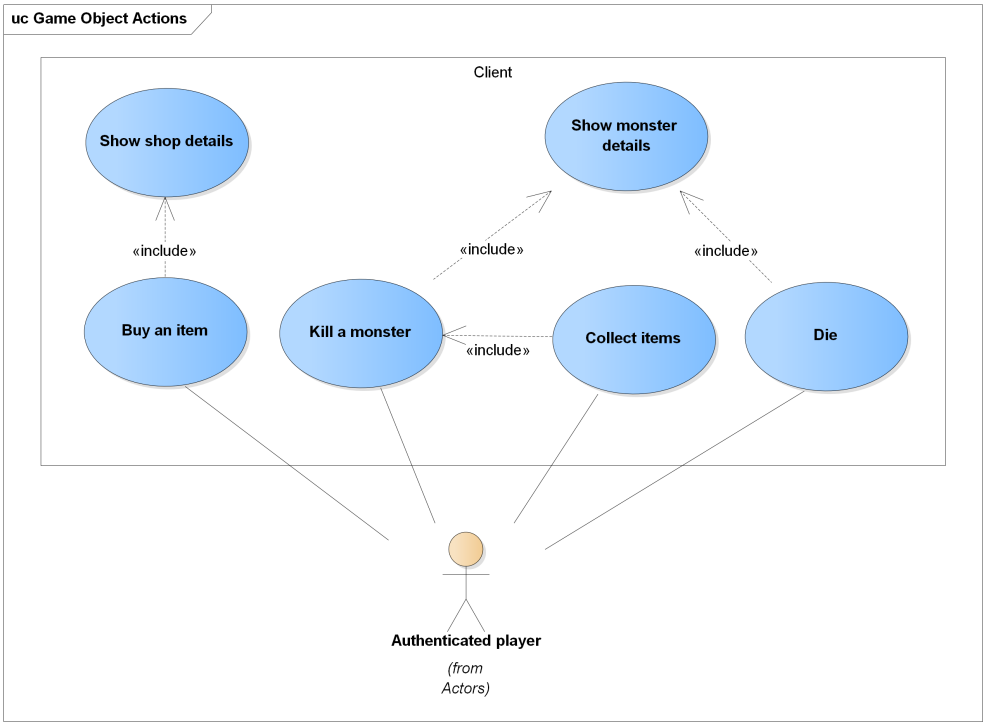

Figure 4.5: Game object actions

4.3.1.3 OpenStreetMap

As part of the OpenStreetMap project, there are different providers of raw geographic data. This also means that there is no guarantee as for the availability of the data and the quality can vary at different locations. The main advantage, on the other hand, is that they are free to use. Many servers can be found that also provide vector geographic data and there is an option of hosting a custom map server. *"Welcome to OpenStreetMap, the project that creates and distributes free geographic data for the world. We started it because most maps you think of as free actually have legal or technical restrictions on their use, holding back people from using them in creative, productive, or unexpected ways."*[13]

4.3.2 Mapzen

The **Mapzen** mapping platform offers several APIs for requesting map data including vector data. *"Mapzen Vector Tiles are powered by several major open data sets and we owe a tremendous debt of gratitude to the individuals and communities which produced them."*[19] Mapzen uses data mostly from OpenStreetMap, but also from a few other data sources. It provides several layers of worldwide vector map data, these are:

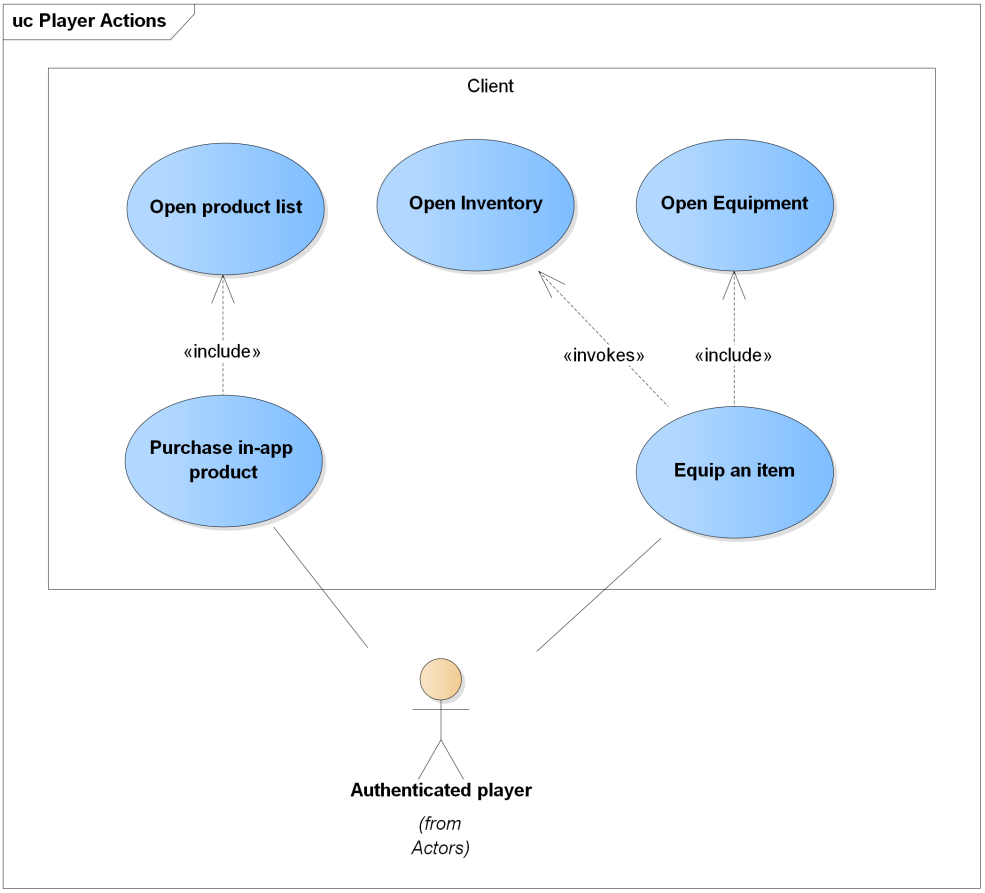

Figure 4.6: Player actions

- boundaries,
- buildings,
- earth,
- landuse,
- places,
- pois (Points of Interest),
- roads,
- transit,
- water.

Provider	Free requests	Vector data	Data quality
Google Maps	25,000 per 24 hours	No	Great
Mapbox	50,000 per month	Yes	Great
OpenStreetMap	Differs	Yes	Differs
Mapzen	50,000 per month	Yes	Great

Table 4.1: Summary of map providers

4.3.3 Chosen map solution

One of the first ideas was to use Google Maps for displaying the world map. I have already used it in my previous projects that I have worked on. But there is no official plugin for Unity for displaying Google Maps such as for example the Google Maps Android API available for Android. I wanted to make the game easily deployable on different platforms in the future. For this reason, creating a Unity plugin for supporting the Maps Android API did not seem like a good idea. I created a test project for displaying the map using the Google Static Maps API. As mentioned previously, this API provides static raster images of map data. The images had to be displayed next to each other. Google also puts a logo to each tile image that cannot be removed. That of course did not provide a pleasant user experience. After some research, I found out that I can easily obtain vector data, so I decided to draw my own map. Both Mapbox and Mapzen present a good option for achieving this goal. I decided to choose the **Mapzen** platform, but the design of the game allows me to switch to a different provider if needed. The prototype uses four of the Mapzen's landuse kinds with a set of different hexagonal tiles for each of them:

- farmland,
- forest,
- meadow,
- residential.

Table 4.1 summarizes the analyzed map providers.

4.3.4 Vector map data

4.3.4.1 Landuse

"Landuse polygons from OpenStreetMap represent parks, forests, residential, commercial, industrial, university, sports and other areas."[20] These polygons serve as a base layer for displaying the map. Each landuse contains a *kind* attribute, which defines the type of the landuse. This is important to determine the look of the tiles.

4.3.4.2 Roads

"More than just roads, this OpenStreetMap and Natural Earth based transportation layer includes highways, major roads, minor roads, paths, railways, ferries, and ski pistes."[21]

4.4 Microtransactions

There is an official support for in-app purchases in Unity. As part of their **Unity Services**, they offer the **IAP** system. *"Unity IAP makes it easy to implement in-app purchases in your application across the most popular App stores."*[22] Therefore this was the obvious choice for implementing microtransactions in the game.

4.5 User interface

The user interface chosen for this prototype had to be simple and clear. The game has two screens, the authentication screen has a sign-in button, an optional input field for entering a username during registration and some text labels with help. The actual game screen displays the world map with an overlay on top of it. There are some basic player attributes and the username displayed as part of the overlay. The buttons for accessing the in-app products, the equipment and the player's inventory are in the bottom-left part of the screen. There are several types of windows that can be displayed on top of both screens. They all follow the same design principle; the windows do not fill up the whole screen and they dim the background when displayed. Most of them have a top bar with a close button, a header with an icon and some text, the main content and buttons at the bottom. Figure 4.7 shows an example of the *monster's* inventory window.

Figure 4.7: The *monster's* inventory window

Chapter 5

Design

This chapter takes the results from the analysis part, describes the required actions and game object types in detail, specifies the API and designs the structure and components of the game prototype.

5.1 Actions

The client is required to confirm each taken action with the server. This means that before applying the action results, the client needs to wait for a successful server response. For that purpose, a window informing the user of an ongoing action is shown each time. If the response from the server is not successful, the action results are not applied. Following is the list of actions that are being sent to the server along with their outcomes.

5.1.1 Authentication

Figure 5.1 shows an activity diagram for the authentication process.

5.1.1.1 Access Code

Authorizing the player's actions on the server is done by receiving an *Access Code* from the server during authentication and sending it with every request since then.

5.1.1.2 User registration

The authentication process starts immediately after opening the game. The player needs to sign into his Google Play account first. After signing into Google Play Games, the *Google ID token* is retrieved and sent to the server. If there is no player found corresponding to the *ID token*, it returns an answer to the client saying that a username must be chosen. The user then chooses

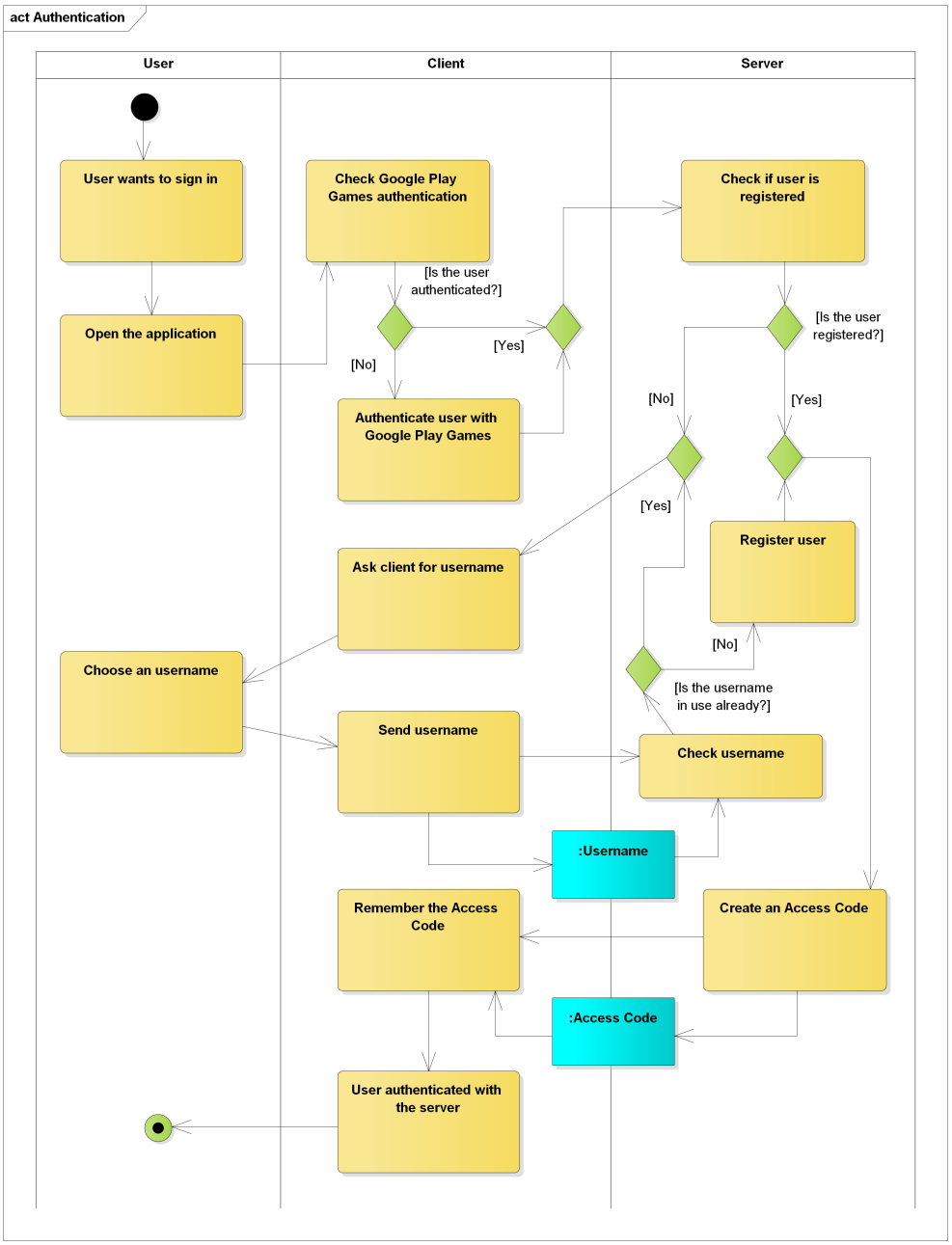

Figure 5.1: Authentication activity diagram

a username and the client sends a new request including the username to the server. The server registers the user and returns the *Access Code*.

5.1.1.3 Sign-in

If the player is found on the server, the client just receives the *Access Code*.

5.1.2 Retrieving game objects

The game objects are retrieved based on the player's location, which is being sent to the server. The first request is sent right after starting the game. Further requests are sent after a certain distance is passed by the player. With each response, all existing game objects are removed from the map and replaced with new ones.

5.1.3 Buying an item

When the user opens a window with details of a *shop*, he can choose an item to buy. The selected item is sent to the server. In a successful response, the server includes the updated player's inventory. The current player's inventory is replaced with the updated one. The *price* taken from the item's type is subtracted from the player's *gold* on the client.

5.1.4 Killing a monster

To kill a *monster*, the user needs to open its window with details. There is an attack button, which simulates a fight. The player can attack the *monster* only if there is a weapon equipped in the right-hand slot, this is checked first. If there is, the *attackDamage* of the weapon is found. The *monster* has the *attackDamage* attribute as well. To simulate a hit, a damage is computed first by taking the *attackDamage* and multiplying it by a random number from 0.5 to 1.5. This is the same for both the player's and the *monster's* hits. The player starts first, the computed damage is subtracted from the *monster's health*. The same thing is then repeated for the *monster* (its hit damage is subtracted from the player's *health*). Once the *monster's health* reaches a value equal or less than 0, a request to kill the *monster* with the updated player's *health* is sent to the server. If successful, the client updates the player's *health* with the updated value, adds the value of the *xp* attribute of the *monster* to the player's *experience* and adds the value of the *gold* attribute of the *monster* to the player's *gold*. The player is then presented with an option to open the *monster's* inventory.

5.1.5 Killing the player

If the player's *health* drops to or below 0 first, a request to kill the player is sent to the server. In this case, the server returns the complete player's profile with updated attributes (restores his *health* and takes some *gold* away). These attributes are then updated for the player.

5.1.6 Collecting items

After killing a *monster*, the server still expects another request with a list of items to be collected. After that, the kill action is marked as complete. In the *monster's* inventory window, an option to collect all items is presented to the player. The player can also just close the window. In both cases, a list of items (an empty list when the window is closed) is sent to the server. When successful, the client receives a response with the updated player's inventory, which replaces the old one. The killed *monster* is also removed from the game on the client.

5.1.7 Equipping an item

To equip an item, the player needs to open the equipment window, then select a slot which opens the inventory window. There the player chooses an item to equip. A request is then sent with the selected item. After a successful response is received, the client assigns the item to the selected slot.

5.1.8 Purchasing an in-app product

To purchase a product, the player needs to open the window with a list of products. After choosing a product and confirming the purchase, the purchased product is sent to the server for confirmation. Once a confirmation is received, the client consumes the product in the same way the server does. In the case of this prototype, the player can buy *gold*. So, the purchased *gold* is added to the current player's *gold* at this point.

5.2 Game object types

This section describes the game object types that appear in this prototype along with their attributes and values.

5.2.1 Skeleton

Skeleton is a *monster* that can be attacked by the player.

Attributes	
health	50
xp	25
gold	20
attackDamage	6

5.2.2 Goblin

Goblin is a *monster* that can be attacked by the player.

Attributes	
health	75
xp	35
gold	25
attackDamage	5

5.2.3 Health Potion

Health Potion is an item that can be bought by the player.

Attributes	
addHealth	100
price	20

5.2.4 Shop

This is a *shop* with items (selling Health Potions).

Attributes	
shop	0

5.2.5 Sword

Sword is a weapon that can be equipped.

Attributes	
price	100
attackDamage	10
slot	hand

5.3 API

The API for the prototype has been specified in cooperation with my colleague Jakub Čech. This section describes just some of the supported API actions (the specification was simplified), the full listing can be found in attachment B. The numbers in the response sections are the HTTP status codes.

5.3.1 GET /login

The Login endpoint verifies the Google ID token and generates an *Access Code* for future requests. When successfully authenticated, the player's profile and the *Access Code* are returned.

5.3.1.1 Parameters

token the *Google ID token*

5.3.1.2 Response

200 Successfully logged in, player's profile and the *Access Code* are returned.

403 Invalid token.

404 User not found, registration needed.

5.3.2 GET /user/inventory

The User Inventory endpoint returns all the items in the player's inventory and the information about what is equipped in which slot.

5.3.2.1 Parameters

accessCode the *Access Code*

5.3.2.2 Response

200 List of items and a map of equipment.

403 Player not logged in.

404 User not found.

5.3.3 POST /action/buy

This action allows to buy items from a *shop*.

5.3.3.1 Parameters

accessCode the *Access Code*

shopid the id of the *shop*

itemId the id of the item to buy

5.3.3.2 Response

200 Player's inventory.

400 Invalid data.

403 Invalid *Access Code*.

404 User not found, registration needed.

500 Unexpected error.

5.4 World map

A key part of the game is the world map. This section describes how the map is drawn on the screen. Figure 5.2 shows an example of the world map in the game.

5.4.1 Tiles

The map is divided into tiles. Each tile is an equivalent for a tile provided by the Mapzen API. The default size of a tile is 256x256 pixels which also equals to the size of a tile returned by the API. But the displayed tiles are scaled based on the resolution of the screen of the device, Full HD (1920x1080 pixels) is the reference resolution. Each tile is responsible for downloading the necessary data and displaying a part of the map. Therefore, a tile holds several layers of map vector data.

5.4.1.1 Landuse

The landuse layer represents a kind of terrain on the map. It is defined as a set of polygon points which mark the borders of the landuse. Instead of drawing simple polygons, I decided to use tiles for displaying it. Simple rectangle tiles look a little too jagged on the map. So, a better solution is to use hexagonal tiles.

Each map tile fills its area with hexagonal tiles representing a landuse kind (a tile can have multiple landuses in it). The landuse tile is randomly chosen from a set of tiles defined for each landuse kind so it does not look too uniform.

For each hexagonal landuse tile, the map tile needs to decide what landuse it belongs to. It does that by looking through the landuse polygons and deciding whether the tile's coordinates fall into the specific polygon.

5.4.1.2 Roads

Road is defined as a series of line points. These lines are simply drawn on the map.

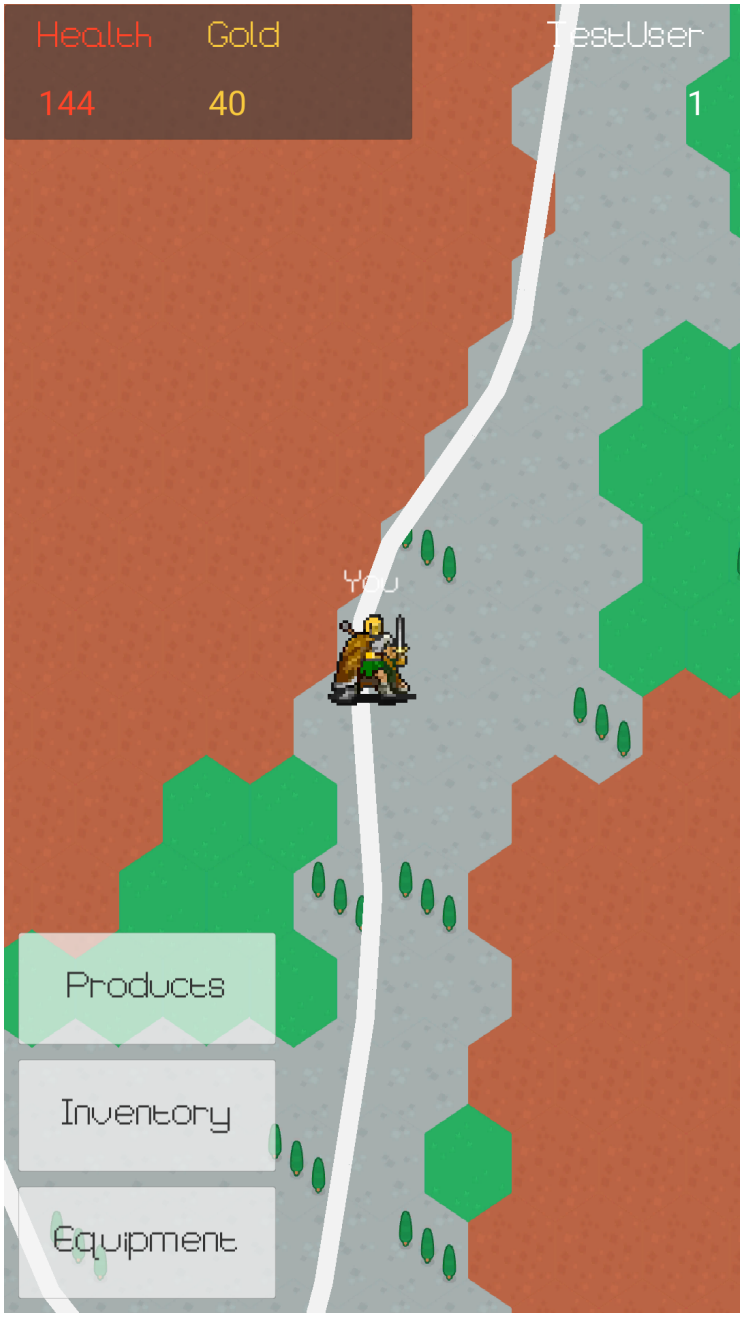

Figure 5.2: The game world map

5.5 Unity

This part describes the structure of the Unity project. It is divided into several folders following a common convention.

5.5.1 Editor

As per the Unity's recommended practice, the *Editor* folder contains the **NUnit** tests [23].

5.5.2 Plugins

This folder contains plugins required by the game.

5.5.3 Prefabs

This folder contains the GameObject *prefabs*. Prefabs in Unity are used for instantiating *GameObjects* in the game. There are two subfolders for different types of prefabs.

5.5.3.1 Game

All the game objects that are used in the game are located here. This includes the monsters, buildings, items and the player prefab. Also, there are prefabs used for displaying the map. The *LandusePrefab* represents a landuse kind and holds the polygon points. The *LanduseTile* prefab renders the actual hexagonal tiles on the map based on the landuse it belongs to.

5.5.3.2 UI

This folder holds different UI components that can be easily reused and composed together. All the windows in the game are built from these.

5.5.4 Scenes

The prototype has two main scenes and one test scene.

5.5.4.1 AuthScreen

This scene is the initial scene that is loaded upon the start of the application. It presents the user with an option to sign into his Google Play Games account. It also contains an input field that is shown if a username needs to be set during the registration. It deals with the player's authentication, initiates the location updates and loads the *MapScreen*.

5.5.4.2 MapScreen

This scene represents the actual game screen with the world map accessible only after a successful authentication.

5.5.4.3 PrefabsTest

This scene is a part of the integration tests. These Unity-specific tests are designed to run in a separate scene [24].

5.5.5 Resources

This folder holds the resources used by the game. It includes the *Road Material* used for drawing roads on the world map [25].

5.5.6 Scripts

The *Scripts* folder contains all the written code for the project.

5.5.6.1 API

The *API* subfolder defines an interface for the API, communicates with the server and returns responses.

5.5.6.2 Auth

The *Auth* subfolder deals with the authentication of the player and provides functionality for the authentication screen.

5.5.6.3 Camera

This subfolder contains all the necessary functionality for the camera, such as computing the orthogonal size.

5.5.6.4 Common

The *Common* subfolder contains common classes for the whole project.

5.5.6.5 Game

This is the main part of the game which connects all the other parts. It also holds the scripts for the game objects.

5.5.6.6 JSON

The *JSON* subfolder processes JSON responses and returns appropriate game entities.

5.5.6.7 Map

The *Map* subfolder parses the map tiles, displays the world map and obtains the device's location.

5.5.6.8 Network

This subfolder handles the HTTP communication with the server.

5.5.6.9 Purchasing

The *Purchasing* subfolder deals with in-app purchases in the game.

5.5.6.10 UI

The *UI* subfolder provides functionality for displaying the user interface and its elements, such as dialogs.

5.5.7 Sprites

The *Sprites* folder holds the image resources for the game, UI elements and the icon.

5.5.8 Tests

This folder contains the scripts for integration testing.

5.6 Class design

This section shows the design of the main components in the *Scripts* folder. A complete listing with diagrams can be found in attachment C. Some classes that inherit from *MonoBehaviour* contain public attributes for them to be accessible in the Unity editor for an easy configuration [26].

5.6.1 API

Figure C.1 shows a class diagram for the *API* package.

5.6.2 Game

The *Game* package is split into multiple class diagrams. Figure C.6 shows a class diagram for a part of the *Game* package with the *Player*. Figure C.7 shows a class diagram for a part of the *Game* package with the *MapGameObject*.

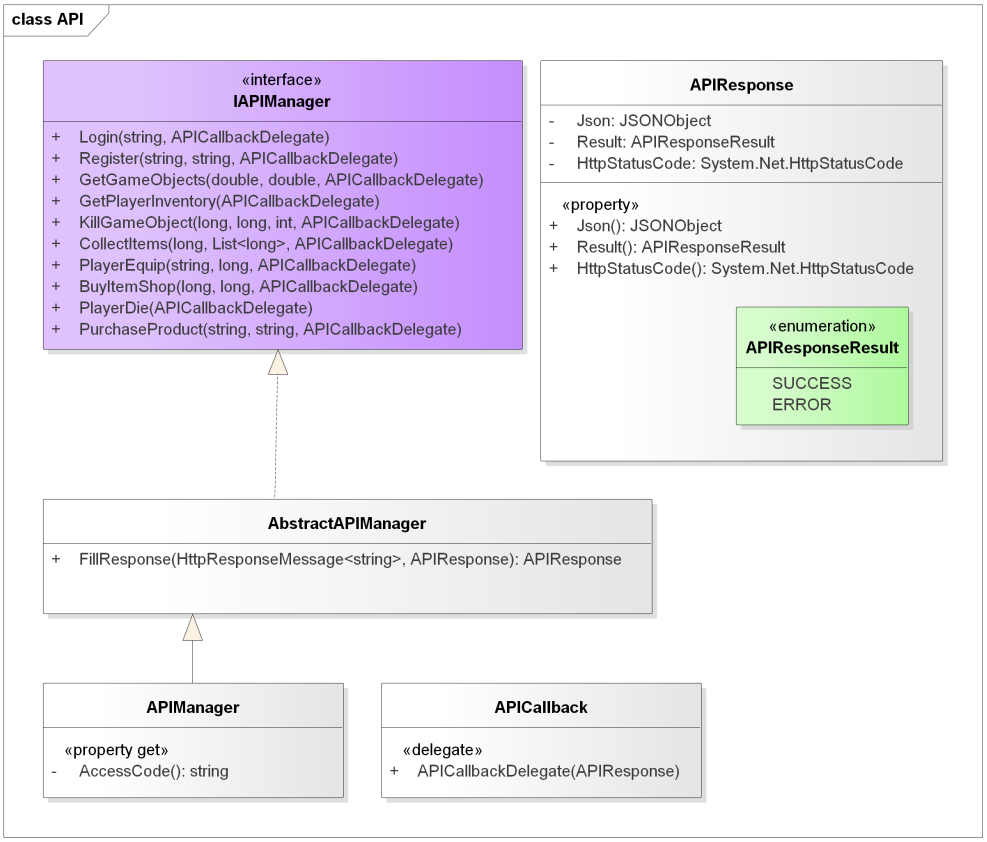

Figure 5.3: API package class diagram

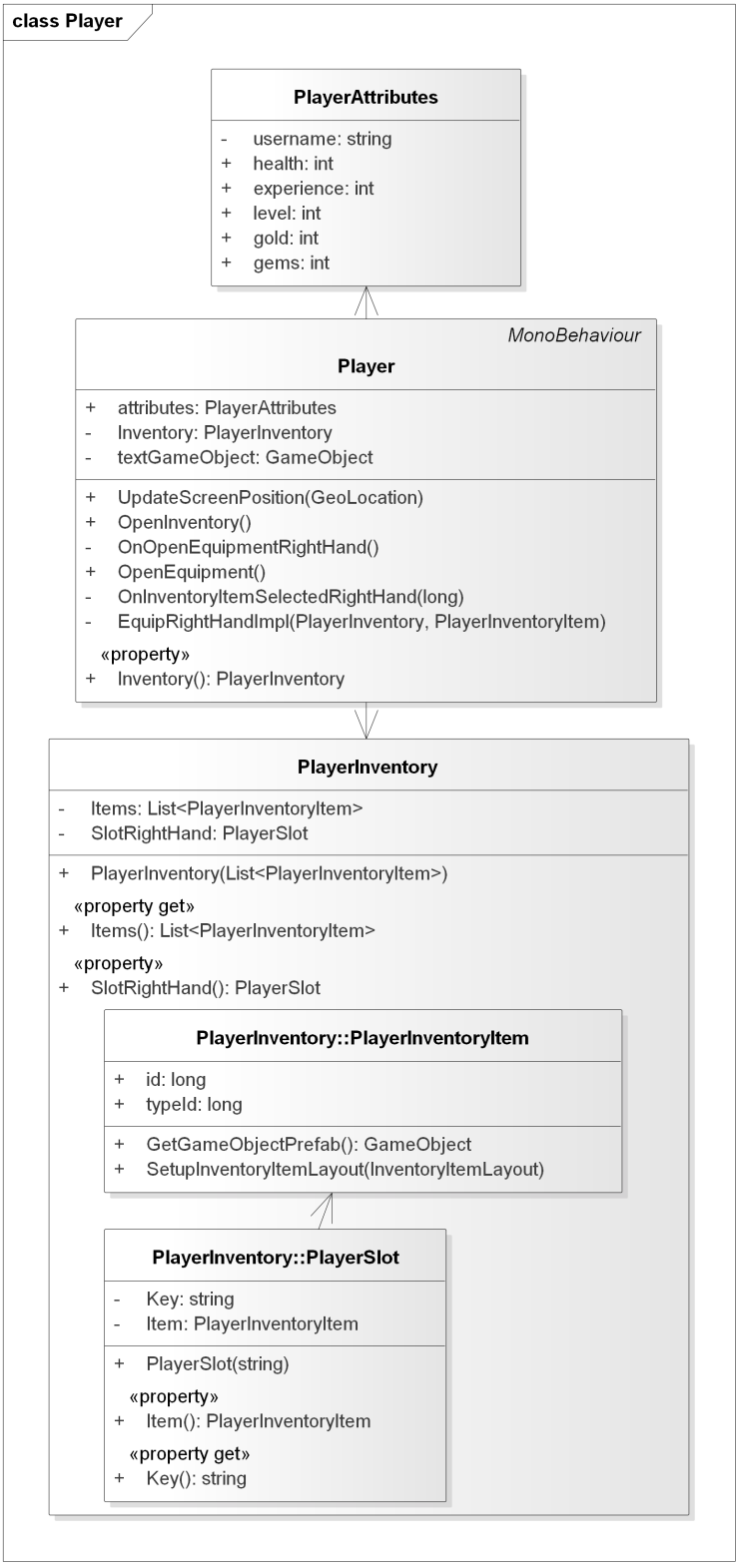

Figure 5.4: Player part of the Game package class diagram

Figure 5.5: Part of the Game package class diagram

Chapter 6

Implementation

The implementation chapter describes the plugins, each component from the *Scripts* folder in the Unity project and some Unity-specific details.

6.1 Assets and plugins

The game utilizes several assets and plugins.

6.1.1 Google Play Games plugin for Unity

For retrieving the necessary data for authenticating the player using his Google Play Games account, I used the **Google Play Games plugin for Unity**. It is an open-source project created by Google to support many features of the Google Play Games API [10]. The server uses the Google Play Games *ID token* to authenticate the user with Google servers.

It requires several steps for configuration. The game needs to be created in the **Google Play Developer Console**. By configuring the game in the Console, a *client ID* is created. In the case of this project, an *Android app* and a *Web app* had to be created on the *Linked apps* page. After importing the plugin in Unity, it is easy to configure it in the UI. The Web app's client ID must be pasted into the plugin's Unity configuration for the server to be able to access the user's details and the authentication to work. The same *client ID* is used on the server for validation.

6.1.2 Http Client

The **Http Client** plugin from the **Unity Asset Store** is used for creating the HTTP requests [27].

6.1.3 JSON Object

For working with JSON objects, the project contains the **JSON Object** scripts from the **Unity Asset Store** [28].

6.1.4 Unity IAP

The system for implementing in-app purchases is the **Unity IAP**. It is a part of the **Unity Services** and needs to be enabled in Unity. After enabling, Unity offers an option for automatically importing the plugin into the project [29].

6.1.5 Unity Test Tools

The **Unity Test Tools** package from the **Unity Asset Store** contains features for integration testing of the game [30].

6.2 Game scripts

This section describes the functionality of all the components in the *Scripts* folder in more detail.

6.2.1 IAPIManager

This interface defines all the API actions as single methods. Each method has several parameters including the *APICallback.APICallbackDelegate* callback for notifying the caller of a finished API call.

6.2.2 AbstractAPIManager

This is an abstract class implementing the *IAPIManager* interface. It holds an *INetworkManager* instance. It has the *FillResponse* method that takes an HTTP response and a *APIResponse* as parameters. The *APIResponse* is populated with data based on the HTTP response.

6.2.3 APIManager

The *APIManager* is a child of the *AbstractAPIManager*. It implements all the methods corresponding to the API actions and contains different *string* constants for creating the API calls. It uses the *NetworkManager* in the parent to communicate with the server and receives HTTP responses. After an HTTP call is finished, it creates an *APIResponse*, uses the *FillResponse* method to populate the response and invokes the callback delegate.

6.2.4 APIResponse

APIResponse holds a JSON response, an *APIResponseResult*, which is an enum describing the result of the response, and the actual HTTP status code. This response is passed to the *APICallback.APICallbackDelegate*.

6.2.5 APICallback

This class contains only the *APICallbackDelegate*.

6.2.6 PlayGamesAuth

The script used on the Authentication screen is the *PlayGamesAuth*. First, it activates the Google Play Games plugin and starts listening for location updates. Then it checks whether the user is already authenticated with Play Games. If not, it uses the plugin to prompt the user to sign into their Play Games profile. Once it is finished, it uses the plugin again to get an *ID token*. If there is a token returned and it is not empty, it can finally send it to the server. The server might return a response saying that the user does not exist yet. In that case, an *InputField* is shown to the user along with a prompt to choose a username. Once the user chooses one and clicks the sign-up button, a registration request is sent to the server. The server checks whether the username is already in use. If it is, the user is notified to choose a different username. If the registration succeeds, the server returns a successful authentication response. It is the same response returned as if the player already existed on the server. Once the authentication succeeds, the *Access Code* and the *PlayerAttributes* are retrieved from the response. These are set to the *GameManager*. After that, the script attempts to start the game. If the player's location is not obtained at this point, it shows a window with a message telling the player that the game is waiting for a location. When the location is obtained, the *MapScreen* is loaded.

There are three public fields to be able to specify the minimum and maximum length of the username as well as a regular expression directly in the Unity editor.

6.2.7 CameraSize

CameraSize computes the orthographic size of the camera for a correct displaying of sprites on all resolutions. The *pixelsPerUnit* attribute holds the value corresponding to the *Pixels Per Unit* attribute of the game sprites.

6.2.8 GenericSingleton

This abstract class provides an implementation of the *Singleton* pattern. It also preserves the *GameObject* it is attached to across multiple scenes. It

uses generics for the children to be able to specify their type as a type for the *Singleton* instance.

6.2.9 SimpleSingleton

Just like *GenericSingleton*, this is an abstract class that implements the *Singleton* pattern. Unlike its counterpart, it does not preserve the *GameObject* across multiple scenes.

6.2.10 SimpleObjectPool

SimpleObjectPool is an implementation of an object pool that provides instances of a given *GameObject*. It has a public field holding the *prefab* that is being instantiated. There is a stack holding unused instances of the *GameObject* prefab. If the stack is empty, a new instance is created. The *PooledObject* component is assigned to each object. It holds a reference to the original object pool that it came from. When an object is returned to the pool, this component is checked to determine the original pool. If it comes from the same pool it is being returned to, it is put on the stack. If not, the object is simply destroyed.

6.2.11 GameManager

GameManager is a subclass of the *GenericSingleton* class and is the main part of the game. It contains other managers that are accessed from all parts of the game. These are the *IAPIManager*, the *IJSONManager*, the *MapManager* and the *IAPManager*. Also, the *LocationService* is part of the *GameManager* to obtain the location in any part of the game. The game object prefabs are contained in an instance of the *GameObjectPrefabs* class.

It also specifies the server and API paths. There is an *Access Code* field for authenticating the user with the server.

It has an event listener that listens for loading of the scenes. After the *MapScreen* is loaded, *GameManager* calls the *Init* method. As a part of the initialization, there are several *GameObjects* looked up by a tag. These include the *MapManager*, the *Player*, and the *GameObjectPrefabs*. The player has the attributes assigned and an API call is made to get the player's inventory. If the player's location is obtained at this point, his position on the map is updated, and the *GameManager* starts retrieving the game objects. The game objects are retrieved periodically only after a specified distance is travelled by the player. The old game objects are deleted and replaced with the new game objects from the server.

6.2.12 GameObjectPrefabs

This class holds the game object prefabs and creates a dictionary of them. First, the game objects are assigned to an array in the Unity editor. The script then goes through the array. For each game object prefab, it gets a type id from the *MapGameObject* component and adds the prefab to the dictionary using the type id as a key. There is a *GetPrefab* method taking a type id as a parameter and returning the corresponding prefab from the dictionary.

6.2.13 GameStates

It holds the current game states. They are represented by *boolean* properties with getters and setters. Upon the creation of the object, all the states are set to false.

6.2.14 Inventory

This class provides functionality for a game object's inventory.

6.2.15 MapGameObject

This is a common class for game objects that can be displayed on the map. It is added to all game objects in the game as a component. It holds some basic attributes of an object, such as a type id, an icon, a name and a description. The *UpdatePosition* method puts the object on a position on the map corresponding to its geo location and enables the rendering of the object. It is called by the *GameManager*. The object holds its attributes of type *MapGameObjectAttributes* and the *Inventory*. There is a dictionary of type attributes, but these are only populated for the prefabs and not for each instance of the prefab. For accessing the type attributes in each instance of the prefab, there is a reference to the original prefab.

6.2.16 MapGameObjectAttributes

The *MapGameObjectAttributes* class represents the basic attributes of a game object. It contains several properties, the first property is the *Id*, identifying the game object. The *LocationId* property is the id of the *Location*, which is a server entity. This id is used for interacting with an object on the specific location, in the case of this game for killing a *monster*, and along with the actual id of a game object fully specifies it. The *Name* is optional and can override a game object prefab's name. The *Desc* (description) is optional as well and works the same way. The *Location* property determines the geographic location of an object. This property is not used in a case of nested objects.

6.2.17 MapGameObjectTypeAttribute

This class represents a single type attribute for a game object. It has a key that identifies the attribute and is used as a key for the type attributes dictionary of the *MapGameObject* prefabs. The type attribute can have either an integer or a string value. The class is serializable to be editable in the Unity editor.

6.2.18 Monster

The *Monster* script provides functionality for the *monster* game object type. It has a reference to the *MapGameObject* component of the *GameObject* for accessing all the necessary attributes. It has several callbacks for interacting with the UI. There are three actions that are performed by this script:

- killing a monster,
- killing the player,
- collecting items.

6.2.19 Player

This class represents the player in the game. It holds the player's attributes and inventory. It can update the player's position on the world map based on the geographic location. There are methods for opening the inventory and equipment windows as well as an implementation for equippping an item in the right-hand slot.

6.2.20 PlayerAttributes

This class represents the basic attributes of the player. It is the username, health, experience, gold and gems. There is also the player's level which is computed dynamically based on the experience. The class is serializable to be editable in the Unity editor.

6.2.21 PlayerInventory

PlayerInventory provides functionality for the player's inventory, holds the items and the right-hand slot (just for this prototype, more slots can be included in the future). It has two nested classes.

6.2.21.1 PlayerInventoryItem

PlayerInventoryItem represents an item in the player's inventory. It has a method for populating an item layout in the inventory.

6.2.21.2 PlayerSlot

PlayerSlot represents a slot for an item from the player's inventory. It has a key for identification and a reference to the equipped item.

6.2.22 Shop

The *Shop* script provides functionality for the *shop* game object type. It has a reference to the *MapGameObject* component of the *GameObject* for accessing all the necessary attributes. It can populate an item layout with the name and the price of an item. It has a callback for choosing an item in the *shop* UI and implements the buy item action.

6.2.23 NetworkManager

NetworkManager holds an instance of the *HttpClient* class from the **Http Client** library. Since the *HttpClient* uses a delegate for a callback after the request is finished, *NetworkManager* accepts this delegate as one of its parameters in the HTTP methods.

6.2.24 IJSONManager

This interface defines the methods for parsing JSON responses from the server.

6.2.25 JSONManager

This is an implementation of the *IJSONManager*. Each method takes a *JSONObject*, parses it, and converts it to an appropriate game entity.

6.2.26 JSONConstants

The *JSONConstants* class holds the constants for parsing JSON responses.

6.2.27 GeoLocation

GeoLocation represents a geographic location with the latitude and longitude components.

6.2.28 IMapProjection

This interface defines a projection of the map. It has four methods, *GetTileSize* returns the size of a tile, *GeoLocationToScreenPosition* takes a given geographic location and converts it to a corresponding screen position. The *ScreenPositionToGeoLocation* method does the exact opposite. The *Distance* method takes two geographic locations and returns the distance between them.

6.2.29 Landuse

This class is a polygon representation of a landuse. It contains the landuse polygon points which are assigned to the *PolygonCollider2D* component. The *CollidesWithWorldPosition* method takes a point and checks for a collision with the polygon collider. It is used by the Mapzen tiles to draw the hexagonal tiles. The *GetRandomTilePrefab* gets an object from the tile pool and sets up the sprite by choosing a random one from the *TileSprites* array.

6.2.30 LocationService

This script obtains the device's location. Getting the location in Unity is straightforward. Location is a part of the *Input* interface, which provides all kinds of inputs, such as keyboard, mouse and acceleration data. It needs to be started first and once it gets to the *Running* state, the data can be accessed (in the case of this game the latitude and longitude components). There is no support for listeners in the location input, so it needs to be polled periodically. *LocationService* keeps a list of listeners that get notified when the location changes, but only after the specified update distance is passed. Also, the location is obtained only after a specified time interval is elapsed.

6.2.31 MapManager

This is a base abstract class for displaying the map. I defined no interface for the map manager to be able to add it to the Unity editor and modify it since Unity does not support interfaces in the editor. The main parameter is the zoom, which defines the magnification level for displaying the map. There is a dictionary of tiles with a screen position as the key. Based on the camera, the *MapManager* adds and removes tiles from the view. There is the abstract *CreateMapTile* method, which creates a new tile on the map. The *CreateMapProjection* creates a projection for the map. *MapManager* also supports moving the camera view to a certain geographic location, which happens every time a location update is received.

6.2.32 MapTile

This abstract class represents a single tile on the map. It keeps its geographic location and screen position. The method to be implemented in the subclass is *LoadTileImpl*, which loads and displays the tile data.

6.2.33 MapzenVectorMapManager

This is a subclass of the *MapManager*. There are several parameters defined that can be modified in the Unity editor for requesting data from Mapzen.

It contains public arrays of *Sprites* with the hexagonal tiles to be displayed on the map. There is a dictionary created from those which can be accessed with a landuse kind as the key. In the implemented *CreateMapTile* method, it creates a *GameObject* with the *MapzenVectorTile* component. The created projection is the *MapzenVectorMapProjection*.

6.2.34 MapzenVectorMapProjection

This is an implementation of the *IMapProjection* for the Mapzen vector map. Upon creation, it computes the tile size based on the screen resolution (it accesses the *UIManager* for that).

6.2.35 MapzenVectorTile

This class is a subclass of the *MapTile* class and represents a single Mapzen vector tile. It loads the vector data from Mapzen and contains the functionality for drawing hexagonal tiles based on the landuses. The roads are drawn by the *Road* script. It also supports caching of the tiles; each tile is saved to a single file to a persistent directory of the application.

6.2.36 Road

The *Road* script provides functionality for drawing a road. When the script starts, it adds the *LineRenderer* component to the attached *GameObject* and assigns the given road points to it. There is also a parameter for setting the road width.

6.2.37 INetworkManager

This interface defines some basic HTTP methods used for communication with the server, these are *GET*, *POST* and *PUT*. It takes an URL for each method and uses a callback with a *string* parameter to notify the caller after the response is received from the server.

6.2.38 NetworkManager

NetworkManager implements the HTTP methods from the *INetworkManager* interface. It holds an instance of the *HttpClient* from the *Http Client* plugin. The *HttpClient* has the corresponding methods implemented and accepts the callback parameter as well.

6.2.39 IAPManager

This class is the manager for purchasing products in the game. It implements the *IStoreListener* and follows the Unity-recommended implementation [29].

The *BuyProductID* method initiates the purchasing process for a given product id. After the purchase is completed by the user, the *ProcessPurchase* method returns the *PurchaseProcessingResult.Pending* indicating that the product is not consumed yet. After that, the manager sends the product to the server for validation and consumption. After a successful response from the server is received, the product is consumed on the client as well and marked as *Complete*. If anything goes wrong before marking the product as *Complete*, the *ProcessPurchase* method is called again automatically when the app is launched next time (and after the *IAPManager* is initialized).

6.2.40 Products

The *Products* class contains a list of products defined as *string* constants.

6.2.41 BringToFront

BringToFront is a simple script that brings the *GameObject* (with an UI element) it is attached to to the front when enabled.

6.2.42 ChoiceWindow

ChoiceWindow is a window with yes and no buttons that presents a choice. There is a message that needs to be specified along with optional callbacks for the buttons.

6.2.43 EquipmentModalWindow

This is the player's equipment window which displays the right-hand slot, either empty or with an equipped item. There is callback that can be specified for clicking the right-hand slot.

6.2.44 HUDManager

HUDManager manages an overlay with the player's information, such as a username and *health*.

6.2.45 InventoryItemLayout

InventoryItemLayout represents the layout for a single item entry in an inventory. It has a listener for click events. The *Setup* method allows to populate the item's layout with an icon, a name and a description.

6.2.46 InventoryList

This UI script populates and displays an inventory. For opening the inventory window, an icon and a name are specified to be shown in the header. Also,

the inventory items to be displayed are passed to it along with the necessary callbacks.

6.2.47 MessagesContainer

This class contains the messages that are being displayed in the UI defined as *string* constants.

6.2.48 MonsterInventoryList

MonsterInventoryList populates and displays a *monster's* inventory. There are two callbacks for when the user decides to collect all items or just exits the inventory.

6.2.49 MonsterModalWindow

This is a window for a *monster*. The parameters passed to the *Open* method are the icon and the name of a *monster* as well as callbacks for attacking the *monster*, opening its inventory and closing the window. The *OnMonsterKilled* method is called when the *monster* is killed. It accepts a message to be displayed as its parameter (for displaying the attack result).

6.2.50 OkWindow

OkWindow is a window with a message and an ok button. The callback for the ok button click can be specified in the *Open* method.

6.2.51 PlayerInventoryList

The *PlayerInventoryList* class populates and displays a player's inventory. For opening the inventory, the player's inventory items and a callback for an item click are passed to it.

6.2.52 ProductLayout

ProductLayout represents the layout for a single in-app product. For the setup, it requires the id of a product and a callback that is invoked when the product is clicked.

6.2.53 ProgressModalWindow

This is a window for displaying progress in the form of a message that is specified in the *Open* method.

6.2.54 PurchasingUI

PurchasingUI represents the UI for purchasing in-app products. It populates a scroll view with the given products. There is also a callback that needs to be specified for when a product is clicked.

6.2.55 UIManager

This script is the basic component of the UI. It computes a factor for the current resolution based on the specified basic resolution. This factor is accessed by the map components for displaying the map correctly on all resolutions. It also monitors the UI elements in the view (windows) by having two methods for adding and removing an element that are called by different UI scripts.

6.3 Unity-specific setup

This section describes some of the components contained in the Unity editor.

6.3.1 User interface

Each scene in the game contains a *Canvas* with UI components in it. *"The Canvas is the area that all UI elements should be inside. The Canvas is a Game Object with a Canvas component on it, and all UI elements must be children of such a Canvas."*[31]

6.3.2 HUD

On the *MapScreen*, there is a *GameObject* inside the *Canvas* that contains the UI elements with player's information.

6.3.3 Modal windows

The *Canvas* contains several *GameObjects* that represent modal windows. Each of the windows contains a *Panel* that is stretched on the whole screen with transparency set to a certain level to slightly dim the objects behind it. Inside this full screen *Panel* there is another *Panel* that contains elements of the actual modal window and fills only a part of the screen. This *Panel* has multiple *Panels* inside it depending on the specific modal window. A generic modal window has an *Image* component, a few *Text* components, *Buttons* and a top *Panel* with a close button. The inventory windows also have the *Scroll Rect* script in them along with other necessary components required for displaying the scroll view.

Chapter 7

Testing

At the beginning of the development phase, mock responses were used for simulating communication with the server. The game features have been user-tested by using both real and mock locations for the device. The game has been deployed to several Android emulators with different resolutions and versions to validate the correct UI behavior. Usability testing has not been conducted since this is just a prototype of a real game. To showcase the Unity's testing capabilities, two types of tests are included.

7.1 Unit tests

These tests are not typically applicable in a large scale for Unity development since most parts of a game usually involve working with multiple *GameObjects* and components and that is when the types of tests in the next section are used.

7.1.1 JSONTests

These tests validate the parsing of mock *JSON* responses.

7.2 Integration tests

The *Test* folder contains **Dynamic Integration Tests** for testing larger parts of the prototype [24].

7.2.1 TestPrefabs

The *TestPrefabs* script tests the *GameObjectPrefabs* used on the *MapScreen* to ensure it contains all the described five game object types.

Conclusion

The goal of the thesis was to create a prototype of the client part of a role-playing game with features of augmented reality (AR). I analyzed existing similar games on the market. Then, in cooperation with my colleague Jakub Čech, I specified the rules and features of the game prototype.

I analyzed and tested a few options for displaying the world map. Based on that, I chose a solution. I gained knowledge about supporting micro-transactions in the game. Then I designed the structure and components of the prototype. I implemented the chosen map solution, added functionality for obtaining the device's location, added a microtransaction functionality, implemented communication with the server, user authentication, parsing and displaying the game objects on the world map and added the required functionality. Finally, I created the user interface, tested all the features and released the prototype. The final tile-based game world map driven by vector data satisfied the set condition of creating an innovative solution for displaying it.

I used the Unity engine with the *Microsoft Visual Studio* IDE for development and *GIT* as a versioning system. *Enterprise Architect* has been used for creating the diagrams. Communication with the server has utilized *JSON* content.

There were many features that had to be left out of the prototype because of the scope of this thesis, for example a quest system or an option for using items (like consuming a potion to add player's health), to mention some of them. Also, the map can be further improved by adding more types of tiles corresponding to more landuse types. The design should easily allow me to extend the prototype by including additional features and to create a fully-fledged game that can be released on the market.

Bibliography

[1] Parallel Kingdom - screenshot. [online], [cit. 2017-05-12]. Available from: http://www.parallelkingdom.com/img/theme/pages/pk_media/screenshots_page/pieces/screenshot_android_2_full.png

[2] Parallel Kingdom Reaches One Million Players. *ParallelKingdom.com - Latest Press Releases [online]*, January 2012, [cit. 2017-04-21]. Available from: http://www.parallelkingdom.com/Media/PR/PR_013012_One_Million.pdf

[3] Parallel Kingdom. [online], [cit. 2017-04-21]. Available from: http://www.parallelkingdom.com/

[4] Parallel Kingdom - Guide - Overview. [online], [cit. 2017-04-21]. Available from: http://www.parallelkingdom.com/guide/en/page1.aspx/

[5] Google Play - Ingress. [online], [cit. 2017-04-21]. Available from: https://play.google.com/store/apps/details?id=com.nianticproject.ingress/

[6] Pokémon Go. [online], [cit. 2017-05-12]. Available from: http://www.pokemongo.com/

[7] Google+ Community - Maplord testers. [online], [cit. 2017-05-11]. Available from: https://plus.google.com/communities/105545818418311721004

[8] Android Studio and SDK Tools. [online], [cit. 2017-05-13]. Available from: https://developer.android.com/studio/index.html

[9] Google Maps Android API. [online], [cit. 2017-05-13]. Available from: https://developers.google.com/maps/documentation/android-api/

[10] GitHub - Google Play Games plugin for Unity. [online], [cit. 2017-05-10]. Available from: https://github.com/playgameservices/play-games-plugin-for-unity

[11] Google Maps APIs. [online], [cit. 2017-05-12]. Available from: https://developers.google.com/maps/

[12] Mapbox. [online], [cit. 2017-05-12]. Available from: https://www.mapbox.com/

[13] OpenStreetMap Wiki. [online], [cit. 2017-04-22]. Available from: https://wiki.openstreetmap.org/wiki/Main_Page

[14] Google Static Maps API. [online], [cit. 2017-05-12]. Available from: https://developers.google.com/maps/documentation/static-maps/

[15] Foursquare. [online], [cit. 2017-05-12]. Available from: https://foursquare.com/

[16] Uber Technologies. [online], [cit. 2017-05-12]. Available from: https://www.uber.com/

[17] Mapbox - Pricing. [online], [cit. 2017-04-22]. Available from: https://www.mapbox.com/pricing/

[18] Mapbox - Unity. [online], [cit. 2017-04-22]. Available from: https://www.mapbox.com/unity/

[19] Data sources in Mapzen Vector Tiles. [online], [cit. 2017-04-20]. Available from: https://mapzen.com/documentation/vector-tiles/data-sources/

[20] Layers in Mapzen's vector tiles - Landuse. [online], [cit. 2017-04-20]. Available from: https://mapzen.com/documentation/vector-tiles/layers/

[21] Layers in Mapzen's vector tiles - Roads (Transportation). [online], [cit. 2017-04-20]. Available from: https://mapzen.com/documentation/vector-tiles/layers/

[22] Unity Manual - Unity IAP. [online], [cit. 2017-04-20]. Available from: https://docs.unity3d.com/Manual/UnityIAP.html

[23] Unity - Manual: Editor Test Runner. [online], [cit. 2017-05-12]. Available from: https://docs.unity3d.com/Manual/testing-editortestsrunner.html

[24] Bitbucket - Unity-Technologies / UnityTestTools / wiki / IntegrationTestsRunner. [online], [cit. 2017-05-12]. Available from: https://bitbucket.org/Unity-Technologies/unitytesttools/wiki/IntegrationTestsRunner

[25] Unity - Manual: Creating and Using Materials. [online], [cit. 2017-05-10]. Available from: https://docs.unity3d.com/Manual/Materials.html

[26] Unity - Scripting API: MonoBehaviour. [online], [cit. 2017-05-10]. Available from: https://docs.unity3d.com/ScriptReference/MonoBehaviour.html

[27] Unity Asset Store - Http Client. [online], [cit. 2017-05-10]. Available from: https://www.assetstore.unity3d.com/en/#!/content/79343

[28] Unity Asset Store - JSON Object. [online], [cit. 2017-05-10]. Available from: https://www.assetstore.unity3d.com/en/#!/content/710

[29] Unity Tutorials - Integrating Unity IAP In Your Game. [online], [cit. 2017-05-09]. Available from: https://unity3d.com/learn/tutorials/topics/ads-analytics/integrating-unity-iap-your-game

[30] Unity Asset Store - Unity Test Tools. [online], [cit. 2017-05-12]. Available from: https://www.assetstore.unity3d.com/en/#!/content/13802

[31] Unity Manual - Canvas. [online], [cit. 2017-04-20]. Available from: https://docs.unity3d.com/Manual/UICanvas.html

Appendix A

Acronyms

API Application Programming Interface

AR Augmented reality

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

IDE Integrated Development Environment

JSON JavaScript Object Notation

RPG Role-playing game

Appendix B

API

B.1 GET /login

The Login endpoint verifies the Google ID token and generates an *Access Code* for future requests. When successfully authenticated, the player's profile and the *Access Code* are returned.

B.1.1 Parameters

token the *Google ID token*

B.1.2 Response

200 Successfully logged in, player's profile and the *Access Code* are returned.

403 Invalid token.

404 User not found, registration needed.

B.2 POST /login/register

The Registration endpoint creates a new player on the server. The profile is initialized with default values. If the username is taken or if the player already exists, then an error is returned.

B.2.1 Parameters

username the new player's username

token *Google ID token*

B.2.2 Response

200 Successfully registered and logged in, player's profile and *Access Code* returned.

409 Either the username exists or the user is already registered. See error details.

B.3 GET /user

The User endpoint returns the player's profile.

B.3.1 Parameters

accessCode the *Access Code*

B.3.2 Response

200 User profile and *Access Code* returned.

403 Invalid *Access Code*.

404 User not found, registration needed.

B.4 PUT /user/die

The User Die endpoint kills the player. After the death, the health is restored and some amount of gold is removed from the player based on his level.

B.4.1 Parameters

accessCode the *Access Code*

B.4.2 Response

200 Player's profile and *Access Code* returned.

403 Invalid *Access Code*.

404 User not found, registration needed.

B.5 GET /user/inventory

The User Inventory endpoint returns all the items in the player's inventory and the information about what is equipped in which slot.

B.5.1 Parameters

accessCode the *Access Code*

B.5.2 Response

200 List of items and a map of equipment.

403 Player not logged in.

404 User not found.

B.6 POST /purchase

The Purchase endpoint offers support for in-app purchases. The purchase is verified and then assigned to the player. To be accepted, it cannot be cancelled or consumed.

B.6.1 Parameters

accessCode the *Access Code*

productId the id of the product to buy

token the purchase token

B.6.2 Response

200 Player's profile.

400 Invalid data.

403 Invalid *Access Code* or the purchase is not valid.

404 User not found, registration needed.

500 Unexpected error.

B.7 GET /location

This retrieves all nearby locations in a 200-m radius from the provided coordinates. The locations are returned along with their associated objects.

B.7.1 Parameters

lat the latitude

lon the longitude

accessCode the *Access Code*

B.7.2 Response

200 List of nearby locations with game objects assigned to them.

500 Unexpected error.

B.8 POST /action/buy

This action allows to buy items from a *shop*.

B.8.1 Parameters

accessCode the *Access Code*

shopid the id of the *shop*

itemId the id of the item to buy

B.8.2 Response

200 Player's inventory.

400 Invalid data.

403 Invalid *Access Code*.

404 User not found, registration needed.

500 Unexpected error.

B.9 POST /action/collect

This action allows collecting items from a *monster's* inventory after a kill. It can be called only once after a kill.

B.9.1 Parameters

accessCode the *Access Code*

killConfirmedCode the kill confirmed code

gameObjects the list of items to be collected

B.9.2 Response

200 Player's inventory.

400 Invalid data.

403 Invalid *Access Code*.

404 User not found, registration needed.

500 Unexpected error.

B.10 PUT /action/equip

This action equips an item from the player's inventory to the specified slot.

B.10.1 Parameters

accessCode the *Access Code*

itemId the id of the item to be equipped

slot the slot key

B.10.2 Response

200 Success.

400 Invalid data.

403 Invalid *Access Code*.

404 User not found, registration needed.

500 Unexpected error.

B.11 POST /action/kill

The kill action is performed on the selected object and location. The location is temporarily excluded from future requests to /location. The player's health is updated, experience and gold are added. It returns a *killConfirmedCode* which is needed to perform the collect action.

B.11.1 Parameters

accessCode the *Access Code*

locationid the id of the location

gameObjectId the id of the game object

health the new player's health

B.11.2 Response

200 One-time code for kill confirmation.

400 Invalid data.

403 Invalid *Access Code*.

404 User not found, registration needed.

500 Unexpected error.

Appendix C

Class diagrams

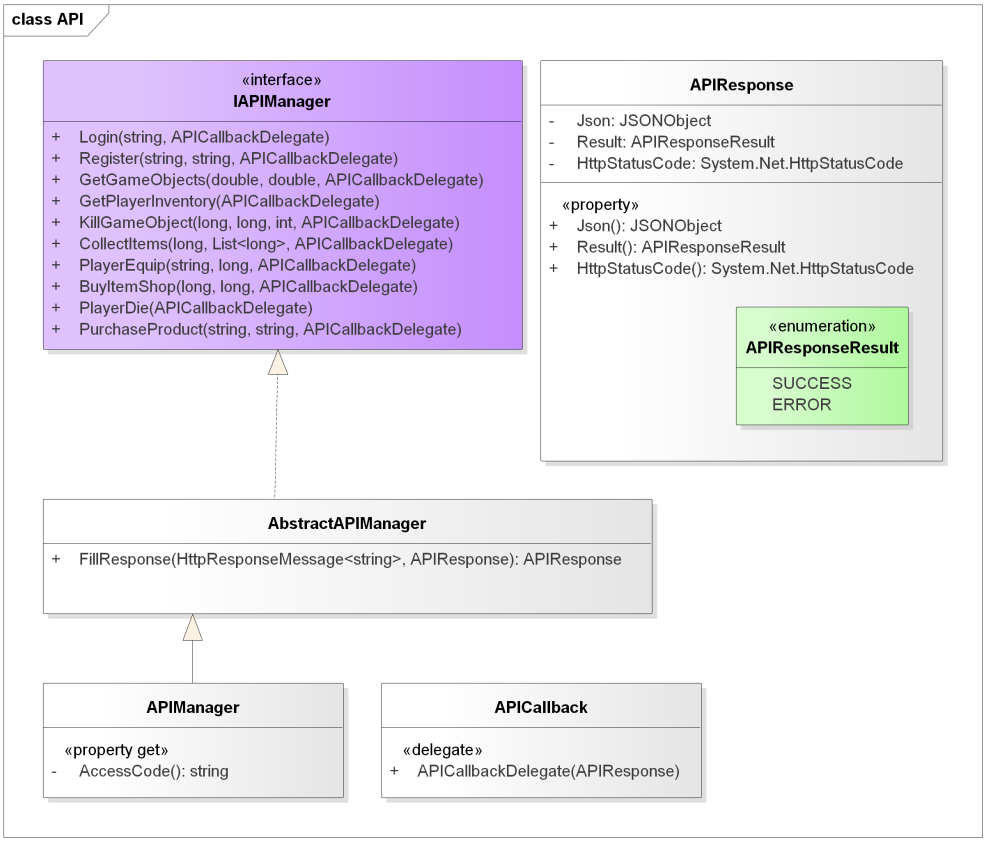

Figure C.1: API package class diagram

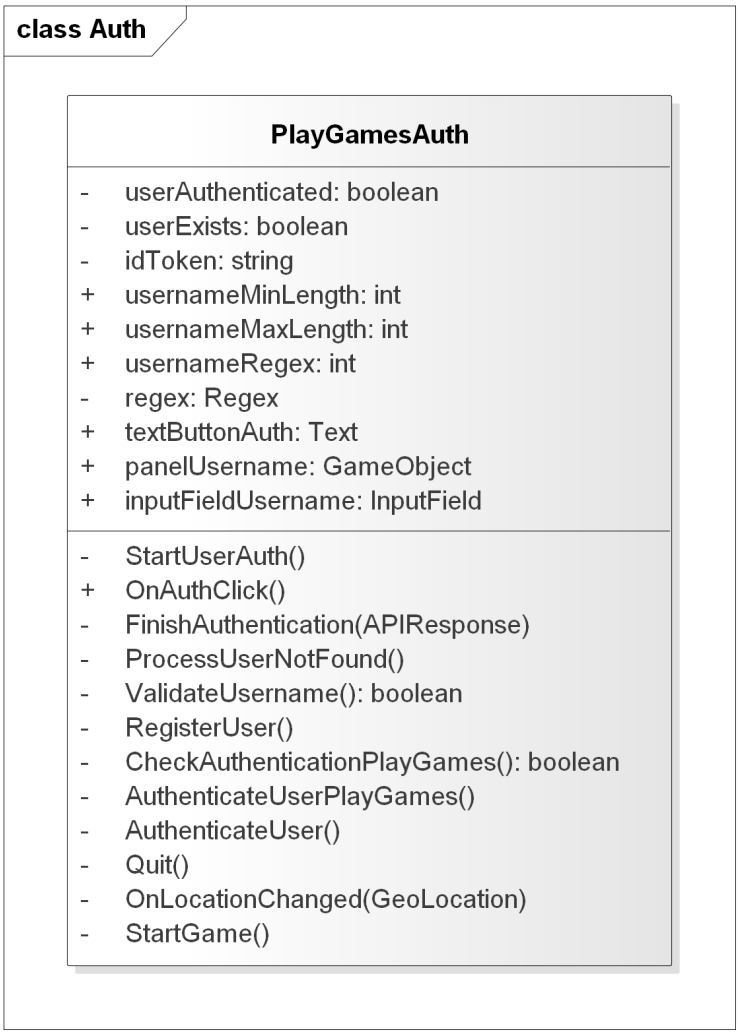

Figure C.2: Auth package class diagram

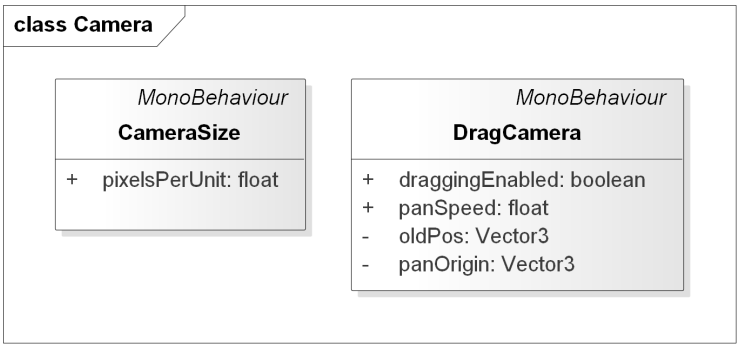

Figure C.3: Camera package class diagram

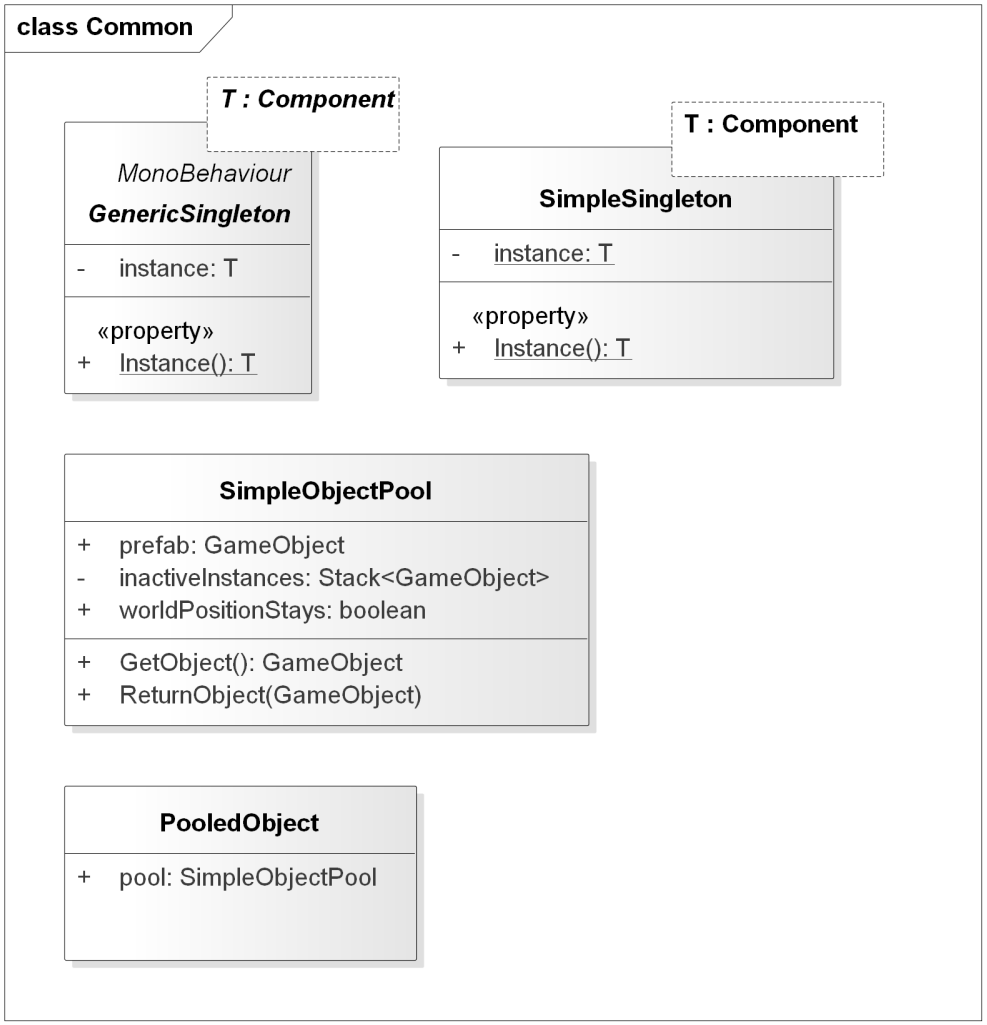

Figure C.4: Common package class diagram

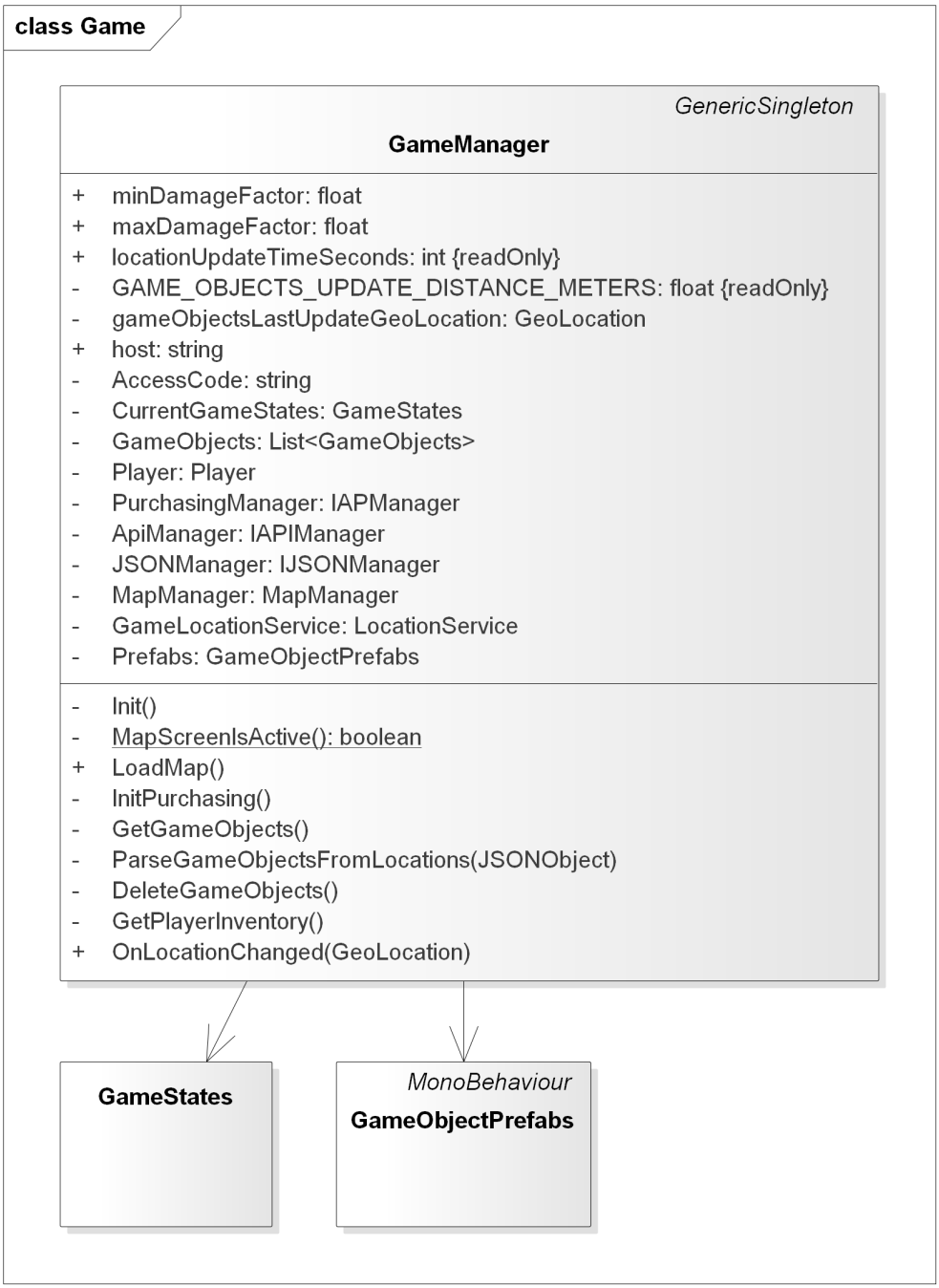

Figure C.5: Part of the Game package class diagram

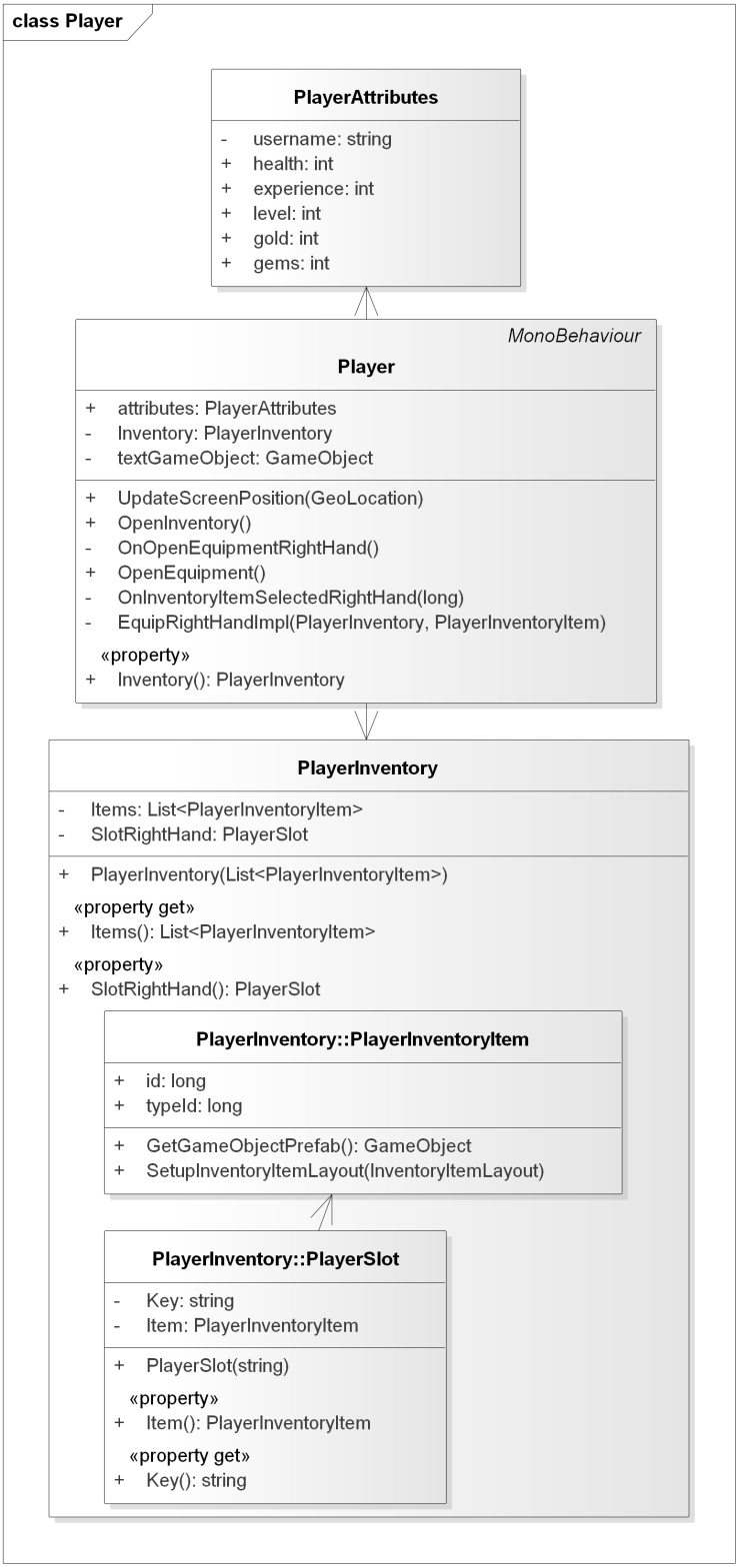

Figure C.6: Part of the Game package class diagram

Figure C.7: Part of the Game package class diagram

Figure C.8: Part of the Game package class diagram

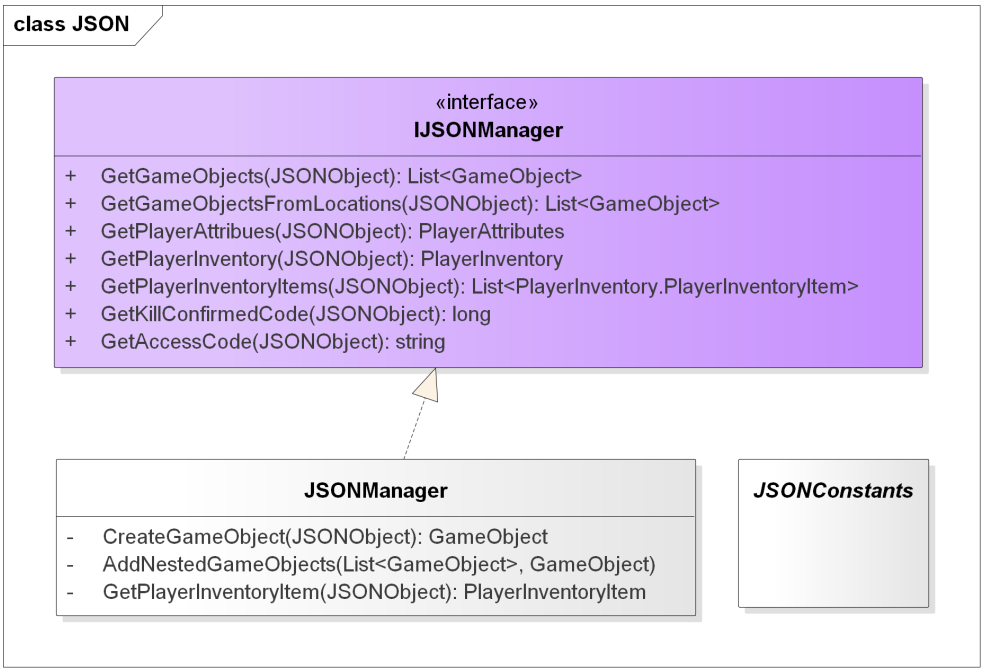

Figure C.9: JSON package class diagram

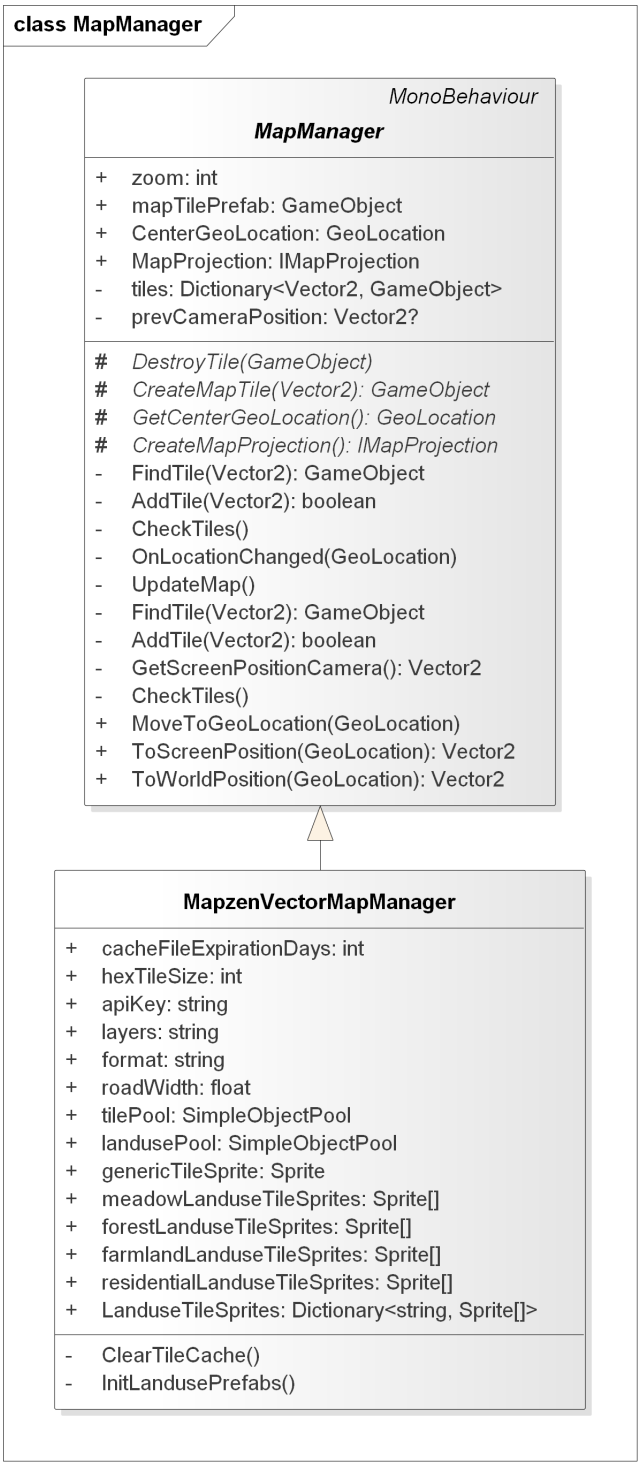

Figure C.10: Part of the Map package class diagram

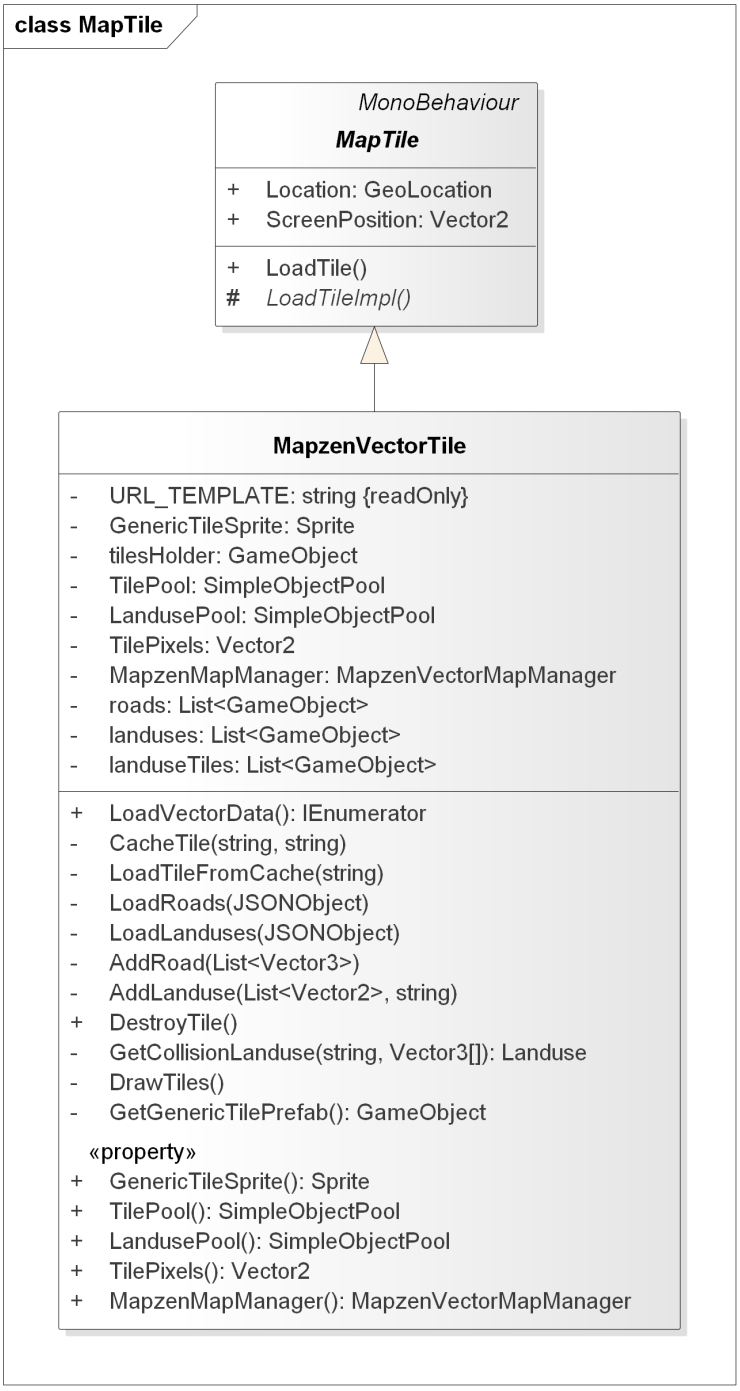

Figure C.11: Part of the Map package class diagram

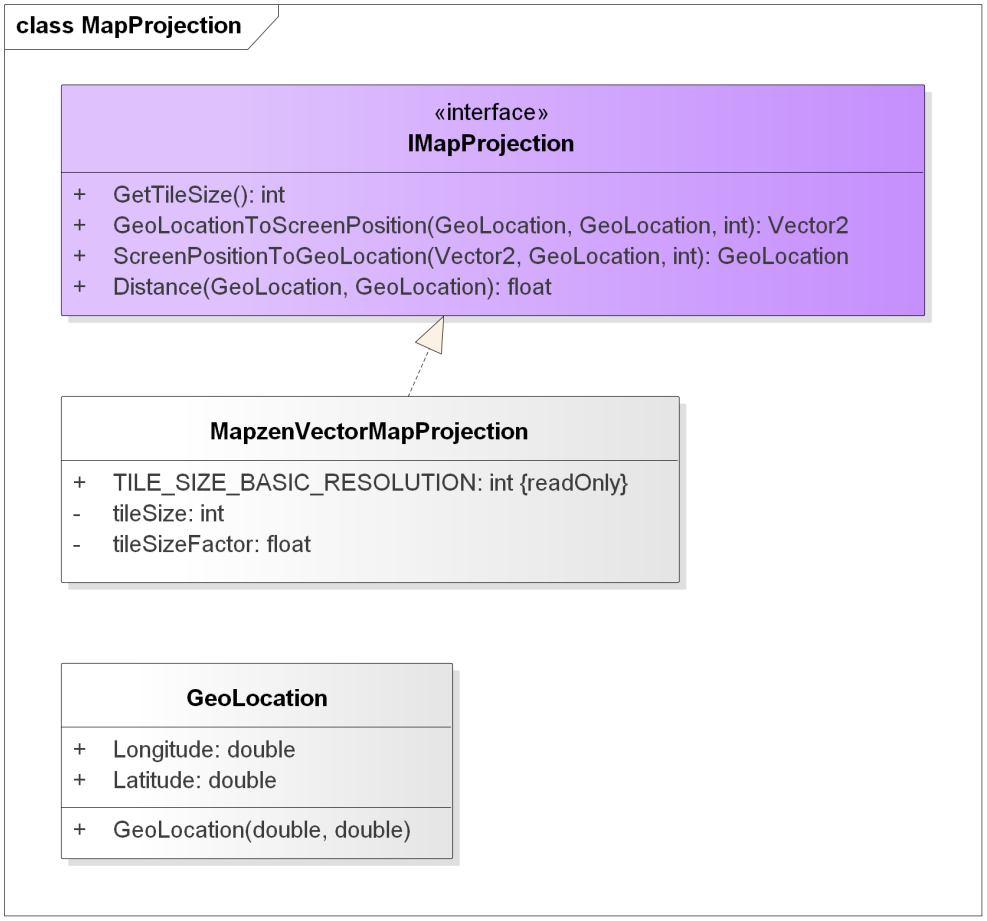

Figure C.12: Part of the Map package class diagram

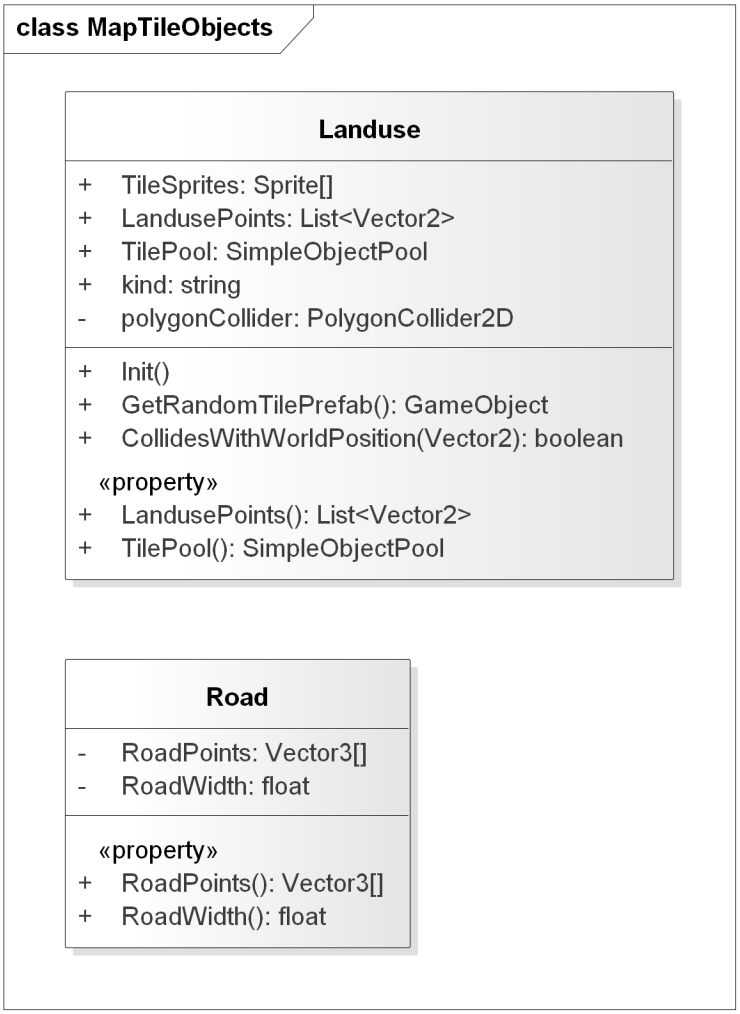

Figure C.13: Part of the Map package class diagram

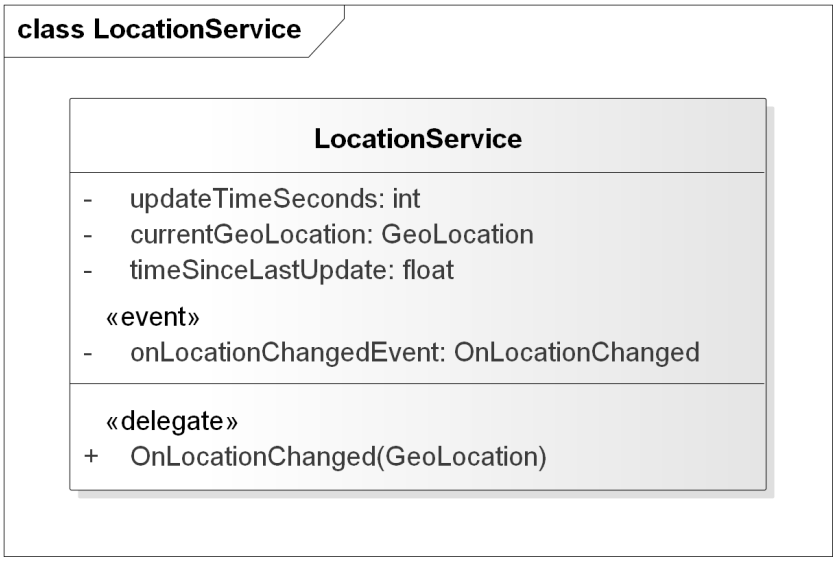

Figure C.14: Part of the Map package class diagram

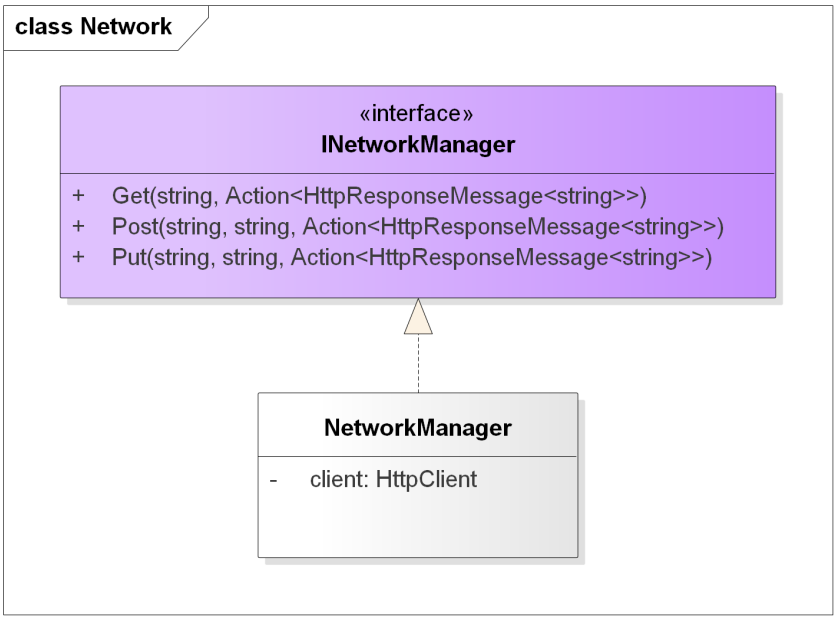

Figure C.15: Network package class diagram

Figure C.16: Purchasing package class diagram

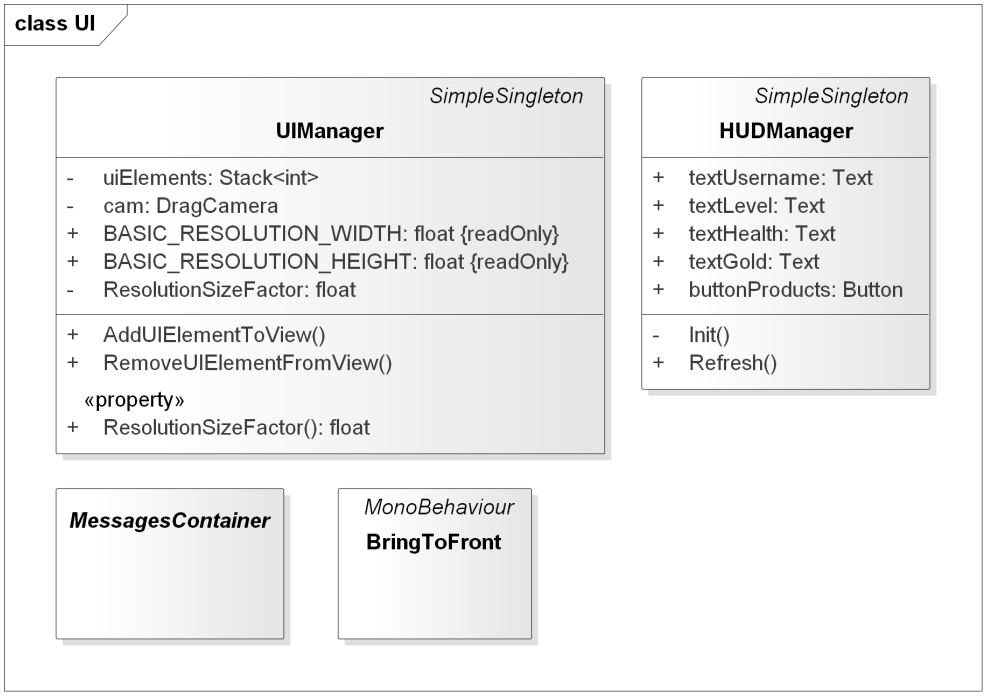

Figure C.17: Part of the UI package class diagram

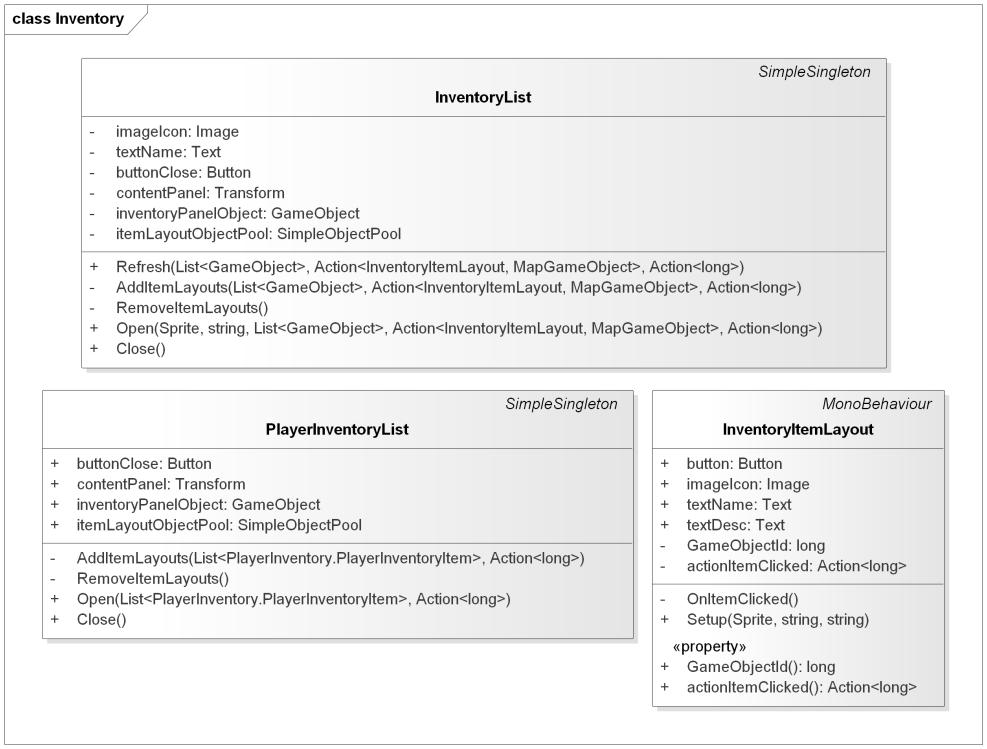

Figure C.18: Part of the UI package class diagram

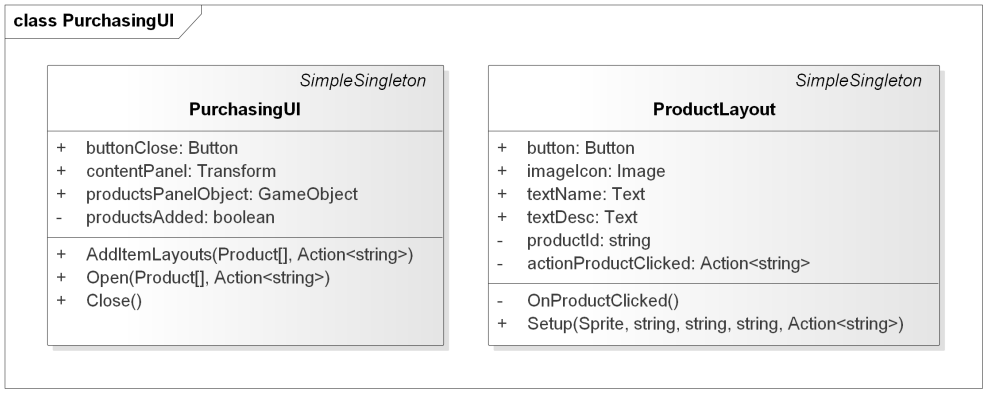

Figure C.19: Part of the UI package class diagram

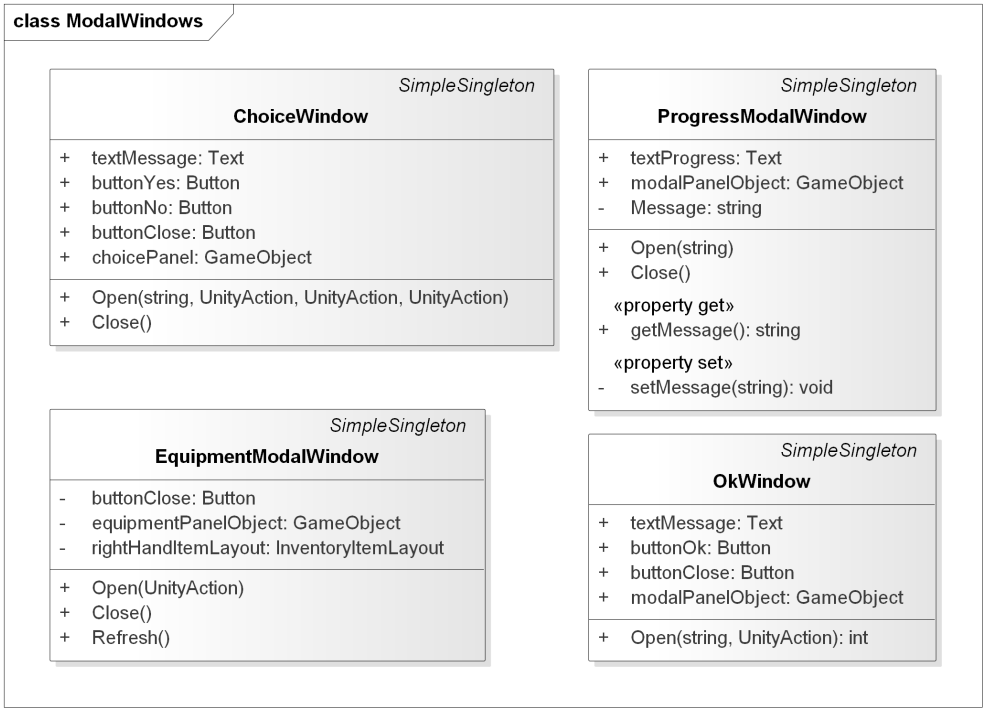

Figure C.20: Part of the UI package class diagram

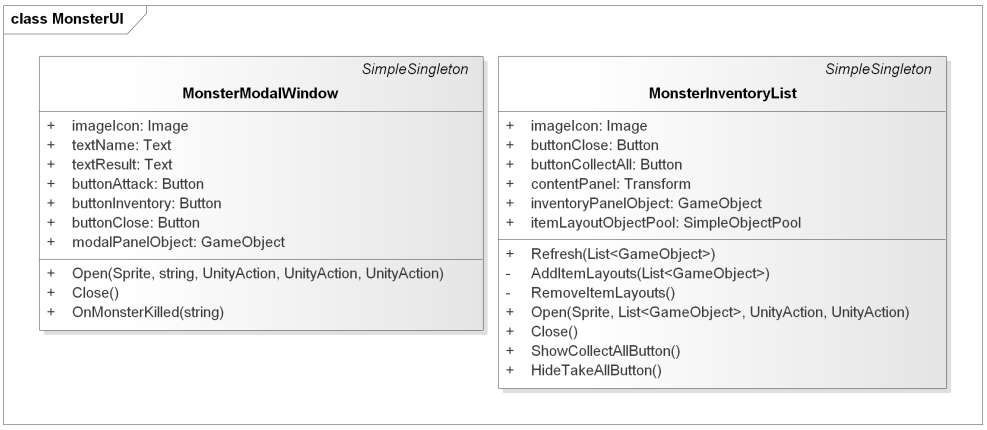

Figure C.21: Part of the UI package class diagram

Appendix D

Contents of enclosed SD Card

- `readme.txt` ..... the file with additional information
- `app` ..... the directory with the APK file
- `doc` ..... the directory with documentation
- `screenshots` ..... the directory with game screenshots
- `src` ..... the directory with source code
- `thesis` .. the directory of LaTeX source codes of the thesis and the output
  - `img` ..... the directory with thesis images
  - `_BachelorsThesis.tex` ..... the LaTeX source code of the thesis
  - `_BachelorsThesis.pdf` ..... the Bachelor's thesis in PDF format
  - `mybibliographyfile.bib` ..... the bibliography file
  - `FITthesis.cls` ..... the LaTeX template