System design document for the Speedtype project (SDD)

Contents

Version: 1.0
Date 2012-05-17

Author: Daniel Larsson

This version overrides all previous versions.

1 Introduction

1.1 Design goals

The goal is to make the design very loosely coupled in order for it to be possible to switch both GUI and model in possible future application additions or changes. The design must support the addition of new game modes and settings. The model must be possible to test completely isolated. For usability, see RAD.

1.2 Definitions, acronyms and abbreviations

All definitions and terms regarding the core Speedtype game are as defined in the reference section.

- GUI, graphical user interface
- Java, platform independent programming language.
- JRE, the Java Run time Environment. Additional software needed to run a Java application.
- SDK, Software Development Kit. Needed for developing Android applications.
- Host, a phone where the game will run.
- Round, one complete game ending with game over or possibly canceled.
- Score, the score for the player during one round.
- Activity, a class that holds the GUI, comparable to a JFrame in Swing.
- Service, an application component representing an application's desire to perform a long-running operation in the background, without interaction with the user. I.e. background music.

2 System Design

2.1 Overview

The application will use a classic MVC model.

2.1.1 Rules

There are very few rules to the game. The ones that do exist are very simple, and also vary between the different game modes. Therefore not much refactoring can be done. The main rule of all game modes is what happens when a player gives input in the form of a letter. Depending on which game mode is being played, completely different actions will be taken by the application.

2.1.2 The model functionality

The abstract class GameFactory enables the possibility to create several GameModes easily without heavy modification to the code. Words are stored in a SQLite database for the application to support future additions to the game logic. That way the dictionary is dynamic and allows for the application to add new words. The solution also provides support for future additions of new languages.

2.1.3 Value objects

The model classes expose some functionality, based on the abstract class GameModel. If the application needs any of the functionality from the models it calls the functions in the abstract class. The view classes are also based on the abstract class GameView which features all functionality the application might need. The only object that is completely free from functionality is the class Word. While sending data between the GUI and model the class Controller is used.

2.1.4 Unique identifiers

The database is the only globally unique identifier used. Everything else is directly connected or accessible without an identifier.

2.1.5 Event handling

The application uses three different types of input events. These are from the keyboard, touchscreen and accelerometer sensors.

All input event listeners are set in the menus activity. The input events are passed on to the controller, where they are handled in different ways. Most common is that they are passed on to the model.

2.1.6 Internal representation of text

In Android, the proper way to store strings used for layout is to gather them in an xml file. This way developers can easily add new languages to the application later on, without having to change anything but the strings file. The strings are fetched via the context, and therefore they are not available from packages that do not need them.

2.2 Software decomposition

2.2.1 General

The application is decomposed into the following packages:

- Activity. This is a kind of frame for the GUI, and this is what is created first in the application, i.e. it is the application entry class.
- Application. This is a vital class in Android, that handles events such as onTerminate(), to close the application.
- Controller. This manages user input events and updates the model.
- Model. Implements the application logic. This is separated from drawing, because of the fact that it must work on different platforms.
- Util. Contains global operation, i.e. words and the dictionary.
- View. This is where the drawing is done. The model decides what the view should be drawing.

2.2.2 Decomposition into subsystems

The application is built upon diffrent abstract classes where they easily can be extracted into subsystems if needed to.

2.2.3 Dependency analysis

The application was developed with as loosely coupled modules as possible. The basic idea is that no class has a connection to the view, and the view fetches all data to be drawn from the model. A complete analysis is shown in Figure 2.

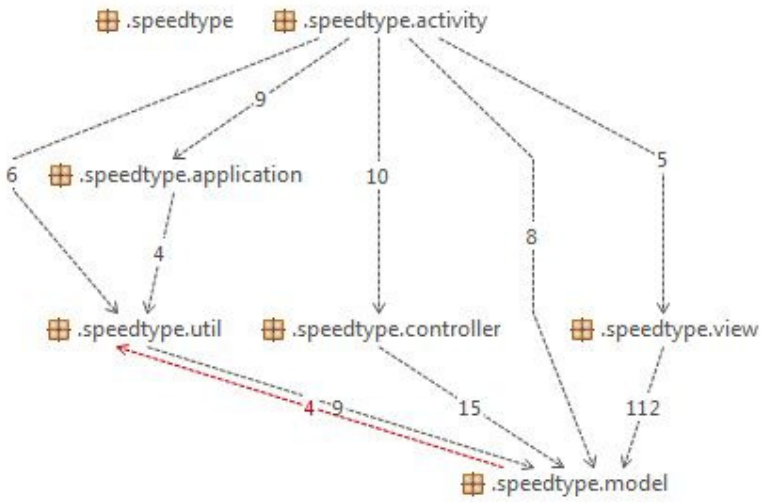

Figure 2: Structure analysis using STAN.

2.3 Concurrency issues

There are two possible threads running in this Application. One thread is the Speedtype application and the other one is running the different game modes. Services, such as background music, is ran in the main thread which for us means the Speedtype application thread. The concurrency issues are minimal and won't raise any problems.

2.4 Persistent data management

To store persistent data, the application uses an SQLite database. The first time the application is started, data is read from a text file to the database. Everytime the application is started, the dictionary is filled with words from the database.

2.5 Access control and security

There should not be any problems with access control or security. There is no instance variables exposed through getters except the two functions getActiveWord and getNextWord in gameModel, which in extension is two strings which is ok to expose.

2.6 Boundary conditions

When installed on the host it runs as any other phone application.

3 References

1. MVC, see http://en.wikipedia.org/wiki/Model-view-controller
2. Inspiration taken from Z-type: http://www.phoboslab.org/ztype/