System design document for Bomberman

Table of Contents

Version: 2.0

2012-05-16

Andreas Rolén, Adrian Bjugård, Joakim Persson, Viktor Anderling

@Override
This version overrides all previous versions.

1 Introduction

1.1 Design goals

The design must penis be loosely coupled to make it possible to switch GUI and/or partition
the application into a client-server architecture. The design must be testable i.e. it should be possible to isolate parts (modules, classes) for test. For usability see RAD.

1.2 Definitions, acronyms and abbreviations

- GUI, graphical user interface.
- Java, platform independent programming language.
- JRE, the Java Run time Environment. Additional software needed to run an Java application.
- Host, a computer where the game will run.
- Round, one complete game ending in a winner.
- Match, a number of rounds ending in a winner or if canceled.
- Game, a number of matches ending in a winner or if canceled.
- Shop, a phase between two rounds where the players can buy upgrades with the points they earned during the rounds.
- Power-up abilities, different abilities that can be upgraded by picking up items in game or buying upgrades in the shop. Includes speed, bombstack, bombrange ???
- Abilities, different abilities that can be upgraded at the shop by buying upgrades. These abilities are health points, ???
- MVC, a way to partition an application with a GUI into distinct parts avoiding a mixture of GUI-code, application code and data spread all over.
- LWJGL, A Lightweight Java Game Library.
- Slick, A Game Library based on LWJGL.

2 System design

2.1 Overview

This application uses the MVC model.

2.1.1 Rules

The Bomberman Game rules could vary by implementing different game modes. We handle it by having a IGameController interface that all subclasses must implement if they should be a game mode. This gives the model the flexibility to change game mode dynamically and enables developers to easily plug in their own game mode. However we have not extracted all the game rules handling code to the GameController class. The Player class keeps count internally on how many matches, rounds they have won in a game. The Model interface delegates all that are related to the game rules to the active rules class.

2.1.2 The model functionality

The models functionality is exposed by the interface IBombermanModel. At run an unique instance of the BombermanModel will be generated that the rest of the application will integrate with. This minimizes the number of objects in the model that the rest of the application needs to keep track of. This is also very convenient with several states using the same parts of the model. Through the interface the application can request functionality like updating the game and notifying it that a round is over etc.

2.1.3 Event handling

We have designed the model without using any kind of system for notifying potential listeners about changes in the model except for the audio system(see 2.1.7). The model is designed to easily be extended with an observer/observable system, however considering our implementation there is no direct need for such a system.

Instead we utilize the benefits from an external Game Library called Slick. Slick continues repaints the views which for us means that we don’t need a way to notify our views that the model has been updated and instead it ask the model for the values it need. However if we replaces the graphical representation of the game we will consider implementing an observer/ observable system for the views.

2.1.4 Map Generation and Loading

In order to load and create maps easily there is a desperate need for a simple system. We have solved the problem by a creating a system where we read from a .txt file that have been formatted in a special way(see Appendix). Our subsystem then converts the .txt files into the Tile matrix that we use in the game. This systems enables us to easily create new maps and since we created the formatting ourselves it is really easy to read and work with.

2.1.5 Game States Design

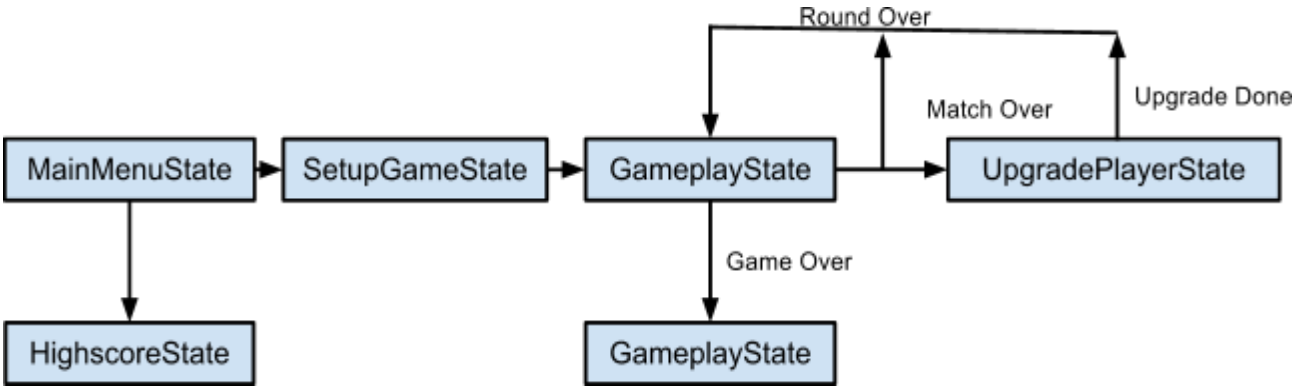

Figure 1: Game Flow

Even a basic game contains separate states for different views. In order to keep our design easily maintainable and expandable we have separated the application into different states consisting of and controller and view For an overview of the the different states in the application see figure 1. This design works well with the slick framework enables us to split up the application in smaller and more manageable pieces. However one problem that occurs is that sometimes two states needs access to the same part of the model. In order to avoid ugly dependencies we solved the problem by using one global interface to talk to the model(see 2.1.2).

2.1.6 Input System

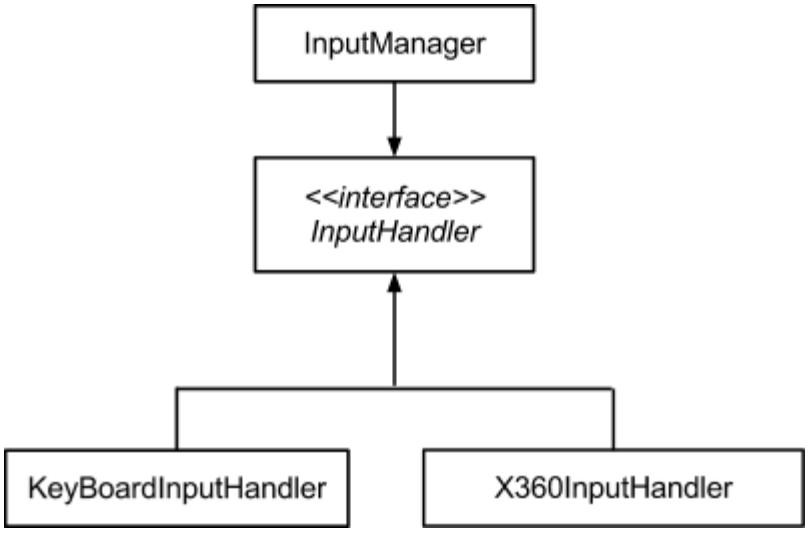

Figure 2: Input System

Through the slick framework we got support for connecting Xbox 360 controllers to the computer and use them in the game. This raises a potential problem managing two different input system and listening to their respective events. We solved this problem by creating a system with an InputManager class and InputHandlers that registers themselves in the inputmanager(see figure 2). This lets the state controllers interact with one interface without needing to worry about anything else. This means that we can easily add support for other input types as long as they implement the InputHandler interface.

2.1.7 Audio system

2.2 Software decomposition

2.2.1 General

- audio, the package including all the audio classes.
- controller, the package including the statechanging and the input package.
- controller.input, the package including all the input classes.
- gui, the package including all the GUI related classes.
- model, the package including all modelrelated classes.
- model.constants, the package including all the constants.
- model.map, the package including all the map related classes.
- model.player, the package including all the player related classes.
- model.tiles, the package including all the tiles related classes.
- model.utils, the package including the position related classes.
- model.tiles.bombs, the package including the bombtile classes.
- model.tiles.factory, the package including the poweruptile factory.
- model.tiles.nonwalkable, the package including the non walkable tile classes.
- model.tiles.walkable, the package including the walkable tile classes.

Package diagram. For each package an UML class diagram in appendix

2.2.2 Layering

The layering is as indicated in Figure (missing) . Higher layers are at the top of the gure.

2.2.3 Dependency analysis

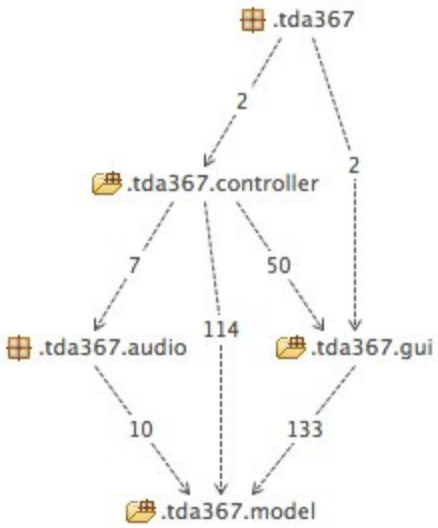

Figure 2: Dependency analysis

Dependencies are as shown in Figure(NB: not final draft). There are no circular dependencies between the packages.

2.3 Concurrency issues

NA

2.4 Persistent data management

NA

2.5 Access control and security

NA

2.6 Boundary conditions

NA. Application launched and exited as normal desktop application (scripts).

3 References

1. MVC, see http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller
2. Slick, see http://slick.cokeandcode.com/
3. LWJGL, see http://lwjgl.org/

APPENDIX