Emergent Architecture Design StandUp Game

f 12th June 2015

Delft University of Technology

Nick Cleintuar	4300947
Martijn Gribnau	4295374
Jean de Leeuw	4251849
Benjamin Los	4301838
Jurgen van Schagen	4303326

Contents

1 Introduction

This document will provide insight into the architecture and the design of our product. It contains both the goals we seek to accomplish as well as an explanation of how the product is constructed.

1.1 Design Goals

In order to maintain the quality of the product during the development certain design goals are set. During the development these goals are kept in mind and should always be prioritised. These design goals are described in the next subsections.

1.1.1 Reliability

The product has to be reliable. This means that it should always be playable.

We guarantee the reliability of our product through testing. Testing is done using unit testing, integration testing, and real life testing. We also seek reliability by preventing mistakes made during development. This is enforced by programming in pairs as well as using version control with Pull request as well as static analysis tools: Checkstyle, Findbugs, and PMD.

1.1.2 Manageability

The product has to be manageable. Changes or extra features should be easily integrated in the product with minimal effort.

Manageability is obtained by using design patterns in the product. Design patterns provide structure to the project. Depending on the design pattern used it becomes really simple to make changes to the code.

Another approach to reach manageability is by using KISS. By keeping the product and implementations simple it becomes easier to change and add to it in the future.

1.1.3 Usability

The product has to be easy to use and have a minimal learning curve. This is important for new players to feel attracted to the game and prevent frustration by unclear instructions.

The game should be fun to play for most users. Finding what is fun and what is not is done by having real users test our game, as well as gathering their feedback. By doing this during the entire production, there are many chances to make changes or rethink the concept of our product.

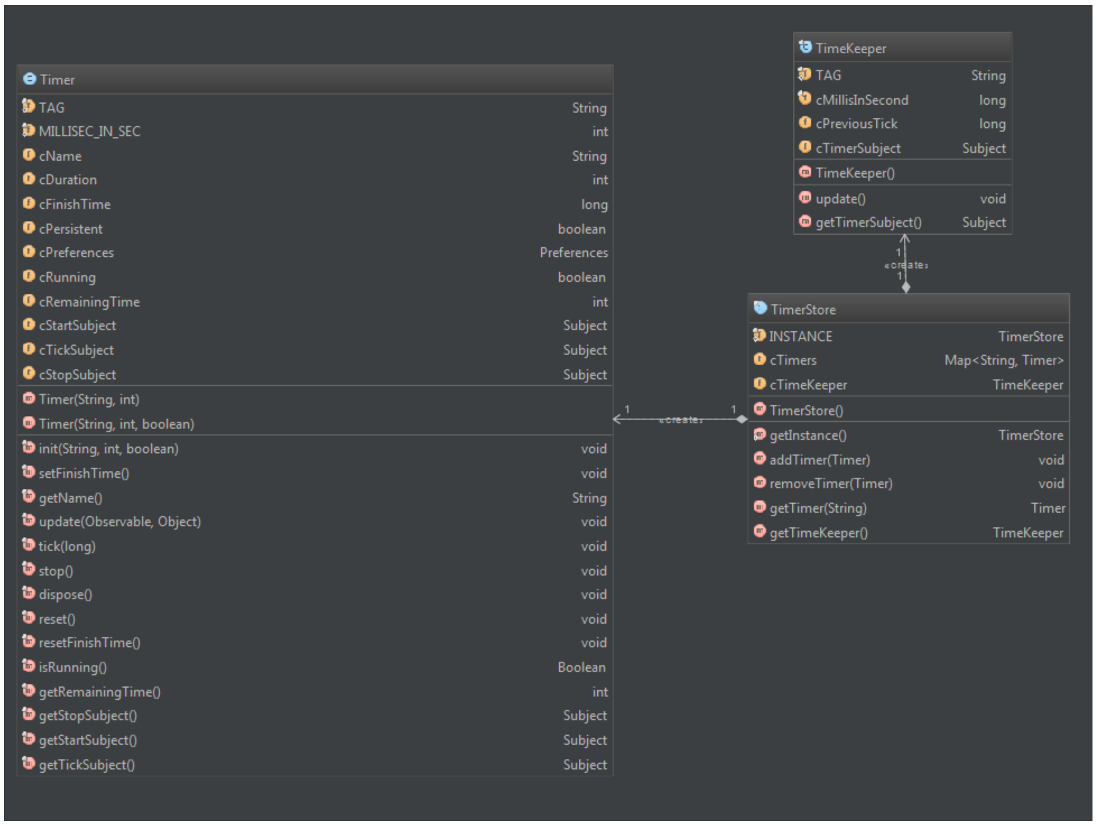
2 Software Architecture Views

This section provides insight in the architecture of our product. It is divided in four paragraphs. The first paragraph explains how the subsystems of our product work together. In the second paragraph we dive into how the hardware and the software of our product work together. The third paragraph provides insight in how the data is stored and handled. The last paragraph we discuss where collisions between subsystems might occur and how we prevent them.

2.1 Subsystem decomposition

Our product consists out of multiple subsystems. Each subsystem is explained in its own subsection. After that the interaction between the subsystems is explained.

2.1.1 Timers



The Timer subsystem consists of three major components. The first part is the TimerStore. The TimerStore is a singleton which maintains a list of timers that exist within the application. Timers can be added using the *addTimer* method and be removed using the *removeTimer* method. We're also able to retrieve specific timers easily using the *getTimer* method.

The TimerStore also has a TimeKeeper, our second component of the subsystem. The TimeKeeper keeps track of every individual timer and makes them update every second using his *update* method.

2.1.2 Assets

This part will be written once implemented. Three types of assets are used. The first asset type is Image. The second asset type is Sound. The third asset type is Music.

The separation of images can be defined as:

1. Skin images. These define the general application design and layout as seen by the player.
2. Reward images. Collectible images require an area of the image to have the colour RGB (120, 120, 120). This area will be used to create multiple coloured rewards based on single images.
3. Event images. These images are used to visualize actions during events.

Sound and music are used to give feedback to the user, for example when an event completes. Also a background music is used to increase the appeal of playing the game. The difference between sound and music is that sound is played after being loaded into memory in one piece. Music is played as a stream and is loaded into memory as a buffer.

adding this week: event occured sound, task complete sound

The Assets class loads the game assets such as images and sound files in memory when the application is started. Because loading a file from the flash memory in the RAM takes a few deca seconds, a loading screen is shown during this period of time.

2.1.3 Graphical User Interface

Since the game mechanics are going to be refactored we decided to wait before writing this part. The graphical user interface

2.1.4 Game mechanics

This part will be refactored in the next iteration. We left it out for now.

2.1.5 Interaction between subsystems

This part will be written once more subsystems have reached their final stage.

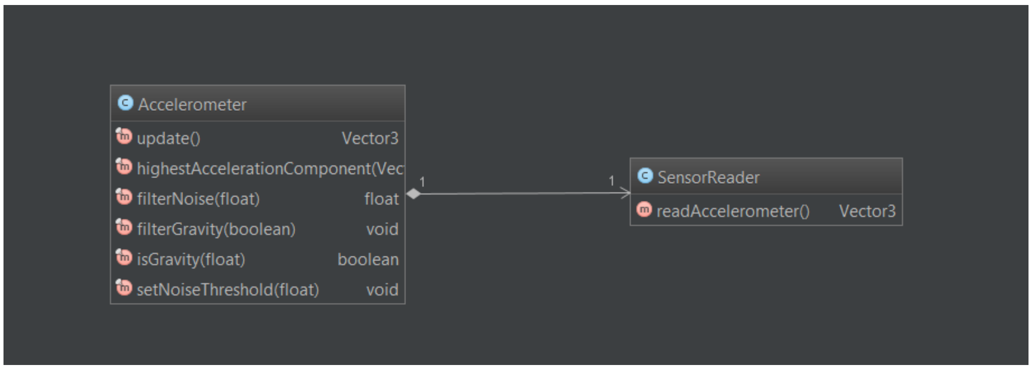
2.2 Hardware/Software mapping

For the product to be useful the user needs a device with access to a gyroscope and an accelerometer. These are needed to complete tasks and to measure activity. Hardware that is not necessary, but still very important to the functionality, is an internet connection and Bluetooth. These are used to communicate with other devices and the server. The server is used to store group information. In the subsections we will explain how the hardware is used in the software.

2.2.1 Gyroscope

This part will be written once implemented.

2.2.2 Accelerometer


Accelerometer
update() Vector3
highestAccelerationComponent(Vec
filterNoise(float) float
filterGravity(boolean) void
isGravity(float) boolean
setNoiseThreshold(float) void
1
1
SensorReader
readAccelerometer() Vector3

The class SensorReader reads the raw data from the hardware using the Gdx.input module. We then process it in the Accelerometer class. The accelerometer's update method updates the values of the Accelerometer every 1/60th of a second. As an example of methods we use for processing the input, we can name the *filterNoise* method. The Accelerometer never has an exact value of zero for each of its input channels, even when you let it lay down on a table for example. For this, we made the *filterNoise* function which automatically sets the acceleration to zero if it doesn't exceed a certain constant.

We also have to distinguish whether or not the input is actually a person moving the phone or that it is gravity 'pulling' the phone. The Accelerometer class also takes care of this with the *filterGravity* and *isGravity* methods.

2.3 Rewards

2.3.1 Collection

The collection is the place where gathered rewards gained from events are stored.

2.3.2 Server interaction

This part will be written once implemented.

2.4 Persistent data management

Data also has to be stored for the users. This is both user data as well as group data. For the user data local storage is used. For group data a server is used (see also: Server Storage). These will now be explained in further detail.

To store data persistently when no internet connection is available, a local

2.4.1 Device storage

The only storage we currently use on the device, is the preferences of the mobile phones. One of the main advantages of storing it in the preferences, is that it is easy to do in the source code. There is no need to play around with writing to the file and the file path.
Secondly, it also prevents users from viewing or tampering with the file itself. Generally, the preferences are not viewable or require a bit of work to be able to view them. To be able to 'edit' the values stored in the preferences, you will have to delete them.

2.4.2 Server storage

We have started with setting up a server where we store user data, so for example their collection of fishes.

2.5 Concurrency

Collisions between subsystems can occur on multiple levels throughout our product. On server level two users might try to change group specific data at the same time, this will be mediated by the database management system. Collisions might also occur within the application, when two subsystem would like to access the same recourse.

Glossary

KISS Keep It Simple, Stupid. 2

Pull request Pull request is a way to inform others of the changes you have made on a git repository and provides the opportunity for others to review your work and suggest changes. 2