NTNU – Trondheim
Norwegian University of
Science and Technology

TDT4240 - Software Architecture

Architectural Description Document

Food Brawl

Group A6
Android:

Kjetil Aune
Annie Aasen
Mikal Bjerga
Nikola Radenkovic
Jonathan Brusch Nielsen Trapnes

Primary focus attribute:
Modifiability

Secondary focus attribute:
Testability

February 27, 2015

Contents

1 Introduction

This document describes the architecture of our "TANK" artillery strategy game for Android developed by Annie Aasen, Mikal Bjerga, Nikola Radenkovic, Jonathan Brusch Nielsen Trapnes and Kjetil Aune. The game takes inspiration from the Worms series. It's a turn-based multiplayer game where the objective is to hit and destroy the enemy tank with different types of ammo. When the round is over, players can visit the store where they can buy new weapons and tank upgrades. We have decided to use different types of food as ammo.

2 Architectural drivers

2.1 Functional Requirements

- Playable as an offline multiplayer game - The game should be a turn-based multiplayer game. As a start, there will be a maximum of two players, but it should also be easy to extend with more players. There will be no need for an internet connection since the game is played locally on the phone.

2.2 Non-functional Requirements

- Support rapid design changes

2.3 Technical Constraints

- Must be run on android - The game will initially be developed for Android devices only, using the Android SDK. Since many Android devices have a fairly small screen size, the game should be simple and easy to play on small screens. It should still run well on larger screens, in example tablets.
- Must use libGDX - LibGDX is the chosen development framework, and the game must then be implemented using the functionality of this framework.

2.4 Quality Requirements

- Testability - must be divided into small modules for easy testing
  - To ensure that the game is testable, it must be divided into as small modules as possible. This will make the game easier to test, which is our secondary focus attribute.
- Modifiability - need to be easy to add new features
  - The game needs to be easily modifiable, meaning that it should not take a long time to extend the game with new features, in example new ammunition and teams. The MVC pattern will be utilized to increase the modifiability of the game.

2.5 Business constraints

- Short development time - The game must be completed in 9 weeks. This is a short amount of time, and we therefore need to keep the game simple and easy to implement.
- Developer inexperience - Since none of the developers have any experience with the libGDX framework, and only little experience with Android development in general, we will need to spend time learning how to use the development tools.

3 Stakeholders and Concerns

Stakeholder	Concerns
Users (Anyone running the program)	Is the game easy and intuitive to use? Is the game fun to play?
Developers (Group A6)	Will it be possible to develop and test the application in 9 weeks? Is the game easily modifiable? Is our program easy to test?
Evaluators (ATAM-groups and course staff)	Does the application meet all the functional requirements? Does the application meet all non-functional requirements? Is the implementation easy to understand (e. g. intuitive variable names and explained through comments) and well-documented? Is the documentation complete and straightforward?

4 Viewpoints

We have chosen to use a logic view, development view and process view, as explained in *The "4+1" View Model of Software Architecture* [1]. We have opted not to use the physical view and scenarios from the same paper, as we have deemed them to be unnecessary. The physical view is not needed as we will only use a single device, running a single process, which does not require any documentation to understand. Using the scenario view was discussed, but abandoned, due to the simplicity of our game. If we choose to modify it into a more complex game, e. g. with multiple game modes or different ways of playing the game (e.g online), a scenario view would have to be added to the documentation.

5 Architectural Tactics

The two main quality attributes we have chosen are modifiability and testability. To this end, we have focused on choosing tactics to ensure these two (mostly modifiability).

As we have decided to use the MVC architectural pattern, there are some modifiability tactics we must apply. These are:

- Encapsulation - This tactic is used on a high level, with the model part of MVC encapsulating the functional core data and functionality, and on a low level, within the model, where different data and function are encapsulated into Vehicle, Ammunition, Inventory, etc. (see logic view).
- Use an Intermediate - The controller part of MVC works as an intermediate between the input given in the view(s) and the data in the model. Another is the view, which is an intermediate between the data in the model and the device the view is shown on.
- Increase Semantic Coherence - By the definition of MVC, the model should contain everything we need to run the software (i. e. our game), demanding that all the necessary data and functions are present in the model. This is also used by separating Vehicle, with functionality to operate a vehicle, and Inventory, with functionality to keep track of the items and upgradres, from Player (see logic view).
- Use Runtime Binding - Views are bound at runtime, as different view can be bound to the data in the model at different times during execution. The views can also be opened and closed dynamically.

Other modifiability tactics we have used are:

- Raise the abstraction levels - The two abstract classes Ammunition and Vehicle (see logic view) are meant to making changes, especially additions of new types of ammunition or different vehicles, easier after the game has been fully implemented.
- Isolate the expected change - Several parts of the game are more prone to changes than the other, e. g. the store, which can be modified to contain new types of ammunition og upgrades or other items that are conceived after the game is completed. An example of a fixed part would be the game settings.

Another tactic we have utilized is I/O-management. This is a testability tactic and is achieved through the use of MVC. The view manages the output to the user, and

the input is managed by the controller. This way, we separate the interface from the implementation.

We have no specific tactics for ensuring performance or availability, as these are not deemed important to our game. This is because it runs locally for a small number of players, meaning an implementation with low resource-use and proper testing will suffice for performance and availability. Security is not an issue for this game, as no information is sent beyond the device we are using and no sensitive data is being stored.

6 Architectural and Design Patterns

Our game will be implemented using the architectural pattern known as MVC (Model-View-Controller). In the Model we are going to have the behavior of the application. The model manages the logic, rules and data of our game. The model updates the View; which is the visual output of information. We're going to have several views, e.g. one for each player. The users of our application uses the Controller. The controller takes the input and converts it to commands for the model.

The design pattern we will utilize is the Singleton pattern. The Singleton pattern is used to restrict instantiation of a class to one single object. By using the Singleton pattern we make sure that we have one global instance. In our game, we intend to implement the Singleton pattern on the settings of the game and the screen size. We intend to use Singleton as little as possible as it makes testing more difficult. This is noticeable when it comes to unit testing. Unit tests are supposed to be repeatable, but since singleton have a state, the changes need to be rolled back when the test is complete.

7 View

In this chapter, you should discuss the results you have obtained from your implementation. These can be correctness results, i.e whether the implementation behaved as expected, or numerical results that express runtime or energy measurements.

7.1 Logic View

Our logic view (see Figure 7.1) is based on the logic view from The "4+1" View Model of Software Architecture [1], but was built with a more recent modeling tool, UML. The modelling is also influenced by examples using the MVC-pattern from the lectures.

The logic view has been separated the different classes into the different parts of MVC. The model contains the classes which hold information about the game, such as game settings and players. These classes are specified with their most important fields and methods, e. g. a vehicle's ability to fire or the initial damage of a certain ammunition type. It also covers the relationship between the classes, specifying which classes are connected and how many of these relations are allowed. The controller simply contains the different controllers which will be used to change values in the model, based on what the user does in the view (GUI). The view contains a GUI- and an Android-class, to illustrate that the view mainly consists of a GUI and that we wish to implement this GUI on a device running the Android OS. Implementing the GUI will be made easier by using the framework libGDX, but this is not shown in the view part, as libGDX will be utilized for other operations as well (e. g. game logic).

As previously mentioned, we want to use libGDX to ease development. We have not included this in the logic view, as the full extent to which we will utilize libGDX is not decided, due to inexperience with the framework.

7.2 Process View

Figure 7.2 shows the process view of the application. It is divided into the different scenes you will encounter when playing the game. You start at the "Main menu".

The double arrows symbolizes a splash screen that will be put on top of the scene you came from (e.g. when going from "Main menu" to "Settings", the "Settings" scene will be put on top of "Main menu" and removed when abandoned.). A single arrow will remove the current scene and generate the subsequent.

When starting the game, you have a couple of choices: you can get information about the game ("how to play" and who developed it), change the settings or start a game with predefined settings. When playing the game, you can pause the game and from

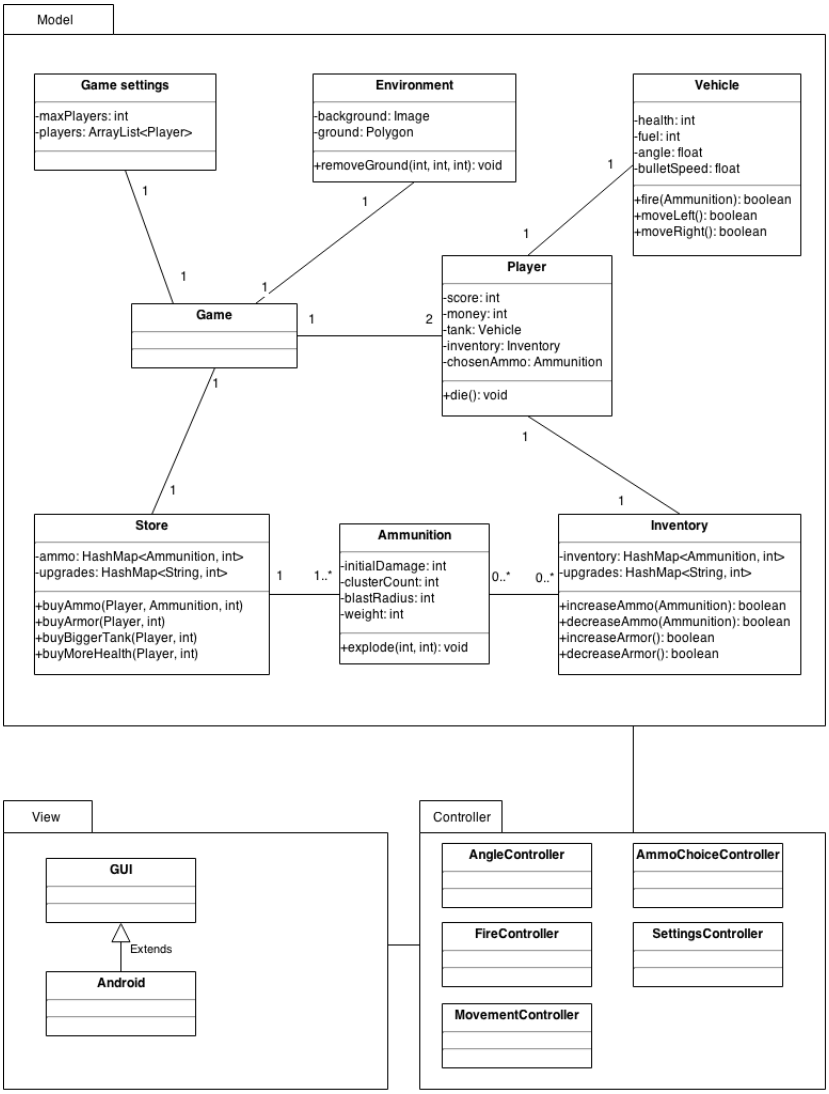

Figure 7.1: Logic view

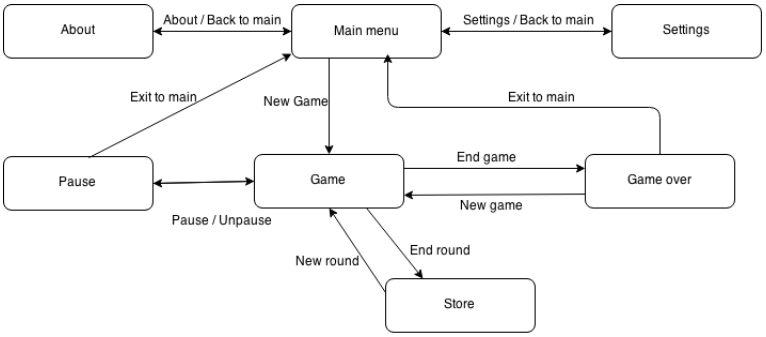

Figure 7.2: Process view

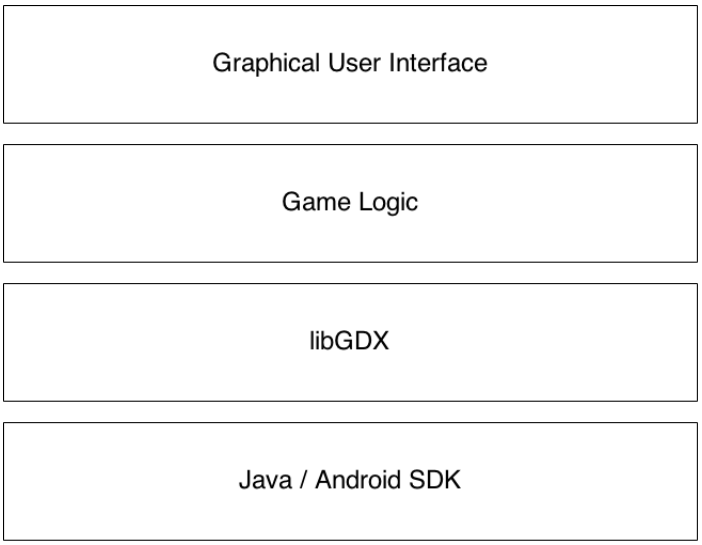

Figure 7.3: Development view

there end it. When a round is over, you will be presented with the store, where you can buy upgrades and ammunition. When this is done, you will start a new round. When the game is over (i.e. a player has won) you will be presented with two choices: you can either start a new game with the same settings, or you can exit the game and go to the main menu.

7.3 Developement View

The project is written in Java using the Android SDK. libGDX is the framework chosen for the game programming. These two gives us the foundation for writing our game logic and the game logic is used when writing the GUI. See Figure 7.3.

8 Consistency Among Views

There are no inconsistencies among the views of our architecture. The views do not contradict one another. In addition, our views are contained in each other. Process View is contained in Logic View; the Logic View is contained in Development View.

9 Architectural Rationale

We have chosen the Model View Controller as our architectural pattern. The MVC pattern separates the data from the display, which allows each to change without the other changing. This increases our primary focus attribute, modifiability, because it keeps the various pieces separate and easy to modify independently. The pattern also enhances or secondary focus attribute; testability. It naturally separates different application concerns into different, independent software pieces, which makes unit testing of the individual pieces a lot easier.

As for design patterns, we want to use the Singleton Pattern. This is a good way to store the game settings, since they will be used across multiple components. A drawback with this pattern is that it may make the game less testable. Therefore we will only use this pattern in a limited way.

10 Issues

In this chapter, you should discuss the results you have obtained from your implementation. These can be correctness results, i.e whether the implementation behaved as expected, or numerical results that express runtime or energy measurements.

11 Changes

In this chapter, you should discuss the results you have obtained from your implementation. These can be correctness results, i.e whether the implementation behaved as expected, or numerical results that express runtime or energy measurements.

Bibliography

[1] Krutchen. *The "4+1" View Model of Software Architecture"*. 1995.