System design document for Treasure Pleasure.

Team: Skyriders
Date: 2018-10-16
Version: 1.0
Members: David Weber Fors, Felix Rosén, Jesper Naarttijärvi,
John Gidskehaug Lindström and Oskar Lyrstrand

1. Introduction

1.1. Purpose

This Software Design Document provides design details for the TreasurePleasure mobile application and its implementation. With this document we want to clarify how we have planned to design our project and what the current architecture looks like.

1.2. Design goals

The design of the android code follows **MVP-pattern**. The advantage of using a MVP pattern over a MVC is that we can keep android specific views separate from the Controller/**Presenter** which makes the Presenter more easily testable. We decided to have one Top Model to whom the Presenter exclusively sends it requests.

To implement a map in the app, we employed the **Google Maps API.** To adhere to MVP pattern we try to keep the model clean from references to any Android classes. We also wanted to keep android functionalities, images and other assets separate from the model.

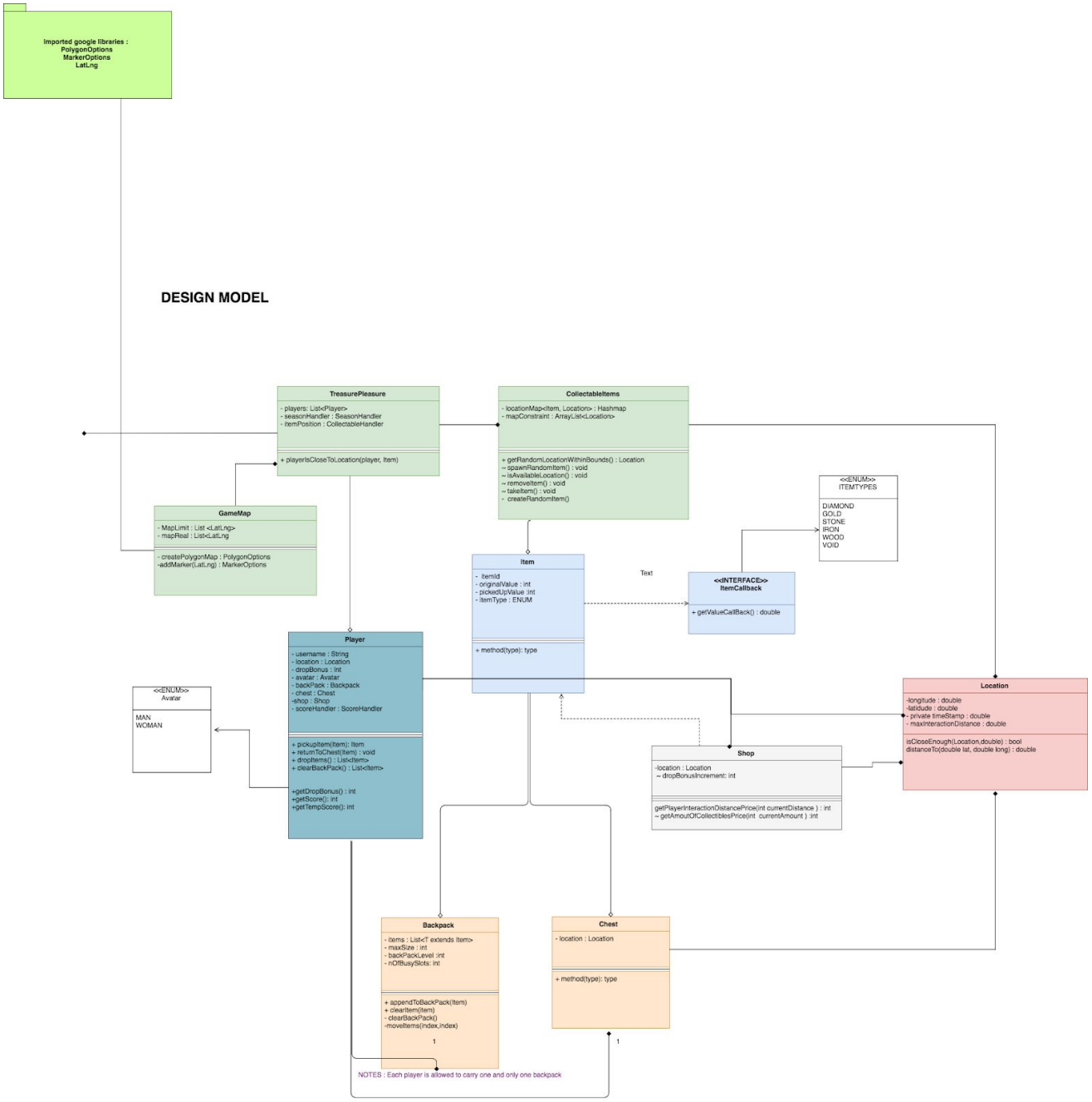
We also implement a wrapper class when using other 3rd party libraries or Android specific classes. Using a wrapper has the benefit that it's easier to swap one library for a new one. It's easier due to the fact that we only need to change the wrapper class and not our inner classes when swapping a library.

1.3. Definitions, acronyms and abbreviations

- **Activity -** Represents a single screen with a user interface.
- **Fragment -** Describes a portion of a user interface. A modular section of an activity, with its own life cycle and input events.
- **Activity LifeCycle -** A collection of callbacks to handle transitions between states. States are changed when the user navigates through, in to, or out of; the app**.**
- **MVP -** Model,View, Presenter. A design pattern for android native apps.
- **Singleton model** - Our main model is a Singleton, i.e. the instantiation of this class will always refer to the same object.
- **SDK -** Software Development Kit.
- **API -** Application Program Interface
- **Google Maps SDK -** A sdk that enables us to pinpoint a location on a map and add custom markers.
- **Presenter -** The "middle-man" in MVP. Contains presentation logic.
- **R class**, dynamically generated to identify all assets, created during the build process.
- **Static variables -** A variable that is common to all instances.
- **Top Model -** TreasurePleasureModel; the one to whom the Presenters exclusively speaks to, i.e. there are no dependencies for other Models.

- **Interaction radius -** A value that is used by the model to calculate if a two locations are close enough to interact.
- **Markers -** A marker is a custom google icon that's pinned on the map. In our game all objects appended to the map is a marker.

2. System architecture

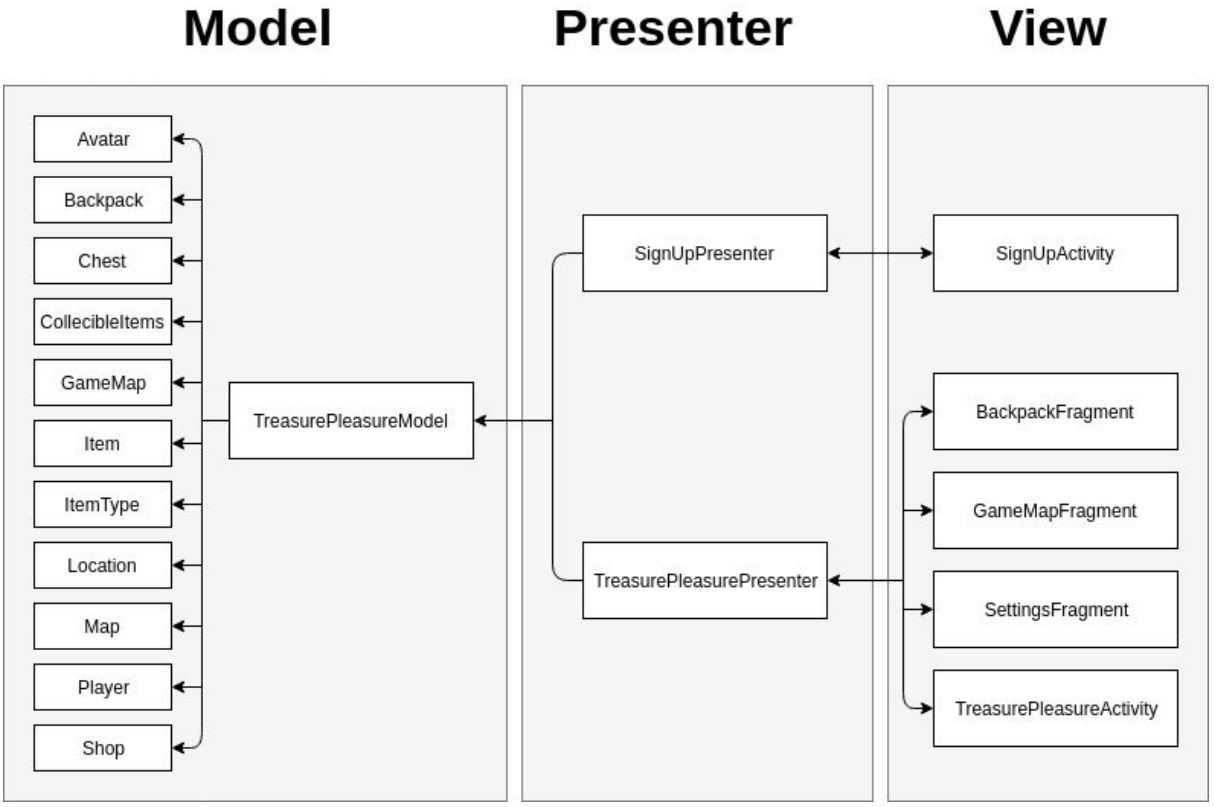


The application follows a MVP architecture. MVP, short for model-view-presenter, is a common architectural design pattern in Android. The user interacts with the view, which in turn communicates with the Presenter. The Presenter sends request for data or manipulation of data to the model and tells the view what to do with the result.

For example when a user interacts with the UI the view forwards the action to the presenter. The presenter then interprets the user action and calls the model to execute/hand the appropriate jobs/data. When the presenter retrieves the result from the model it tells the view what to display.

2.1. Subsystem decomposition

Initially we tried to have one **Presenter** for each **View** because that seems to be the general approach when using MVP in Android. However we realized that this type of approach was a bit redundant for the scope of our application. We decided to merge all of the Presenters into one, thus achieving simpler implementation with the drawback of having more code in one Presenter.



The project still has multiple Views; two **Activities** that are two different screens, and multiple **Fragments** that appears on top of either of these Activities.

2.2. Views: Activities & Fragments

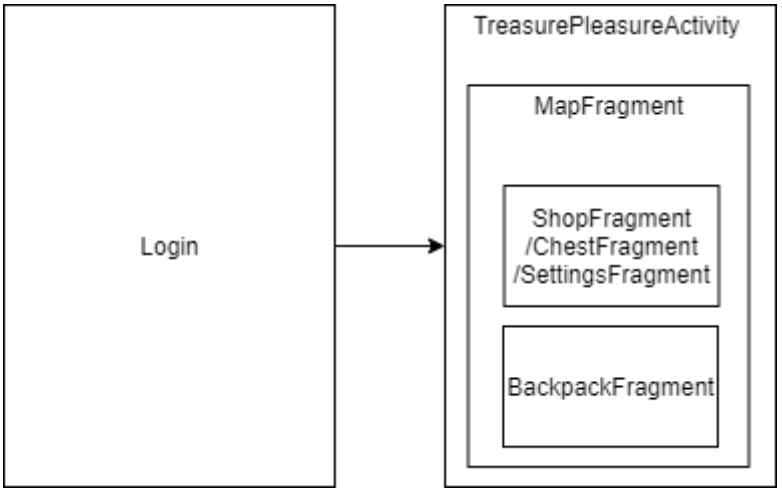
There are two activities in the application. On startup of the application a SignUpActivity will be displayed for the user, which initially instantiates the **Singleton** model of the application via its Presenter.
If the player signs up and succeeds the SignUpActivity will be destroyed along with its presenter and next Activity will be started. The idea is that the player will be prompted to

create a player first time they start the game, next time the Player will automatically jump into the game. [1]

The main activity for the game is the TreasurePleasureActivity, it instantiates and holds the TreasurePleasurePresenter. This presenter then utilizes the singleton model that was instantiated by the SignUpActivity.

In addition to the TreasurePleasureActivity there exists some **fragments**:

- A BackpackFragment widget is inflated if backpack button is clicked.
- A SettingsFragment widget is inflated if settings button is clicked.
- A ChestFragment widget is inflated if the player interacts with a chest on the map.
- A ShopFragment widget is inflated if the player interacts with a shop on the map.



The fragments are all placed on the Activity_Main.xml. On fragment inflation a reference to the Presenter is passed to the fragment. After passing the reference, all communication to or from the fragments will go via the presenter. When a fragment is no longer being displayed, it will be destroyed and any references to the fragment is set to null.

2.2.1. Known issues

The fragments can be buggy and the UX not consistent. For example there are buttons for functionalities in the SettingsFragment that are not yet implemented. And some of the Fragments have a close button while others only have a toggle button.

Handling identification of markers in GameMapFragment; marker click event fails sometimes. A user may encounter this problem in two ways:

- “Player not close enough to interact with item” instead of “Store/Chest not close enough”.
- Opening store/chest may require an extra click.

2.3. Presenter

The app has two Presenters, one created by the SignUpActivity, and the other created by TreasurePleasureActivity. The Activities create their respective Presenter, then establish a “two-way communication” between Presenter and Activity. This setup allows the Presenter to

[1] At the moment of writing this document this is not part of the implementation, there is no login screen.

handle requests initiated by the User in the view. Requests destined for the model is passed on, and appropriate UI updates are called.

2.3.1. Tests

Not yet implemented, however this is going to be implemented in the future.

2.3.2. Known issues

The TreasurePleasurePresenter instantiates a hard-coded name and Avatar. This was a quick fix before the SignUpActivity is implemented correctly where the User should set these parameters.

As a consequence of not having a Presenter for each View, the TreasurePleasurePresenter is bulky. The Presenter still requires attention, it will be reorganized to have consistent name conventions and the methods will be reordered.

2.4. Model

In accordance to the **MVP** pattern the model is separated from any view or controller, in other words the model does not know of any presenters nor views. We also use a **top model** that is public and this is the only model the presenters knows. The top model is responsible for hiding the inner structure of all other models and convert classes to standard types, e.g. the top model does not expose the Class Player, it returns a unique string to the presenter instead of the player object. The model also handles the data and business logic of the app.

2.4.1. Tests

TreasurePleasureTest.java tests the Top Model, other tests are for the specific classes.

2.4.2. Known issues

A known issue is when we try to remove a **marker** from the map. As of now, some markers references are not correct which means that we can’t update certain markers. An example of this is when a user tries to collect a item, the view tells the presenter that a marker has been pressed and the presenter does the appropriate arrangements and then tries to remove the marker. But the markers reference has been corrupted and we can’t remove the marker from them map, which means that the marker stays on the map and points to item Null. A work around for this is to clear the map every time marker.remove is called and then redraw all the markers, it’s both expensive and risky to do this and when a solution is found it should be changed.

An other known issue is the algorithm in Item for creating new random items, the algorithm is hard coded and will not update if we add new ItemTypes.

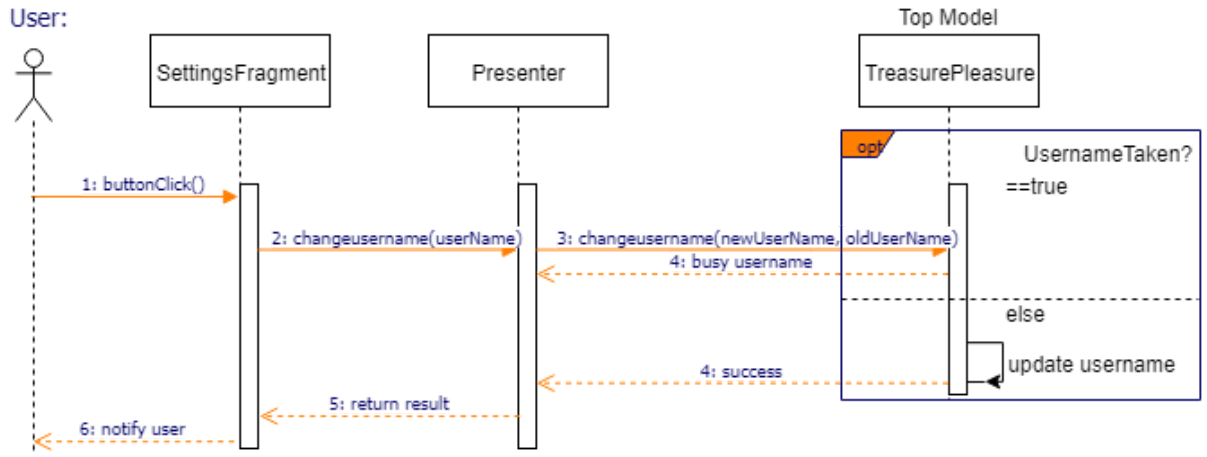
2.5. Third party dependencies

2.5.1. Google maps SDK

The SDK is an API provided by google that accesses Google's servers to display a map. We can append objects on the map with given coordinates. The SDK has built in functionalities for user gestures, which makes a player able to interact with placed objects on the map. The google map SDK is also used for responding when a players position is changed. The SDK tells us this by giving callbacks when the users location was updated.

2.6. System flow example

Let's investigate what the communication flow through the system can look like by examining an example. A typical call starts with a user action, followed by a call from the active UI to the model via the presenter. Consider the user story "As a player I want to have a choosable nick...". Below is a Sequence Diagram demonstrating a name update.



3. Persistent data management

3.1. Data handling

3.1.1. Global game settings

To handle all predefined global game settings (such as: how many slots in backpack, the interaction with item distance, how many items to spawn..), variables are separated into its on class in our data package. These values are implemented as static values, so that this class does not need to be instantiated. These values always are accessible.

3.1.2. Persistent Data

To handle persistent data, the app employs SharedPreferences APIs from google. A sharedPreference object point to a XML file containing key-value pairs. These files persist between app sessions. In our implementation current score is stored during the Main Activity, TreasurePleasureActivity, lifecycle callback method onPause. onPause is guaranteed to run if the activity is killed. In other words, score is saved when the main activity is killed. The main activity is always active, this ensures the integrity of the score. Score is fetched during the Main Activity onCreate method. More variables will be added to persistent storage during future development.

3.1.3. Image and layout assets

Layout definitions and images are separated from code and stored in app/res folder. Resources are accessed using resource IDs generated in the project **R Class**. To connect **Model Items** with associated images we introduced a class: AndroidImageAssets. AndroidImageAssets stores relevant image IDs and key value pairs to be used by the **Presenter**.

4. Implementations/functionalities explained

4.1. Functionalities

4.1.1. Location handling and position related calculations

We handle GPS coordinates in our Location class. The reason why we don't use the built in location service is because it's not possible to unit-test it on fake coordinates/an emulator. Defining our own location handler made it so that we could extend with more functionality e.g. checking if a given location is within a specified area and checking the distance between two Longitudes and Latitudes in meters. However, since we use **google maps SDK** we depend on the SDK to update our current location and give us callbacks in the view.

4.1.1.1. Game map - markers, spawning etc.

The model has two coordinates which tells the view where to focus on the map. These coordinates represent the northwest and southeast corners of the map. The top model also tells the view where to draw the polygon that displays borders of the map.

When creating a new **marker,** the presenter asks the model to generate a new coordinate within the map borders. The newly generated coordinates are used by the Presenter, delegating the view to draw a new marker.

4.1.2. Score calculation

Each item type has a value, and each player has a value multiplier. For a more dynamic experience, we introduced a feature allowing a user to trade score points to increase

different aspects of the game. For example a user can improve the value of items found on the map. The player class holds a multiplier, that manipulates item value when its put in the chest.

4.1.3. Item generation

The map always contains a constant amount of Collectibles. When a item is collected by a player the model automatically spawns a new random item at a random location such that two items have at least 3 **interaction radius** between them. Items are spawned according to their representative value, e.g. a diamond has a span of 95-100 in value and a 5% spawn chance while wood has a value of 0-35 and 35% spawn chance.

4.2. Exceptions and error handling

The easiest way to describe our usage of Exceptions is done with an example. Remember that the user interacts with the view, which in turn, communicates with the Presenter. The presenter sends a request for data or manipulation to the model and tells the view what to do with the result.

Changing user name: Assume the Presenter attempts to run the function changeUsername, residing in the model. If the username was successfully changed the presenter tells this to the view, but if the request was not successful (e.g. because the username is taken) the model sends an Exception. The Exception is caught in the Presenter, which then changes the state of the View accordingly. Now the Presenter can just take the Exception message and forward it as a string to the View. It doesn't have to know what went wrong it just have to know that something went wrong. Alternatively we could have used side-effects in the model and used switch cases in the Presenter to decode explicitly what went wrong.

5. Access control and security

The application has two types of users, debug mode and normal mode. If the debug flag is set in Data, debug mode is activated and some functionalities of the game are different. For example when in debug mode, interaction distance is infinite, allowing the user to collect items from any point on the map.

6. References

Google Maps SDK: https://developers.google.com/maps/documentation/android-sdk/intro
Google Styleguide: https://github.com/google/styleguide