Emergent Architecture Design StandUp Game

f 19th June 2015

Delft University of Technology

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

TU Delft Delft University of Technology

Challenge the future

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.

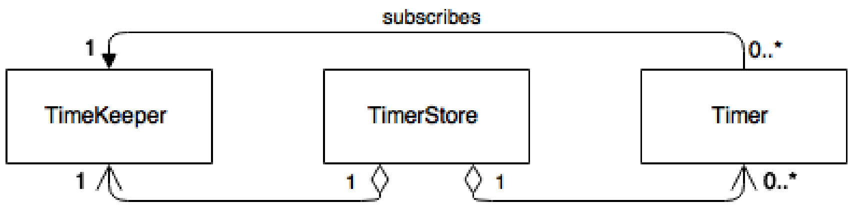
2 Software Architecture Views

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 *TimerStore*, the *TimeKeeper*, and multiple *Timers*.

As the name suggests, the *TimerStore* stores individual *Timers*. Each *Timer* can always be returned using its *name*. When a *Timer* is added to the *TimerStore* it is also immediately subscribed to the *TimeKeeper*.

The *TimeKeeper* acts as an observable metronome. To prevent many *Timers* from individually checking if one second has passed, we have the *TimeKeeper* do this task. When it notices that it has been a second since the last tick it will notify its *subscribers*, the *Timers*.

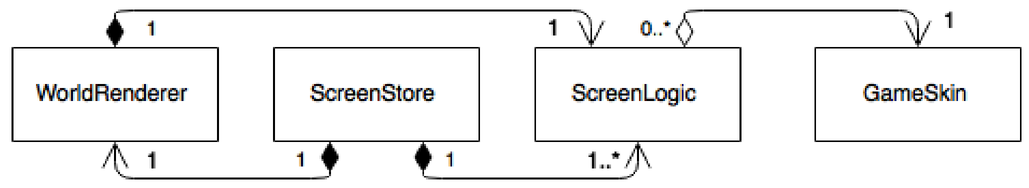
Finally we have the *Timers*. *Timers* are given a name, duration, and if they are persistent or not. Persistent *Timers* are added to the *preferences* of the device and can be recreated upon restart of the application.

Timers have three different *Subjects*. These *Subjects* act as *Observables* that can be observed by various components in the game. Each *Subject* identifies a different state of the *Timer*: *Start*, *Stop*, and *Tick*. *Ticks* are fired once every second while the *Timer* is running. *Tick* will notify its observers with the remaining time for the *Timer* to finish.

2.1.2 Assets

We have a Singleton *Assets* class that uses the *AssetManager* provided by LibGDX. The *Assets* is used to asynchronously load all the images that we require in the game. It also has convenience methods to access the various types of assets, for example: *Textures*, *Sounds*, and *Music*.

2.1.3 Graphical User Interface

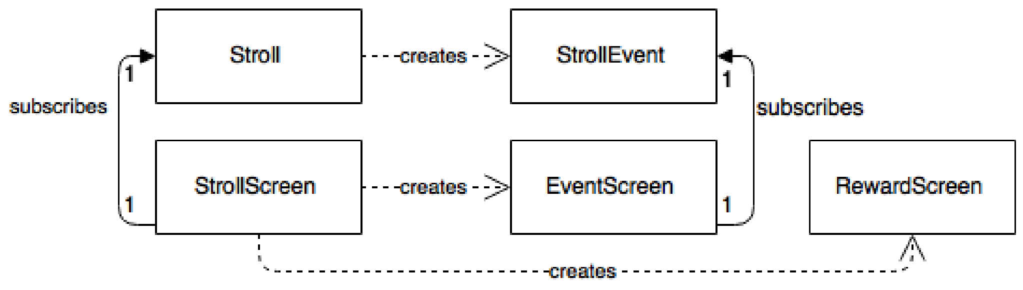


The *WorldRenderer* manages the display of elements and does many rendering tasks that are consistent on all screens. It contains a single *ScreenLogic* that defines the UI elements for that *Screen*. The *WorldRenderer* applies the components from the current *ScreenLogic* to its stage, which can then be seen and interacted with.

ScreenLogic is stored inside the *ScreenStore*. This is used to cache the screens and allow various observers to remain active while the screen is not the active screen.

Every *Screen* has a reference to the *GameSkin* class. The *GameSkin* contains various style definitions. It has easy accessor methods for things like *buttons*, *labels*, *textfields*, and more. *Screens* can use the *GameSkin* to easily create rich user interfaces.

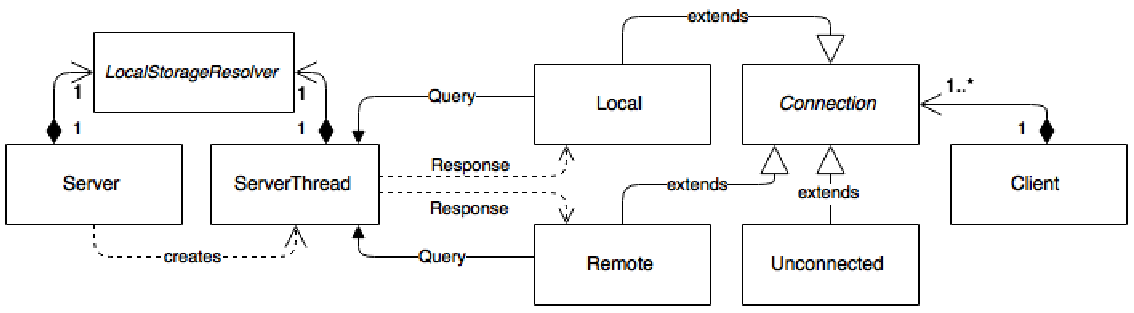
2.1.4 Game mechanics



An important part of the game is the ability to go on a *Stroll*. When in a *Stroll*, the player has a change to receive an event. The most significant problem we encountered was the need to separate the *Stroll*- and *Event-logic* from their *Screens*. Without doing this, testing the Stroll was impossible without creating the Screen as well.

To solve this issue, *Stroll* and *Event* contain an observable *Subject*. The *Screens* can then subscribe to these Subjects and receive updates about the state of the either the *Stroll* or the *Event.*

2.1.5 Client and Server



The *Server* and the *Client* together create backbone of the application. To store and retrieve information on every visit is extremely important for a properly functioning game.

The *Server* class initialises the access point for clients to connect to. To do so it uses the Java *ServerSocket*. When all the initialisation is completed, the *Server* goes in a permanent loop where it keeps waiting for any incoming connections. When a connection is established, the *Server* creates a new *Thread* that interacts with the *Client* and provides information.

One of the features of the *Server* is that it is used for both local and remote storage. The same server runs on the android device itself as well as on hosted server. This makes it easy for the *Client* to interact with both. The server is provided with a *LocalStorageResolver*. This resolver contains the proper connection with the database associated to that server.

The Client contains various methods that interact with a *Connection*. This *Connection* defines the state in which the Client is with a given server. When *Unconnected*, it will immediately reply *false* on any calls.

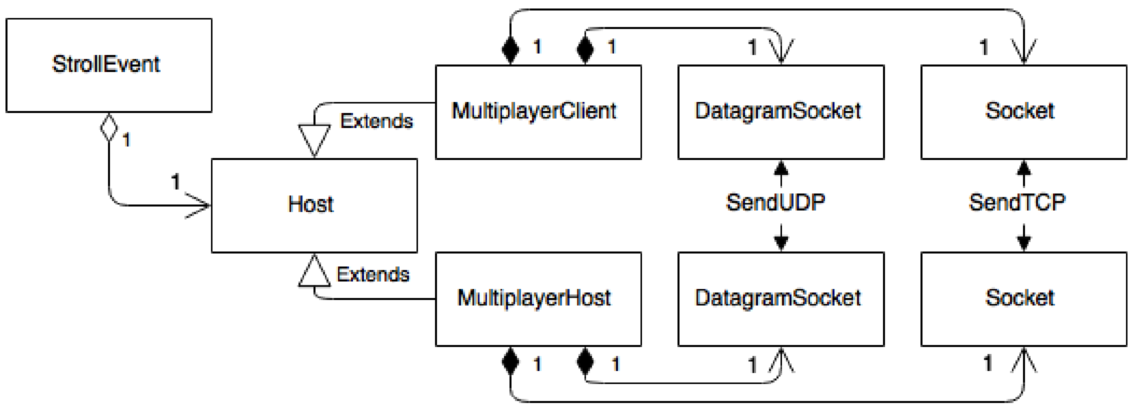
The *Local* connection is used for internal storage, and thus immediate replies are necessary. Since this is quickly received when the server is on the same device, the *Local* connection runs in a *blocking* method. This will ensure data is first retrieved and then immediately accessible.

The *Remote* connection however is run using a *threaded* approach. This is to prevent the application from freezing when no reply is received from the Server.

A final thing to notice is that the application provides the *Client* with a *ResponseHandler* whenever a request is made. This *ResponseHandler* updates the data *appropriately* when a *Response* is received. For the *Local* connection this is done immediately.

Since the *remote* queries are run in a separate *Thread* it needs to be added differently. This is done by adding a new *Runnable* to the *Gdx.app.postRunnables()* which will run that code before the next render cycle. This updates data appropriately without interfering with the rest of the application.

2.1.6 Multiplayer



Stroll *Events* can also be played with other players. These *multiplayer* events require a connection between the two parties. First, the *MultiplayerHost* gets a random code from the *remote Server*. The server stores the *IP* of the host in a database assigned to that code.

The *MultiplayerClient* can supply this code to the *remote Server* which then returns the *IP* of the host. This is used by the *Client* to connect to the *Host*.

When the two devices connect with each other, they open both a *TCP* and a *UDP* connection with each other. The *TCP* connection is used for messages that are not send often, but are required to properly make it to the other side. This includes, for example, sending the starting location of various *Actors* in *Events*.

UDP messages are used for rapid messaging, for example the rotation of a boat. This needs to be updated real-time on the other device as well. Since these messages are send at 30 messages per second, worrying about packet loss is not an issue.

2.1.7 Notifications

Notifications are send to remind the player a *Stroll* is ready to be played. This is an android only feature, implemented by interfacing a *NotificationController* through the game launcher, this requires some Android specific configurations can be made. The *Android* module implements the notification using Androids *NotificationManager*.

2.1.8 Aquarium

The *Aquarium* is for displaying a *Collection* maintained by a group on a shared screen. This feature is represented in the *Aquarium* module. This module consists of a connection component and a view component.

The connection with the remote server is made by the *Connector*. The *Connector* uses a *ScheduledExecutorService* to fetch collection data of the aquarium on a fixed interval. When the *Collection* is changed it uses an observer pattern to notify the view of the update.

The view of the collection contains a *CollectibleRenderer* for each collectible. Since the collectibles are only displayed and contain no game logic, the movements of each collectible fish are handled in the *CollectibleRenderer*.

2.1.9 Interaction between subsystems

Many of the subsystems discussed in the previous subsections are implemented as *Singletons*. This is done because these components are required at various locations. The *Singleton* subsystems are: *Timers*, *Assets* and *Client*.

The Timers are used for *Events* and game states. *Assets* are used for music in *Events* as well as the *GameSkin*. The *Client* is used in almost everything that needs some data stored or retrieved, this includes *GUI*, *Events*, and settings.

There has been put a lot of effort into separating every components from the others. Though *Observables*, or in our case *Subjects*, we provide other parts of the application about changes in the game.

2.2 Hardware/Software mapping

The product is developed using smart-phones in mind. Currently only Android smart-phones are supported, but iOS devices should also be possible since the game is built using the LibGDX framework. The problem however is that iOS development requires a Mac, which none of the developers had.

Two necessary hardware components are required. Both are accessible through the LibGDX library:

2.2.1 Accelerometer

The accelerometer is used for various parts in the game. The most significant to our context being: The detection of movement during a stroll. The player should only receive events when he or she is actually moving. The accelerometer is also used in many of the events that can be played.

The accelerometer is divided in two parts. The first being the state detection and the second being the game interaction.

For the state detection we used the *accellibandroid* library provided by Rafael Bidarra and authored by Job Becht, Wouter Groen, Olivier Hokke, Eric Rijnboutt. This library contains an *Observable* that notifies its listeners when activity state changes. Our own code observes this *Observable* and in turn notifies other components in the game.

The game interaction is handled by the *Accelerometer* class. This is a class that uses the LibGDX framework default accelerometer functions and enhances them for easy accessibility within the game. Enhancements are for example: gravity filtering and noise filtering. Noise filtering *flattens* the input vectors to a 0 value when below a certain threshold. Otherwise control can be hard and change unintentionally.

The *Accelerometer* class can be created anywhere in the code. This is because various Events require various settings. By calling the update method on the *Accelerometer* instance the event can update its state according to the changed vector.

2.2.2 Vibrator

The Vibrator is used to give the player feedback about the walking state during the stroll as well as when an event or stroll is available. When the players have put their phone away, they should still be aware of changes in the game. Since this is built into LibGDX, there was no need for us to further improve on this and we simply call the LibGDX methods.

2.3 Rewards

2.3.1 Collection

When the player completes an event during a stroll, he or she receives a reward. This is a random *Collectible* from a set of *Collectibles*. Each type of *Collectible* has its own shape and rarity. On top of the *Collectible* rarity, each collectible also has a random *hue* assigned to it that further increases its rarity. At the end of the *Stroll* the *Collectibles* are added to the players personal *Collection*.

2.3.2 Group

Players can also create and join groups. After creating or joining a group the player can donate fish from their personal *Collection* to the group. The group collection can then be viewed in the game, but also be displayed on a monitor using the *Aquarium*.

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. Group data is stored on a *Server*. These will now be explained in further detail.

2.4.1 Device storage

Two kinds of device storage are used. Timers are stored in the *preferences* of the player. Player data and Collection data is stored in an internal *SQLite* database.

The *preferences* are accessed using built in LibGDX methods. Every *persistent* timer is stored using its name. When the game launches, the *preferences* are read and the Timers are reinitialised on the time difference with the stored timestamp and the current timestamp.

The local database stores the players *Id*, *Username* and *GroupId* and *timestamps* for the *persistent Timers*. in the *User* table.

Personal *Collection* is also stored locally. It does so by storing a *Key*, *OwnerId*, *Type*, *Hue*, *Amount*, *Date* and *GroupId*.

2.4.2 Server storage

The remote server stores information about groups. This includes information about *Users*: *Username*, *UserId*, *GroupId*.

Group *Collections* are also stored remotely. It does so by storing a *Key*, *OwnerId*, *Type*, *Hue*, *Amount*, *Date* and *GroupId*.

Event Hosts are also stored using a *Code*, *IP* and *CURRENT TIMESTAMP*

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. We have done multiple optimisations to avoid this as much as possible including:

The *Server* creates a new *Thread* that handles communication with the incoming connections. This prevents the *Server* from only being able to handle a single request at a time.

Client instances contain a *buffer* that queues the outgoing *Queries*. This is to avoid one query being send and not receiving a reply. If another *Query* is send to the server and it happens to send data to the first request, errors are going to pop up.

The Client has both *local* and *remote* connection possibilities. Depending on which kind of connection is made, the process is run in a separate *Thread*. This is to prevent the application from blocking when waiting for an event, or when there is no network connection.

We have many *Subjects* that create the ability to diversely apply the *Observer* pattern. This greatly reduces the number of errors that occur because references are less likely to be *null*.

Anything that needs to run when a *Thread* has finished is put in the *Gdx.app.postRunnables()*. By doing this, the data is updated before the next rendercycle. This prevents concurrent modification exceptions.

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